PYTHON

FLASK & django

FULL STACK PYTHON FOR WEB DEVELOPMENT

Build Web Applications In Python Using django & Flask Framework

Emenwa Global

Python
Flask & Django

Full Stack Python for Web Development

www.emenwa.com

Contents

Introduction To Full Stack Python Web Development

Full stack developers are hard to find these days! You are among a tiny percentage that will stand out as a professional web developer.

As technology advances, the technological world undergoes rapid change. The days when a developer can easily keep a job with only one programming language for years without picking up new skills are long gone. Many of us enter the world of programming and web development knowing only one or two technologies, like Java, C++, or JavaScript, but that is no longer enough.

Before now, web developers used to work in groups on specialized projects, such as front-end development carried out by a different team of programmers and back-end development written by a different team of programmers, also referred to as server-side developers.

Nowadays, everyone is looking for full-stack developers, someone who is knowledgeable in both front-end and back-end technology and can work independently to develop a fully functional web application.

Two ways that people can use a computer to make websites are Django and Flask. Web developers use two different programs to create sites and web apps. These two programs are called frameworks, and they help people make fun, cool sites that look nice and run fast. Django is one of the top frameworks because it is open source and works well. But you must learn

about web apps to build different web pages and website templates. You will need to create different apps from scratch to develop a single web app. The second way, Flask, is simpler and easier.

Flask is a newer framework that is easier to learn for building simple web apps. That is a lovely place to start learning web development.

That is why you should be happy that you are on this journey to learn how to build websites and web apps with Flask and Django. Two in one!

Learning a framework that complements your primary area of expertise is preferable. For example, a Python developer would learn more from Django and Flask than from Angular. Similarly, a JavaScript developer would learn React and Node JS better than Django and Node JS. This book is only for developers who are familiar with Python programming language.

Full-stack development with Python

The Python programming language has many advantages, one of which is a relatively quick development cycle. The career opportunities that Python as a full-stack engineer may open up for you, however, may be its best feature. Full-stack and back-end Python engineers are still needed. Python is frequently used in data science and machine learning, so as a full-stack engineer, you can add these to your repertoire of back-end skills.

Back-end development using Python

Flask, Django, Turbogears, CherryPy, Pyramid, Bottle, and Falcon are just a few Python back-end frameworks. However, we'll talk about Django and Flask, the two most widely used frameworks.

Django

The developer community for the free and open-source project Django is sizable. As a result, its security, user, and role management, as well as database migration management feature, frequently improve. Additionally, RESTful Web APIs are fully supported by the REST framework in Django.

Flask

Another well-liked Python web framework is Flask. It's referred to as a

micro-framework and is lighter than Django. The back end of APIs is frequently developed using Flask. The Flask community has lots of pluggable features available.

Front-end development using Python

Python front-end development is still in its infancy in comparison to back-end development. Because of this, the front-end typically uses HTML, JavaScript, and CSS. In this course, you will learn how to implement Bootstrap with Django and Flask to create a beautiful front-end interface.

The first part of this book focuses on the Flask framework, and the latter part discusses Django. We hope you find it interesting.

Typically, it should not take you more than a few weeks at most to learn Flask and start to develop apps. However, that depends on your other commitments and reasons for learning.

This book is divided into short chapters, which are isolated lessons. Many teachers would write their books and tutorials as a long lesson where they create an example app and update it throughout the book to demonstrate concepts and tasks. That is not the case here. In this book, we include examples in each lesson to illustrate the concepts, but we have examples from other chapters that may not even be related to the previous. Hence, the book is not meant to be combined into one large project.

This book will help you learn Flask by building a series of projects and showing you verifiable screenshots so that you can use the skills to create different projects with Flask. Please, as you read this book, I recommend opening your computer and implementing the codes as we go. The lessons in this book will help you create a web application on your own.

The second part of this book is for Django. 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 tedious 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.

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, resulting 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.

Part 1 Flask

Chapter 1 – Learning the Strings

Assuming that you are an intelligent programmer, you must identify and use specific terms and conventions that guide the format of Python codes. You might even know some of these conventions. This chapter discusses them. It will be brief.

The PEP Talk

A PEP is an abbreviation for "Python Enhancement Proposal." Python.org indexes and hosts these proposals. PEPs are classified into several categories in the index, including meta-PEPs, which are more informative than technical. On the other hand, technical PEPs analyze enhancements to Python's internals.

PEPs such as PEP 8 and PEP 257 guide how we write our code. Guidelines for coding style are included in PEP 8. PEP 257 specifies procedures for docstrings, the widely used method of documenting code.

PEP 8: Style Guide for Python Code

Python code should follow PEP 8 as the coding style. This is like a format for writing Python programs. You can read about it if you want.

When your code grows to multiple files with hundreds or thousands of lines of code, PEP 8 style will make it much more readable. Furthermore, if your project will be open source, potential contributors will likely expect and feel at ease working with code written with PEP 8 in mind.

One crucial suggestion is to use four spaces per indentation level. Not tabs. If you violate this convention, switching between projects will be difficult for you and other developers. Inconsistency like this is annoying in any language. Because that indented space is vital in Python, switching between tabs and spaces could result in errors that are difficult to debug.

PEP 257: Docstring Conventions

Another Python standard is covered by PEP 257, and it is called docstrings.

A docstring is a string literal that appears as the first statement in the definition of a module, function, class, or method. A docstring of this type becomes the object's __doc__ unique attribute.

Relative imports

When developing Flask apps, relative imports make things a little easier. The idea is straightforward.

For example, if you are developing an app and need to import User model from myapp/models.py module. You might use the app's package name, such as myapp.models. This would indicate the location of the target module relative to the source using relative imports. We use a dot notation instead of a slash, with the first dot representing the current directory and each subsequent dot representing the following parent directory.

```
# myapp/views.py

# An absolute import gives us the User model
from myapp.models import User

# A relative import does the same thing
from .models import User
```

This method makes the package much more modular, which is a good thing. Now you can change the name of your package and use modules from other projects without changing the import statements.

In summary, what will help you advance in your development journey is to

- follow the style used in this book,
- follow the coding style shown in PEP 8,
- use docstrings defined in PEP 257 to document your app,
- import internal modules with relative imports.

Application Directory

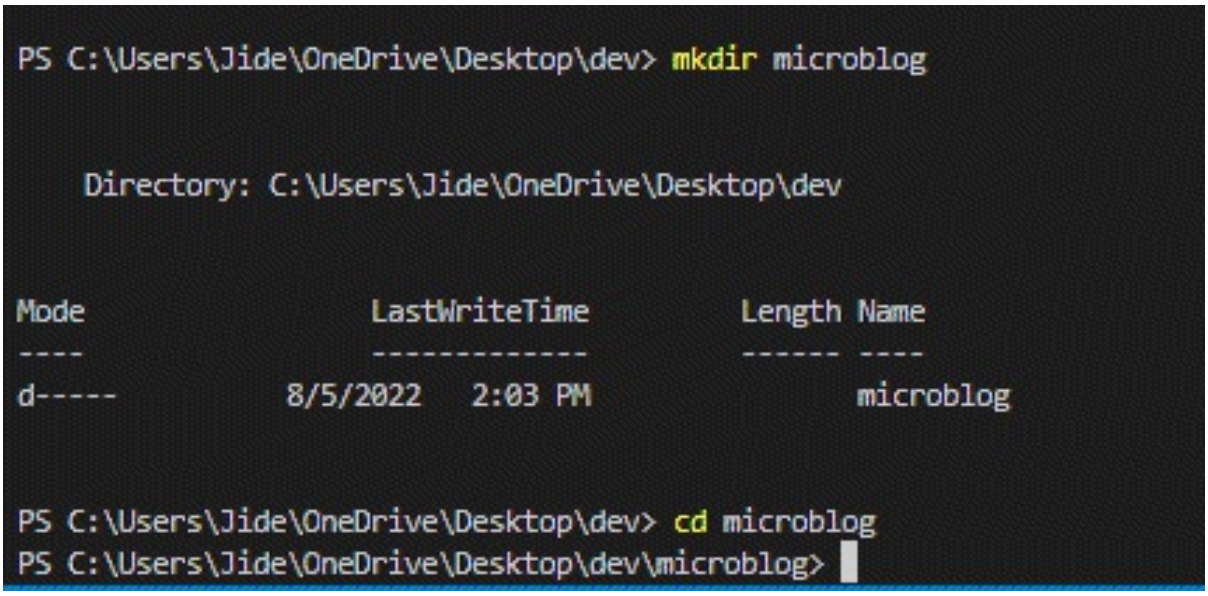
I assume you are new to Flask, but you can use Python. Otherwise, I highly recommend starting your journey by learning Python basics.

Anyway, open your Python text editor or IDE and let us start. The first task is to create a folder where your project will sit. I use Visual Studio Code.

Open the Terminal, and type in the following code:

mkdir microblog

That will create the folder. Now cd into your new folder with cd microblog.



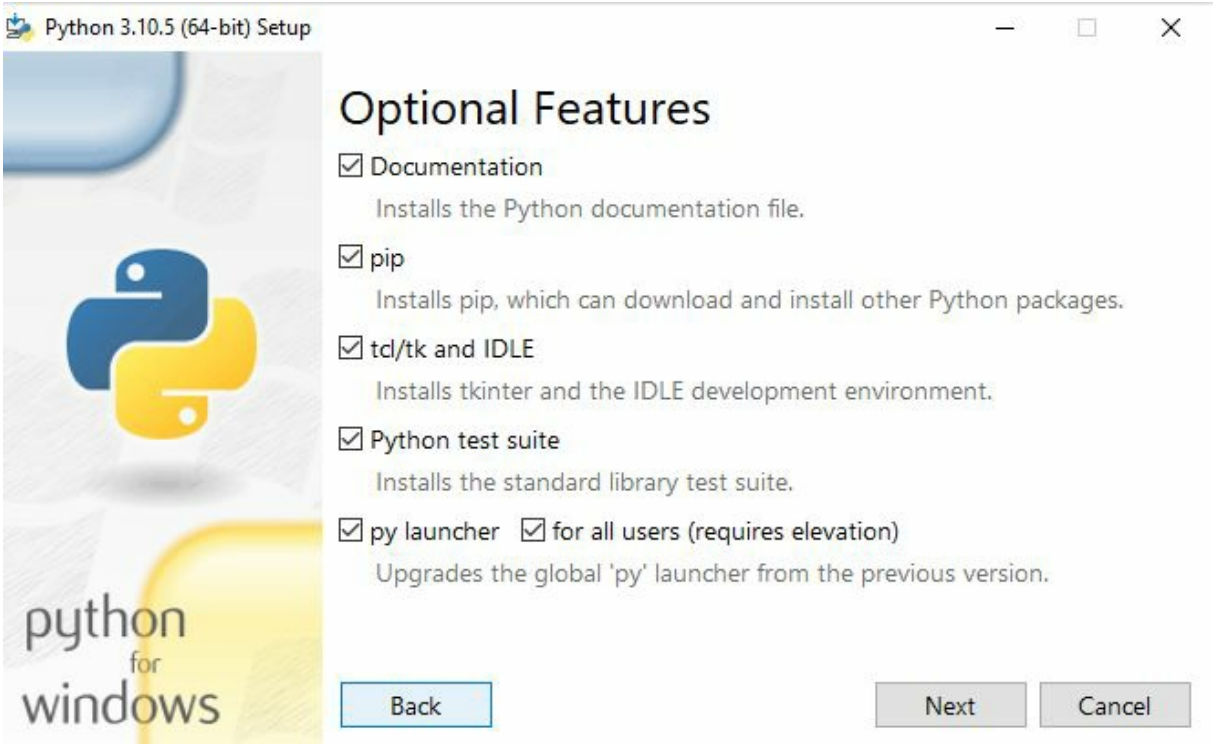
Installing Python

Install Python if you don't already have it on your computer. It is possible to download an installer for Python from the official website if your operating system does not include a package. Please keep in mind that if you're using WSL or Cygwin on Windows, you won't be able to use the Windows native Python; instead, you'll need to download a Unix-friendly version from Ubuntu or Cygwin, depending on your choice.

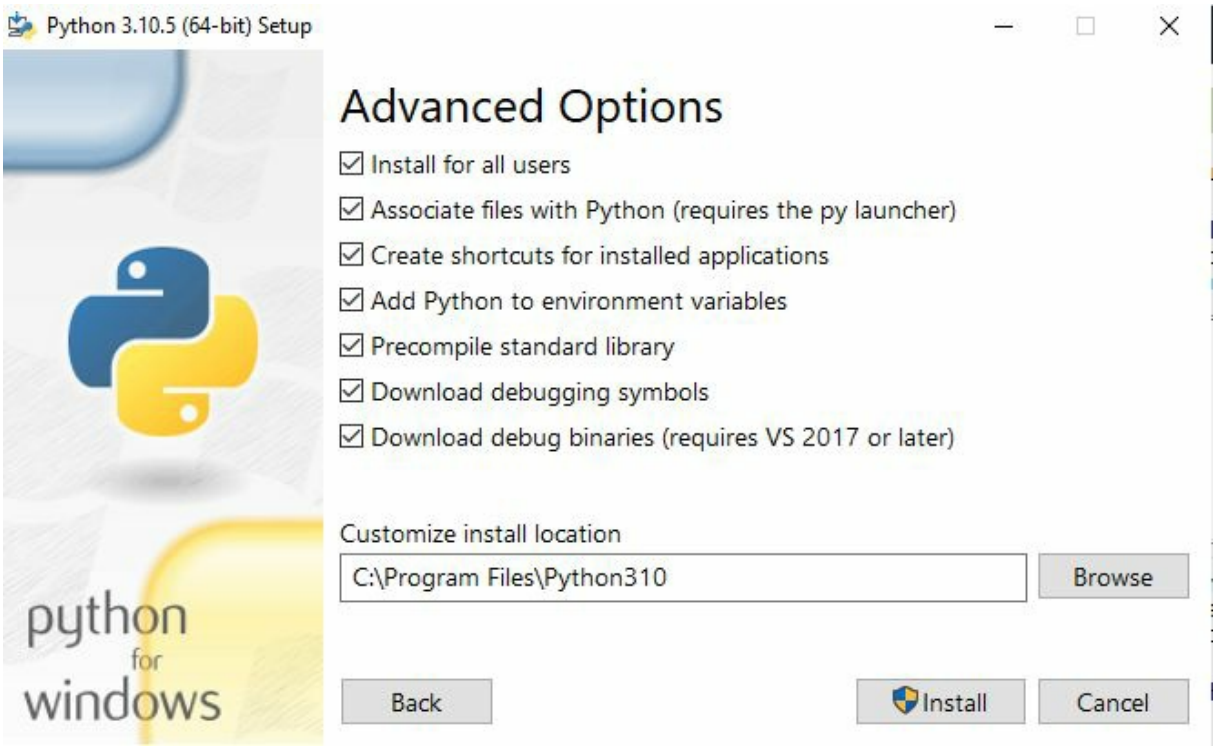
Installing Python

First, go to Python’s official website and download the software. It is pretty straightforward. Now, after downloading, run the program.

When installing, click on Customise, and you can check these boxes. Most significantly, pip and py launcher. You may leave out the “for all users” if the computer is not yours.



Click on next. Check Add Python to environment variables," and install.



Install Pip

Pip is a Python package manager that will be used to add modules and libraries to our environments.

To see if Pip is installed, open a command prompt by pressing Win+R, typing "cmd," and pressing Enter. Then type "pip help."

You should see a list of commands, including one called "install," which we'll use in the next step:

It's time for Flask to be installed, but first, I'd like to talk about the best practices for installing Python packages.

Flask, for example, is a Python package available in a public repository and can be downloaded and installed by anyone. PyPI (Python Package Index) is the name of the official Python package repository (some people also refer to this repository as the "cheese shop"). A tool called pip makes it easy to install a package from the PyPI repository.

You can use pip to install a program on your computer as follows:

```
pip install <package-name>
```

Unfortunately, this method of installing packages does not work in many cases. To run the command above, you'll need to be an administrator on your computer, which means you'll need to be logged in as an administrator. Even if you don't have to deal with that, think about what happens when you do the installation described above. The package will be downloaded from PyPI and added to your Python installation using the pip command-line tool. After that, all your Python scripts can access the package you just installed. Suppose you've finished a web application using Flask version 1.1, which was the most current version of Flask when you started but has since been replaced by Flask version 2.0. You'd like to use the 2.0 version for a second application, but if you replace the 1.1 version you already have installed, you could end up causing problems for your older application. Do you see what I'm getting at? It would be ideal if Flask 1.1 and Flask 2.0 could coexist on

the same computer. Flask 1.1 will work as a backend for your older application, and the newer app will have the current version at the time.

Python uses virtual environments to deal with the issue of maintaining various versions of packages for various applications. That is what we will discuss in the next chapter.

CHAPTER 2 – VIRTUAL ENVIRONMENTS

You can install more software now that the application directory has been configured. For your app to function correctly, you will require various software. You must first have at least the Flask package, and the Python installed. If not, you might be reading the incorrect book. The environment for your program is everything that must be accessible for it to run. There are numerous things we can do to set up and maintain the environment for our app. This chapter is focused on that.

When installing packages privately without impacting the Python interpreter that is already installed on your system, you can do so in a virtual environment, which is a duplicate of the Python interpreter.

Virtual environments are highly useful because they prevent the system's Python interpreter from becoming clogged with mismatched packages and versions. You may make sure that applications only have access to the packages they require by setting up a virtual environment for each project. This allows you to create more virtual environments and keeps the global interpreter clean. Additionally, since virtual environments may be created and operated without administrator privileges, they are superior to the system-wide Python interpreter.

Use virtualenv to manage your environment

virtualenv is a program that isolates whatever application you are developing in a virtual environment. A virtual environment implies that all the software your program depends on is stored in a single folder. This means that the software is only usable by your application.

The Python interpreter is a type of virtual environment (a copy). Installing packages in a virtual environment has no impact on the Python interpreter used by the entire system. Only the copy is. As a result, creating a separate virtual machine just for each application is the best way to ensure you can install any version of your packages. Additionally, virtual environments do not require an administrator account because they are owned by the user who creates them.

Instead of using system-wide or user-wide package directories, we can download them to a separate, dedicated folder for our application. For each project, we can choose the version of Python we want to use and which dependencies we want to have available.

It's possible to switch between various versions of the same package with Virtualenv. This kind of scalability can be crucial when working on an older system with multiple projects requiring different software versions.

As a result of using virtualenv, you'll be limited to a small number of Python packages on your machine. Virtualenv will be one of these. Pip may be used to install virtualenv.

Virtual environments can be created as soon as virtualenv is installed on your computer. Run the virtualenv command in your project's directory to get started. The virtual environment's destination directory is the only parameter required.

```
pip install virtualenv
```

Now that we've installed virtualenv, we can make different environments to test our code. But it can be hard to keep track of all of these places. So we'll pip install another package that will help us.

Install virtualenvwrapper

virtualenvwrapper is a package that lets you control the virtual environments that virtualenv makes. Use the following line to install the virtual wrapper for our Flask projects.

```
pip install virtualenvwrapper-win
```

Make a Virtual Environment

The structure of the command that makes a virtual environment looks like this:

```
python -m venv virtual-environment-name
```

The -m venv suggestion launches the venv package from the source file as a standalone script with the name given as an argument.

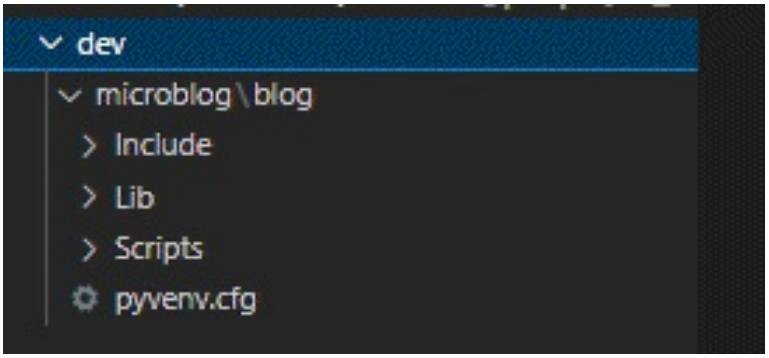
Inside the microblog directory, you will now make a virtual environment. Most people call virtual environments "venv," but you can call them something else if you'd like. Make sure that microblog is your current directory, and then run this command:

```
python3 -m venv venv
```

(you can use any name different from venv)

After the command is done, you'll have a subdirectory called venv (or, like mine, blog inside the microblog folder. This subdirectory will have a brand-new virtual environment with a Python interpreter for this project only.

```
PS C:\Users\Jide\OneDrive\Desktop\dev> cd microblog
PS C:\Users\Jide\OneDrive\Desktop\dev\microblog> python -m venv blog
PS C:\Users\Jide\OneDrive\Desktop\dev\microblog>
```



Now, activate the virtual environment by using the following line:

```
blog\Scripts\activate
```

```
PS C:\Users\Jide\OneDrive\Desktop\dev\microblog> blog\Scripts\activate
(blog) PS C:\Users\Jide\OneDrive\Desktop\dev\microblog>
```

Once done with activating the virtual environment, You'll see "(blog)" next to the command prompt. The line has made a folder with python.exe, pip, and setuptools already installed and ready to go. It will also turn on the Virtual Environment, as shown by the (blog).

The PATH environment variable is updated to include the newly enabled virtual environment when you activate it. A path to an executable file can be specified in this variable. When you run the activation command, the name of the virtual environment will be appended to the command prompt as a visual cue that the environment is now active.

After a virtual environment has been activated, the virtual environment will be used whenever python is typed at the command prompt. Multiple command prompt windows necessitate individual activation of the virtual environment.

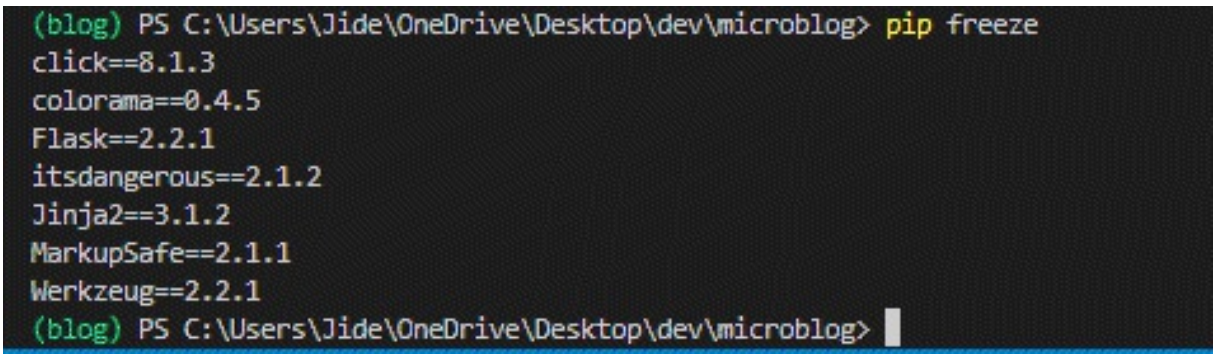
Installing Python Packages

All virtual environments have the pip package manager, which is used to install Python packages. Similar to the python command, entering pip at a command prompt will launch the version of this program that is a part of the active virtual environment.

Make sure the virtual environment is active before running the following command in order to install Flask into it:

```
pip install flask
```

When you run this prompt, pip will install Flask, and every software Flask needs to work. You can check the packages installed in your virtual environment using pip freeze. Type the following command:

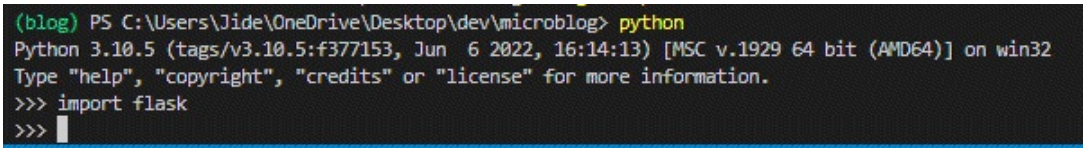


Type deactivate at the command prompt to return your Terminal's PATH environment variable and the command prompt to their default states once you've finished working in the virtual environment.

Each installed package's version number is shown in the output of pip freeze.

Most likely, the version numbers you get will be different from those shown here.

You can also make sure Flask was installed correctly by starting Python and trying to import it:



If there are no errors, you can give yourself a pat on the back. You are ready to move on to the next level.

CHAPTER 3 – PROJECT ORGANISATION

Flask doesn't help you to organize your app files. All of your application's code should be contained in a single folder, or it could be distributed among numerous packages. You may streamline developing and deploying software by following a few organizational patterns.

We'll use different words in this chapter, so let's look at some of them.

Repository - This is the folder for your program on the server. Version control systems are typically used to refer to this word.

Package: This is a Python library that holds the code for your application. Creating a package for your project will be covered in greater detail later in this chapter, so just know that it is a subdirectory of your repository.

Module: A module is one Python file that other Python files can import. A package is nothing more than a collection of related modules.

Patterns of organization

Most Flask examples will have all the code in a single file, usually called app.py. This works well for small projects with a limited number of routes and fewer than a few hundred lines of application code, such as those used for tutorials.

```
app.py
config.py
requirements.txt
static/
templates/
```

When you're working on a project that's a little more complicated, a single module can get cluttered. Classes for models and forms must be defined, and they will be mixed in with the script for your routes and configuration. All of this can slow down progress. To solve this problem, we can separate the different parts of our app into a set of modules that work together. This is called a package.

```
config.py
requirements.txt
run.py
instance/
    config.py
yourapp/
    __init__.py
    views.py
    models.py
    forms.py
    static/
    templates/
```

This listing's structure lets you group the different parts of your application in a way that makes sense. Model class definitions are grouped together in models. The definitions of routes and forms are in views.py. and forms.py, respectively (we have a whole chapter for forms later).

This table gives a breakdown of the parts included in the majority of Flask projects. You will likely have many additional files in your repository, typical of Flask apps.

run.py	This file is executed to launch a development server. It obtains a copy of the application from the package and runs it. This will not be used in production but is heavily utilized throughout the development phase.
requirements.txt	This file lists all Python packages on which your application depends. You can have different files for development and production dependencies.
config.py	This file contains most of the variables your project needs for

	configuration.
/instance/config.py	This file includes configuration variables that should not be tracked by version control. This includes API keys and database URIs with embedded passwords. Additionally, this contains variables unique to this instance of your program. For example, you may have DEBUG = False in config.py but DEBUG = True in instance/config.py on your local development system. Because this file will be read after config.py, DEBUG = True.
/yourapp/	This is the package that contains your application.
/yourapp/__init__.py	This file initializes your application and assembles its diverse components.
/yourapp/views.py	This is where route definitions are made. It may be separated into its package (yourapp/views/), with related views organized into modules.
/yourapp/models.py	This is where you define the application's models. Similar to views.py, this may be separated into many modules.py.
/yourapp/static/	This directory contains the public CSS, JavaScript, images and other files you want to make public via your app. It is accessible from yourapp.com/static/ by default.
	This is where you'll put the Jinja2

/yourapp/templates/	templates for your app.

Initialization

All Flask applications need to create an application instance. Using a protocol called WSGI, pronounced "wiz-ghee", the web server sends all requests from clients to this object so that it can handle them. The application instance is an object of the class Flask. Objects of this class are usually made in this way:

```
from flask import Flask

app = Flask(__name__)
```

The only thing that has to be given to the Flask class constructor is the name of the application's main module or package. Most of the time, the __name__ variable in Python is the correct answer for this argument.

New Flask developers often get confused by the __name__ argument passed to the application constructor. Flask uses this argument to figure out where the application is, which lets it find other files that make up the application, like images and templates.

