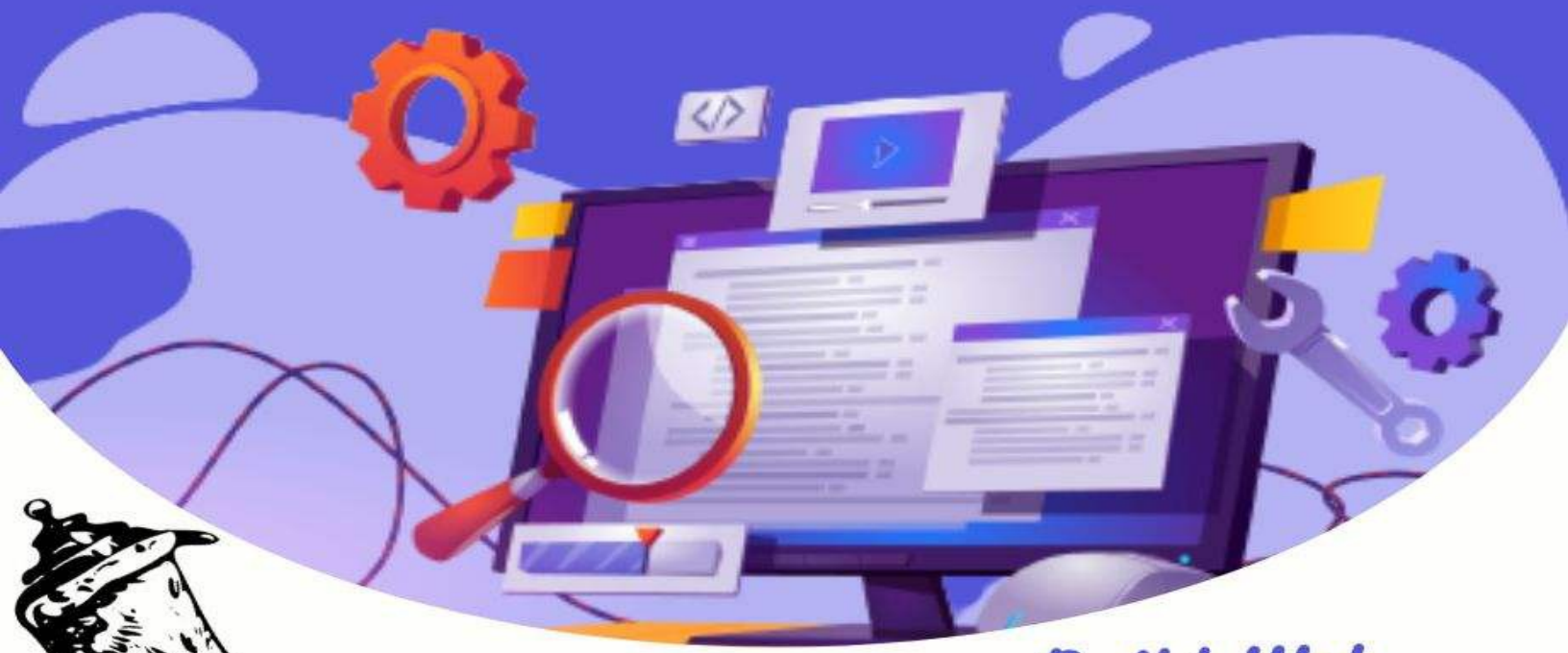
PYTHON

FLASK

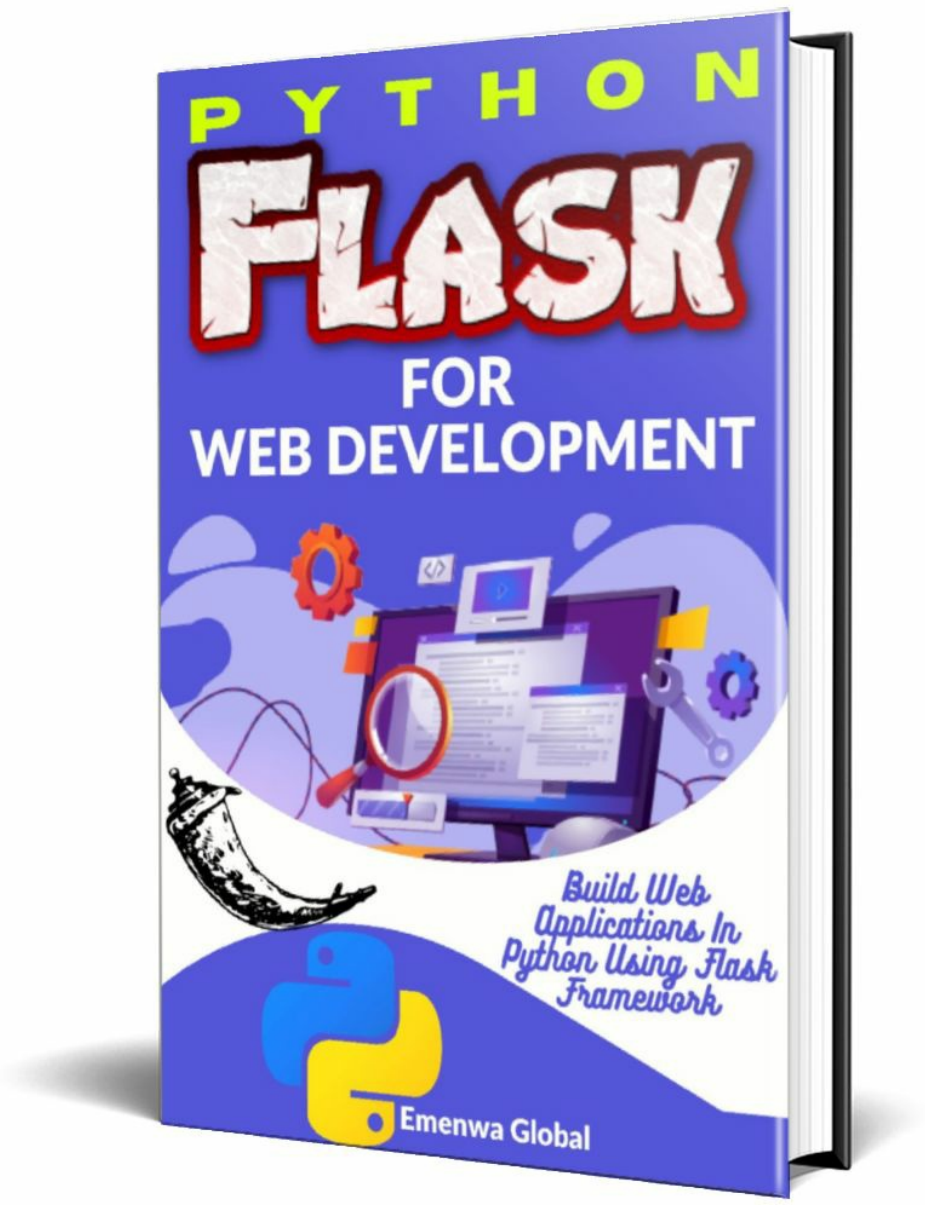
FOR WEB DEVELOPMENT



Build Web Applications In Python Using Flask Framework

Emenwa Global

Python Flask for Web Development



Build Web Applications in Python Using Flask Framework

www.emenwa.com

Contents

INTRODUCTION

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.

What is Flask?

Flask is a micro web framework written in Python. Python is very easy to learn, and there are lots of people that like it. Flask gives programmers a lot of freedom to write software how they want. It also has a lot of extensions that make adding new functionality easy.

Flask is a program that helps you build websites. Flask helps you by giving an easy way to create new features, like spreadsheets or games. Flask allows you to do whatever you want, so you can make whatever you want. What else? Flask is free!

Flask is a web framework that makes it easy to create new websites. It was created by a person called Armin Ronacher. Flask is similar to Django, which

is another framework. Like Django, Flask helps develop websites with forms and queries because it allows you to use programming code (Python) tools to change HTML pages into the format you want.

Python is a language with many modules or frameworks that let you use Python to build your website. But Flask and Django are the ones that people like the most. Flask gives you more options for web programming.

To grasp what Flask is, you must be familiar with a few basic terms.

The Web Server Gateway Interface (WSGI), pleasantly pronounced "whiskey," has emerged as the industry standard for creating Python web applications. The WSGI standard creates a recognizable interface for web servers and online applications.

Werkzeug. Requests, response objects, and other valuable features are implemented by this WSGI toolkit. This enables it to be built upon a web framework. One of the foundations upon which Flask is constructed is Werkzeug.

jinja2. Many users use the Python templating engine jinja2. A web templating system generates dynamic web pages using a template and a specific data source.

Python is the language used to create Flask. The Werkzeug WSGI toolkit and Jinja2 template engine serve as the basis for Flask. Pocco created both of them.

Why Do Most People Use Flask?

Scalable

Flask can quickly help you grow a tech project like a web app as a micro-framework. If you want to make a small app that can grow rapidly and in ways you haven’t thought of, it’s a good option. It is easy to use and has few dependencies, so it works well even as it gets bigger and bigger.