Blueprints

At some time, you may discover that there are numerous interconnected routes. If you're like me, your initial inclination will be to divide opinions. Py into a package and organize the views as modules. It may be time at this stage to incorporate your application into blueprints.

Blueprints are essentially self-contained definitions of your application's components. They function as apps within your app. The admin panel, front-end, and user dashboard may each have their own blueprint. This allows you to group views, static files, and templates by component while allowing these components to share models, forms, and other features of your application. Soon, we will discuss how to organize your application using Blueprints.

CHAPTER 4 – ROUTING & CONFIGURATION

Web applications that run on web browsers send requests to the web server to the Flask application instance. For each URL request, the Flask application instance needs to know the code to execute, so it keeps a map of URLs to Python functions. A route is a link between a URL and the function that calls it.

Modern web frameworks employ routing to aid users in remembering application URLs. It is useful to be able to browse directly to the required page without first visiting the homepage.

The Python programming language has them built in. Decorators are often used to sign up functions as handler functions that will be called when certain events happen.

The app.route decorator made available by the application instance is the easiest way to define a route in a Flask application. This decorator is used to declare a route in the following way:

```
@app.route("/")
def index():
    return "<h1>Hello World!</h1>"
```

In the previous example, the handler for the application's root URL is set to be the function index(). Flask prefers to register view functions with the app. route decorator. However, the app.add_url_rule() method is a more traditional way to set up the application routes. It takes three arguments: the URL, the endpoint name, and the view function. Using the app.add_url_rule(), the following example registers an index() function that is the same as the one shown above:

```
def index():
    return "<h1>Hello World!</h1>"
```

```
app.add_url_rule("/", "index", index)
```

Similar to index(), view functions manage application URLs. Going to

http://www.example.com/ in your browser would cause the server to run index if the app runs on a server with the domain name www.example.com (). This view function's return value is the response the client receives. This answer is the page displayed to the user in the browser window if the client is a web browser. As we'll see later, a response from a view function could be as straightforward as an HTML string, or it might be more intricate.

You'll notice that many of the URLs for services you use on a daily basis have sections that can be modified if you pay attention to how they are constructed. For instance, https://www.facebook.com/your-name> is the URL for your Facebook profile page. Your username is a part of this, making it particular to you. Flask can handle these URLs using a special app.route decorator. The steps to configure a route with an active portion are as follows:

```
@app.route("/user/<name>")
def user(name):
   return "<h1>Hello, {}!</h1>".format(name)
```

The portion of a URL for a route that is enclosed in angle brackets changes. Any URLs that match the static portions will be mapped to this route, and the active part will be supplied as an argument when the view function is called. In the preceding illustration, a personalized greeting was provided in response using the name argument.

The active components of routes can be of other kinds in addition to strings, which are their default. If the id dynamic segment has an integer, for example, the route /user/int:id> would only match URLs like /user/123. Routes of the types string, int, float, and path are all supported by Flask. The path type is a string type that can contain forward slashes, making it distinct from other string types.

URL routing is used to link a specific function (with web page content) to its corresponding web page URL.

When an endpoint is reached, the web page will display the message, which

is the output of the function associated with the URL endpoint via the route.

View decorators

Decorators in Python are functions used to tweak other functions. When a function that has been decorated is called, the decorator is instead invoked. The decorator may then perform an action, modify the parameters, pause execution, or call the original function. You can use decorators to encapsulate views with code to be executed before their execution.

Configuration

When learning Flask, configuration appears straightforward. Simply define some variables in config.py, and everything will function properly. This simplicity begins to diminish while managing settings for a production application. You might need to secure private API keys or utilize different setups for various environments. For example, you need a different environment for production.

This chapter will cover advanced Flask capabilities that make configuration management easier.

A basic application might not require these complex features. It may be just enough to place config.py at the repository's root and load it in app.py or yourapp/__init__.py.

Each line of the config.py file should contain a variable assignment. The variables in config.py are used to configure Flask and its extensions, which are accessible via the app.config dictionary — for example, app.config["DEBUG"].

```
DEBUG = True # Turns on debugging features in Flask
BCRYPT_LOG_ROUNDS = 12 # Configuration for the Flask-Bcrypt extension
MAIL_FROM_EMAIL = "abby@example.com" # For use in application emails
```

Flask, extensions, and you may utilize configuration variables. In this example, we may use an app.config["MAIL_FROM_EMAIL"] to specify the default "from" address for transactional emails, such as password resets. Putting this information into a configuration variable makes future

modifications simple.

Instance folder

Occasionally, you may be required to define configuration variables containing sensitive information. These variables will need to be separated from those in config.py and kept outside of the repository. You may hide secrets such as database passwords and API credentials or setting machine-specific variables. To facilitate this, Flask provides us with the instance folder functionality. The instance folder is a subdirectory of the repository's root directory and contains an application-specific configuration file. We do not wish to add it under version control.

```
config.py
requirements.txt
run.py
instance/
  config.py
yourapp/
  __init__.py
  models.py
  views.py
  templates/
  static/
```

How to use instance folders

If you want to load configuration variables from an instance folder, you can use the function app.config.from_pyfile(). First, set the instance_relative_config = True when creating your app with the Flask() function. The app.config.from_pyfile() will load the file from the instance/ folder.

```
# app.py or app/__init__.py

app = Flask(__name__, instance_relative_config=True)
app.config.from_object("config")
app.config.from_pyfile("config.py")
```

Now, instance/config.py can contain variable definitions identical to those in config.py. Additionally, you should add the instance folder to the ignore list of your version control system. To accomplish this with Git, add instance/ to a new line in.gitignore.

Secret keys

The instance folder's private nature makes it an ideal location for establishing keys that should not be exposed to version control. These may include your application's private or third-party API keys. This is particularly crucial if your program is open source or maybe in the future. We generally prefer that other users and contributors use their own keys.

```
# instance/config.py

SECRET_KEY = "Sm9obiBTY2hyb20ga2lja3MgYXNz"
STRIPE_API_KEY = "SmFjb2IgS2FwbGFuLU1vc3MgaXMgYSBoZXJv"
SQLALCHEMY_DATABASE_URI = (
    "postgresql://user:TWljaGHFgiBCYXJ0b3N6a2lld2ljeiEh@localhost/databasename"
)
```

Configuring based on environment variables

Don’t add the instance folder under version control. This means you cannot trace configuration changes to the config setup in your instance. If you have one or two variables, this may be overlooked. Still, you don't want to risk losing precisely calibrated setups for different environments (production, staging, development, etc.).

Upon load, Flask gives us the option to choose a configuration file based on the value of an environment variable. As a result, we can store different configuration files in our repository and load the appropriate ones as needed. Once a large number of configuration files have been produced, we can move them into the appropriate configuration directory.

We'll take advantage of the app.config.from envvar() function to figure out which configuration file to import.

```
# yourapp/__init__.py
```

```
app = Flask(__name__, instance_relative_config=True)

# Load the default configuration
app.config.from_object("config.default")

# Load the configuration from the instance folder
app.config.from_pyfile("config.py")

# Load the file specified by the APP_CONFIG_FILE environment variable
# Variables defined here will override those in the default configuration
app.config.from_envvar("APP_CONFIG_FILE")
```

Variable Rule

App routing is the process of mapping a certain URL to the function designed to complete a given action. The most recent Web frameworks employ routing to aid users in remembering application URLs.

To hard-code each URL while creating an application is pretty inconvenient. Creating dynamic URLs is a better way to handle this problem.

Using variable elements in the rule parameter allows you to create URLs on the fly. Variable-name> is the name of this variable component. It is passed as a parameter to the function that corresponds to the rule.

Let's examine the idea of variable rules in great detail.

Dynamic URLs can be created with the use of variable rules. They are essentially the variable sections added to a URL using the variable name> or converter: variable name> tags. It is passed as a parameter to the function that corresponds to the rule.

Syntax:

@app.route('hello/<variable_name>')

OR

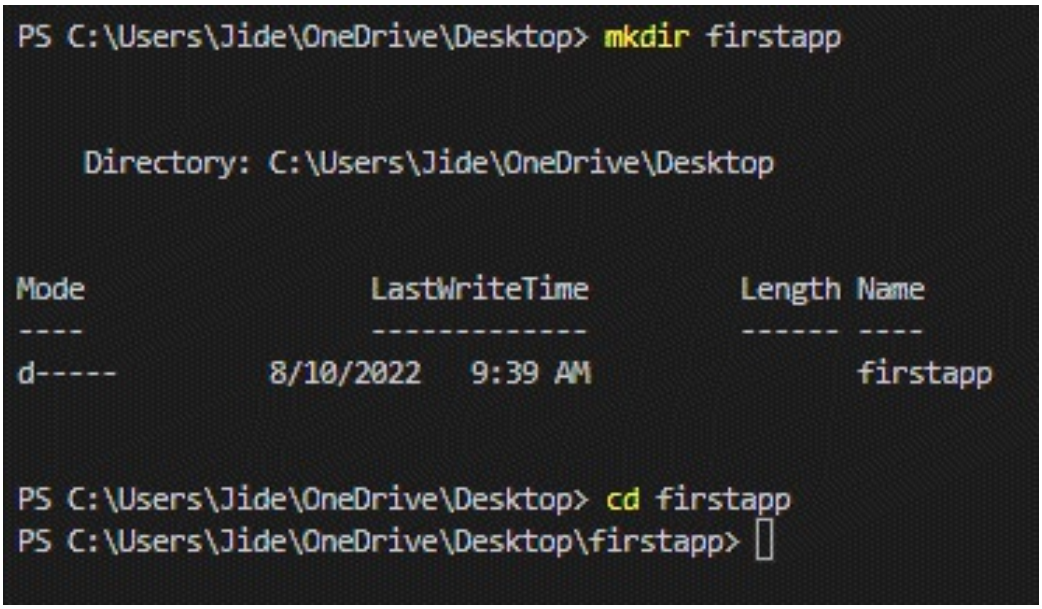
@app.route('hello/<converter: variable_name>')

Chapter 5 – Build A Simple App

You've understood the various parts and configurations of a Flask web application in the previous sections. Now it's time to write your first one. In the example below, the application script defines an application instance, a single route, and a single view function, as we've already said.

I'll be using Visual Studio Code, which has installed the Python extension.

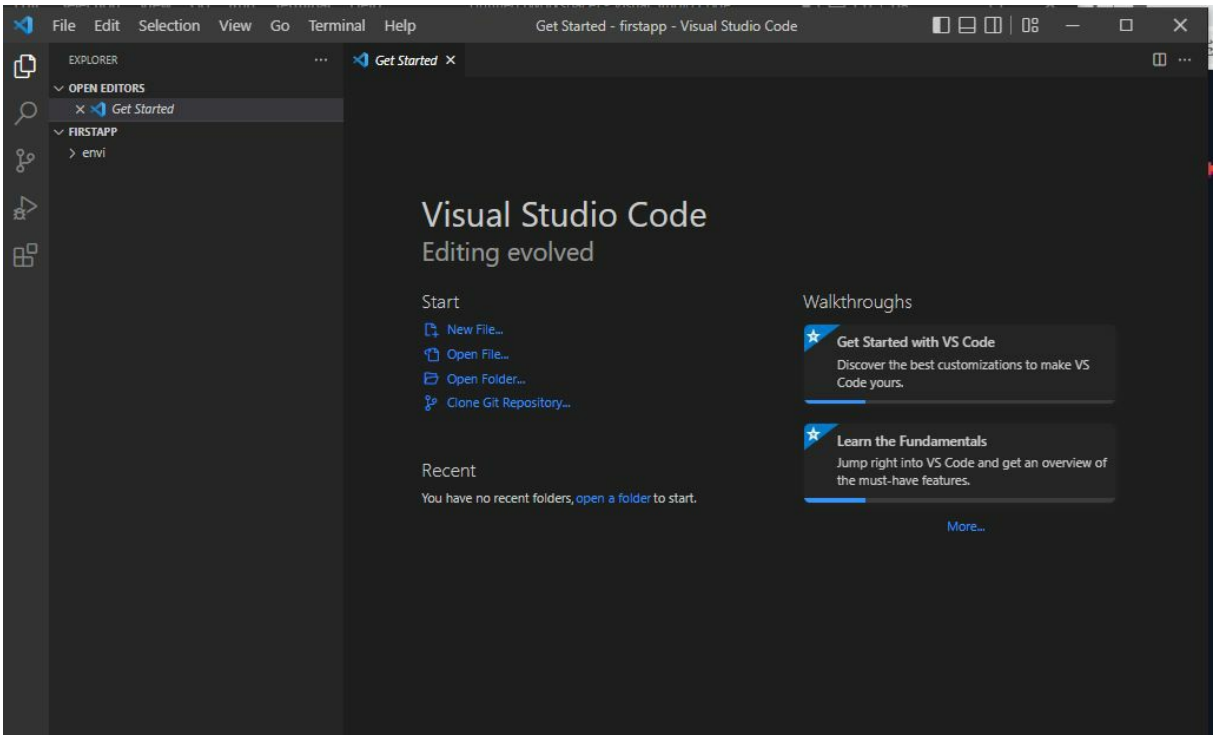
The first step is to create a project folder. Mine is firstapp. Name yours whatever.



After you have cd that folder, create a virtual environment. I will name mine envi.

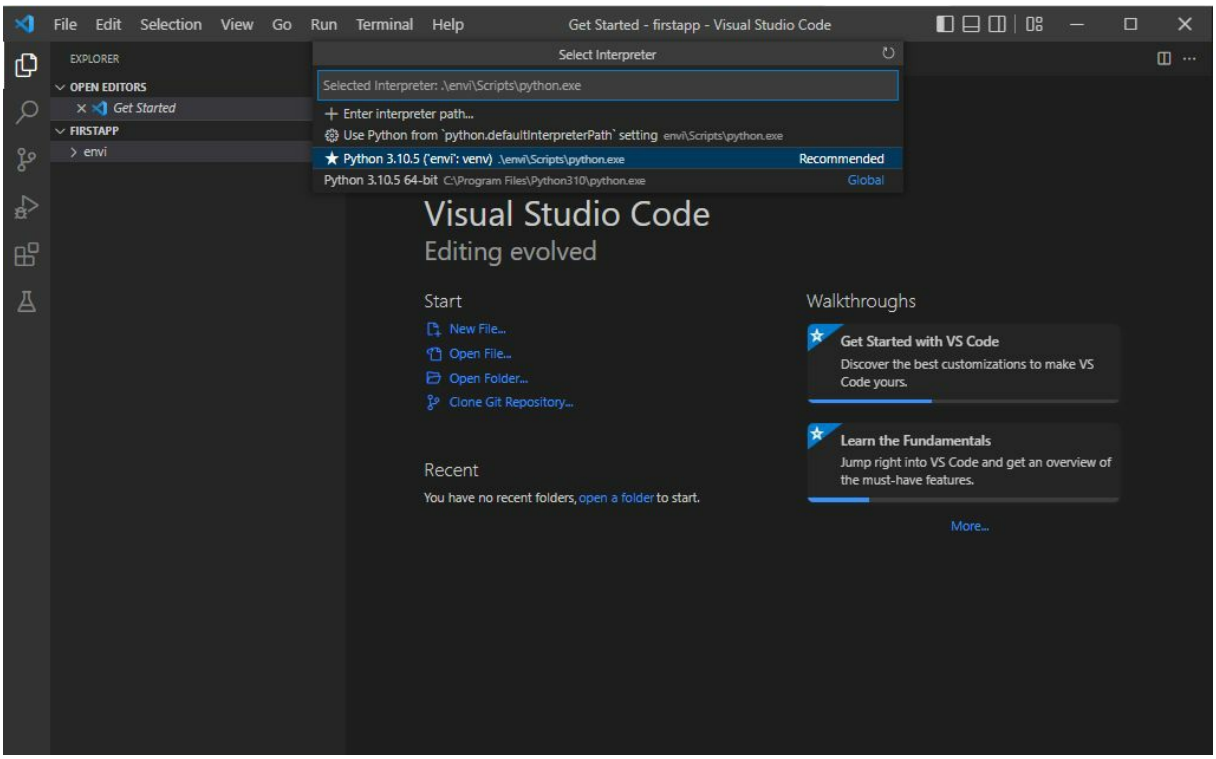
python -m venv envi

Now, type code in the Terminal, and run. Visual Studio Code will open in a new window. Now, open the app folder in the new window like this:

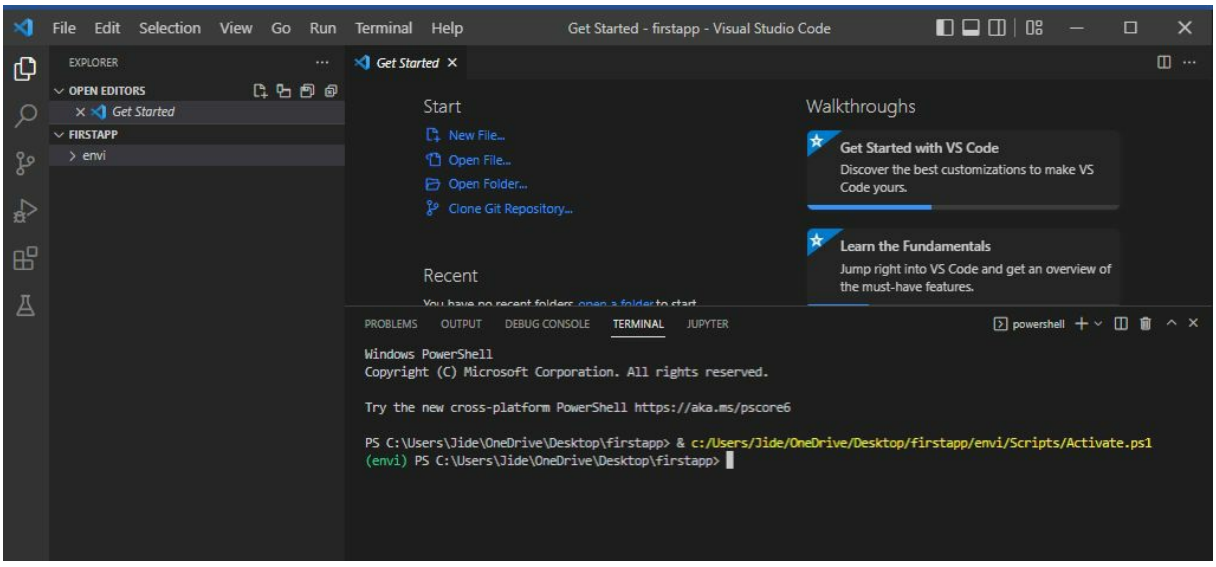


Next, open the Command Palette. Go to View and click on Command Palette (or press Ctrl+Shift+P). Select Python: Select the Interpreter command.

This means you want to see interpreters that are available to VS can locate. Here's mine.



Go to the Command Pallete again and search Terminal. Click on Terminal: Create New Terminal (SHIFT + CTRL + `)



Can you see the name of your virtual environment at the bottom left corner? Mine has the "envi" as the name of my virtual environment.

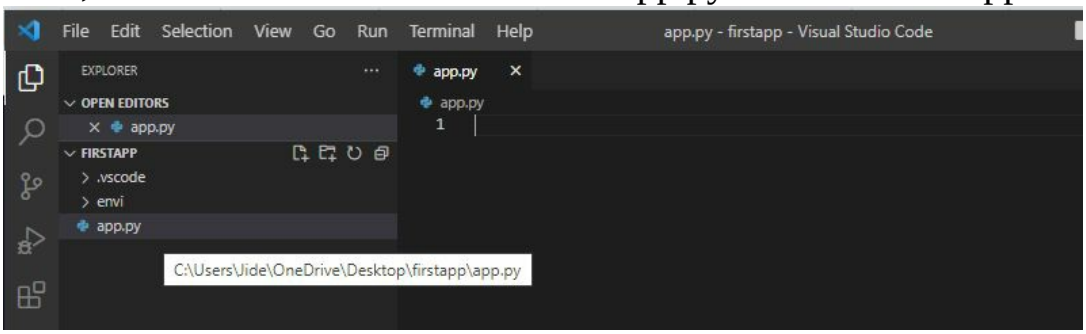
Now that the Virtual environment is active, install Flask in the virtual environment by running pip install flask in the Terminal.

```
(envi) PS C:\Users\Jide\OneDrive\Desktop\firstapp> pip install flask
Collecting flask
  Downloading Flask-2.2.2-py3-none-any.whl (101 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 101.5/101.5 KB 324.7 kB/s eta 0:00:00
Collecting Jinja2>=3.0
  Using cached Jinja2-3.1.2-py3-none-any.whl (133 kB)
Collecting Werkzeug>=2.2.2
  Downloading Werkzeug-2.2.2-py3-none-any.whl (232 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 232.7/232.7 KB 547.9 kB/s eta 0:00:00
Collecting click>=8.0
  Using cached click-8.1.3-py3-none-any.whl (96 kB)
Collecting itsdangerous>=2.0
  Using cached itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting colorama
  Using cached colorama-0.4.5-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Using cached MarkupSafe-2.1.1-cp310-cp310-win_amd64.whl (17 kB)
Installing collected packages: MarkupSafe, itsdangerous, colorama, Werkzeug, Jinja2, click, flask
Successfully installed Jinja2-3.1.2 MarkupSafe-2.1.1 Werkzeug-2.2.2 click-8.1.3 colorama-0.4.5 flask-2.2.2 itsdanger
ous-2.1.2
```

When you start a separate command prompt, run envi\Scripts\activate to activate the environment. It should begin with (envi), indicating that it is engaged.

The actual app

Now, we will create a new file named app.py inside the firstapp folder.



In app.py, we will add a code to import Flask and construct a Flask object instance. This object will serve as the WSGI application.

```
from flask import Flask

app = Flask(__name__)
```

We will now call the new application object's run () function to run the main app.

```
if __name__ == "__main__":
   app.run()
```

We develop a view function for our app to display something in the browser window. We will construct a method named hello() that returns "Hello, World!

```
def hello():
   return "Hello World!";
```

Now, let us assign a URL route so that the new Flask app will know when to call the hello() view function. We associate the URL route with each view function. This is done with the route() decorator in front of each view function like this.

```
@app.route("/")
def hello():
   return "Hello World!"
```

The complete app.py script is like this:

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def hello():
   return "Hello World!"
```

```
if __name__ == "__main__":
   app.run(debug=True, host="0.0.0.0", port=3000)
```

Development Web Server

Using the flask run command, you can start a development web server for Flask applications. This command looks in the FLASK_APP environment

variable for the name of the Python script that includes the application instance.

To run the app.py application, first, make sure the virtual environment you set up earlier is running, and that Flask is installed in it.

```
python -m flask run
```

When the server fires up, it goes into a loop that receives requests and handles them. This loop will keep going until you press Ctrl+C to stop the program.

Open your web browser and type http://localhost:5000/ in the url bar while the server is running. The screenshot below shows what you'll see once you're connected to the app.

Now that is the base url we set a route to. Adding anything else to the URL will mean that your app won't know how to handle it and will send an error code 404 to the browser.

The app.run() method can also be used to programmatically start the Flask development web server. In older Flask versions that didn't have the flask command, the server had to be started by running the application's main script, which had to end with the following code:

```
if __name__ == "__main__":
    app.run()
```

This is no longer necessary because of the flask run command. However, the app.run() function can still be helpful in some situations, such as unit testing.

Chapter 6 - Dynamic Routes

Let's now consider an alternative routing method. The next illustration demonstrates how an alternative implementation of the program adds a second, dynamic route. Your name appears as a customized greeting when you visit the active URL in your browser.

In this chapter, I will describe variable rules, converters, and an example of dynamic routing.

We've discussed routes, views, and static routing when the route decorator's rule parameter was a string.

```
@app.route("/about")
def learn():
   return "Flask for web developers!"
```

If you want to use dynamic routing, the rule argument will not be a constant string like the /about. Instead, it is a variable rule you passed to the route().

We have learned about Variable Rules. However, read through this script to better get a glimpse of the variable rule:

```
"""An application to show Variable Rules in Routing"""
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def home():
   """View for the Home page of your website."""
   return "This is your homepage :)"
```

```
@app.route("/<your_name>")
def greetings(your_name):
   """View function to greet the user by name."""
   return "Welcome " + your_name + "!"
```

```
if __name__ == "__main__":
   app.run(debug=True, host="0.0.0.0", port=3000)
```

This is the result:

What happens if you add /name?

Like magic! How can you do this? Pretty easy. Follow me.

The first thing you will find different in the new code is the second view function, greetings (). There is the variable rule: /<your_name>.

That means the variable is your_name (whatever you type after the /). We then pass this variable as a parameter to the greetings() function. That is why it is called to return a greeting to whatever name is passed to it. Facebook is not that sleek now, is it?

Converter

The above example used the URL to extract the variable your_name. Flask now converted that variable into a string and passed it to the greetings() function. That is how converters work.

Here are the data types Flask converters can convert:

- Strings: this goes without saying
- int: they convert this only for when you pass in positive integers
- float: also only works for positive floats

- path: this means strings with slashes
- uuid: UUID strings means Universally Unique Identifier strings used for identifying information that needs to be unique within a system or network.

Let us learn about another feature for web apps.

CHAPTER 7 – STATIC TEMPLATES

This chapter will teach you how to create and implement static and HTML templates. You will also learn file structure strategies.

Clean and well-structured code is essential for developing apps that are simple to maintain. Flask view functions have two different jobs, and this can cause confusion.

As we can see, a view function's one obvious purpose is to respond to a request from a web browser. The status of the application, as determined by the view function, can also be altered by request.

Imagine a user signing up for the first time on your website. Before clicking the Submit button, he fills up an online form with his email address and password.

The view method, which manages registration requests, receives Flask's request on the server containing the user's data. The view function interacts with the database to add the new user and provide a response to display in the browser. These two responsibilities are formally referred to as business logic and presentation logic, respectively.

Complex code is produced when business and presentation logic are combined. Imagine having to combine data from a database with the required HTML string literals in order to create the HTML code for a large table. The application's maintainability is improved by placing presentation logic in templates.

That is why a template is necessary. A template is a file that contains placeholder variables for the dynamic parts of a response that are only known in relation to a request's context. The process known as rendering is what gives variables their real-world values in exchange for the final response string. Flask renders templates using the powerful Jinja2 template engine.

Rendering HTML Templates

Flask expects to find template files in the leading application directory's templates subfolder. These templates are actually HTML files.

Flask can render HTML using two methods:

- as string
- using render_template function

A String

You can use HTML as a string for the function.

Here is an example:

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def home():
    return "<h1>There's Something About Flask!</h1>"
```

```
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=3000)
```

When you run that and follow your local host link, this is what you get:

This came out well as a template because we use HTML tags h1. We can use any HTML codes in the scripts, and Flask will read it well.

render_template() function

Now, the string is suitable for simple one-page websites. For big applications, you must add your templates as separate files.

In this case, you create the HTML code and keep the file separate in the folder. You will then call the file in the views function by the file names.

Flask will use the render_template() function to render the HTML templates.

This function uses the following parameters:

- template_name_or_list: that is the template file name or names if they are more than one
- context: these are variables that are in the template script.

The render_template() returns the output using the view instead of a string.

Here is an example:

```
def view_name():
   return render_template(template_name)
```

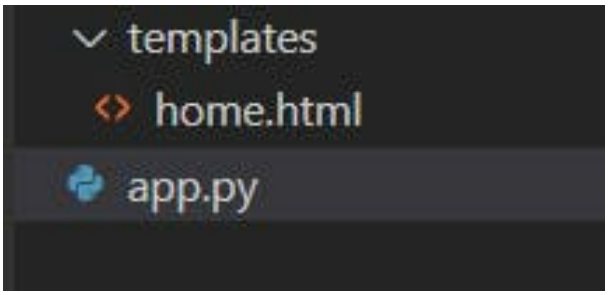
In this case, we would already have a template saved, perhaps as home.html or index.html, with the HTML code in it. When you run the app, Flask will run all the HTML codes included in the script, and the view will display them on the web browser.

File Structure Strategies

When you run the program, Flask will execute the script and run through your \templates folder to find the HTML files you reference in the script. You must place the folder correctly so that there will be no errors. These are the correct file structures that Flask can read:

Module File Structure

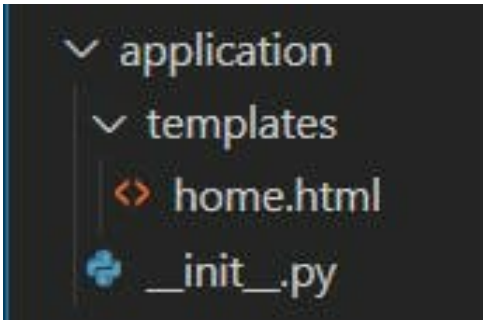
This is a very simple and straightforward structure where all the application logic is in a single .py file. The templates folder will be the same folder as the .py file where the developer keeps the HTML files.



Package File Structure

In many complex apps, the script is divided into separate .py files. In this case, you must present all the .py files in the same package. A package is a folder that contains an __init__.py file.

You must create the templates folder in the main application package to use this structure in your application.



So, let us do it together. First, know what we want to do: we want to render a home.html template with the render_template() function in our web app, app.py.

You will create a templates folder and then create a file inside the new folder and call it home.html.

Fill home.html with this code:

```
<!DOCTYPE html>
<html>
<h1>This is where we say FLASK! :)</h1>

</html>
```

Now, we can change our app.py code to call the HTML code.

```
from flask import Flask, render_template

app = Flask(__name__)
```

```
@app.route("/")
def home():
    return render_template("home.html")
```

```
if __name__ == "__main__":
```

```
app.run(debug=True, host="0.0.0.0", port=3000)
```

Look at the result!

Chapter 8 - The Jinja2 Template Engine

Jinja2 is a Python template engine. It can be used instead of Python's standard string interpolation, that is, adding data to strings. Flask can easily read Jinja2 templates, which is easier to write than the typical Python code. Jinja2 templates have a more natural language format.

Jinja2 templates are written in HTML or XML and then turned into "jinja" bytecode, which the Jinja environment can read and use. The python compiler module turns the templates into bytecode, which is then run by an interpreter that parses and runs jinja scripts built from HTML or XML templates.

Templates are files that contain both static and dynamic data placeholders. To create a finished document, a template is rendered with precise data. The Jinja template library is used by Flask to render templates. Templates will be used in your application to render HTML shown in the user's browser.

We have to place the template file in the templates folder. The templates are in the root folder of the project.

For example, we can have our home.html to be:

```
<!DOCTYPE html>
<html>
{% raw %}
<h1>Hello {{ name }} </h1>
{% endraw %}
<h1>This is where we say FLASK! :)</h1>

</html>
```

We will now write the view function as the following code:

```
@app.route("/user/<name>")
def index(name):
   return render_template("home.html", name=name)
```

The Jinja2 template engine is built into the application by the Flask function render_template(). The template's filename is the first argument to the render_template() function. The rest of the arguments are key-value pairs that

show the real values of the variables in the template.

Variables

A template file is just a normal text file. The part to be replaced is marked with double curly brackets ({{ }}), in which the variable name to be replaced is written. This variable supports basic data types, lists, dictionaries, objects, and tuples. The same as in template.html:

```
{% raw %}

<p> A value form a string: {{ name }}. </p>
<p> A value form a int: {{ myindex }}. </p>
<p> A value form a list: {{ myindex }}.
<p> A value form a list: {{ mylist[3]] }}. </p>
<p> A value form a list: {{ mylist[3] }}.</p>
<p> A value form a list, with a variable index: {{ mylist[myindex] }}. </p>
<p> A value form a dictionary: {{ mydict['key'] }}. </p>
<p> A value form a dictionary: {{ mydict['key'] }}.
<p> A value form a tuple: {{ mytuple }}. </p>
<p> A value form a tuple: {{ mytuple }}.</p>
<p> A value form a tuple by index: {{ mytuple[myindex] }}. </p>

{% endraw %}
```

Filters

As you write your apps, you may want to change some parts of your values in the template when they come on. For example, you may set the code to capitalize the first letter in a string, remove spaces, etc. In Flask, one way to do this is by using a filter.

Filters in the Jinjia2 template engine work like pipes in Linux commands. For example, they can capitalize the first letter of a string variable.

```
{% raw %}

<h1>{{ name | capitalize}}</h1>

{% endraw %}
```

Both filters and the Linux pipeline command can be spliced. For example, you can splice a line to do two things at the same time. Let us write a line to

capitalize values and take out whitespace before and after.

```
{% raw %}

<h1>{{ name | upper | trim }}</h1>

{% endraw %}
```

As you see in the code, we connected the filter and the variable with the pipe symbol |. That is the same as processing the variable value.

Here are some standard filters web developers use:

filter	description
safe	rendering is not escaped
capitalize	initial capitalization
lower	all letters lowercase
upper	all letters uppercase
title	Capitalize the first letter of each word in the value
trim	removes the first blank character
stripttags	removes all HTML tags from the value when rendering

Control structure

Jinja2 has a number of control structures that can be used to change how the template is run. This section goes over some of the most useful ones and shows you how to use them.

Many times, a smarter template rendering is needed, which means being able to program the rendering, such as having a style for boys and the same style for girls. Control structure instructions need to be specified with command markers, and some simple control structures are explained below.

Conditions

This kind of structure is when you use a conditional statement, i.e., an if-else structure in the template.

Here's an example of how you can add a conditional statement to a template:

```
{% raw %}

{% if gender=='male' %}
Hello, Mr {{ name }}
{% else %}
Hello, Ms {{ name }}
{% endif %}

{% endraw %}
```

The view function will be:

```
@app.route("/hello2/<name>/<gender>")
def hello2(name, gender):
    return render_template("hello2.html", name=name, gender=gender)
```

This is no different from the typical python code structure.

loop

If your web page has lists, for example, a control structure you want to use is loops. for loops are better suited. For example, let us display a list with ul.

```
{% raw %}

<ul>
    {% for name in names %}
    <li>{{ name }} </li>
    {% endfor %}
</ul>

{% endraw %}
```

Chapter 9 - Bootstrap Integration with Flask

Bootstrap is the most common CSS framework. It has more than 150k stars on Github and a very large ecosystem that supports it. To make this chapter more useful, we'll look at an open-source Flask project with a beautiful UI styled with Bootstrap. This project comes in two flavors: a low model that uses components downloaded from the official Bootstrap Samples page and a production-ready model with more pages (home, about, contact) and a complete set of features.

What is Bootstrap?

Twitter's Bootstrap is a free web browser framework that makes it simple to construct aesthetically pleasing, clean web pages that function on desktop and mobile platforms with all current web browsers.

Bootstrap is a client-side framework that doesn't directly interact with the server. The user interface elements must be created using HTML, CSS, and JavaScript code once the server sends HTML answers that connect to the appropriate Bootstrap Cascading Style Sheets (CSS) and JavaScript files. It is easiest to do all of this using templates.

Getting Started

The first step in integrating Bootstrap with your program is to modify the HTML templates as needed. The following code will allow you to create an HTML file and view it in your browser.:

```
<!doctype html>
<html lang="en">

<head>
    <title>My First Bootstrap Page</title>

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>

<body>
    <h1 class="text-primary">
        Let's learn Bootstrap
```

```
    </h1>

    <!-- Bootstrap Javascripts -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js"></script>

</body>

</html>
```

However, you can do this even better and faster with a Flask *extension* called Flask-Bootstrap, and you can install it with pip:

```
pip install flask-bootstrap
```

You can initialize Bootstrap into your script very quickly.

Code Flask App with Bootstrap

Now, go back to your app folder. You can delete and start over or open a new base folder. Create a new app.py file and fill it with the following code:

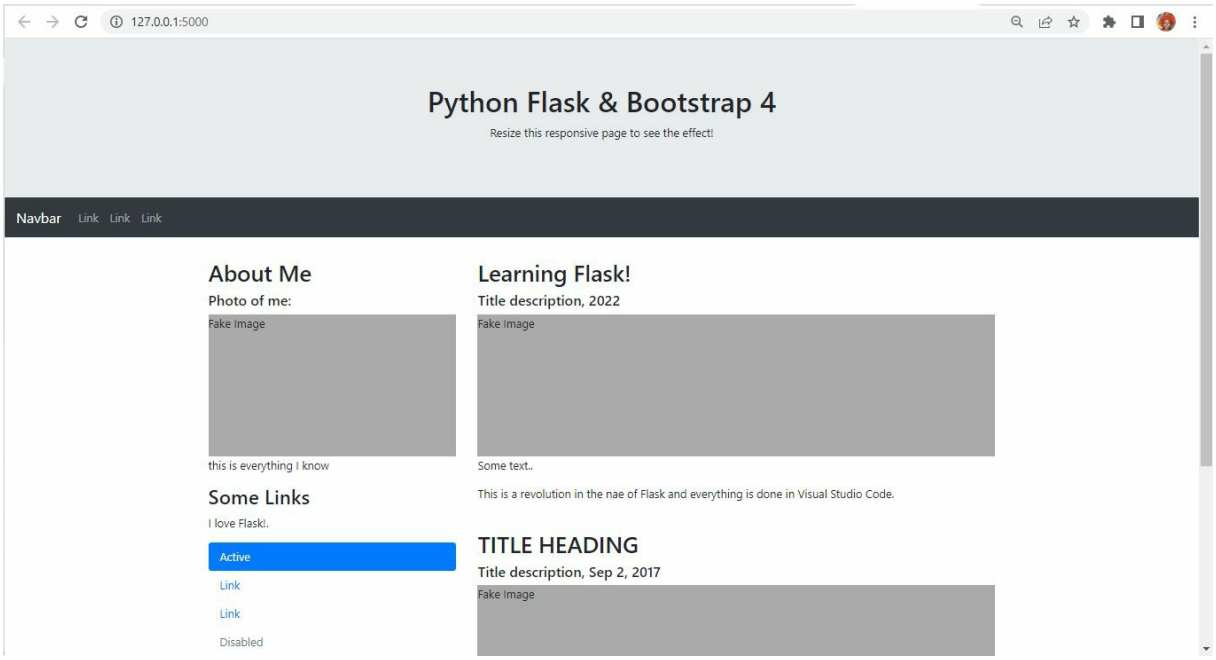
```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def main():
    return render_template('index.html')

if __name__ == "__main__":
    app.run()
```

Once that is saved, you can create an index.html template in the templates folder. Fill it with this demo script I adapted from W3Schools:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Python Flask & Bootstrap 4</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
  <style>
    .fakeimg {
      height: 200px;
      background: #aaa;
    }
  </style>
</head>

<body>

  <div class="jumbotron text-center" style="margin-bottom:0">
    <h1>Python Flask & Bootstrap 4</h1>
    <p>Resize this responsive page to see the effect!</p>
  </div>

  <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#collapsibleNavbar">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="collapsibleNavbar">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link" href="#">Link</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Link</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Link</a>
        </li>
      </ul>
    </div>
  </nav>

  <div class="container" style="margin-top:30px">
    <div class="row">
```

```
        <div class="col-sm-4">
          <h2>About Me</h2>
          <h5>Photo of me:</h5>
          <div class="fakeimg">Fake Image</div>
          <p>this is everything I know</p>
          <h3>Some Links</h3>
          <p>I love Flask!.</p>
          <ul class="nav nav-pills flex-column">
            <li class="nav-item">
              <a class="nav-link active" href="#">Active</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Link</a>
            </li>
            <li class="nav-item">
              <a class="nav-link disabled" href="#">Disabled</a>
            </li>
          </ul>
          <hr class="d-sm-none">
        </div>
        <div class="col-sm-8">
          <h2>Learning Flask!</h2>
          <h5>Title description, 2022</h5>
          <div class="fakeimg">Fake Image</div>
          <p>Some text..</p>
          <p>This is a revolution in the name of Flask, and everything is done in Visual Studio Code.
</p>
          <br>
          <h2>TITLE HEADING</h2>
          <h5>Title description, Sep 2, 2017</h5>
          <div class="fakeimg">Fake Image</div>
          <p>Some text..</p>
          <p>Another long text about how the world is going, and we are here learning about Flask.
What
            a beautiful thing to know, but after this, there is nothing more because we are all enjoying
            exercitation ullamco.</p>
        </div>
      </div>
    </div>

    <div class="jumbotron text-center" style="margin-bottom:0">
      <p>Footer</p>
    </div>

</body>

</html>
```

Run the program by running python -m flask run in the Terminal while your virtual environment is running, and you will see a complete Flask demo website like this:



Let us create a standard website where users can log in, sign up, and register.

Create a Real Flask Website

Create a new folder inside your base folder for your new website project. I call it app. Inside it, we are going to create a new views.py file.

So, let us begin with the homepage. Every website needs a very specific home page, and your home page will likely be very different from the rest of your website.

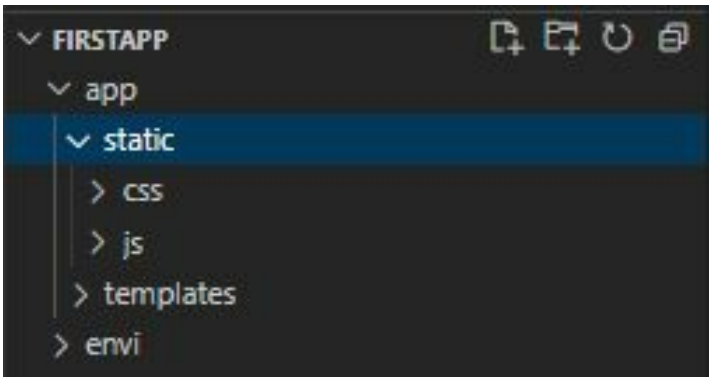
We'll have a separate home page, the only page that doesn't "extend" any header stuff like most pages. Bootstrap takes care of almost all the graphic stuff for you, which is great. You only have to decide where things go; the rest is styled for you. It really does help a lot, too.

To use Bootstrap, you'll need to ‘add’ it to your website. How do we do that?

Getting Bootsrap

It is as simple as installing Python. Go to the official Bootstrap website here: https://getbootstrap.com/. Go to the download page and download it.

Now, extract the zip file. Go to your Terminal and create a static folder inside the new project folder called static. So run mkdir static. Move the two folders js and css to the static folder.



After that, we'll need to look through the documents to see what's available. I usually just quickly scroll through until I see something that looks interesting.

You'll most likely be interested in the pages with components or JavaScript. Below each thing shown is the code that made it. Note that all the features should work if you copy and paste them onto your page. You will need to add the script to the JavaScript. See the videos if you don't know what that means. In short, you just need to include the required javascript file at the end of your HTML body tags. This means that you need to call the javascript functions before you include the javascript function in the script tags.

The file we end up making is this:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>Python Programming Tutorials</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="{{ url_for('static', filename='css/bootstrap.min.css') }}" rel="stylesheet">
  <link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}">
</head>

<header>
```

```
    <div class="navbar-header">
      <a class="navbar-brand" href="/">
        <img style="max-width:120px; margin-top: -7px;"
          src="{{ url_for('static', filename='images/mainlogo.png') }}">
      </a>
    </div>

    <div class="container-fluid">
      <a href="/dashboard/"><button type="button" class="btn btn-primary" aria-label="Left Align"
          style="margin-top: 5px; margin-bottom: 5px; height: 44px; margin-right: 15px">
          <span class="glyphicon glyphicon-off" aria-hidden="true"></span> Start Learning
        </button></a>
      <div style="margin-right: 10px; margin-left: 15px; margin-top: 5px; margin-bottom: 5px;"
        class="container-fluid">
      </div>
    </div>
</header>

<body>

    <script src="//code.jquery.com/jquery-1.11.1.min.js"></script>
    <script type="text/javascript" src="{{ url_for('static', filename='js/bootstrap.min.js') }}"></script>

</body>

</html>
```

This is the main.html file.

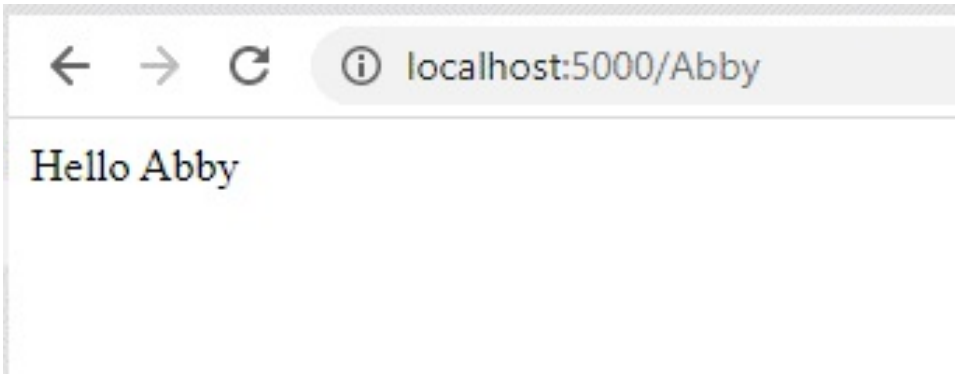
Web App

Let's actually go ahead to start building our first web page or website with flask. I have created a new .py file I call app.py in the project folder. Fill it with the following code:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Welcome to my Main Page <h1>Hello!<h1>"

@app.route("/<name>")
def user(name):
    return f"Hello {name}"

if __name__ == "__main__":
```

```
    app.run()
```

In this code, we have created a new app route. This will create a Hello and put whatever you put in after the slash.



Page redirect

Now, what if you want to redirect different pages from your code? For example, if we're going to get to a separate page, we need to type that actual page, but sometimes a user goes to a page they're not supposed to be. Perhaps they are not authenticated. We need to redirect them to the home page.

We go back to our app.py, and import two modules called redirect and url_for. These two will allow us to return a redirect from a specific function. Here is the new file:

```
from flask import Flask, redirect, url_for

app = Flask(__name__)

@app.route("/")
def home():
    return "Welcome to my Main Page <h1>Hello!<h1>"

@app.route("/<name>")
def user(name):
    return f"Hello {name}!"

@app.route("/admin")
def admin():
    return redirect(url_for("user", name="Admin!"))

if __name__ == "__main__":
    app.run()
```

In this example, we assume that we have an admin page that can only be accessed by someone who's signed in or is an admin. After creating the decorator, we input the redirect to redirect the user to a different page. We then type in the url_for() function, and inside it, we put the name of the function we want to redirect to inside of strings. Restart your server and add the slash admin to be redirected to the home page.

Template inheritance

Template inheritance is an extremely useful tool, so you're not repeating HTML code, JavaScript, or whatever it's going to be throughout your entire website. It essentially allows you to create a base template that every other one of your templates will work off of, and that is what we will use for our website with bootstrap.

I'm also going to be showing you how we can add Bootstrap to our website and just create a basic navbar.

What is Template Inheritance

If we look at the bootstrap website, for example, we can see that this website has a theme, and we can kind of detect that theme by the navbar. You see a specific color, buttons, links and so on. All pages on that website have the same theme in terms of colors and buttons.

It would be boring and stupid to keep writing the code to generate this navbar on every single web page they have because this will stay the same for most of the pages.

Flask at least makes this really easy because we can actually inherit templates. Now I'm going to do to illustrate this is just create a new template.

I'm just going to create a new file. I will save this as base.html, representing the base template or the base theme of my website. It will store all the HTML code that will persist throughout most or the entire website. So, populate the base.html with the following code:

```
<!doctype html>
```

```
<html>

<head>
  <title>Home Page</title>
</head>

<body>
  <h1>{{content}}</h1>
</body>

</html>
```

We'll start working with a few things here, so since this is our base template, we are not going to ever render this template. We'll always use this as something from which the child templates, which will be, for example, index.html, will inherit.

Inheritance essentially means to use everything and then change a few small things are overwrite some functionality of the parent, which in this case is going to be the base.html, so the way that we can allow our child templates to change specific functionality of the base template is by adding something called blocks.

```
<!doctype html>
<html>

<head>
  <title>{% block content %}{% endblock %}</title>
</head>

<body>
  <h1>Abby’s Website</h1>
</body>

</html>
```

You can see the block in the curly brackets with the same tags used to write you know for loops and if statements in HTML code. The name directly after block is the name of the block. We then simply end the block by typing endblock with similar syntax. This says we're going to define a block we're going to call content, and in this block, we will allow the child template to give us some content that we will fill in.

Let us now go to the child template I can inherit. Create a new index.html in the templates folder. Create this block and then tell the block where what content I want. Then it will substitute it inside here for a title and use that title when we render the template.

Here is the code in the index.html file:

```
{% extends "base.html" %}
{% block title %}Home Page{% endblock %}
{% block content %}
<h1>My Home!</h1>
{% endblock %}
```

Our base.html

```
<!doctype html>
<html>

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

<body>
  <h1>Abby's Website</h1>
  {% block content %}
  {% endblock %}

</body>

</html>
```

I'm going to do is actually give some content for that block title, so this is the exact same as what we had in our base template, except this time I'm actually going to put some stuff in between kind of blocks, so I'm going to say and block like that so block content and block and then inside here I'm actually just going to put homepage now what this is going to do is very similar just kind of like an HTML tag where this homepage now will be replaced with whatever this block title is and that will actually show now for us inside title so very useful. I'm going to put something that just says Abby's website, and this h1 tag will be shown on every page no matter what.

Our app.py:

```
from flask import Flask, redirect, url_for, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html")
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

The result:

So, every page we go to will have Abby's Website as the inherited template.

I was saying that we're going to have some more complex components. I'm going to show you how we can add a nav bar now and then how we can use the base template so all our other templates will have that nav bar on it. Let's actually talk about adding Bootstrap.

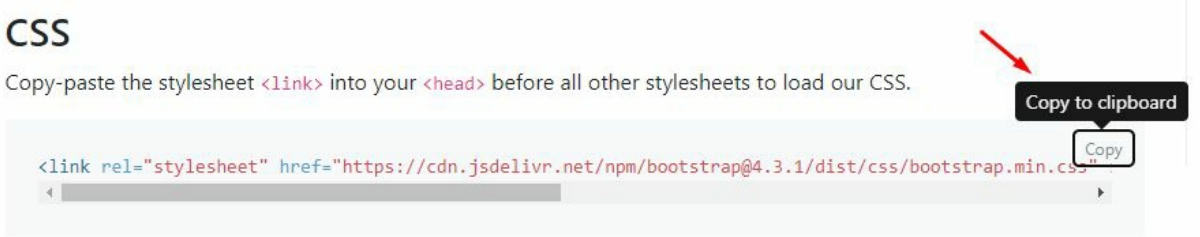
Adding Bootstrap

If you're unfamiliar with Bootstrap, it is a CSS framework for quickly creating and styling your website. To add, it is actually pretty easy.

You will go to the Bootstrap website and grab the codes! It is basically copy and paste. Don't think programmers are magicians. We don't cram stuff. Simply go here on your browser https://getbootstrap.com/docs/4.3/getting-

started/introduction/ and grab the codes.

I'm going to look where it says CSS, and I will copy the link with the copy button.



I'm going to take that CSS link and paste that inside the head tags of my website, in this case, the base.html template. Next, I'm going to go to where it says Js and copy that too and put them at the end of the body.

This will allow us to use a library of different classes and a bunch of different kinds of styling from bootstrap to make our website look nicer.

If you look at the codes, you will see cdn at the end. That means we don't need to download any Bootstrap files because this will just grab the CSS and JavaScript code from the Bootstrap server.

Nav bar From Bootstrap

I will show you how we can just grab a sidebar layout or a navbar layout from the bootstrap website. Go to the sidebar and search for whatever you need. In this case, the nav bar. Look for one that you like, as there are a bunch of different nav bar codes.

Just place the code on the website in the base template right after the first body tag. Any child template will automatically have this nav bar at the top of it.

Our current base.html code:

```
<!doctype html>
<html>

<head>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
    integrity="sha384-
```

```
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <title>{% block title %}{% endblock %}</title>
</head>

<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="#">Navbar</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav"
            aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
                <li class="nav-item active">
                    <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Features</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Pricing</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link disabled" href="#" tabindex="-1" aria-disabled="true">Disabled</a>
                </li>
            </ul>
        </div>
    </nav>
    <h1>Abby's Website</h1>
    {% block content %}
    {% endblock %}
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
        integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
        crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.14.7/dist/umd/popper.min.js"
        integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
        crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/js/bootstrap.min.js"
        integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
        crossorigin="anonymous"></script>

</body>

</html>
```

The new face of the website:

If you wanted to change anything associated with the navbar, obviously, all the codes here so you can change them, but that's just what I wanted to show you regarding how we can add bootstrap.

There are several other frameworks for styling, but I like bootstrap because it's pretty easy.

Chapter 10 – HTTP Methods (GET/POST) & Retrieving Form Data

The templates you worked with in this course are one-way, meaning that information can only flow from the server to the user. Most applications also need the information to flow in the opposite direction, from the user to the server, where it is accepted and processed.

Your website may want to collect data from users instead of serving them. This is done with forms.

With HTML, you can make web forms that users can use to enter information. The data from the form is then sent to the server by the web browser. This is usually done as a POST request.

We'll talk about HTTP methods in this chapter. The standard way to send and receive information from and to a web server is through HTTP methods. Simply put, a website runs on one or more servers and sends information to a client (web browser). The client and the server share information using HTTP, which has a few different ways to do this. We will talk about the ones called POST & GET that are often used.

GET

GET is the most common way of getting or sending information to a website. GET is the most commonly used HTTP method to retrieve information from a web server, depending on how this information is going.

POST

POST is a way of doing this securely, so GET is an insecure way of getting the most commonly used information. People often use the POST method to send information to a web server. It is often used when sending sensitive information, uploading a file, or getting form data. With POST, you can send data to a web server in a safe way.

A basic example of that is when we type something in the URL bar or in the address bar. For instance, if you have your local server running, you will see a command that pops up saying GET in the console when you go to the home page. Whenever we type something that's not secure, anyone can see it, and the data will be sent to the server here. Then it will return us the actual web page using a GET method.

If we were to use POST, what we would actually do is send secure and encrypted information. Something that cannot be seen from either end and is not stored on the actual web server. That is the difference between GET and POST.

The best way to think of it is whenever you're using a GET command, it's something that's not secure that you don't care if someone sees it. It's typically typed in through the address bar where it's just a link you redirect to, and then with POST, that's something secure. It's usually form data. It's something that we're not going to be saving on the actual web server itself unless we're going to be sending that to it.