Flexible

This is Flask’s main benefit. One of the Zen of Python principles states that

simplicity is better than complexity because it can be easily rearranged and relocated.

This allows your project to quickly change direction and prevents the structure from collapsing when a part is altered. Flask is even more flexible than Django itself because it is simple and can be used to build smaller web apps.

Easy to Navigate

As with Django, navigating easily allows web developers to focus on coding quickly without feeling overwhelmed. The microframework saves web developers time and effort and gives them more control over their code and what's possible.

Lightweight

When we use this term to describe a tool or framework, we're talking about its design—it has a few parts that need to be assembled and reassembled. It doesn't rely on many extensions to function. Web developers have some control over this design.

A developer can split a single Flask app into different modules. Each module works like a separate building block that can do its part of the job. All of this means that the structure's parts are flexible, moveable, and can be tested on their own.

Documentation

Flask users will find many examples and tips arranged in a structured manner, per the creator's theory. Developers are quickly introduced to the framework's features and capabilities, which encourages their use.

Why Do Some Hate Flask?

Few resources

Flask doesn't have an extensive library of tools like Django. Developers have to add extensions like libraries manually. Funnily enough, if you add many extensions, the program may slow down from too many queries.

Large Flask apps are usually complex

Because Flask web app development can take several turns, a web developer

entering midway through the project may fail to understand how it's been developed. Programmers may struggle with the microframework's modular architecture because they must learn all of its components.

Maintenance

Flask is diverse in terms of technologies. However, the company will have to update and implement if a Flask app technology becomes obsolete or withdrawn. Complex apps have higher maintenance and implementation costs.

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.

Let’s start coding!

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

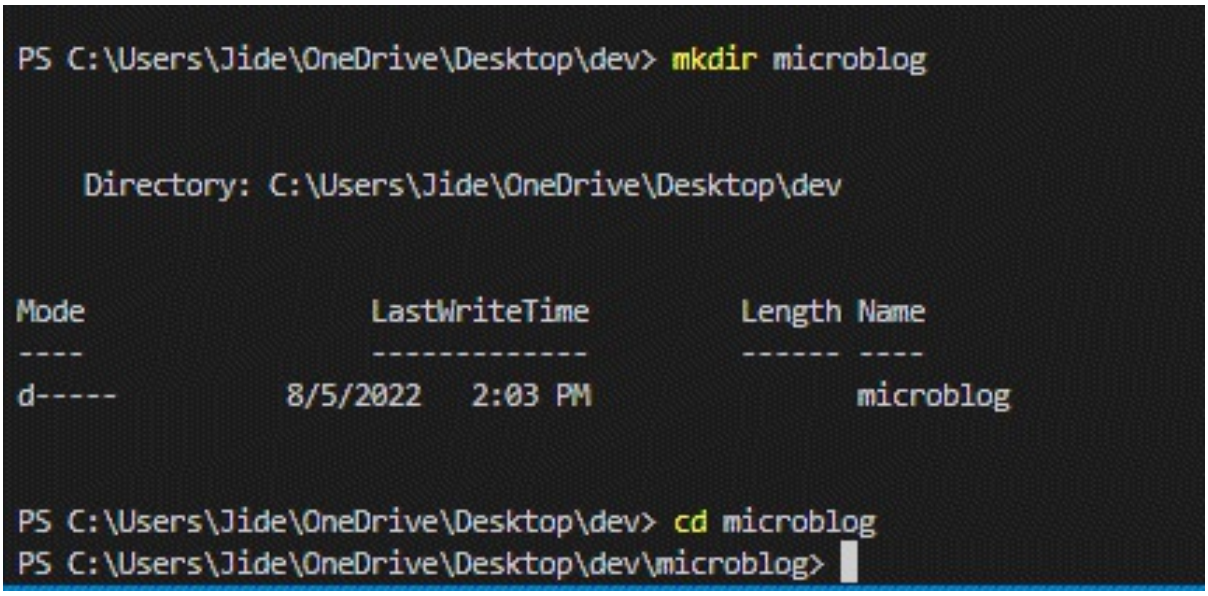
I assume you are new to Flask, but you can use Python. Otherwise, I highly recommend starting your journey by learning Python basics.