Web Forms

Let's now go through a basic example of web forms in a website. You can use the same app.py we have been working with here. You only need to add a few different pages for this example first. Here is what the new code will be like:

```
from flask import Flask, redirect, url_for, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
    return render_template()

@app.route("/<usr>")
def user(usr):
    return f"<h1>{usr}</h1>"
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

So what I want to do is set up a page here for logging in

Login page template

Now, I'm going to go and build out the login page template inside my templates folder. Create a new file and call this login.html; inside, we'll start creating the form. Here is the script to create a form:

```
{% extends "base.html" %}
{% block title %}Login Page{% endblock %}

{% block content %}
<form action="#" method="post">
    <p>Name:</p>
    <p><input type="text" name="nm" /></p>
    <p><input type="submit" value="submit" /></p>
</form>
{% endblock %}
```

So we start by extending that base.html and then do the tags for our title. So essentially, a form is a way to send information to the website.

Whenever we know we will get some information from a form, we need to put our form tags in HTML. I'm just going to specify that here, and we need to say the action this form will take now. The action is essentially just a URL we want to redirect to once this form is submitted.

We've decided how the form will be sent. This means that when the form is sent, we will send a request with the data to the web server.

Note the name of the text input field. We will use this to get the value of the field from our Python code.

So we've created the form. You need to go back to the app.py and render the new template.

Back-End

First, add request to the imports in your app.py script. So the first line is like

this:

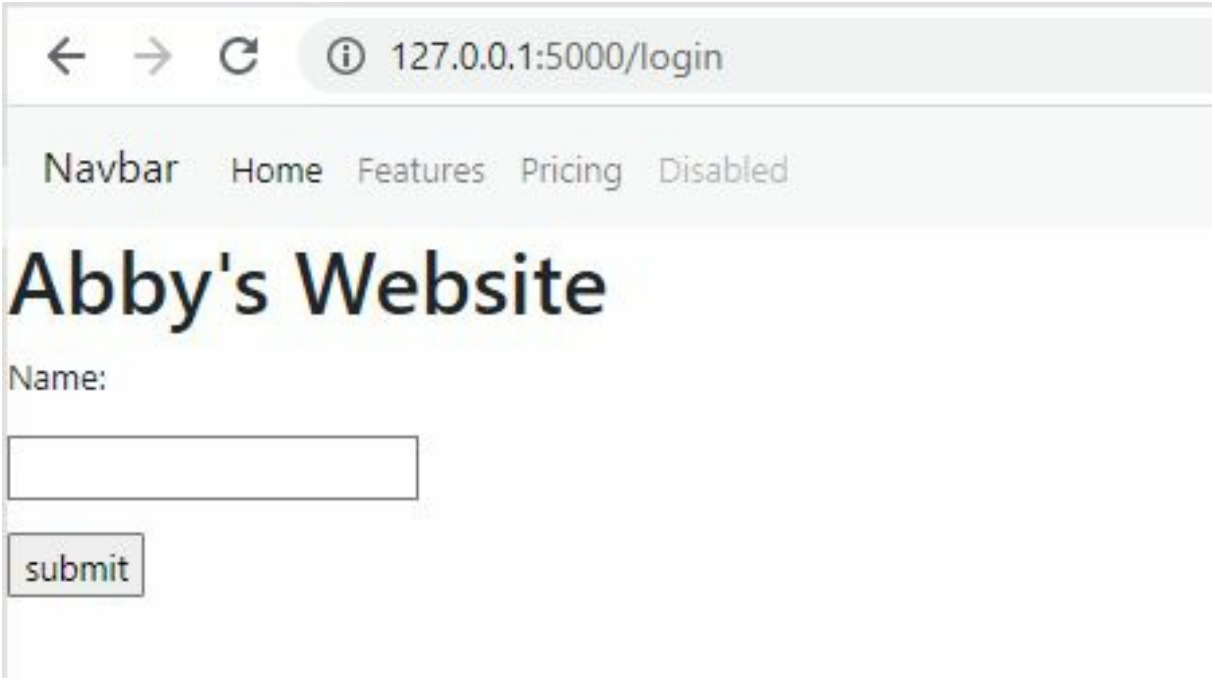
```
from flask import Flask, redirect, url_for, render_template, request
```

Now, add the new login.html in the login function block like this:

```
@app.route("/login", methods=["POST", "GET"])
def login():
    return render_template("login.html")
```

Now, we have rendered this template. We need to figure out how we'll get this information and handle it from this side. You can test your new update by restarting the server in your console and going to the url/login.

You should see a basic little box where we can type some things in, and we have a submit button.



However, when you hit that button, all you see is a hashtag here, and it's different because it's using POST. If you refresh the page, you will get a GET request rather than a POST.

The job of the request we imported is to determine in this login function whether we called the GET request or the POST request. I will show you how we can check whether we reach this page with a GET request or a POST

request.

Basically, all we're going to do is use an if-else clause. We say if request.method == POST, then we're going to do something specific. Otherwise, we'll do something else.

In this case, what I'm going to do is move this render down here, so if we have the get request, what we're going to do is render the log in template because that means you know we didn't click the submit button we're just going to the /login page so let's show it here but if we have POST what I want to do is actually get the information that was from that little name box and then uses that and send us to the user page where we can display the user's name.

So how do we do that? It's pretty easy, so all we need to do is set up a variable that will store our users' names. We need to say user equals request.form, and then we will put the dictionary key that we want for the name corresponding. In the login.html script, we had name = nm, so we'll put nm as a dictionary key in the code. What that's going to do is actually give us the data that was typed into this input box.

```
def login():
   if request.method == "POST":
      user = request.form["nm"]
      return redirect(url_for("user", usr=user))
   else:
      return render_template("login.html")
```

We are using the redirect(url_for) function to make sure that this page will not be blank before we go to the next page. We are telling Flask to use the data from the form to redirect us to the user page. Refresh your server and test it out.

After clicking submit:

You can see that we get redirected to a page that says our name.

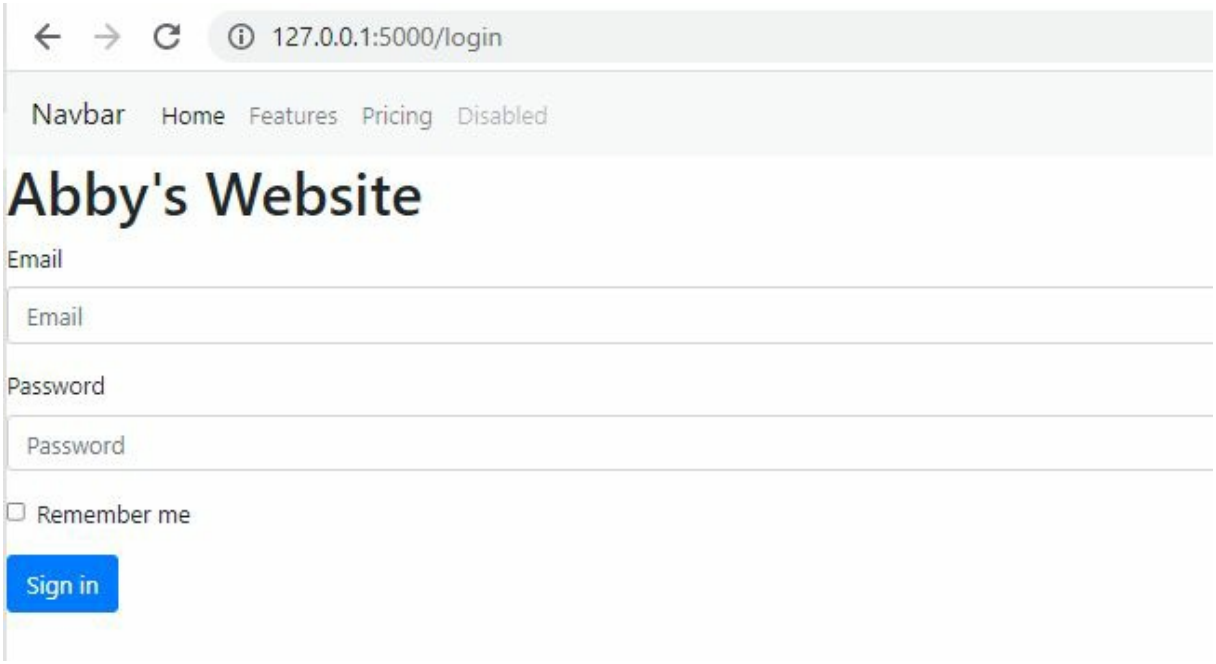
That is how we actually get information from a form, and obviously, if you have more than one info you want, you just add it to the input type in the login.html script. Then you can get all those information by just using the name as a dictionary key on the request.form.

Bootstrap forms

If you want to create a beautiful form, you can use Bootstrap. Just like we got the code for the nav bar, you can get the code for a good form.

```
{% extends "base.html" %}
{% block title %}Login Page{% endblock %}

{% block content %}
<form action="#" method="post">
  <div class="mb-3">
    <label class="form-label" for="inputEmail">Email</label>
    <input type="email" class="form-control" id="inputEmail" placeholder="Email">
  </div>
  <div class="mb-3">
    <label class="form-label" for="inputPassword">Password</label>
    <input type="password" class="form-control" id="inputPassword" placeholder="Password">
  </div>
  <div class="mb-3">
    <div class="form-check">
      <input class="form-check-input" type="checkbox" id="checkRemember">
      <label class="form-check-label" for="checkRemember">Remember me</label>
    </div>
  </div>
  <button type="submit" class="btn btn-primary">Sign in</button>

</form>
{% endblock %}
```

This code is still built on the base template.



You have learned the basics behind this. Notice that request.form comes in as a dictionary, meaning you can access each object using the key.

Most applications need to take information from the user through web forms, store that information, and use it for the user experience. The next chapter is about sessions and cookies in Flask.

Chapter 11 – Sessions vs. Cookies

This chapter is about sessions. Now to try to explain what sessions are, I'm going to give you an example of what we did in the previous chapter and talk about how we could do this better.

So essentially, we had a login page, and once we logged in, we got the user's name, and then we redirected them to a page that showed them their name. But every time we want to see the users' names, we need them to log in again and again.

What if we want to direct to another page and that page wants the user's name? that means we have to set up a way to pass the user's name to that page. For example, if we want to set up a page for a specific user. That means we have to use a parameter, set up another link, and so on. That is not the best way to do things, and sometimes you know you don't want to redirect to a page it says /Abby or /Jo.

What we're going to do to pass around information through the back-end and our different web pages is use something called sessions.

Sessions

Sessions are great because they're temporary. They're stored on the web server and simply there to quickly access information between your website's different pages. Think of a session as something you'll load to use while the user is on your website.

That session will work when they're browsing on the website, and then as soon as they leave, it will disappear. For example, on Instagram or Facebook, when someone logs in, a new session will be created to store their username. Probably some other information as well about what they're doing on the website at the current time, and then as they can go between different pages, those pages can access that session data so it can say okay, so I moved to my profile page this is the profile of Abby I know that because I stored that in a session. So let's show all the information I have stored in the session that only Abby needs to see.

Then, as soon as that user leaves the web page or logs out, all of that session data is erased. And the next time they log in, data will be reloaded into the

session, where it can be used for the rest of the pages.

Sessions or Cookies?

Do you accept cookies? You may have seen this a lot. I just want to quickly explain the difference between a cookie and a session to clear up any confusion.

Cookie: This feature is stored on the client side (in the user's web browser) and is NOT a safe way to store sensitive information like passwords. It is often used to remember where a user left off on a page or their username so that it will be filled in automatically the next time they visit the page.

Session: This is saved in a temporary folder on the web server. It is encrypted and is a safe way to store information. Sessions are often used to store information that the user shouldn't be able to see or modify.

How to set up a Session

I want to do an example where the user logs in, we create a session for them that stores the name, and then we can redirect to another page that doesn't have this /user.

In this basic example, a user logs in, and we will hold their username in a session until they log out.

Let us open our app.py and add session from flask and timedelta from datetime to the first lines.

```
from flask import Flask, redirect, url_for, render_template, request, session
from datetime import timedelta
```

When the user presses login or submit on that login page, we will set up session data based on whatever information they typed in.

I will paste the finished script up here and explain the process:

```
from flask import Flask, redirect, url_for, render_template, request, session
from datetime import timedelta
```

```
app = Flask(__name__)
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

@app.route("/")
def home():
    return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
    if request.method == "POST":
        session.permanent = True
        user = request.form["nm"]
        session["user"] = user
        return redirect(url_for("user"))
    else:
        if "user" in session:
            return redirect(url_for("user"))

        return render_template("login.html")

@app.route("/user")
def user():
    if "user" in session:
        user = session["user"]
        return f"<h1>{user}</h1>"
    else:
        return redirect(url_for("login"))

@app.route("/logout")
def logout():
    session.pop("user", None)
    return redirect(url_for("login"))

if __name__ == "__main__":
    app.run(debug=True)
```

After importing session, the first and most important thing in sessions is the data to set up.

Under the login function, we set the session to the user = user. This is to set up some data for our session to store data as a dictionary just like we've seen this requests.form. If I want to create a new piece of information in my session, I can simply type the name of whatever I want that dictionary key to be and then set it equal to some specific value. In this case, this is the user who clicks Submit to the form.

How do we get that information to use on another page? Next, I will change the redirect to redirect to the user, but I'm not going to pass the user as an argument without passing any information from the user function.

To do that, I need a conditional clause in the user function. This new statement will first check if there's any information in the session before I reference the user's dictionary key. Technically, someone could just type /user and access the user page without being logged.

That is as easy as it is to store and retrieve session data.

Next, the else statement. This is what Flask will do if this session does not exist. If there is no user in my session, that means that the user has not logged in yet or has left the browser and needs to log in again. That is the job of the redirect line.

Now, what does the secret key do? It is essentially the way that we decrypt and encrypt data. The line is usually typed at the beginning of the script as app.secretkey with any string you want.

Session Data

If someone logs out, you probably want to delete all the information associated with their session or at least some of that information. So you need a new page for logout.

The job of the session.pop() function is to remove some data from our session. In the function, we pass in “user”, None. What this is going to do is actually remove the user data from my sessions. This is just how you remove it from the dictionary. Then this none is just a message that's associated with removing that data.

After that, we must return the user to the login page. So we'll say url_for (“login”).

Session Duration

Remember that as it stands, the session data is deleted when the user closes the browser. That is why we need the permanent sessions. Now what I'm going to do to set up the permanent session here is define how long I want a permanent session to last. So you may have sometimes noticed you know you

revisit a website a few days later, and you just log in immediately. You don't actually have to, you know, go through the process, or maybe your information is already typed in, and you just hit login. We will store some of this information in permanent sessions, which means keeping it longer. So that every time you go back to that web page, you can quickly access information that you need, and you don't need to log back.

CHAPTER 12 – MESSAGE FLASHING

In this chapter, we will talk about flashing messages on the screen. Essentially, message flashing shows some kind of information from a previous page on the next page when something happens on the GUI.

For example, say I log in, it redirects me to another page and then maybe on the top of that page, it says logged in successfully, login error, or if I log out, perhaps I'm going to get redirected to another page. Still, I want to show on that other page that I logged out successfully, so I'll flash a message in a specific part of that page so that the user has some idea of what they actually did. This is to give them a little bit more interaction with the page.

flash() Function

Instead of thinking about changing the whole page or passing through some new variables to show on the screen, you can just flash a message quickly with a module called flash().

All you need to do is to import flash. Then you can use this function to display or kind of like post the messages that are to be flashed and then from the different pages, we can decide where we want to flash those, and we'll do that in a second.

We will use the same app.py from the previous section because we will also deal with sessions and log in.

The simple syntax is

flash(message, category)

A basic example of when you might want to flash a message in our app.py script is where a user logs out. When we log out, we go to a logout page that pops our session and redirects us back to the login page. What if we can show a “Logged out successful” message on that page so that they know there wasn't an error?

First, import flash from flask. That means it goes in the first line. So go to the logout function in the script and input the following line before the redirect line:

```
flash("You Have Logged Out Successfully!", "info")
```

The next parameter for this is the category, which is optional. Still, I'm going to put “info” as the category. One of the built-in categories includes a warning, info, and error.

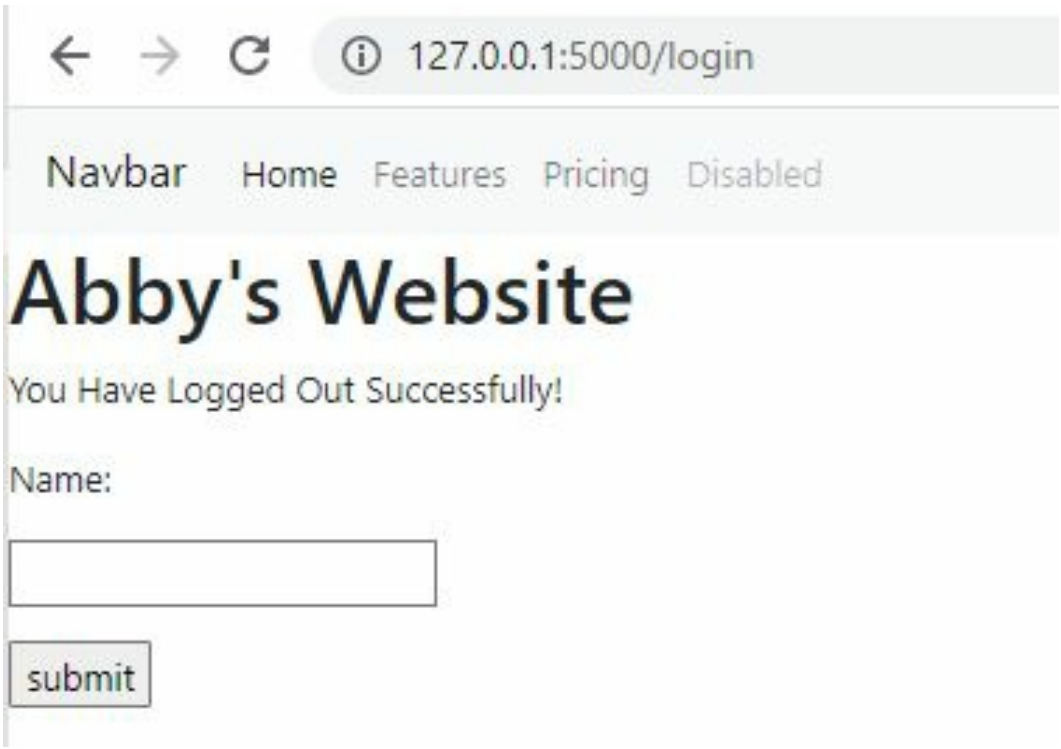
Displaying Flash Message

Now that we have written a message, we need to display the message from our different pages. So go to the login page, and inside the block content, write a templated code here to show all of the flashed messages that come up:

```
{% extends "base.html" %}
{% block title %}Login Page{% endblock %}

{% block content %}
{% with messages = get_flashed_messages() %}
{% if messages %}
{% for msg in messages %}
<p>{{msg}}</p>
{% endif %}
{% endif %}
{% endwith %}
<form action="#" method="post">
    <p>Name:</p>
    <p><input type="text" name="nm" /></p>
    <p><input type="submit" value="submit" /></p>
</form>
{% endblock %}
```

The new thing in this is the with, which is just another Python syntax you can use here. It says to check if there's any to display. We'll loop through them and show them.

Notice that we can have more than one flash message, which means if we go between a few different pages, we'll show two or three flash messages on a specific page.



As you see, when I logged out, I saw the message.

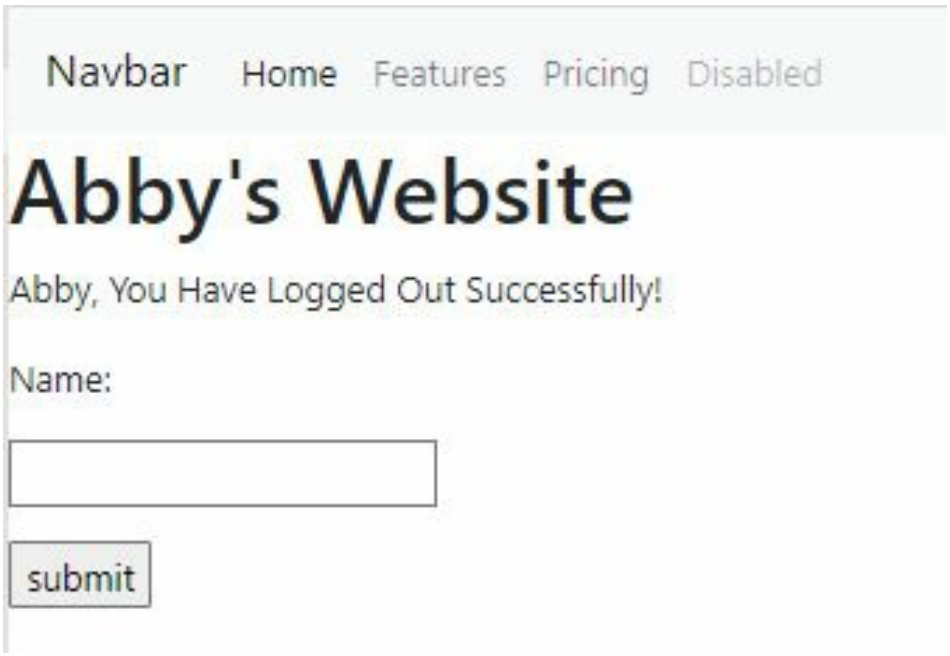
The problem we have now is that this message pops out whenever you type /log out, even if you had not been logged in before. In that case, we could check if we have a user in the session and only if we do will we say you've been logged out.

What will we do? We will add an if statement to check if the user was in session and then display their name and say they have been logged out.

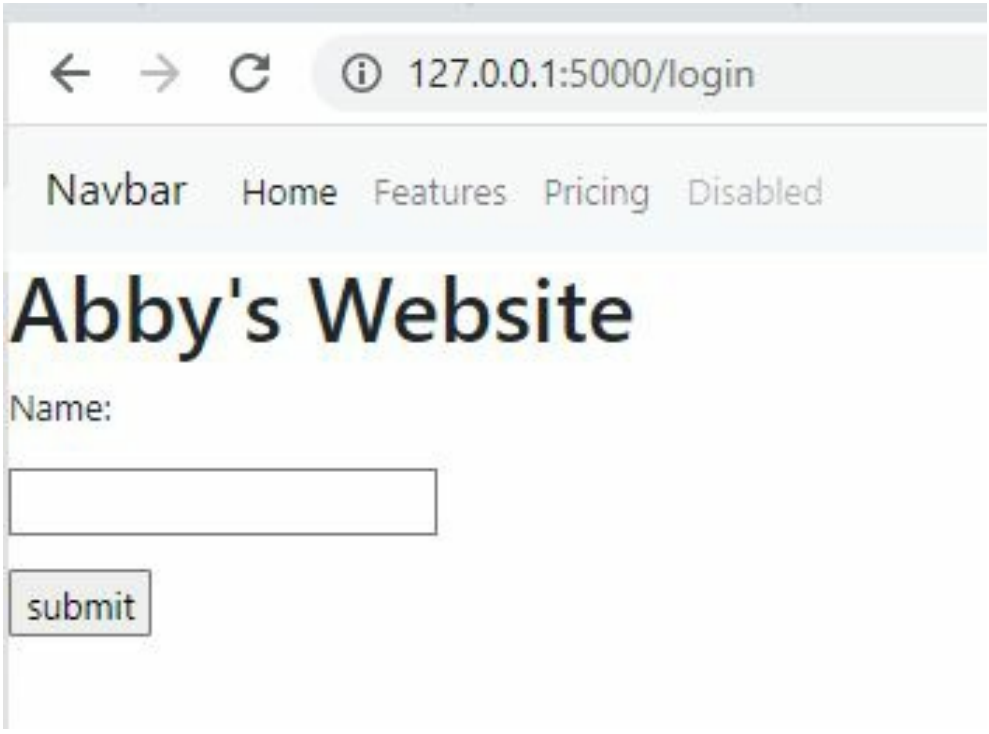
```
from flask import Flask, redirect, url_for, render_template, request, session, flash
from datetime import timedelta

app = Flask(__name__)
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

@app.route("/")
def home():
    return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
    if request.method == "POST":
        session.permanent = True
        user = request.form["nm"]
```

```
        session["user"] = user
        return redirect(url_for("user"))
    else:
        if "user" in session:
            return redirect(url_for("user"))

        return render_template("login.html")

@app.route("/user")
def user():
    if "user" in session:
        user = session["user"]
        return f"<h1>{user}</h1>"
    else:
        return redirect(url_for("login"))

@app.route("/logout")
def logout():
    if "user" in session:
        user = session["user"]
        flash(f"{user}, You Have Logged Out Successfully!", "info")
    session.pop("user", None)
    return redirect(url_for("login"))

if __name__ == "__main__":
    app.run(debug=True)
```

If you run this, you will get the result:



Not logged in, it displays this without the flash message.



Displaying More Than 1 Message

We will change the app.py and create a new user.html file to do this example.

Let's start by creating a new HTML file that we'll use to render the user page. With the app.py right now, we just have some h1 tags. Let us make something that looks a little bit nicer.

```
{% extends "base.html" %}
{% block title %}User{% endblock %}
{% block content %}
   {% with messages = get_flashed_messages() %}
      {% if messages %}
        {% for message in messages %}
          <p>{{ msg }}</p>
        {% endfor %}
      {% endif %}
   {% endwith %}
<h2>User Authenticated</h2>
<p>Welcome, {{user}}</p>
{% endblock %}
```

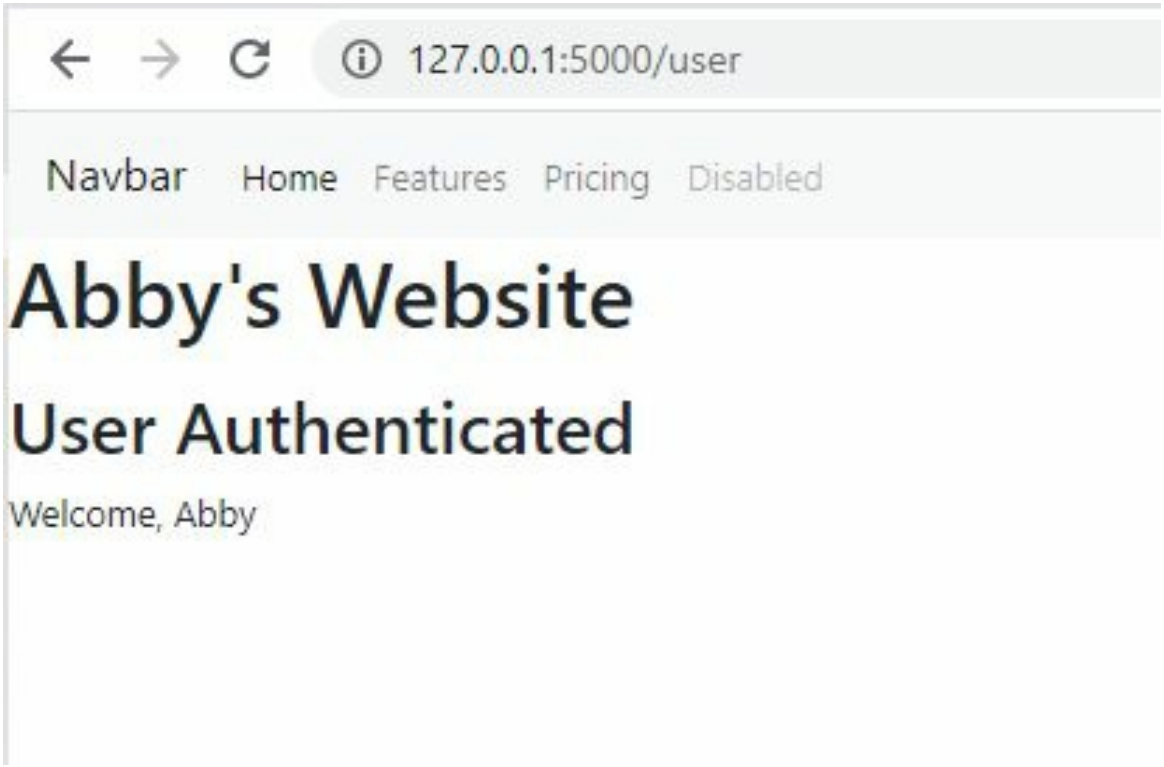
Now, we go to the user function in the app.py file and render the new template.

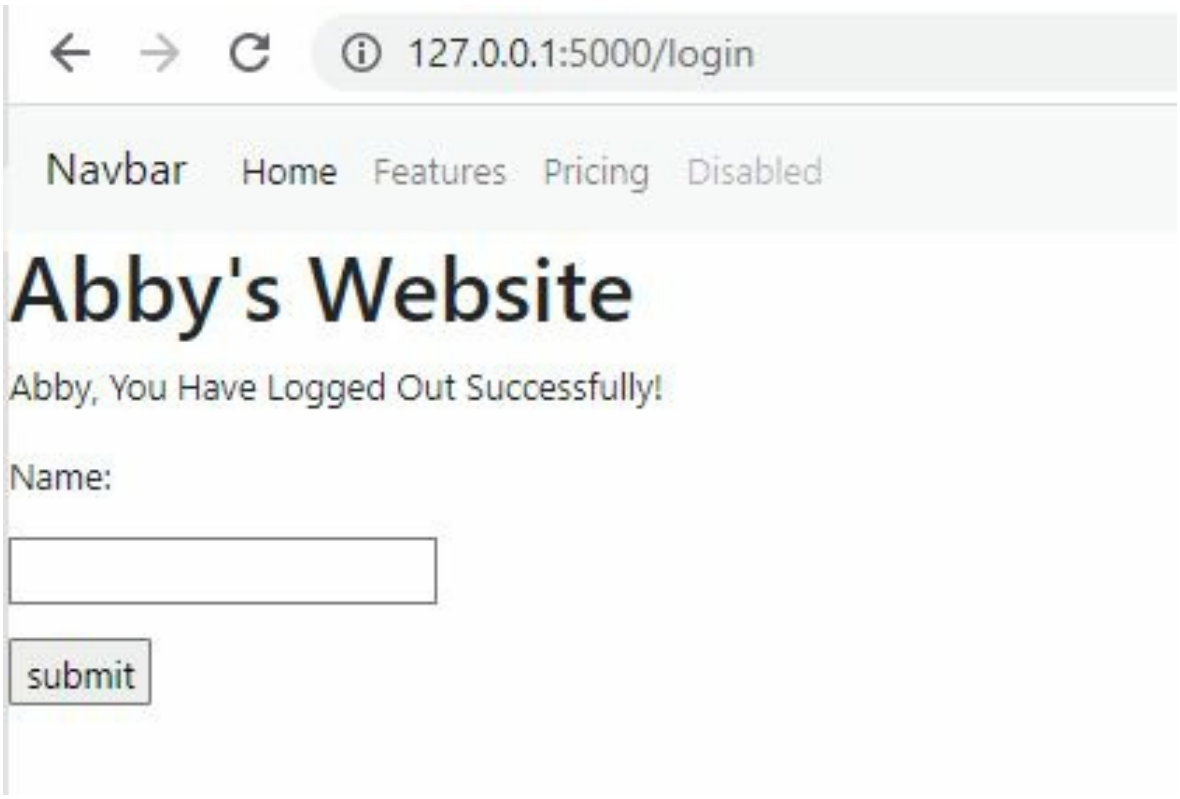
We can also flash a new message after running the log-in function. We could also flash “You are not logged in” when the person tries to enter the /user page from the url.

```
from flask import Flask, redirect, url_for, render_template, request, session, flash
from datetime import timedelta

app = Flask(__name__)
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

@app.route("/")
def home():
   return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
   if request.method == "POST":
      session.permanent = True
      user = request.form["nm"]
      session["user"] = user
      flash("You Are Logged In!")
      return redirect(url_for("user"))
   else:
      if "user" in session:
```

```
            flash("You Are Logged In!")
            return redirect(url_for("user"))
        return render_template("login.html")

@app.route("/user")
def user():
    if "user" in session:
        user = session["user"]
        return render_template('user.html', user=user)
    else:
        flash("You Are NOT Logged In!")
        return redirect(url_for("login"))

@app.route("/logout")
def logout():
    if "user" in session:
        user = session["user"]
        flash(f"{user}, You Have Logged Out Successfully!", "info")
        session.pop("user", None)
        return redirect(url_for("login"))

if __name__ == "__main__":
    app.run(debug=True)
```

Now, let us test it out:





That is the message flashing. In the next chapter, we'll get into the basic database and discuss how to set up a scalable web server.

CHAPTER 13 – SQL ALCHEMY SET UP & MODELS

In this chapter, what we're going to be doing is talking about databases and how we can actually save user-specific information to the database.

Application data is organized and kept in a database. When necessary, the program then issues queries to retrieve specific portions of the data. The majority of online applications make use of relational model-based databases. Because they employ Structured Query Language, these databases are sometimes known as SQL databases. However, document-oriented and key-value databases, also known as NoSQL databases, have gained popularity as alternatives in recent years.

A few database models will be made.

A Flask add-on called Flask-SQLAlchemy makes it simpler to use SQLAlchemy in Flask programs. Strong relational database framework SQLAlchemy is compatible with a variety of database backends. It has both a high-level ORM and a low-level way to access the SQL features of the database.

Creating A Simple Profile Page

We want to collect few data from the user. When the user logs in, they're brought to a page where they can modify some information about themselves. This is called CRUD (create-update-update-delete).

Well, to keep things simple, we're just going to make that information an email. In this program, we will create, and each user will upload an email. When they go there, they can change their email, they can update it, they can delete the email, and we'll save that in a database and then the next time that the user logs in, we'll look for that email, and we'll display it, and then they can change it. This will give you an idea of how we have persistent information in CRUD programs in Flask.

Database Management with Flask-SQL Alchemy

We will need to install it as an extension in our virtual environment. Stop your server in the command prompt or terminal and do a pip install flask-SQLalchemy.

pip install flask-sqlalchemy

Once done, open your app.py and import sqlachemy.

Now we're just going to work on some of the front-end stuff for the website. So we will start by getting the form set up, grabbing some information from the form with a post request and then we'll get into the database.

The first step is the user.html file.

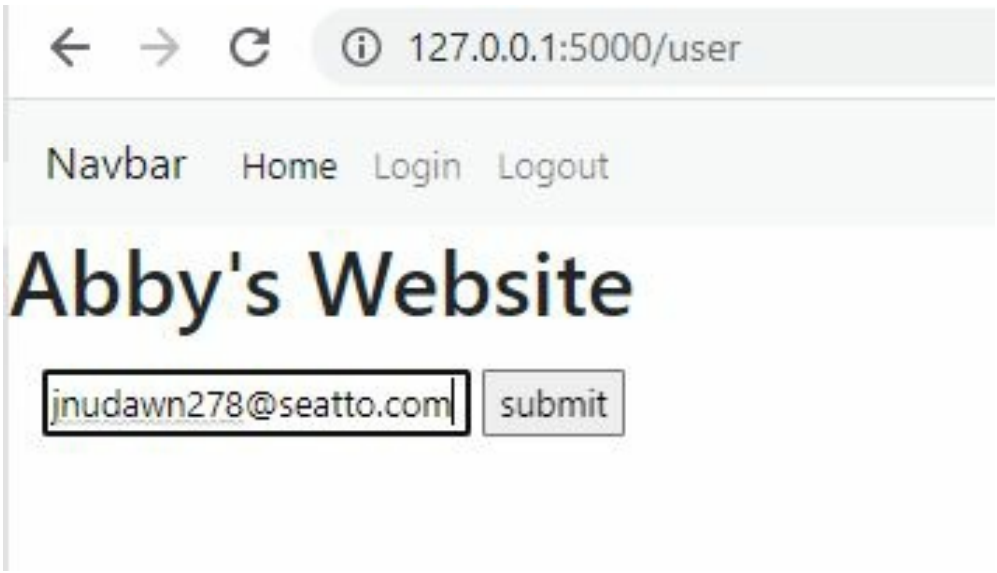
```
{% extends "base.html" %}
{% block title %}User{% endblock %}
{% block content %}
{% with messages = get_flashed_messages() %}
{% if messages %}
{% for message in messages %}
<p>{{ msg }}</p>
{% endfor %}
{% endif %}
{% endwith %}
<form action="#" method="POST">
   <input type="email" name="email" placeholder="Enter Email" value="{{email if email}}" />
   <input type="submit" value="submit" />
</form>
{% endblock %}
```

And look closely at the changes in the app.py file:

```
from flask import Flask, redirect, url_for, render_template, request, session, flash
from datetime import timedelta
from flask_sqlalchemy import SQLAlchemy

app = Flask (__name__)
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

@app.route("/")
def home():
   return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
   if request.method == "POST":
      session.permanent = True
      user = request.form["nm"]
      session["user"] = user
      flash("You Are Logged In!")
      return redirect(url_for("user"))
```

```
    else:
        if "user" in session:
            flash("You Are Logged In!")
            return redirect(url_for("user"))
        return render_template("login.html")

@app.route("/user", methods=["POST", "GET"])
def user():
    email = None
    if "user" in session:
        user = session["user"]

        if request.method =="POST":
            email = request.form["email"]
            session["email"] = email
        else:
            if "email" in session:
                email = session["email"]

        return render_template('user.html', email=email)
    else:
        flash("You Are NOT Logged In!")
        return redirect(url_for("login"))

@app.route("/logout")
def logout():
    if "user" in session:
        user = session["user"]
        flash(f"{user}, You Have Logged Out Successfully!", "info")
        session.pop("user", None)
        session.pop("email", None)
        return redirect(url_for("login"))

if __name__ == "__main__":
    app.run(debug=True)
```

The result:



I've actually made a change in the base template too. I've added this html thing that says div class equals container-fluid. It is a bootstrap class that covers the entire web page.

Here is the entire code:

```
<!doctype html>
<html>

<head>
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
    integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
  <title>{% block title %}{% endblock %}</title>
</head>

<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav"
      aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item active">
          <a class="nav-link" href="/">Home <span class="sr-only">(current)</span></a>
        </li>
        <li class="nav-item">
```

```
            <a class="nav-link" href="/login">Login</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="/logout">Logout</a>
          </li>
        </ul>
      </div>
    </nav>
    <h1>Abby's Website</h1>
    <div class="container-fluid">
      {% block content %}
      {% endblock %}
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
      integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
      crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.14.7/dist/umd/popper.min.js"
      integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
      crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/js/bootstrap.min.js"
      integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
      crossorigin="anonymous"></script>

</body>

</html>
```

I did with sessions to save the users email in a session, and then once we have it in a session, we can change the user page to have a method. That is the change in the user function in the app.py file. Like our login page, you can see the methods=”POST” and “GET”.

I set up a bit of code to collect and save the email in the session. And we use the if statement to check the current method.

You can play around with the code and even show some message to the user to tell them that their email is saved.

If you go to /login, you can see that it still has the email saved. If we close a web browser as we did not use a permanent session, the data will go away and won't be saved.

How to use database

Now it's time to talk about databases. We've done great in saving data but have not set up a database to collect it. The data is saved in the session. This means that the data will disappear once you close the browser or after 5 mins.

To set up a database, you need to create a Flask application object for the project and set the URI for the database to use. Add this line immediately after the __name__

```
app = Flask (__name__)
app.config ['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.sqlite3'
app.config ["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

db = SQLAlchemy(app)
```

We use users because that's what we're going to use, and then again sqlite3. The second line shows that we're not tracking all the modifications to the database. And the last line creates the database.

Models

Now, let us create models for the data we want to collect and save into the database we have created.

```
class students(db.Model):
   _id = db.Column('id', db.Integer, primary_key = True)
   name = db.Column(db.String(100))
   email = db.Column(db.String(100))
```

So that you can understand how databases work, I will explain object relation mapping. Object Relation Mapping is used to identify and store objects. SQL is based on objects. The objects are referenced by tables like Ms Excel. Tables hold the information in the RDBMS server. A relational database management system (RDBMS) is a group of programs experts use to create,

update, administer and otherwise interact with a relational database like SQL.

Object-relational mapping is a method for relating an RDBMS table's structure to an object's parameters. SQLAlchemy helps you do CRUD operations without having to write SQL statements.

You must set up the table, that is, what we want to represent. Any pieces of information can be stored in rows and columns in our database for each object you want to collect. In this case, we want a single column, that is, the name and email of a user.

The columns will represent pieces of information, and the rows will represent individual items. We want to store users, and our users are going to have. In this case, just a name and an email, and that's all we want to store. We define a class to represent this user object in our database. That is why we call it users. You can play around with the names if you want to store more than names, emails, or different information.

Every single object that we have in our database needs to have a unique identification. That is why we set an _id class. The identification could be a string, boolean or an integer. After selecting the db.column name, we set the input type and the length or the maximum length of the string that we want to store. In this case, we use 100 characters.

Lastly, we will need to define the function that shows the database that we are collecting the data from the user.

```
def __init__(self, name, email):
    self.name = name
    self.email = email
```

This __init__() method will take the variables we need to create a new object because technically, we can store some values here that will be None values. Some objects might not actually have a value for that property. For example, say we have gender as an option, and now some people decide not to declare that, and we want to leave that as none.

The next thing to do is go to the bottom of the script to add something db.create_all(). This is a method to actually create this database if it doesn't already exist in our program whenever we run this application.

```
from flask import Flask, redirect, url_for, render_template, request, session, flash
from datetime import timedelta
from flask_sqlalchemy import SQLAlchemy

app = Flask (__name__)
app.config ['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.sqlite3'
app.config ["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
app.secret_key = "hello"
app.permanent_session_lifetime = timedelta(minutes=5)

db = SQLAlchemy(app)
class students(db.Model):
   _id = db.Column('id', db.Integer, primary_key = True)
   name = db.Column(db.String(100))
   email = db.Column(db.String(100))

   def __init__(self, name, email):
      self.name = name
      self.email = email
```

```
@app.route("/")
def home():
   return render_template("index.html")

@app.route("/login", methods=["POST", "GET"])
def login():
   if request.method == "POST":
      session.permanent = True
      user = request.form["nm"]
      session["user"] = user
      flash("You Are Logged In!")
      return redirect(url_for("user"))
   else:
      if "user" in session:
         flash("You Are Logged In!")
         return redirect(url_for("user"))
      return render_template("login.html")

@app.route("/user", methods=["POST", "GET"])
def user():
   email = None
   if "user" in session:
      user = session["user"]

      if request.method =="POST":
         email = request.form["email"]
         session["email"] = email
      else:
         if "email" in session:
            email = session["email"]
```

```
        return render_template('user.html', email=email)
    else:
        flash("You Are NOT Logged In!")
        return redirect(url_for("login"))

@app.route("/logout")
def logout():
    if "user" in session:
        user = session["user"]
        flash(f"{user}, You Have Logged Out Successfully!", "info")
        session.pop("user", None)
        session.pop("email", None)
        return redirect(url_for("login"))

if __name__ == "__main__":
    db.create_all()
    app.run(debug=True)
```

CHAPTER 14 - CRUD

Now, let us create a simple web app where users can create, update, delete and read posts. This is called a CRUD app. We will give our users the ability to store information about books. The database is SQLAlchemy and SQLite.

The app we're making here isn't meant to be valid on its own. But once you know how to write a simple web app that takes user input and stores it in a database, you are well on your way to writing any web app you can think of. So, we'll keep the example application as simple as possible so you can focus on the tools themselves instead of details about the application.

You have learned how to set up and model a database. Now is time to watch it work and build web apps.

The Flask Book Store

First, we need to create a simple database. Then our app lets users to write book titles and upload them as text. They can also read posts that they have added, change or delete them.

CRUD scripts are found in almost every web app out there. Whatever you want to build, you'll need to get user input and store it (let your user create information), show that information back to your user (let your user read data), find a way to fix old or wrong information (let your user update information), and get rid of information that isn't needed (let your user delete information) (allow users to delete information that was previously added). This will make more sense when we see how each CRUD operation works in our web application.

Ensure that the following are installed:

- Flask
- SQLAlchemy
- Flask-SQLAlchemy

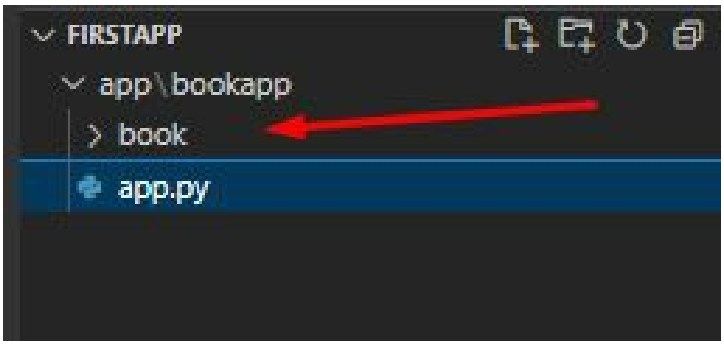
You can install everything with pip by running the following command:

```
pip3 install --user flask sqlalchemy flask-sqlalchemy
```

Remember to install them in your virtual environment.

Your static web page with Flask

Flask is simple. That is one of its biggest selling points as a web framework. We can get a simple page running in only a few lines of code. Create a folder for your project, create a file inside it called bookmanager.py or leave it as app.py.



In this case, book is my virtual environment. You can create a basic page with this code in your app.py:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "This is an app"

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Handling user input in our web application

You know the basics of how this works, but I will just go over it a bit. We have a simple web app that doesn't do much now. We want to create an app to take the users' content. We'll do this by adding an HTML form that sends information from the front-end of our application (what our users see) through Flask and to the back-end (our Python code).

In the Python code for our above application, we set up the string "This is an app." This was fine because it was only one line, but as our front-end code

grows, defining everything in our Python file will become more complex. Flask lets you keep different things separate by using templates.

Templates

Create an index.html file in your templates folder. This is the first template for this project. You can fill it with the following code:

```
<html>

<body>
   <form method="POST" action="/">
      <input type="text" name="book">
      <input type="submit" value="Add">
   </form>
</body>

</html>
```

This is a simple HTML page that has:

A text input that will link any text entered to the name "book."

A simple form

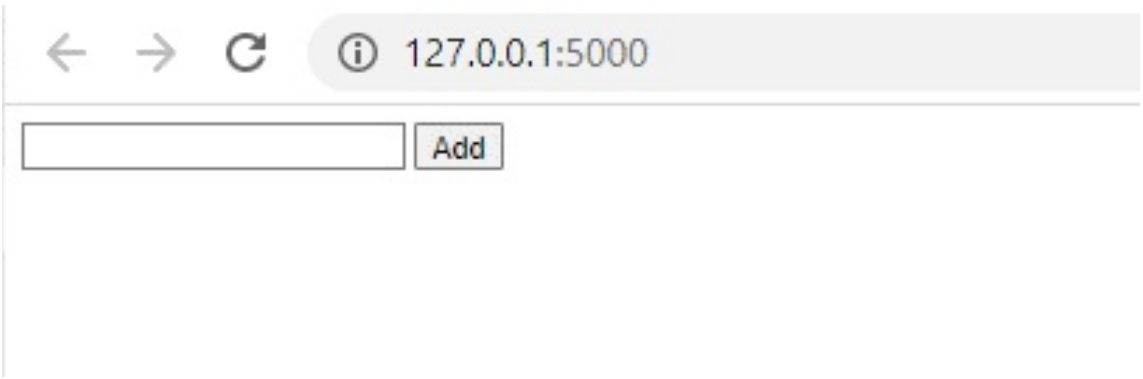
A submit button with the word "Add" on it.

A direction to send the data ("post") to our web application's main page (the / route, which is the same page we set up in our app.py file).

We need to make two changes to the app.py file to use our new template. Add render_template to the imports section, and replace “This is an app” with the following:

```
return render_template("home.html")
```

Start your server and check in your web browser:



This is a simple box where the user can type in text and click "Add." Doing this will send the text to the back-end of our app, and we'll all be on the same page. Before we can try it, our back-end code needs to be changed to handle this new feature.

Back-end

A method="POST" line in our index.html file says that the data in the form should be sent using HTTP POST. We have learned that Flask routes only accept HTTP GET requests by default. What matters to us is that if we send in our form right now, we'll get an error message that says, "Method not allowed." Because of this, we need to change our app.py file so that our web application can handle POST requests.

So import request and update your new home function to be like this:

```
@app.route("/", methods=["GET", "POST"])
def home():
    if request.form:
        print(request.form)
    return render_template("index.html")
```

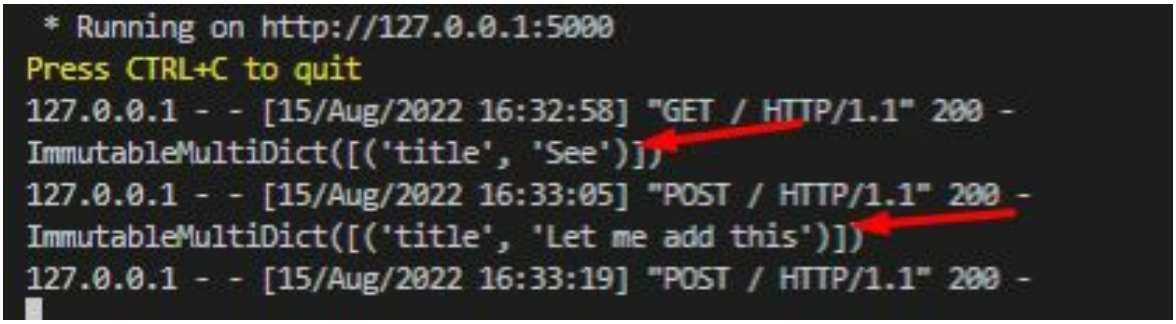
Here's what we did:

We changed our route decorator by adding methods=["GET", "POST"]). This will get rid of the "Method not allowed" error we got when we tried to send the form before. By default, Flask lets all routes accept GET requests. Here, we tell it to let both GET and POST requests.

We use the if request.form to see if the form was just sent by someone. If they did, we can use the request.form variable to get the information they sent

in. We'll just print it out to make sure our form works.

Restart your server and test the page now. Type anything into the box and click "Add." In the console, the string you typed should show up as output, like in the picture below.



Flask stores all of the form data in an ImmutableMultiDict, a fancy Python dictionary. It saved the user's input as a tuple typed into the form, and "title" is the name we gave it in the home.html template.

You see the two items I added. You can play around with this with flash messages or everything. This is only a sample to help open your creative mind.

We are not there yet. Now that we know how to get user input and do something with it let's learn how to store it.

Add a database

To help our Flask app remember our users' input, we need to add the items to a database. We have learned how to set up and model a database in the last chapter. Let us do that now.

```
import os

from flask import Flask, render_template, request

from flask_sqlalchemy import SQLAlchemy

project_dir = os.path.dirname(os.path.abspath(__file__))
database_file = "sqlite:///{}".format(os.path.join(project_dir, "bookdatabase.db"))

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = database_file

db = SQLAlchemy(app)
```

On line 1, we add an import for the os Python library. This lets us access paths on our file system relative to our project directory.

In line 7, we import SQLAlchemy's Flask version (we had to install both Flask-SQLAlchemy and SQLAlchemy). We only import Flask-SQLAlchemy because it extends and depends on the SQLAlchemy base installation.

In lines 9 and 10, we find out where our project is and set up a database file with its full path and the sqlite:/ prefix to tell SQLAlchemy which database engine we are using.

Next, we show our app where to store our database.

Then, we set up a connection to the database and store it in the db variable. This is what we'll use to talk to our database.

Lastly, we set up the database.

This recap summarizes how to set up the SQLAlchemy database for your program. Now we can give the database what to store and how to store it in our database.

For a real book store app, there are many details the user may need to post. The programmer would have to model a lot of information, like the book's author, title, number of pages, date, etc. For simplicity, we are only allowing users to post titles. Add the code below to app.py. This is how each book will be stored in our database. Make sure to add the code below the line db = SQLAlchemy(app) since we use db to define the book model.

```
class Book(db.Model):
    title = db.Column(db.String(80), unique=True, nullable=False, primary_key=True)

    def __repr__(self):
        return "<Title: {}>".format(self.title)
```

Front-end

When a user types in the name of a book, we can now make a Book object and store it in our database. To do this, update the home() function once more

so that it looks like this.

```
def home():
    if request.form:
        book = Book(title=request.form.get("title"))
        db.session.add(book)
        db.session.commit()
    return render_template("index.html")
```

When we get input, we no longer need to send it to the console. Instead, we make a new Book object using our form's "title" field. We assign this new Book to the book variable.

Then, we include the book into our database and save the modifications.

This is the app.py:

```
import os

from flask import Flask, render_template, request

from flask_sqlalchemy import SQLAlchemy

project_dir = os.path.dirname(os.path.abspath(__file__))
database_file = "sqlite:///{}".format(os.path.join(project_dir, "bookdatabase.db"))

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = database_file

db = SQLAlchemy(app)
class Book(db.Model):
    title = db.Column(db.String(80), unique=True, nullable=False, primary_key=True)

    def __repr__(self):
        return "<Title: {}>".format(self.title)

@app.route("/", methods=["GET", "POST"])
def home():
    if request.form:
        book = Book(title=request.form.get("title"))
        db.session.add(book)
        db.session.commit()
    return render_template("index.html")

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Initializing

When we write codes like app.py, Python, Flask, or other frameworks need to run the code every time we run the program. There is a way to run a setup code that will be one-time only.

Open a python shell. You can do this in your Terminal or command prompt by typing Python or Python3. This will open up the Shell. In the shell, run the following 3 lines.

```
>>> from app import db

>>> db.create_all()

>>> exit()
```

You may now return to your online application and add as many book titles as you like. Our CRUD application is complete, and we have reached the C stage, where we can generate new books. The next step is to restore our ability to read them.

Retrieving books from our database

We'd want to retrieve all the latest books from the database and show them to the user every time they visit our web app. Using SQLAlchemy, we can quickly and easily store a Python variable with all the books in our database. Just before the end of the home() method, add a line to retrieve all of the books and change the final line so that the books are passed to our front-end template. Home() should end with these two lines.

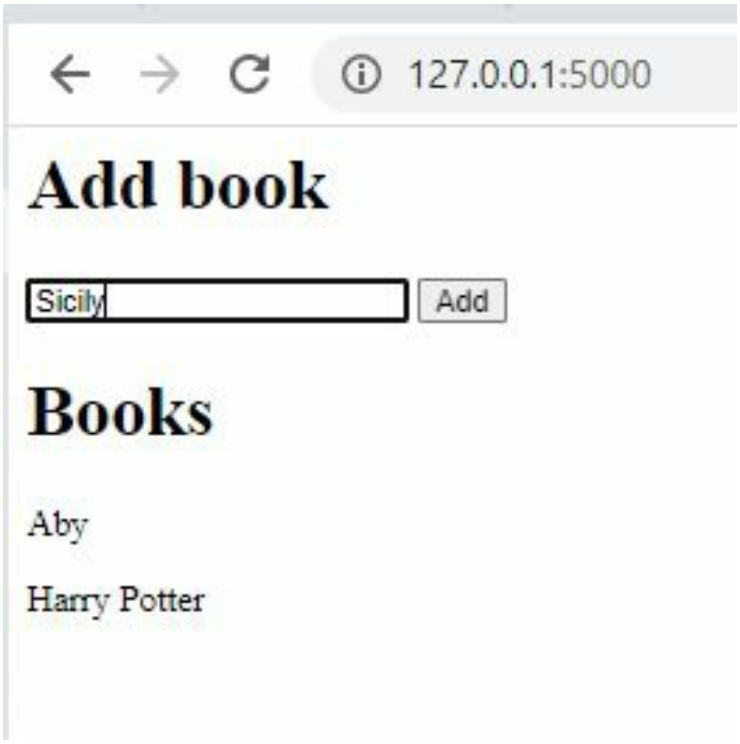
```
books = Book.query.all()
return render_template("index.html", books=books)
```

You can currently render all the books in the home.html file using a Jinja for loop. While you're working on the file, feel free to add the headers we'll need for the form and the list of books to appear. Here is the complete code for the index.html page.

```
<html>

<body>
   <h1>Add book</h1>
   <form method="POST" action="/">
```

```
    <input type="text" name="book">
    <input type="submit" value="Add">
  </form>

  <h1>Books</h1>
  {% for book in books %}
  <p>{{book.title}}</p>
  {% endfor %}
</body>

</html>
```

Simply save the file and refresh the program in your browser to see the changes take effect. You should now see the books as you add them, as shown below.