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.



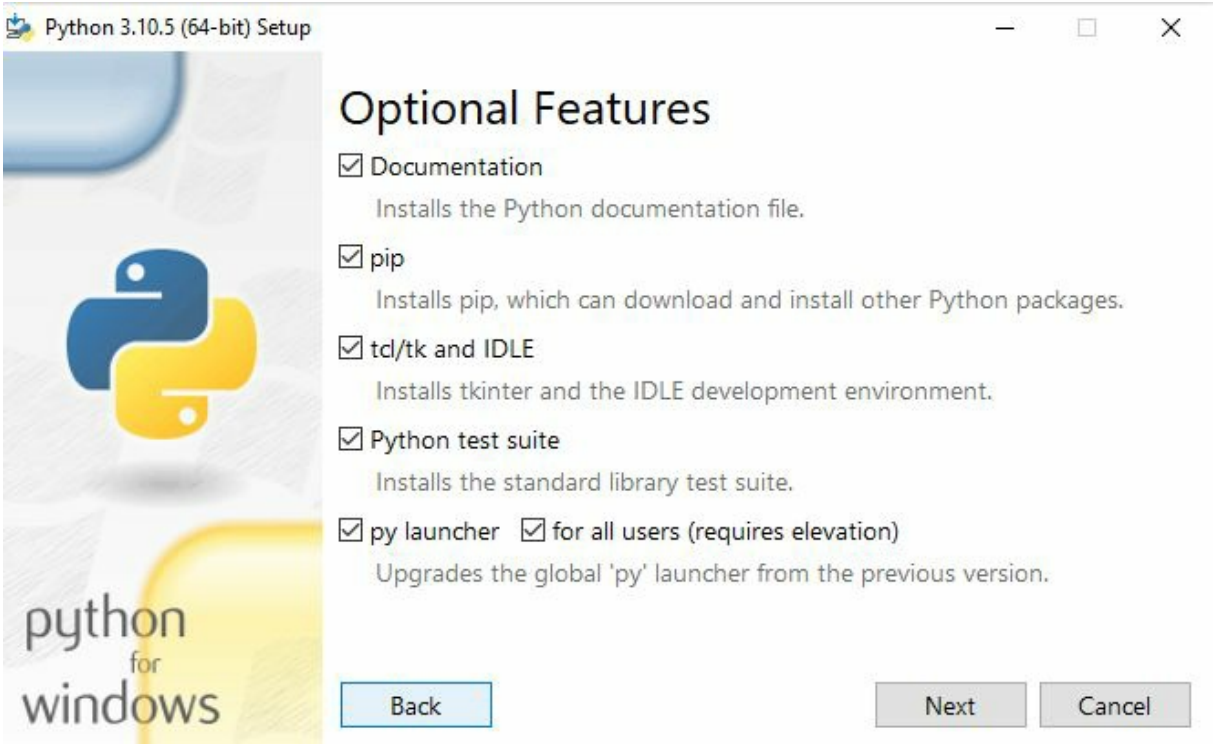
Installing Python

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.

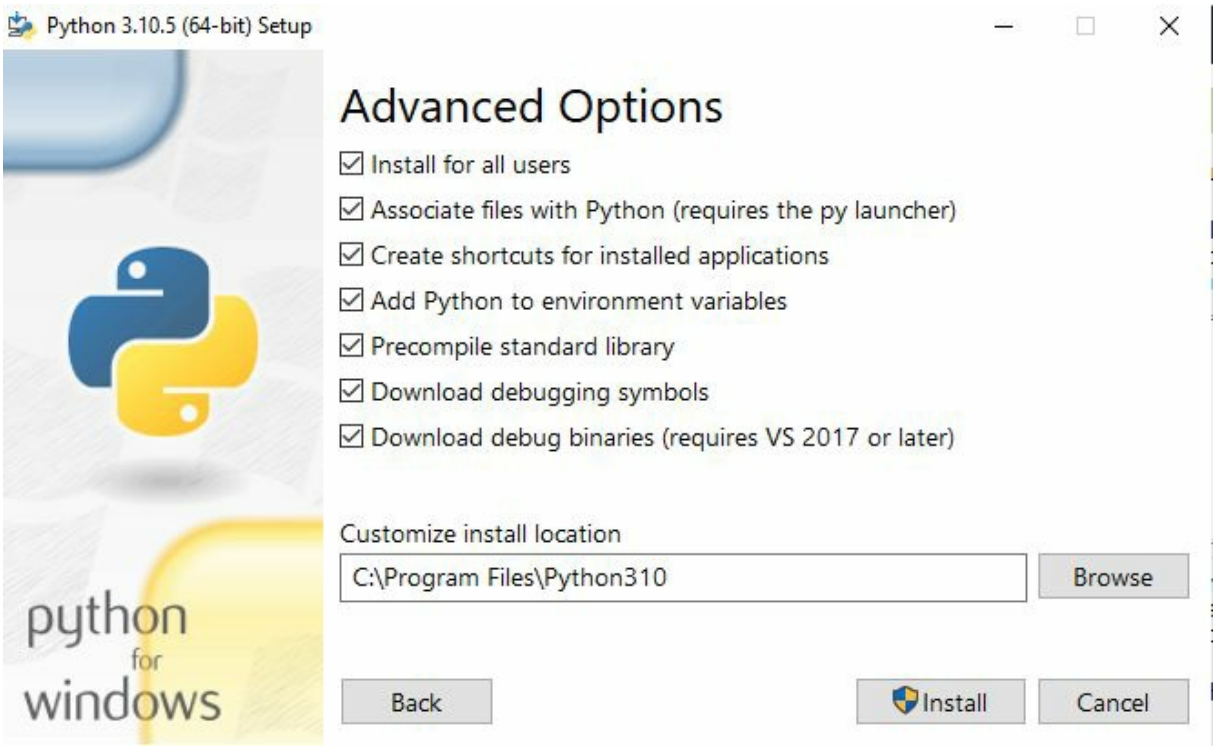
Installing Python

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.

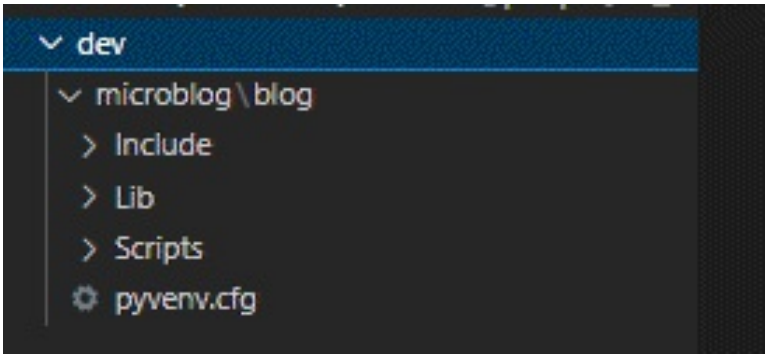
Inside the microblog directory, you will now make a virtual environment. Most people call virtual environments "venv," but you can call them something else if you'd like. Make sure that microblog is your current directory, and then run this command:

```
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.

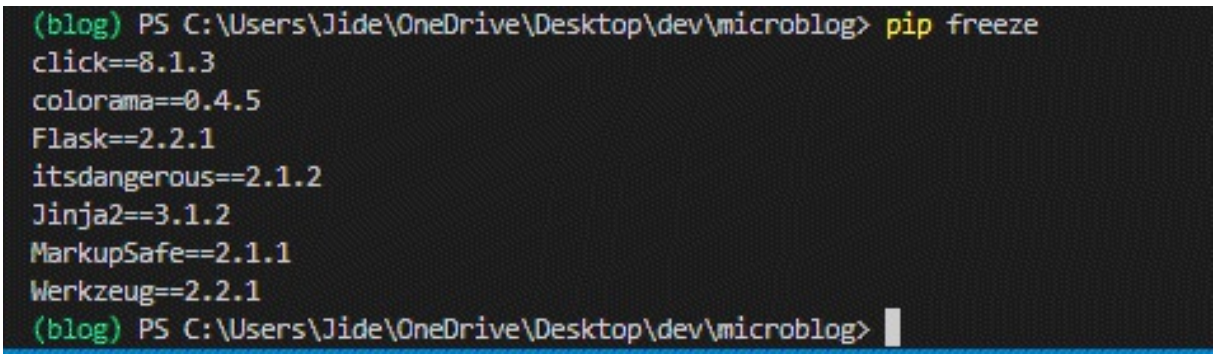
Installing Python Packages

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:

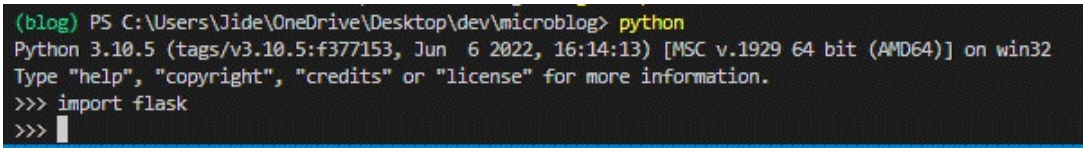


Type deactivate at the command prompt to return your Terminal's PATH environment variable and the command prompt to their default states once you've finished working in the virtual environment.

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

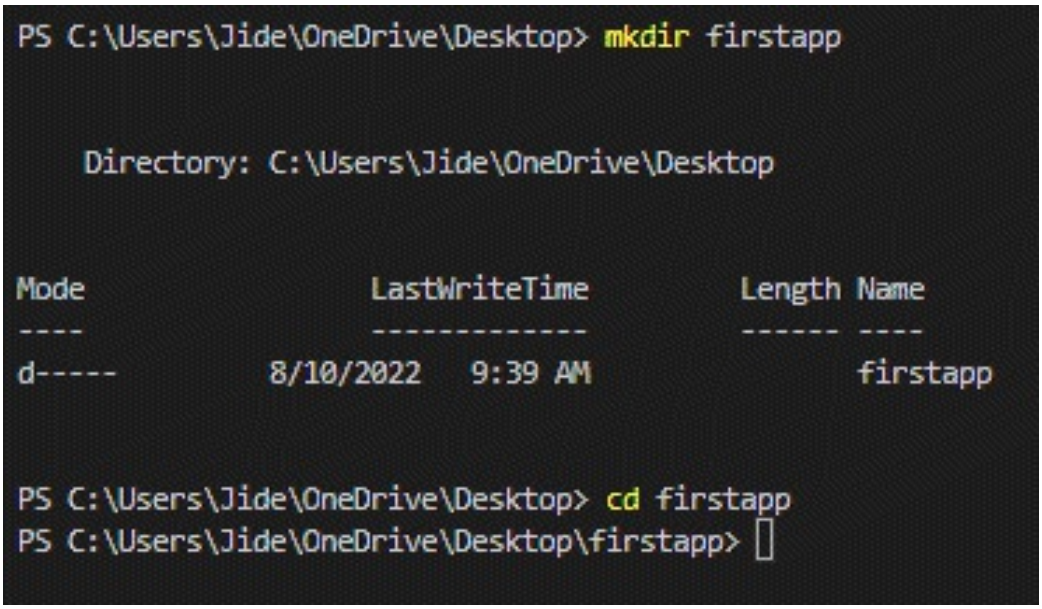