We have successfully finished the C and the R of our CRUD program. That is, our users can now Create and Read their content. Next, How can we update that Aby to a book title?

Updating book titles

Now, the last and probably the most complex part of the project is data updates.

We only present a representation of the data on our front end. As a result, the user won't be able to alter anything. As an alternative, we request that you send us a more recent title while we archive the earlier one. The newly updated book can be found using the old title in our code, which will replace it with the one the user submitted.

We'll create each title in its own distinct form because it's unlikely that the user will want to manually enter both the old and new titles. The previous title will be available to us when the user gives the revised one. We will use a hidden HTML input to get the previous title without having it appear in the user interface.

Change the for loop in our home.html file to the following:

```
{% for book in books %}
<p>{{book.title}}</p>
<form method="POST" action="./update">
    <input type="hidden" value="{{book.title}}" name="oldtitle">
    <input type="text" value="{{book.title}}" name="newtitle">
    <input type="submit" value="Update">
</form>
{% endfor %}
```

The form is similar to the previous form that we used to add new books. Here are a few critical updates:

We first need to direct this form's data submission to the /update app route, not the home page. Since we have not created a decorator for it, we must do that in our app.py file.

We use a secret input on line 4 to provide the "old" title of the book. This area will automatically be filled from the program database. So the user will see it.

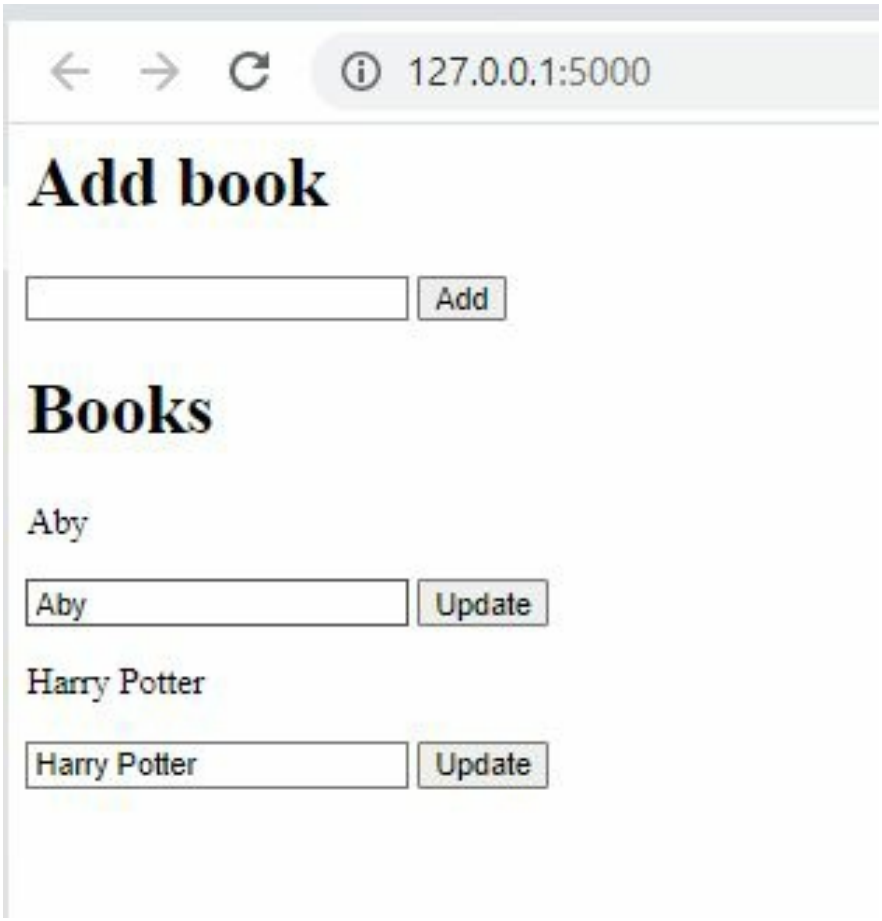
Open your app.py file and add a redirect to the imports list.

Now add the new route decorator for /update with the following block:

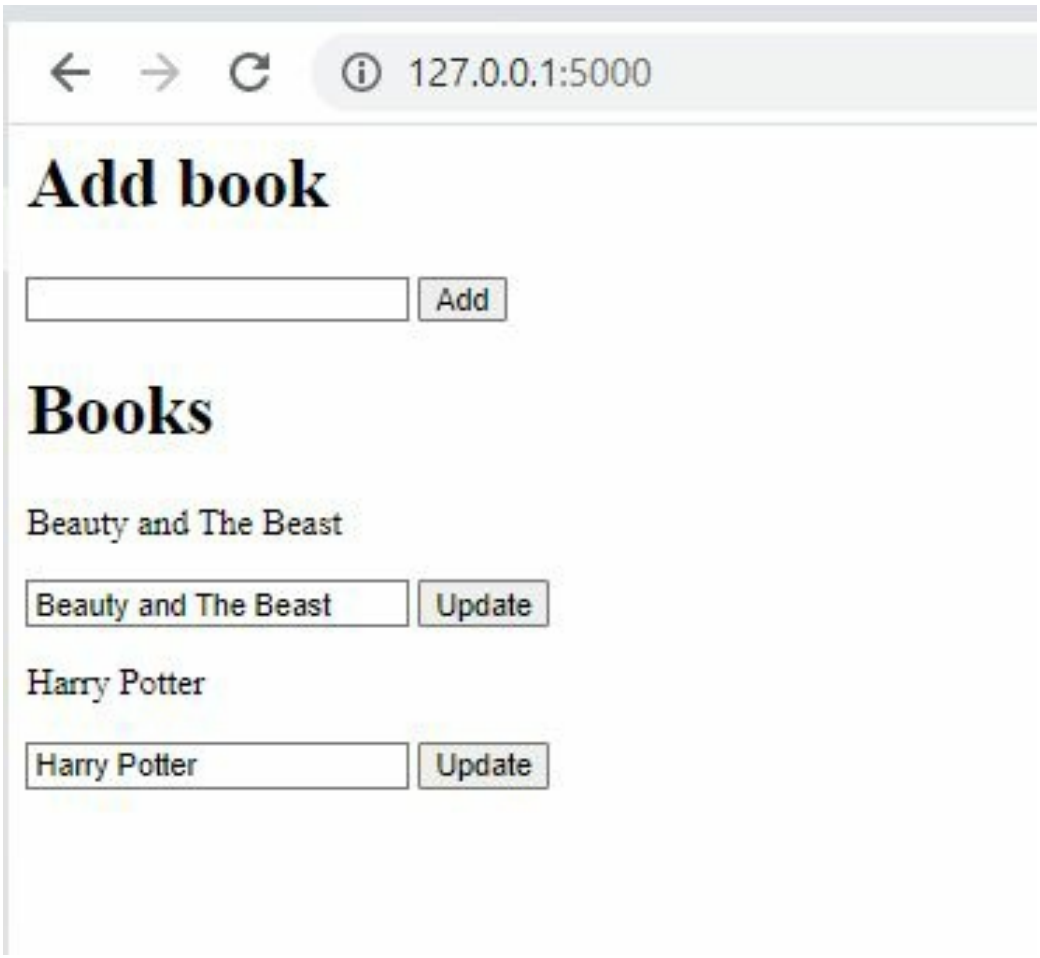
```
@app.route("/update", methods=["POST"])
def update():
    newtitle = request.form.get("newtitle")
    oldtitle = request.form.get("oldtitle")
    book = Book.query.filter_by(title=oldtitle).first()
```

```
    book.title = newtitle
    db.session.commit()
    return redirect("/")
```

If you refresh the application page in your browser, you should see something that looks like the image below. It is possible to alter the titles of already-created books by editing the corresponding input field and clicking the "Update" button.



After updating:



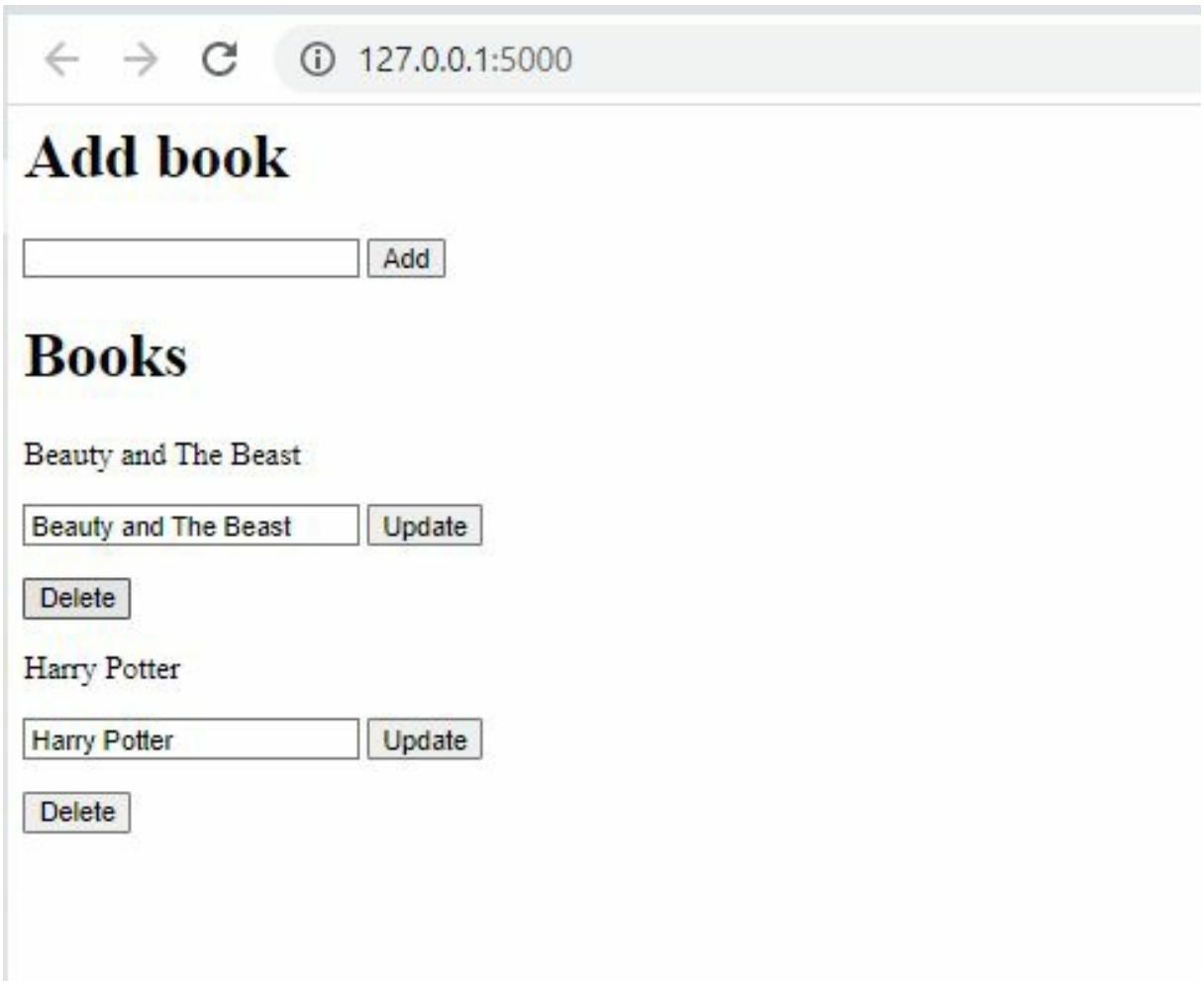
Well done. You have seen how we handle the CRU with Flask. Now, let us give the user the power to Delete books that they no longer want to see.

Deleting books from our database

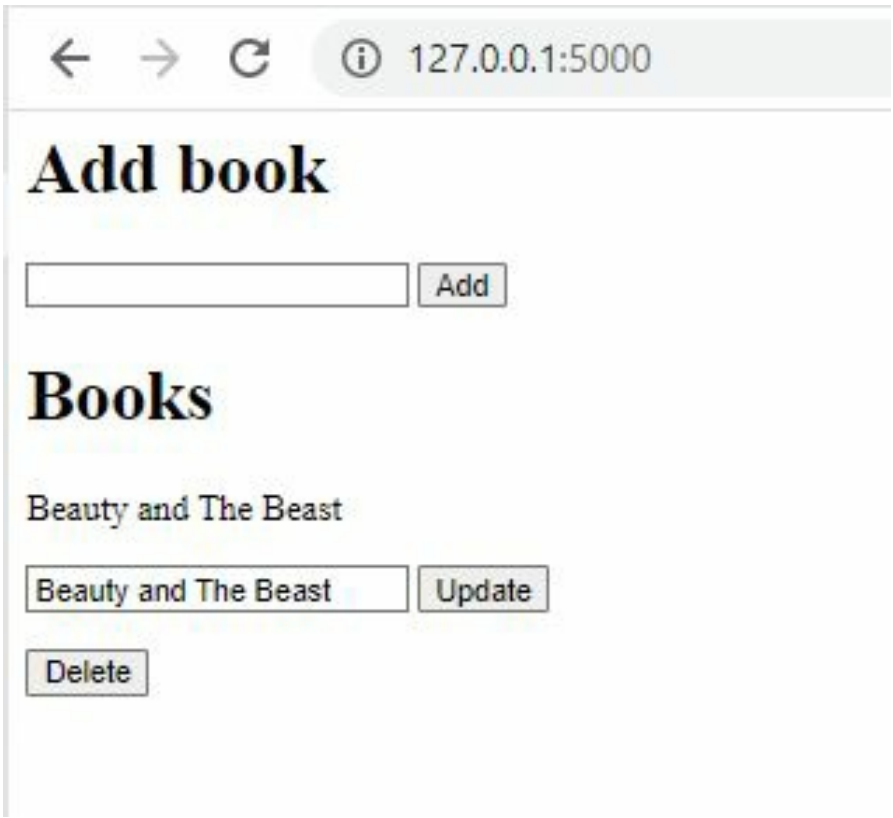
This feature is not completely different from how we created the Update feature. In this case, we don't need to call the old title. Open the index.html file and create another form with the for loop.

```
<form method="POST" action="./delete">
    <input type="hidden" value="{{book.title}}" name="title">
    <input type="submit" value="Delete">
</form>
```

Next, add a new app route decorator in the app.py for the /delete route and create the function.



After deleting:



We have created a Flask app that handles CRUD operations like magic! This is the basics. You can use this to create a standard app anyhow you want by playing around with the code and Bootstrap.

These are the working codes:

app.py:

```
import os

from flask import Flask, render_template, request, redirect

from flask_sqlalchemy import SQLAlchemy

project_dir = os.path.dirname(os.path.abspath(__file__))
database_file = "sqlite:///{}".format(os.path.join(project_dir, "bookdatabase.db"))

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = database_file

db = SQLAlchemy(app)
class Book(db.Model):
    title = db.Column(db.String(80), unique=True, nullable=False, primary_key=True)
```

```
    def __repr__(self):
        return "<Title: {}>".format(self.title)

@app.route('/', methods=["GET", "POST"])
def home():
    books = None
    if request.form:
        try:
            book = Book(title=request.form.get("title"))
            db.session.add(book)
            db.session.commit()
        except Exception as e:
            print("Failed to add book")
            print(e)
    books = Book.query.all()
    return render_template("index.html", books=books)

@app.route("/update", methods=["POST"])
def update():
    try:
        newtitle = request.form.get("newtitle")
        oldtitle = request.form.get("oldtitle")
        book = Book.query.filter_by(title=oldtitle).first()
        book.title = newtitle
        db.session.commit()
    except Exception as e:
        print("Error in updating title")
        print(e)
    return redirect("/")
```

```
@app.route("/delete", methods=["POST"])
def delete():
    title = request.form.get("title")
    book = Book.query.filter_by(title=title).first()
    db.session.delete(book)
    db.session.commit()
    return redirect("/")
```

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Our index.html will look like this:

```
<html>
```

```
<body>
    <h1>Add book</h1>
    <form method="POST" action="/">
        <input type="text" name="title">
        <input type="submit" value="Add">
    </form>

    <h1>Books</h1>
    <table>
        {% for book in books %}
        <tr>
            <td>
                {{book.title}}
            </td>
            <td>
                <form method="POST" action="./update" style="display: inline">
                    <input type="hidden" value="{{book.title}}" name="oldtitle">
                    <input type="text" value="{{book.title}}" name="newtitle">
                    <input type="submit" value="Update">
                </form>
            </td>
            <td>
                <form method="POST" action="./delete" style="display: inline">
                    <input type="hidden" value="{{book.title}}" name="title">
                    <input type="submit" value="Delete">
                </form>
            </td>
        </tr>
        {% endfor %}
    </table>
</body>

</html>
```

The new code now has error handling codes.

When we first learnt about databases, I stated that each object must be distinct. If we try to add a book with the same title twice or change the title of an existing book to one that already exists, an error will occur. The revised code will have a try: except: block around the home() and update() blocks.

CHAPTER 15 – DEPLOYMENT

At long last, our app is ready for release. The deployment has begun. There are a lot of factors to consider, which can make this procedure tedious. When it comes to our production stack, there are also many options to consider. In this section, we'll go over a few key components and the various customization paths available to us for each of them.

Web Hosting

Since the beginning of this tutorial, you have been using your local server, which only you can access. You need a server that is accessible to everyone. There are thousands of service providers that give this, but I use and recommend the three below. The specifics of getting started with them are outside the scope of this book. Therefore I won't be covering them here. Instead, I'll focus on why they're a good choice for Flask app hosting.

Amazon Web Services EC2

Amazon Web Services (AWS) is the most common option for new businesses, so you may have heard of them. I am talking about the Amazon Elastic Compute Cloud (EC2) for your Flask app. The main selling feature of EC2 is the speed with which new virtual computers, or "instances" in AWS lingo, may be created. Adding more EC2 instances to our app and placing them behind a load balancer allows us to swiftly expand it to meet demand (we can even use the AWS Elastic Load Balancer).

For Flask, AWS is equivalent to any other form of the virtual server. In a matter of minutes, we can have it running our preferred Linux distribution, complete with our Flask app and server stack. However, this necessitates that we have some expertise in systems management.

Heroku

Heroku is a platform for hosting applications developed on top of existing AWS capabilities, such as Elastic Compute Cloud (EC2). As a result, we could enjoy EC2's benefits without learning the ins and outs of systems administration.

When using Heroku, we simply push our application's source code repository to their server through git. This is handy when we don't feel like logging into

a server through SSH, configuring the software, and thinking out a sensible deployment strategy. These luxuries don't come cheap, but both AWS and Heroku provide some levels of service at no cost to the user.

Digital Ocean

In recent years, Digital Ocean has emerged as a serious alternative to Amazon Web Services EC2. In the same way that EC2 allows us to easily create virtual servers, Digital Ocean will enable us to create what they call droplets. In contrast to the lower tiers of EC2, all droplets use solid-state drives. The most appealing feature for me is the interface's superior simplicity and ease of use compared to the AWS control panel. If you're looking for a hosting service, I highly recommend Digital Ocean.

Using Flask for deployment on Digital Ocean is similar to using EC2. We're going to install our server stack on a new Linux distribution.

Requirements for deployment

This section will discuss the software that must be installed on the server before we can begin to host our Flask application. As a case study, I will use Heroku as our deployment server. What do you need to deploy to Heroku?

Before Heroku accepts to deploy your app, you need to add two files to your project folder and install the app runner called Gunicorn:

You must create a requirements.txt file to specify your app's dependencies and a special Heroku file called Procfile.

Gunicorn

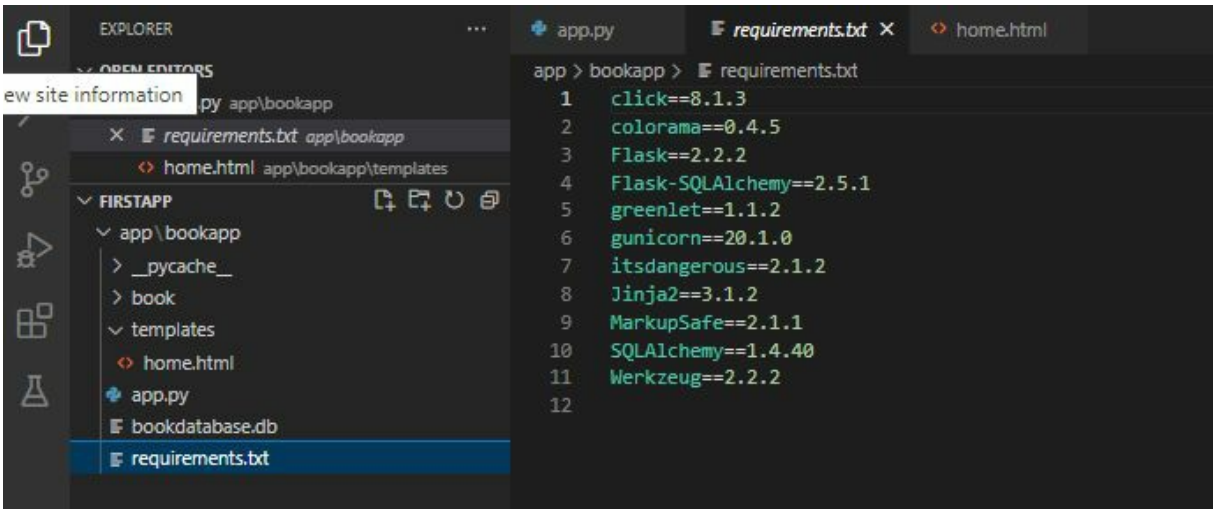
It is easy to get this by installing with pip:

```
pip install gunicorn
```

After that, use the following command to create the requirements.txt file.

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

Your app's dependencies will be determined mechanically by pip and dumped into requirements.txt.



In the end, Heroku will look to the Procfile to determine how to launch our application. It will be instructed to use the gunicorn web server instead of the local development server.

Create a file named Procfile and save it in the project's root folder with the following contents:

web: gunicorn app:app

Replace the first “app” with the name of the module or file for your main flask file and the second “app” with the name of your flask app.

My app’s module name is app because the script is in the file app.py, and the other is the app name also app because that’s the name of my script in the file.

Once you have the requirements.txt and Procfile in your root folder, you can deploy!

Deploy!

While there are a number of options for getting your app up and running on Heroku, git is by far the most straightforward.

Set up Git

A git repository should have been created for your project's directory; all that remains is to make a commit of all of your code. Now run git init to initialize git for your code.

git init

git add .

git commit -m “initial commit”

These three commands will configure and commit your script so that Heroku knows that you are ready for deployment.

Push your Site

Finally, use the following command to push your program up for production into the Heroku environment:

Type heroku create in your terminal and wait for a few minutes. Then run the following line:

git push heroku main

It may take a few seconds to push your code. That command takes your code to the Heroku server. Now is time to switch from SQLAlchemy models to the new PostgreSQL database that Heroku understands. Type python in your command line to open the shell and run the following commands:

```
>>> from app import db
>>> db.create_all()
```

When you type exit(), you will close the shell. To test your web app, type heroku open. It will take you to a free domain name with your code running in production.

Part 2 Django

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.

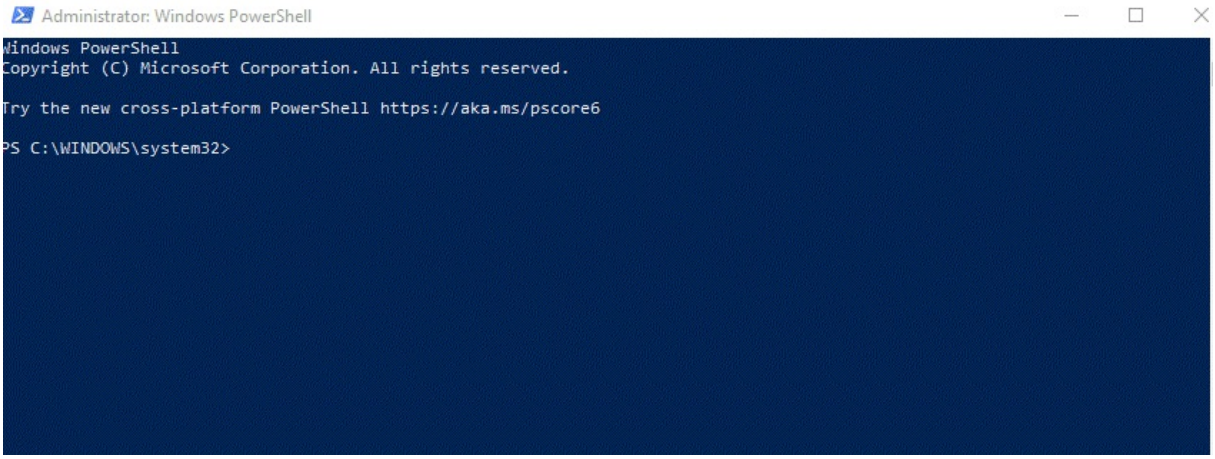
In real life, the command line is also called the console, terminal, shell, prompt, or Command Line Interface (CLI). Technically, the terminal is the

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

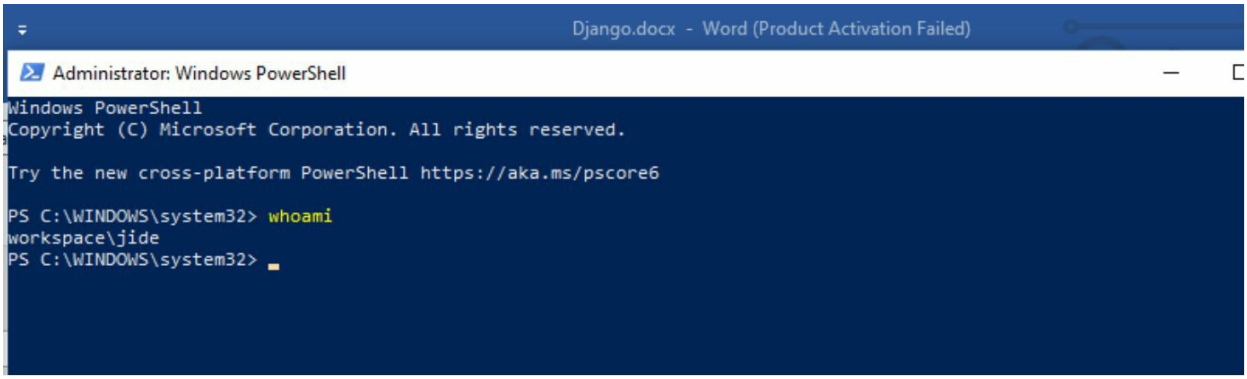
what comes after the percent sign. It's different for each computer and can be changed in the future.