@app.route('hello/<converter: variable_name>')

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.

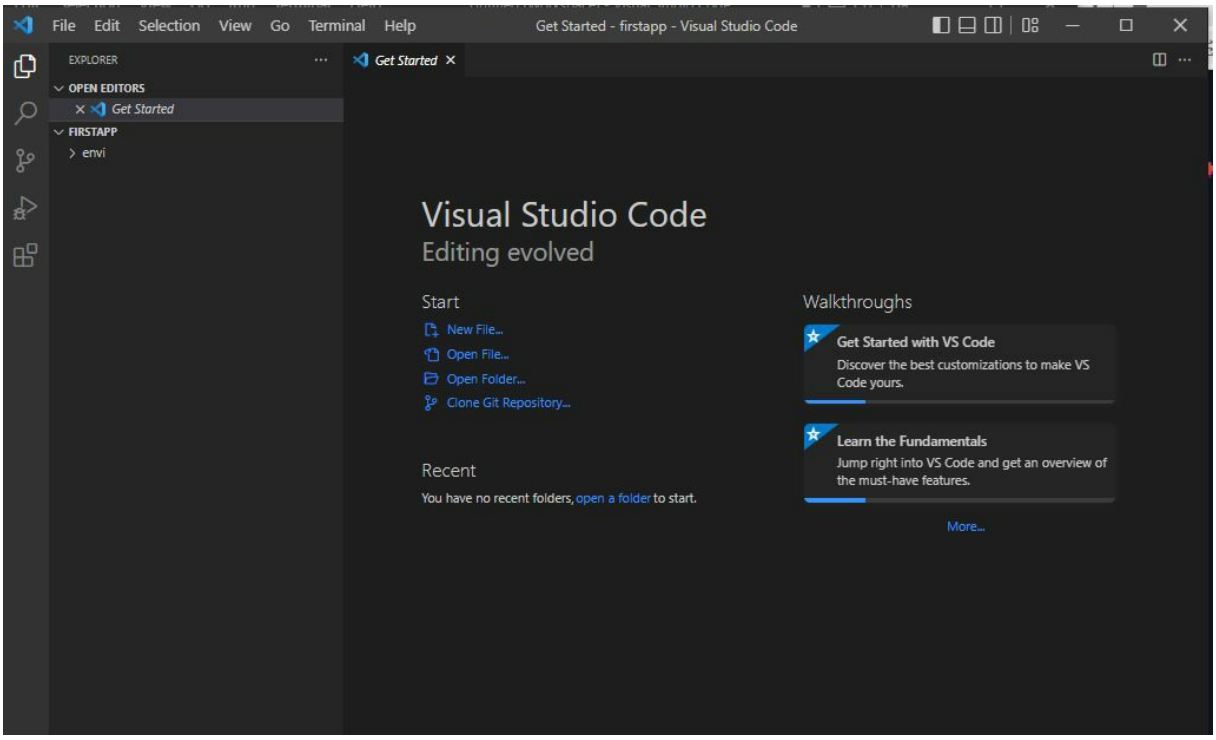
The first step is to create a project folder. Mine is firstapp. Name yours whatever.



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

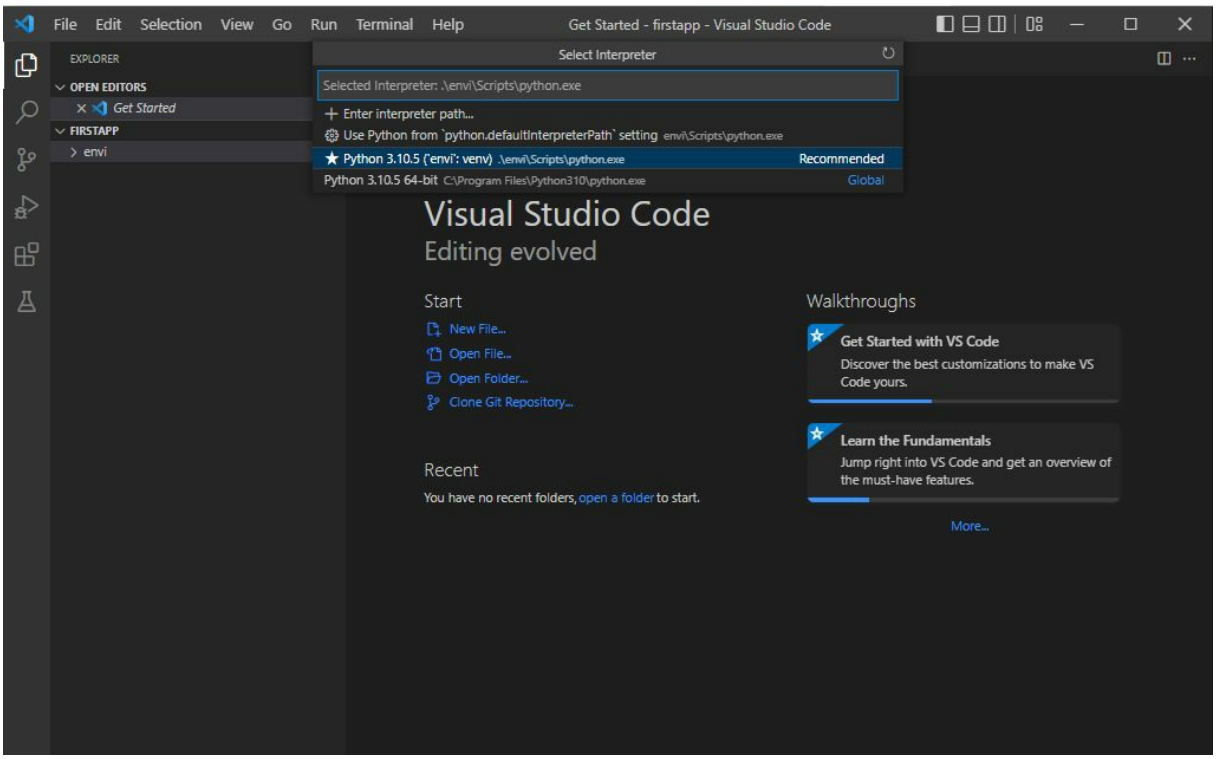
python -m venv envi

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:

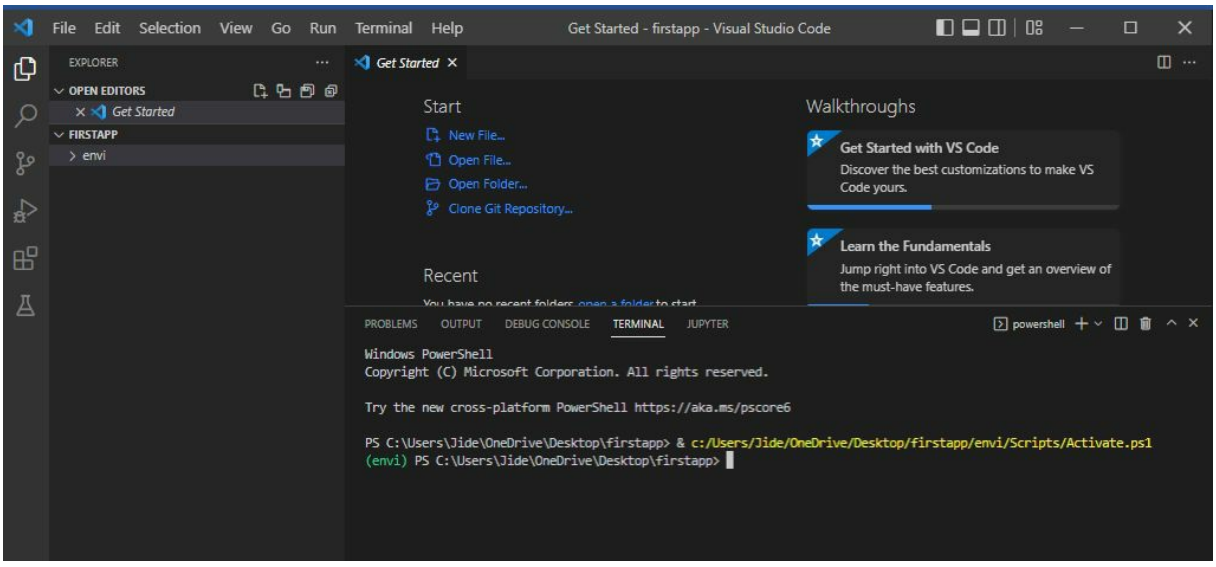


Next, open the Command Palette. Go to View and click on Command Palette (or press Ctrl+Shift+P). Select Python: Select the Interpreter command.

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.

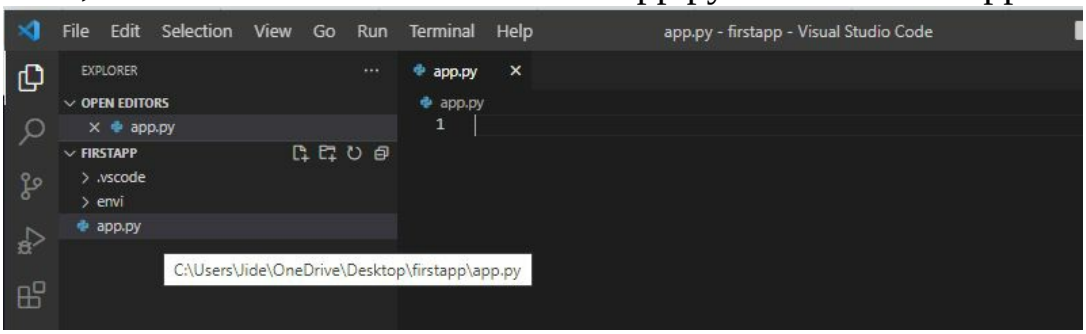
Now that the Virtual environment is active, install Flask in the virtual environment by running pip install flask in the Terminal.

```
(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)
```

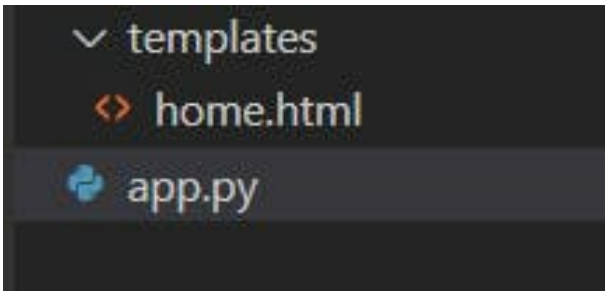
In this case, we would already have a template saved, perhaps as home.html or index.html, with the HTML code in it. When you run the app, Flask will run all the HTML codes included in the script, and the view will display them on the web browser.

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

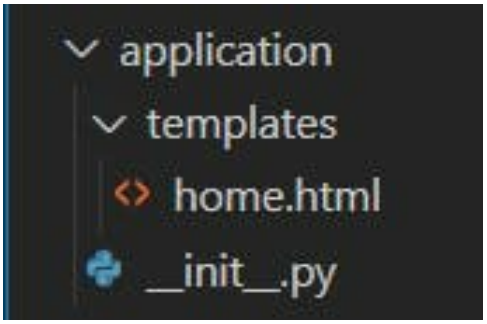
This is a very simple and straightforward structure where all the application logic is in a single .py file. The templates folder will be the same folder as the .py file where the developer keeps the HTML files.



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:

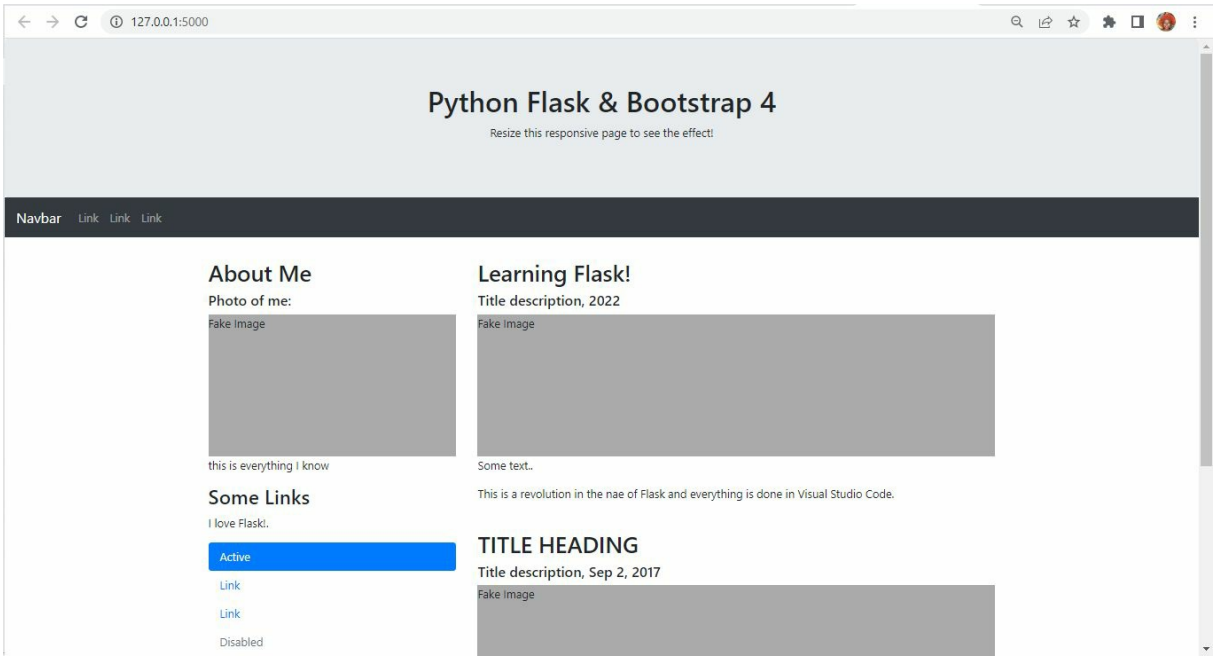
```
from flask import Flask, render_template