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

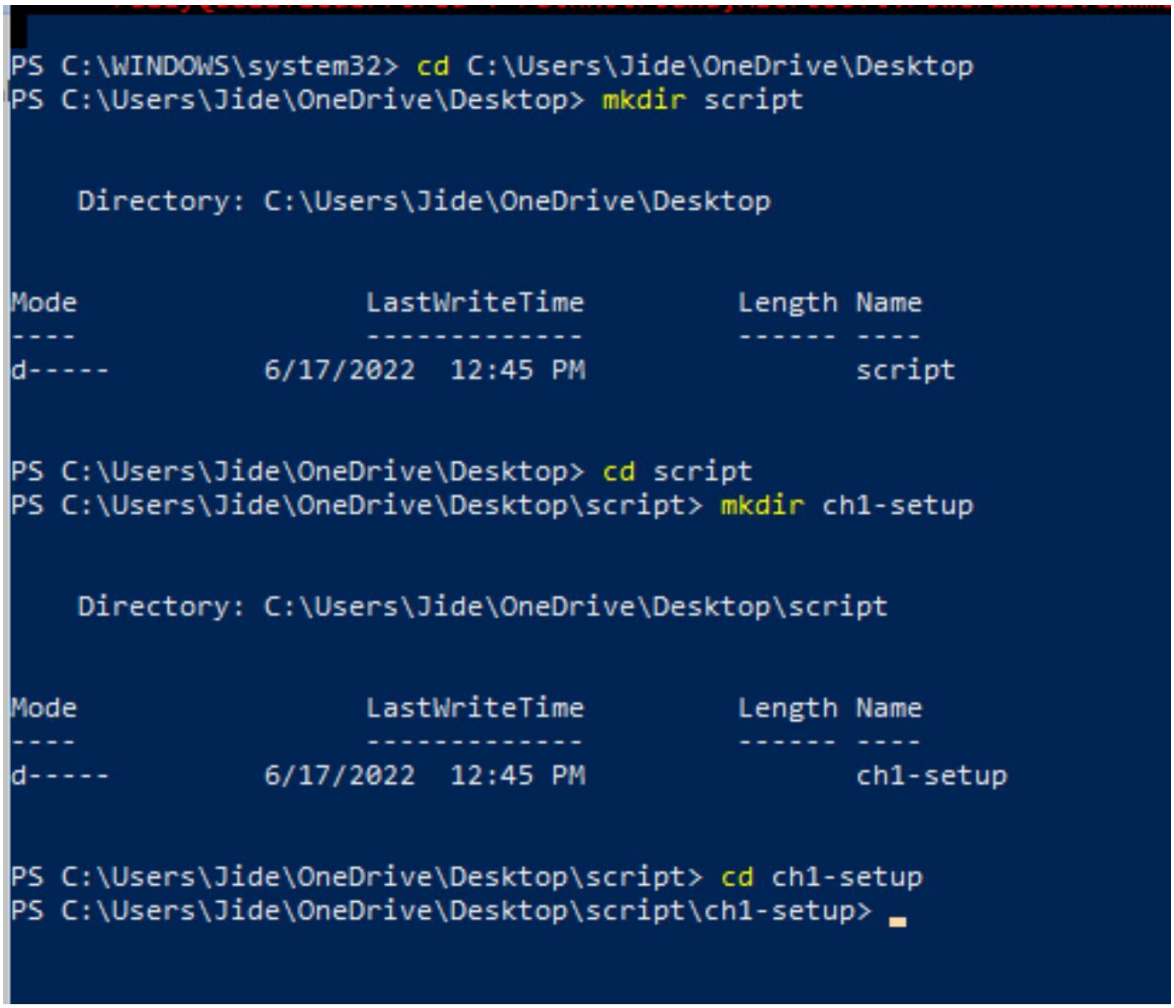
```
% cd desktop
```

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

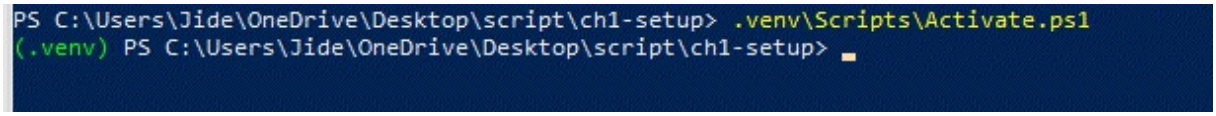
It is up to the developer to choose a good name for the environment, but .venv is a common choice.

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 .

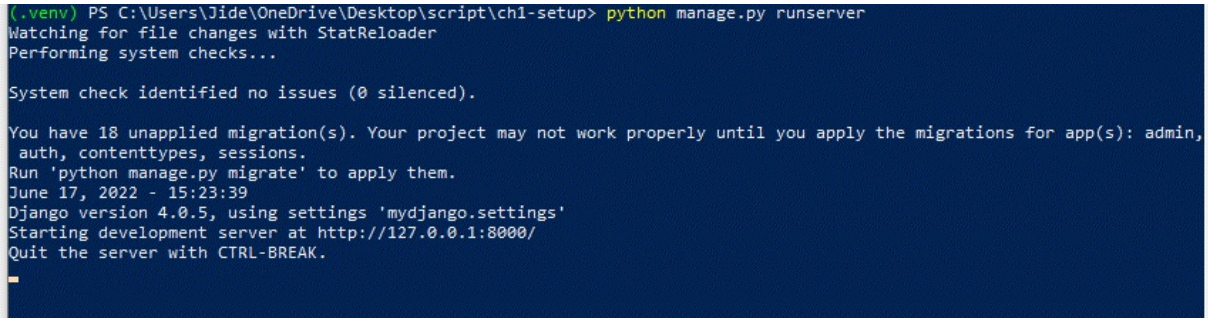
Where mysite is the name of your project. You can use almost any name, but we will use mydjango in this book. This is what the command line loos like:

(.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:

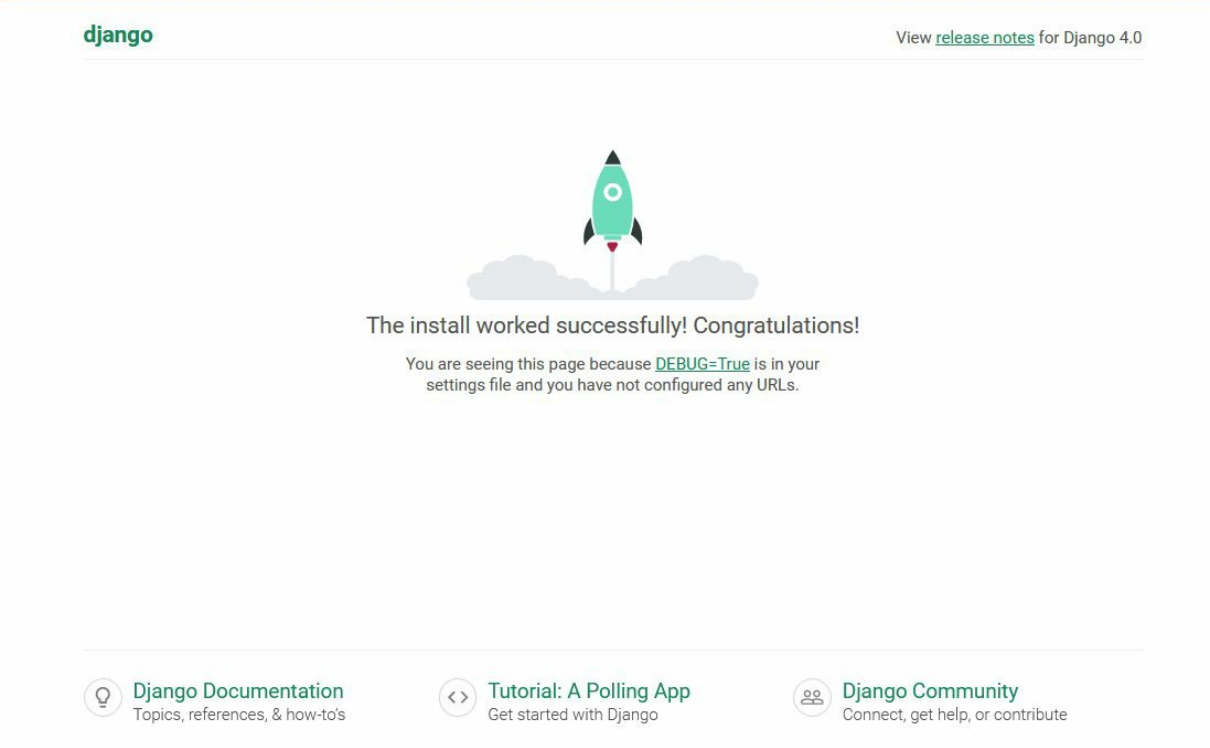
python manage.py runserver



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

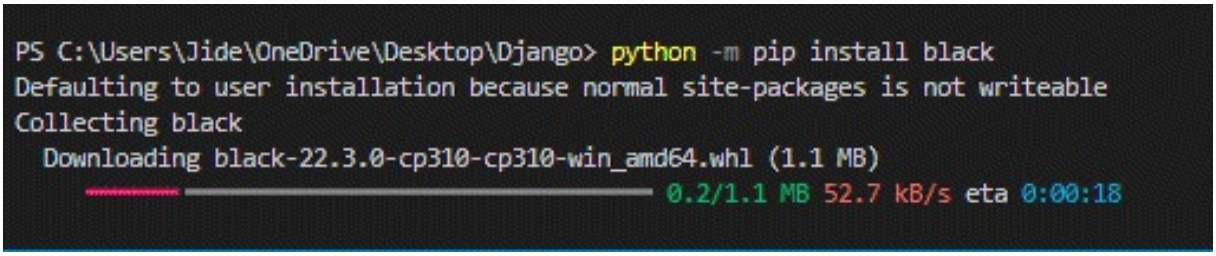
We will set up our Django project on VS Code or Visual Studio Code. If you don’t have that app, go to code.visualstudio.com and download the version for your machine. It is free and cross-platform. It also has a vast community of people or developers that build all sorts of great things for it.

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.

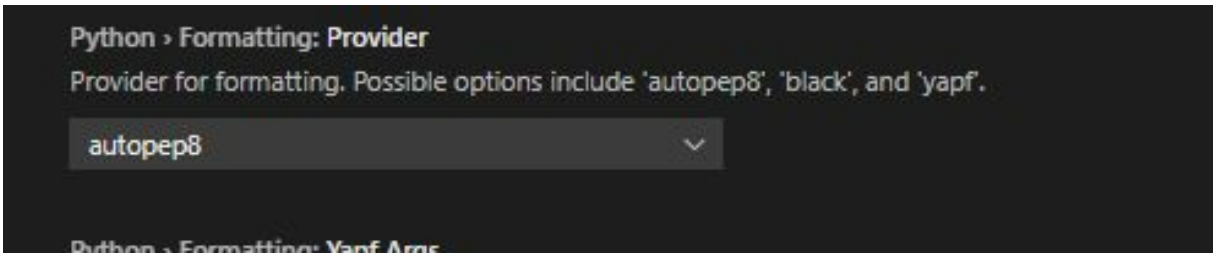
Go to the Extensions tab. Search Python and install the first result with the highest number of downloads. After that, you need to install Black. To do this, go to Terminal, and click on New Terminal. From there, type in the following command:

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.

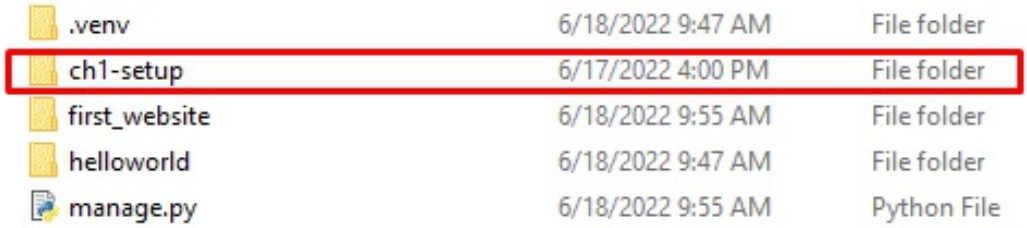
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

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

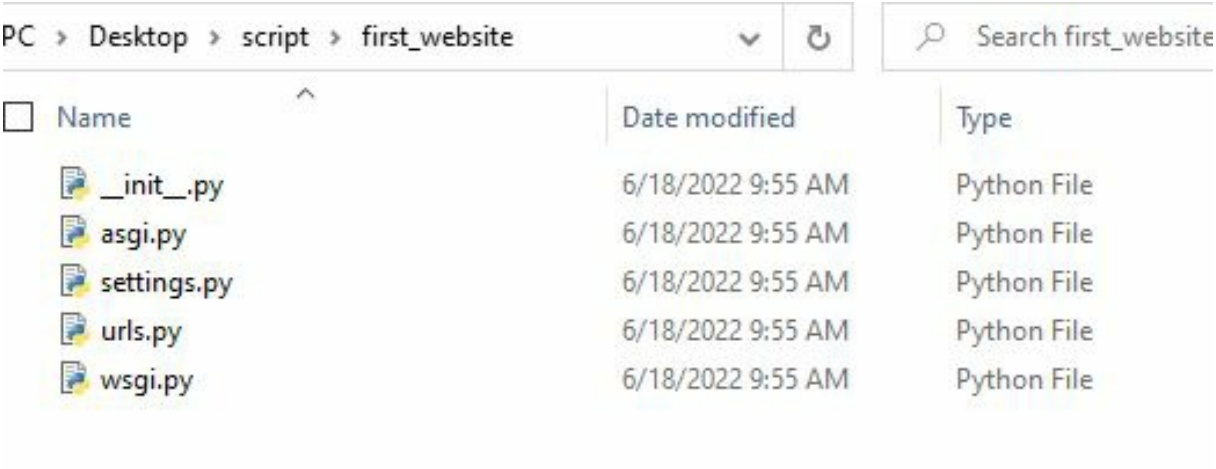
folder we have created in the previous chapter called scripts, and we made another folder within it called helloworld. Then we activated our virtual environment and installed the version of Django we wanted to use for this project.

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.

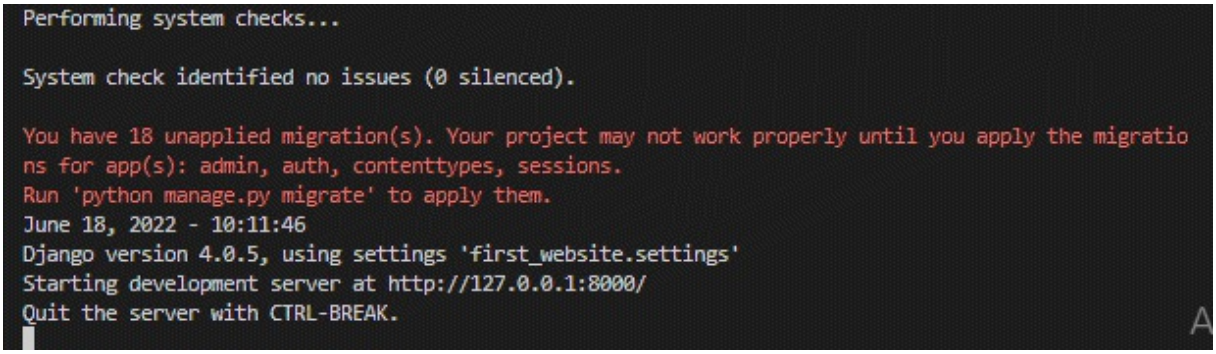
wsgi.py stands for Web Server Gateway Interface. WSGI helps Django serve our web pages.

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/



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:

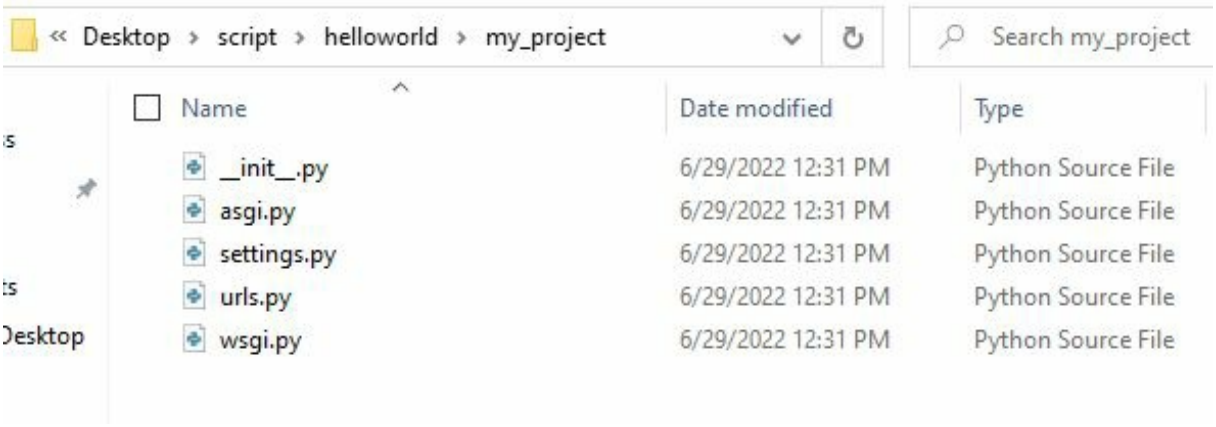
```
.venv\Scripts\Activate.ps1
```
OR

```
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:

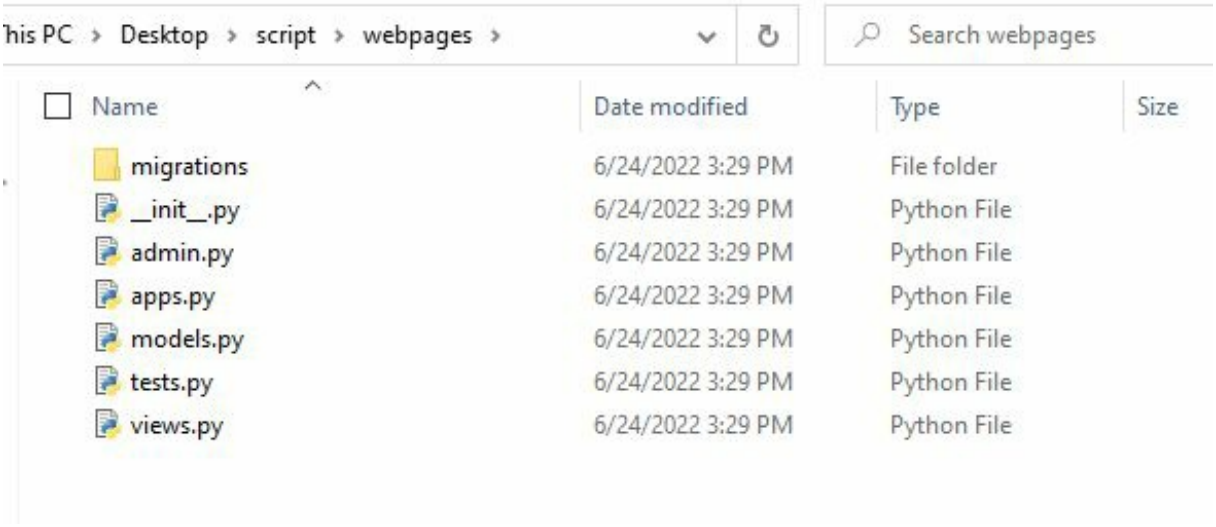
```
python manage.py migrate
```

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.

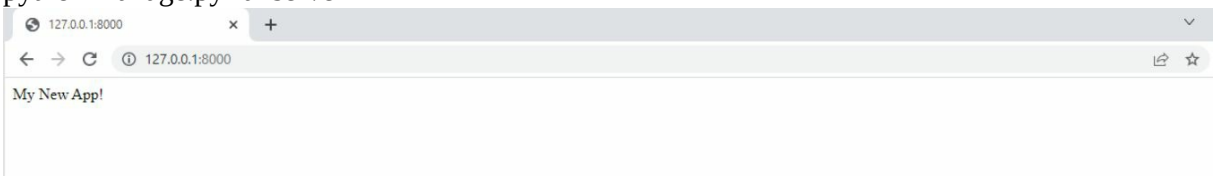
```
script > helloworld > my_project > urls.py > ...
"""my_project URL Configuration

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

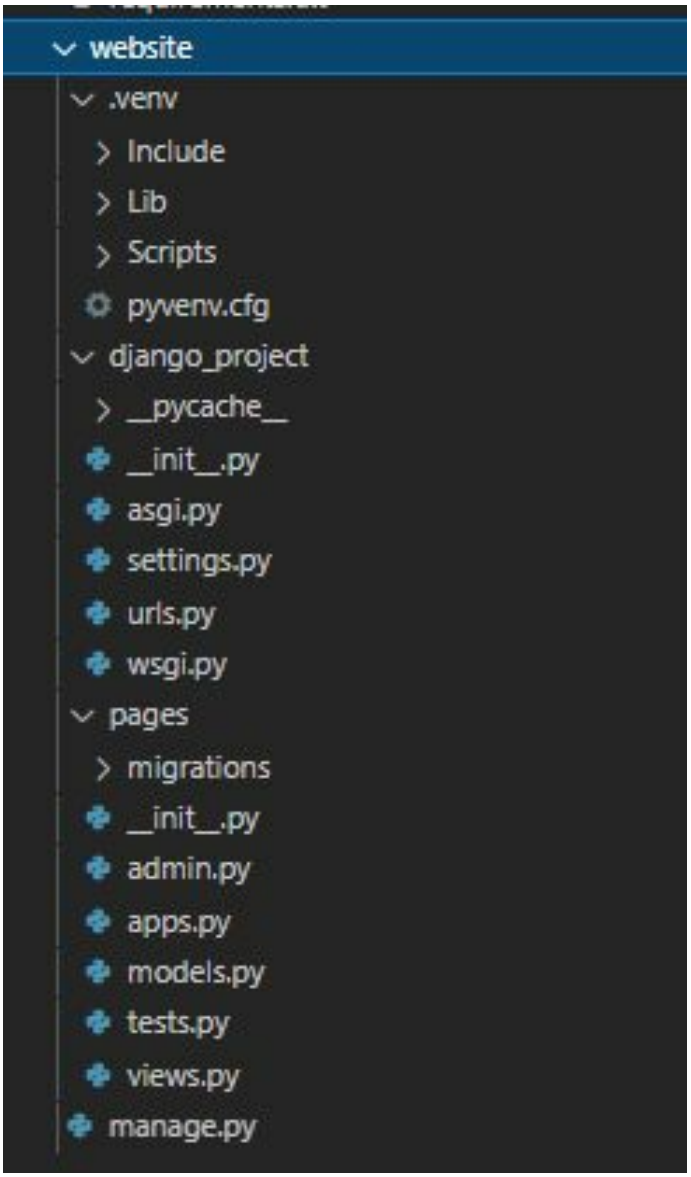
You have learned how to set up Django to create an application in chapter 2. Use the knowledge to

- 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
```

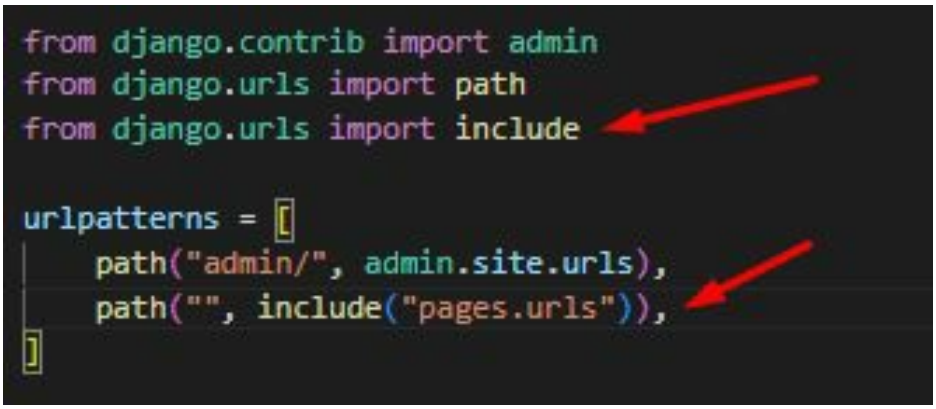
```
class HomePageView(TemplateView):
    template_name = "home.html"
```

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:

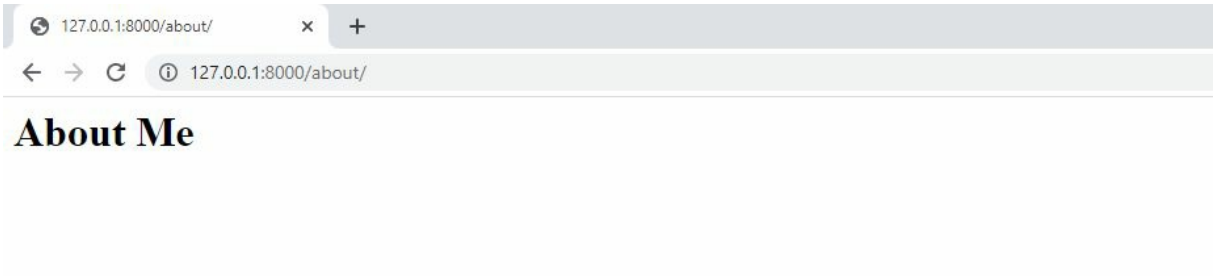
```
class AboutPageView(TemplateView):
    template_name = "about.html"
```

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"),

```
from .views import HomePageView, AboutPageView

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

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

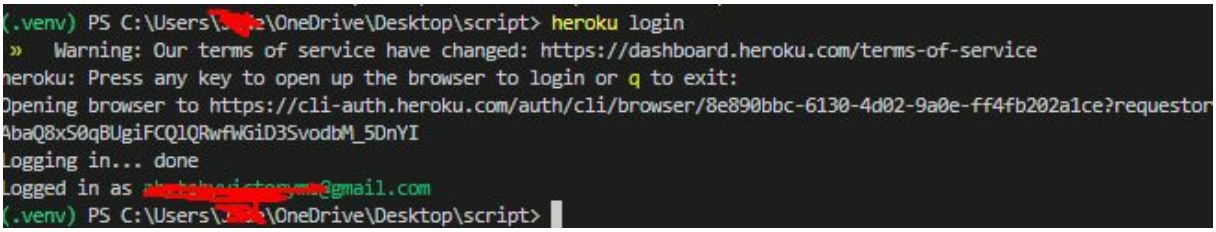
be available everywhere on our machine and not only in the virtual environment. So you can open a new command line terminal for this.