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.

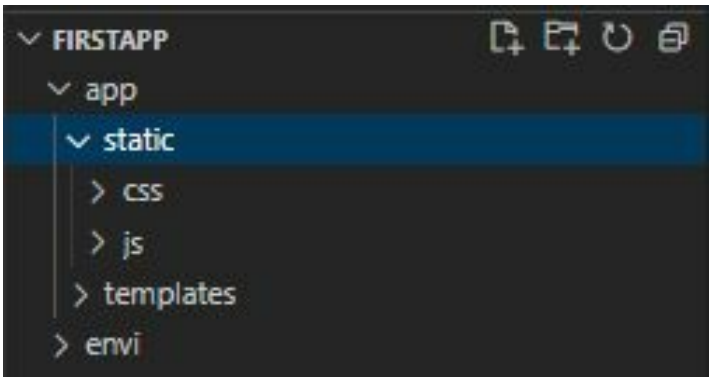
We'll have a separate home page, the only page that doesn't "extend" any header stuff like most pages. Bootstrap takes care of almost all the graphic stuff for you, which is great. You only have to decide where things go; the rest is styled for you. It really does help a lot, too.

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.

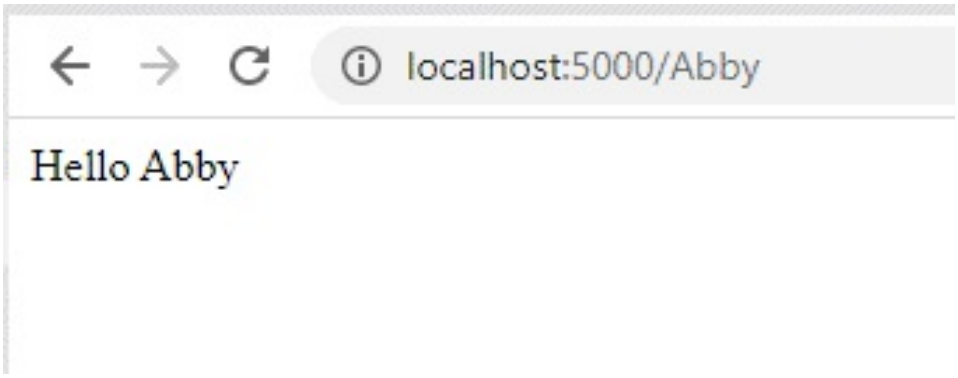
Web App

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.

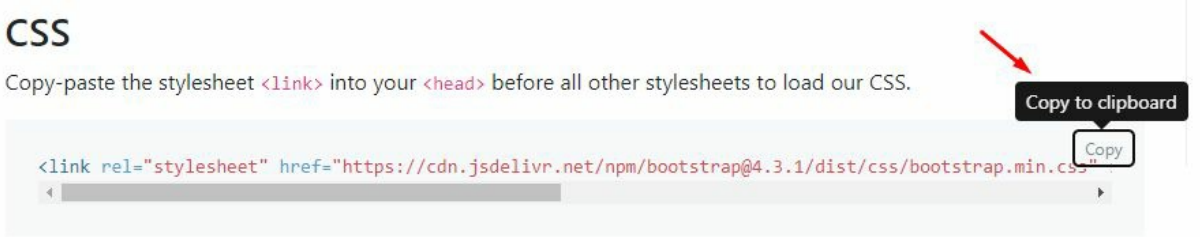
Adding Bootstrap

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:

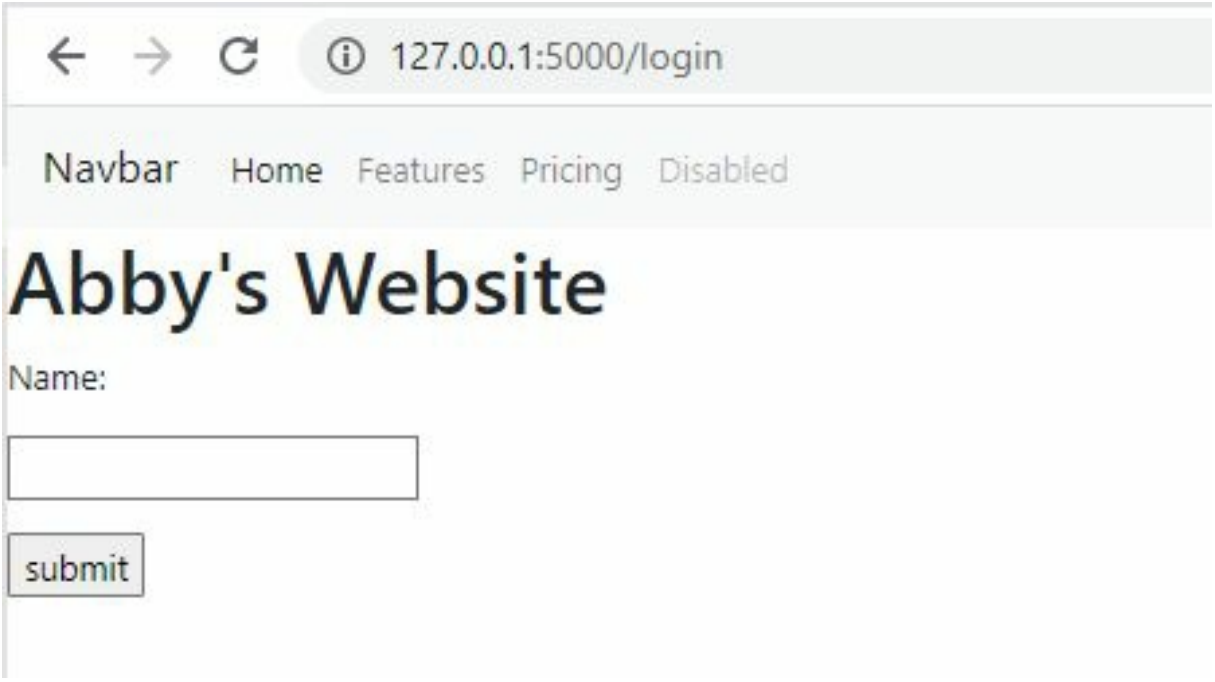
```
from flask import Flask, redirect, url_for, render_template, request
```

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.