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
```

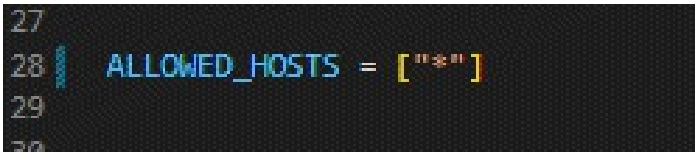
Step two is to make a file called "requirements.txt" that lists all the Python dependencies that our project needs. That is, all of the Python packages we have installed in our virtual environment right now. This is important in case a team member, or we ever want to start over with the repository. It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

We'll create a simple message board application in this chapter so users can post and read brief messages. We're going to use a database for the first time here. We'll examine Django's robust internal admin interface, which enables us to modify our data understandably. After adding tests and deploying the app to Heroku, we'll push our code to GitHub.

There is built-in support for MySQL, PostgreSQL, Oracle, MariaDB, and SQLite as database backends, because of Django's robust Object-Relational Mapper (ORM). The same Python code can be written by developers in a models.py file, and it will be automatically converted into the appropriate SQL for each database. Our settings.py file, located inside the django project folder, only needs the DATABASES section to be modified. This function is really fantastic!

Since SQLite is a file-based database, it is much simpler to use than the other database options, which require running a separate server in addition to Django. This is why Django uses SQLite by default for local development.

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

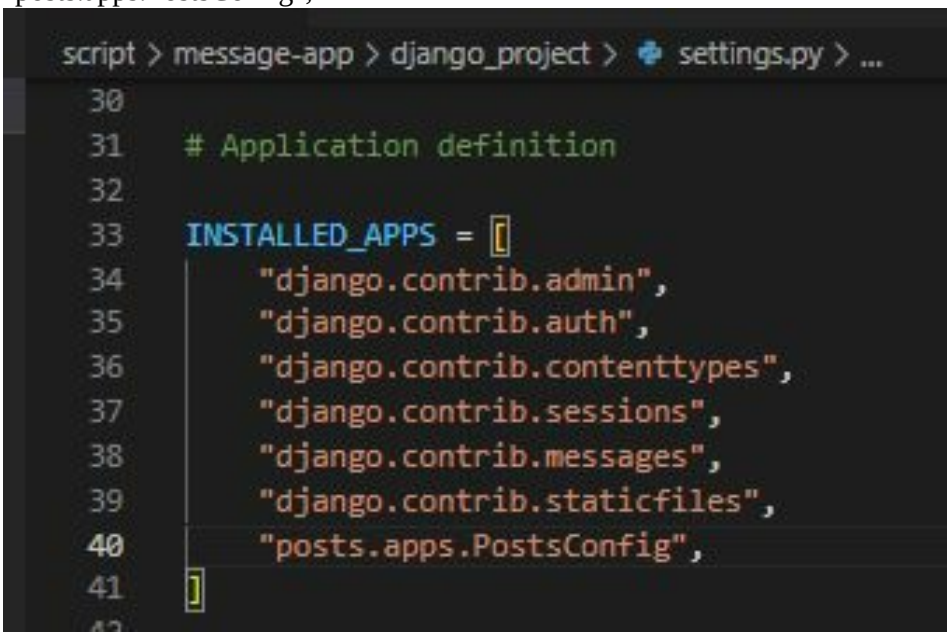
(.venv) > django-admin startproject django_project .

(.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
```

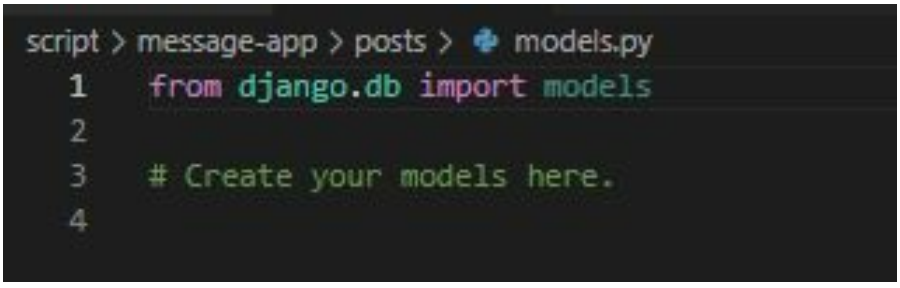
Now go to the local URL on your browser: http://127.0.0.1:8000/

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

The 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 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()
```

Remember 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.

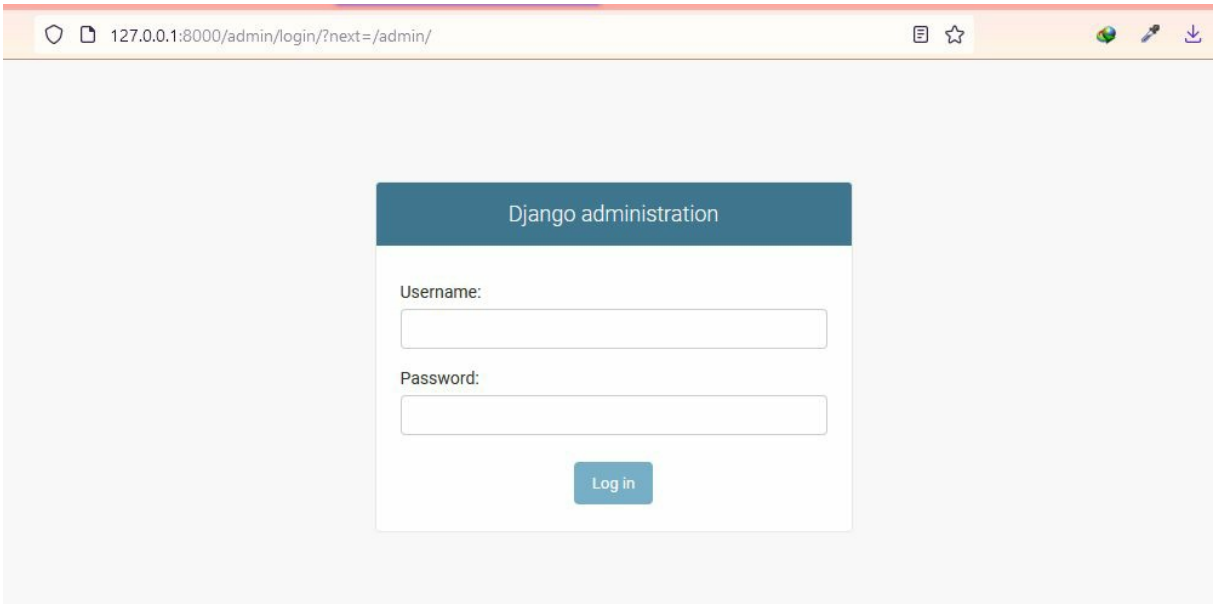
Django Admin

Django's robust admin interface, which allows users to visually interact with data, is a major selling point for the framework. This is partly because 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.

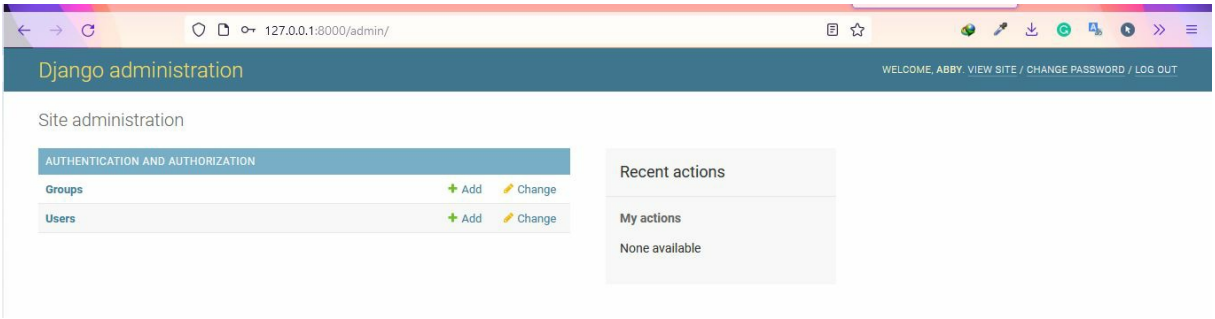
It is necessary to generate a superuser before accessing the Django admin. Type python manage.py createsuperuser into the command prompt and enter a username, email address, and password when prompted.

```
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:



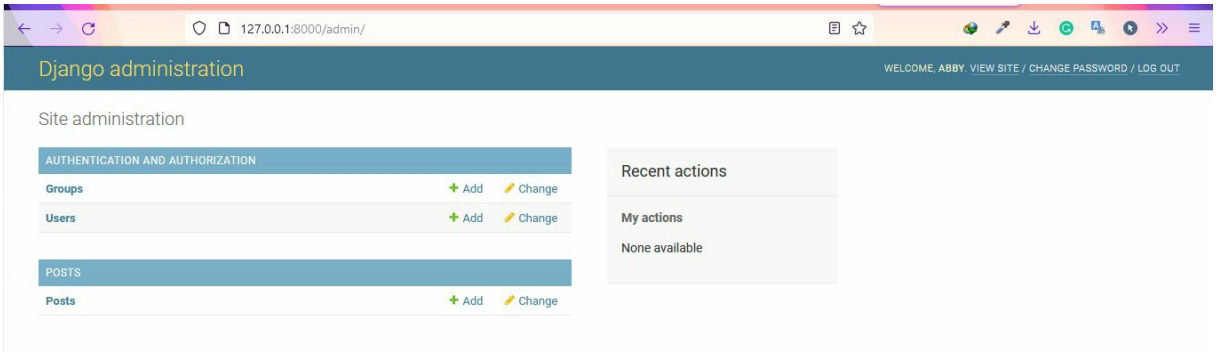
You can adjust the LANGUAGE in the settings.py file. It is set to American English, en-us, by default. You can access the admin, forms, and other default messages in a language other than English.

Our post app does not appear on any primary administration pages. Before this shows 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 for the app to be shown in the admin.

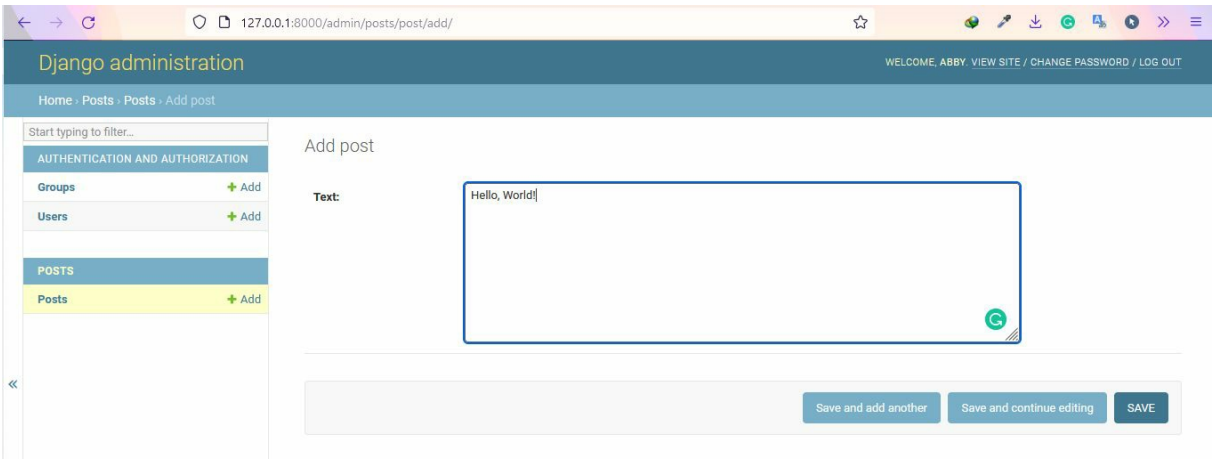
For the Post model to be visible, open admin.py in the posts folder in your preferred text editor and insert the following lines of code.

```
from .models import Post

admin.site.register(Post)
```

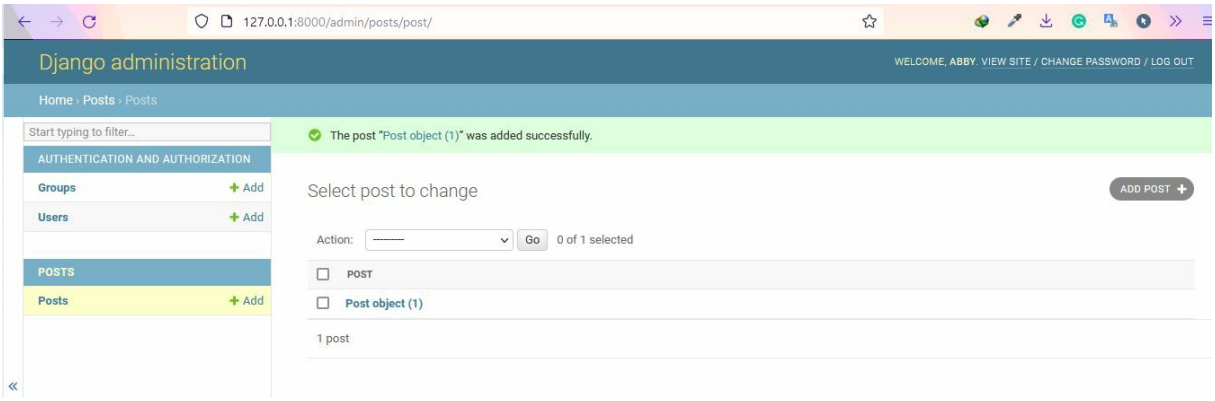
Now, run the server again and go to the page.



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.



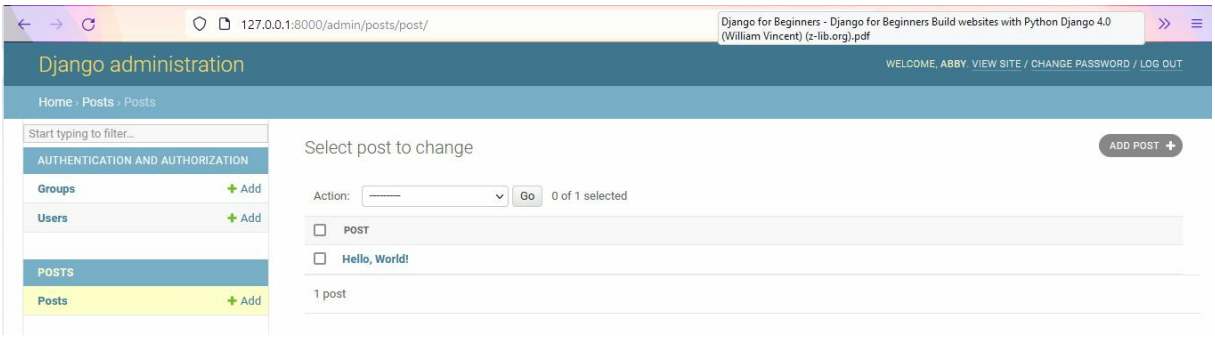
Then, click "Save." This will take you back to the main Post page. Yet, if you take a closer look, you'll notice that our new entry is titled "Post object (1)."



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

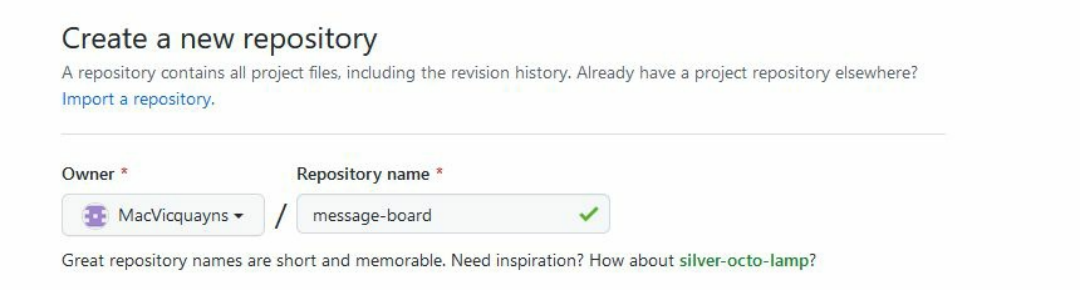
2 files changed, 20 insertions(+), 1 deletion(-)

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:

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

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
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 Ctrl + 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.

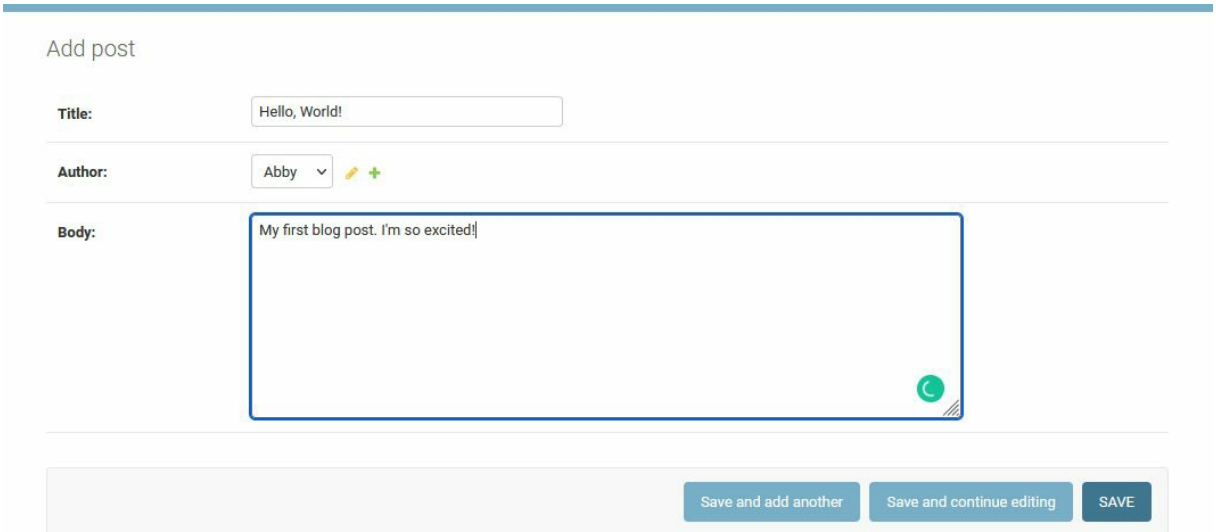
Type the following command and then follow the prompts to create a superuser account with a unique email address and password. For security reasons, your password will not display as you type it.

python manage.py createsuperuser

Now, we update the admin.py file.

```
from django.contrib import admin
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"],
```

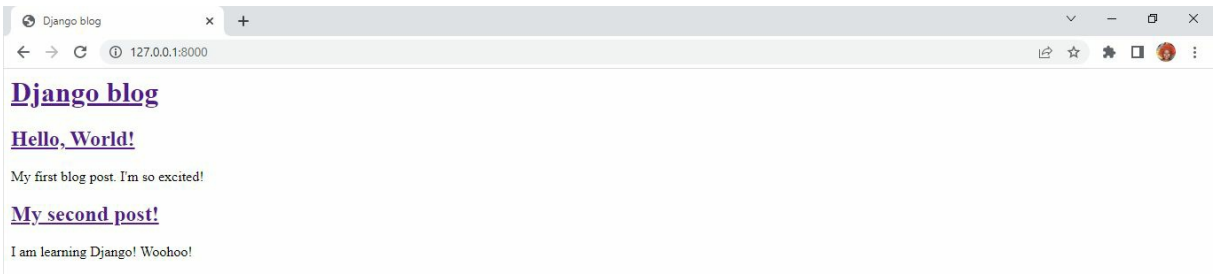
In the base.html file, put the following:

```
<!-- 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!

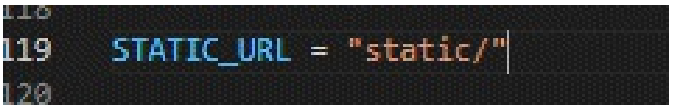
We need to add some CSS to our project to enhance the styling. A fundamental component of any contemporary web application is CSS, JavaScript, and pictures, which are referred to as "static files" in the Django ecosystem. Although Django offers enormous flexibility in terms of how these files are used,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

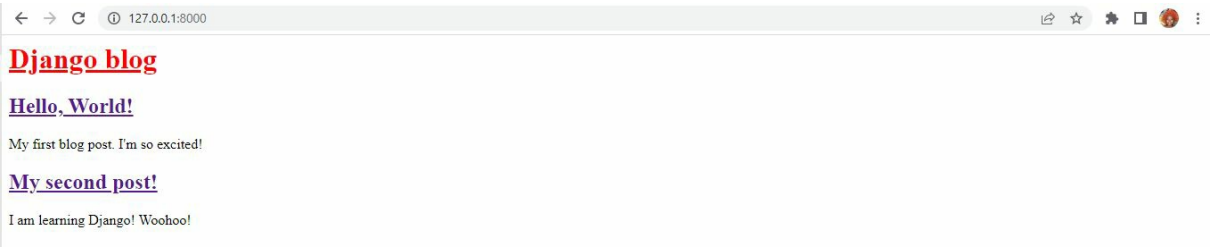
```
mkdir static/css
```

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"),
]
```

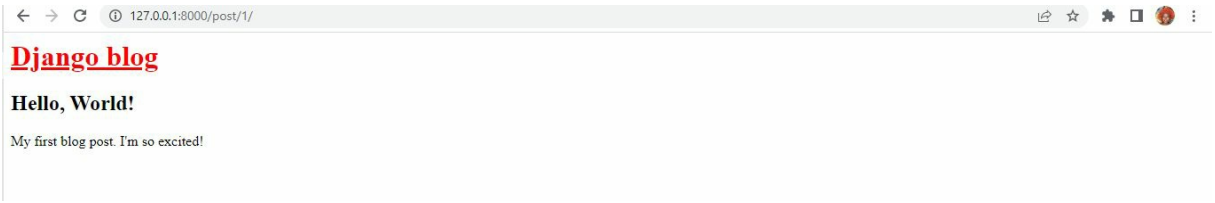
For consistency, we've decided to prefix all blog post URLs with post/. The next thing to consider is the post entry's primary key, which we'll express as an integer, int:pk>. I know what you're thinking: "What is the main factor?" Our database models already have an auto-incrementing primary key86 because 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:

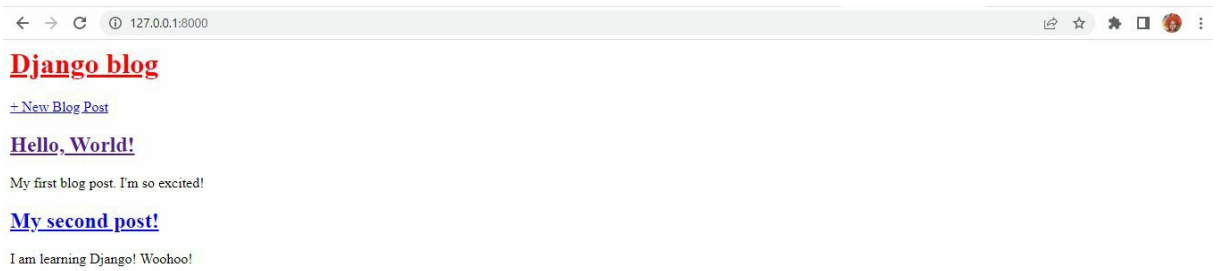
```
{% extends "base.html" %}
{% block content %}
<h1>New post</h1>
<form action="" method="post">{% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Save">
</form>
{% endblock content %}
```

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.

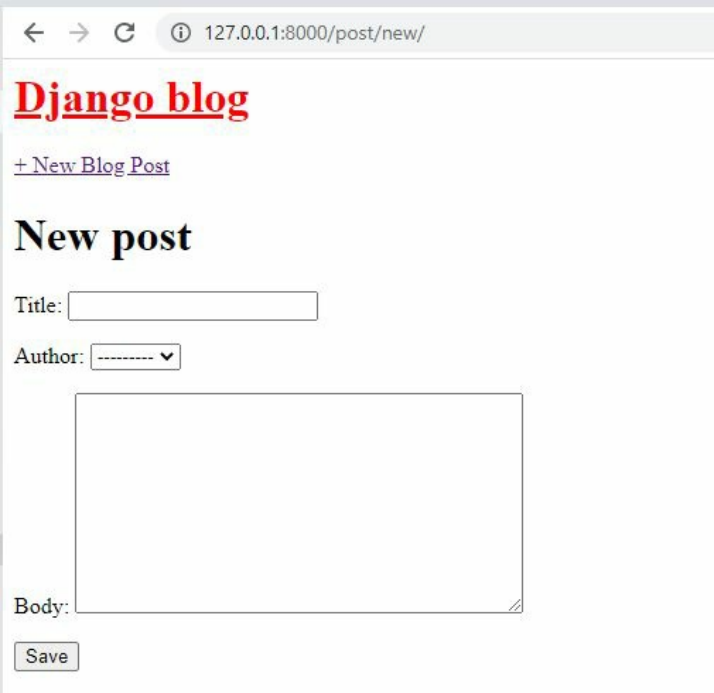
Lastly, set the value "Save" for a submit type input.

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.

```
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\blog>  python manage.py test
Found 7 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.......
----------------------------------------------------------------------
Ran 7 tests in 0.485s

OK
Destroying test database for alias 'default'...
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\blog>
```

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.

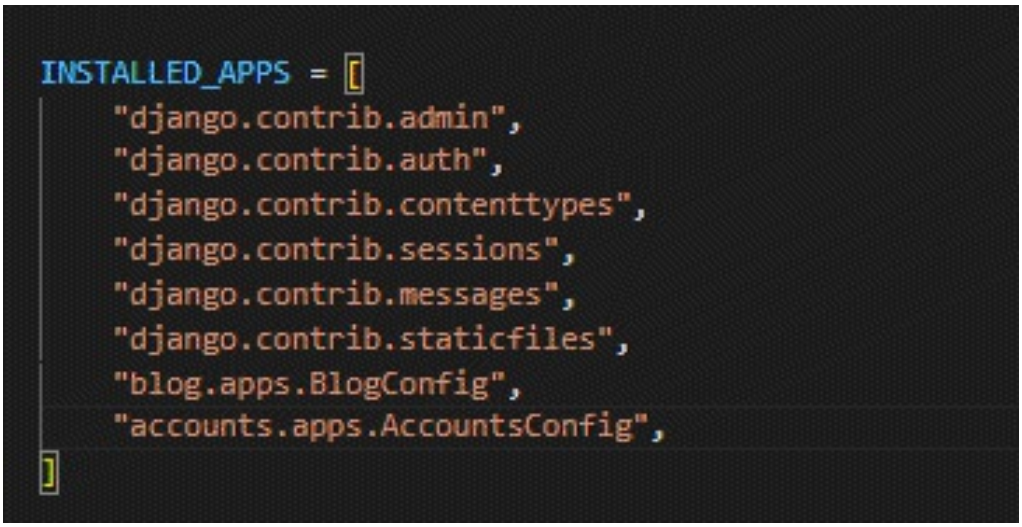
Allow Users to Sign Up

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.

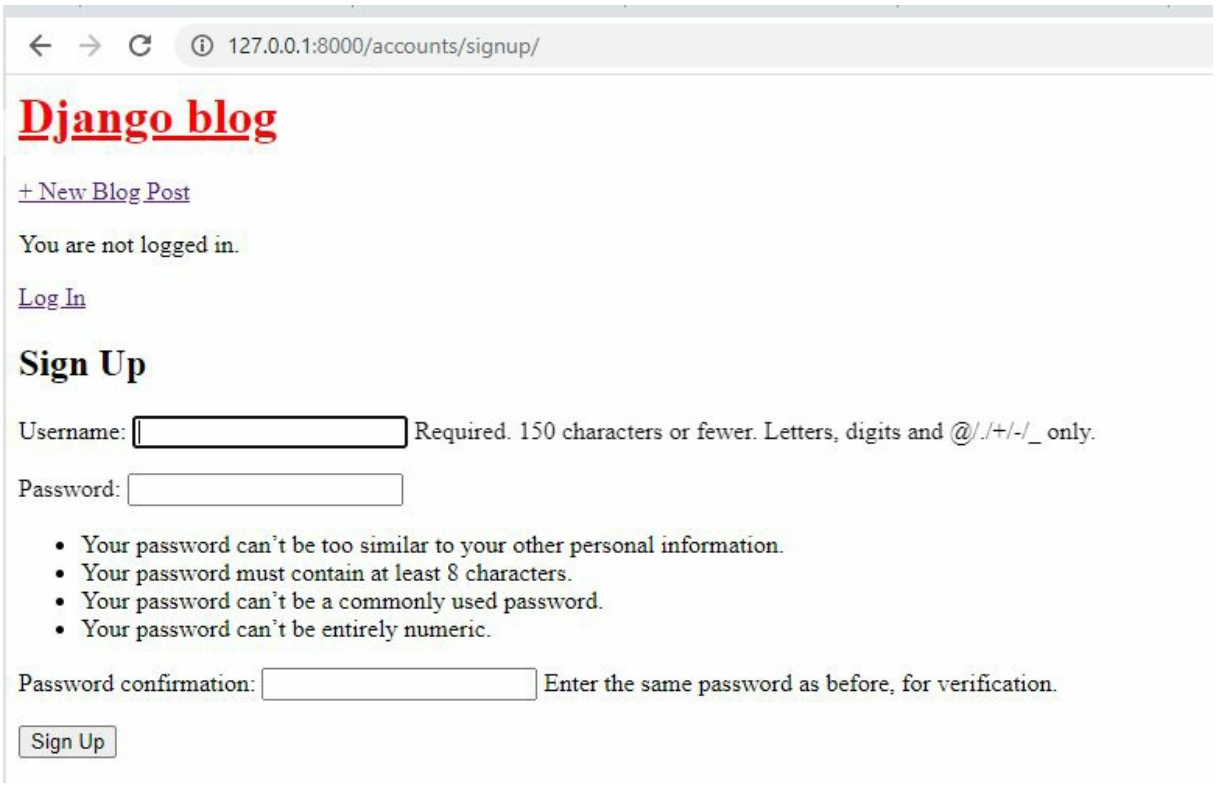
Why is reverse lazy used here rather than reverse? All generic class-based views don't load URLs when the file is imported. Therefore we use reverse's lazy form to load them afterward.

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.