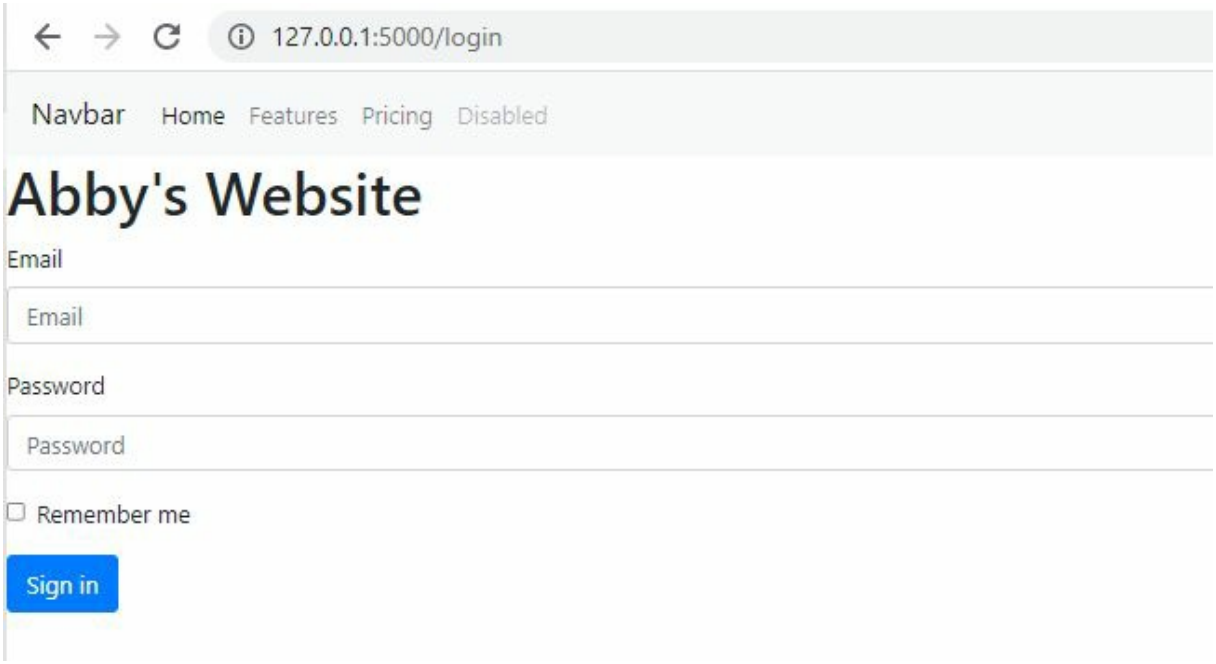
That is how we actually get information from a form, and obviously, if you have more than one info you want, you just add it to the input type in the login.html script. Then you can get all those information by just using the name as a dictionary key on the request.form.

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.

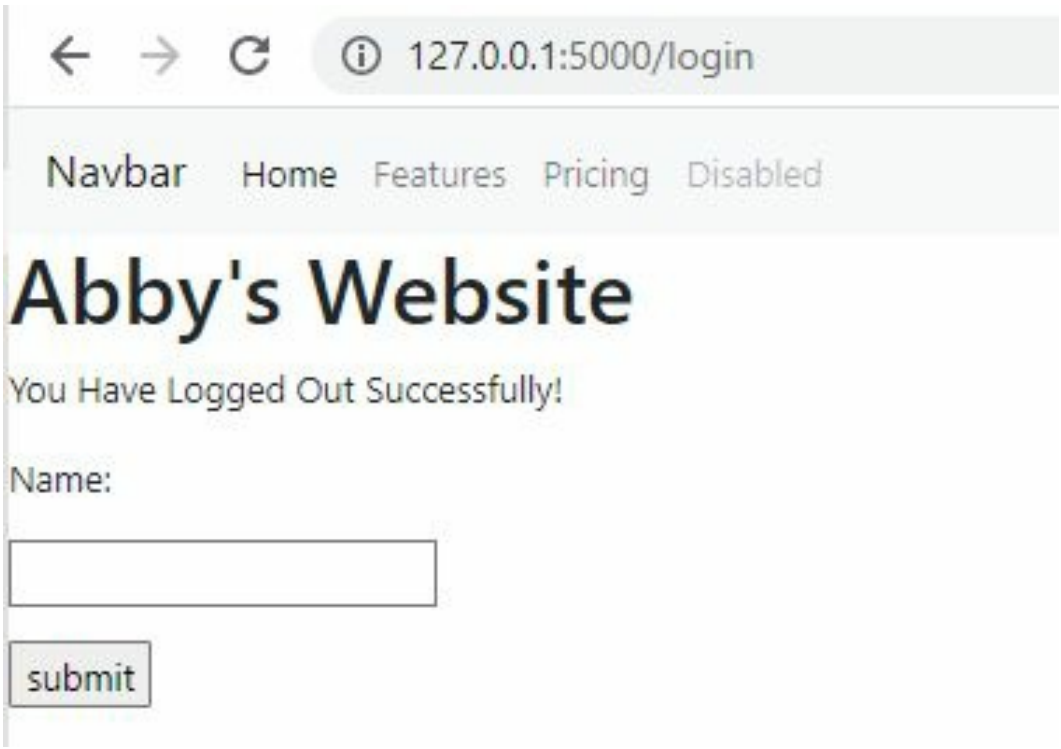
Displaying Flash Message

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.

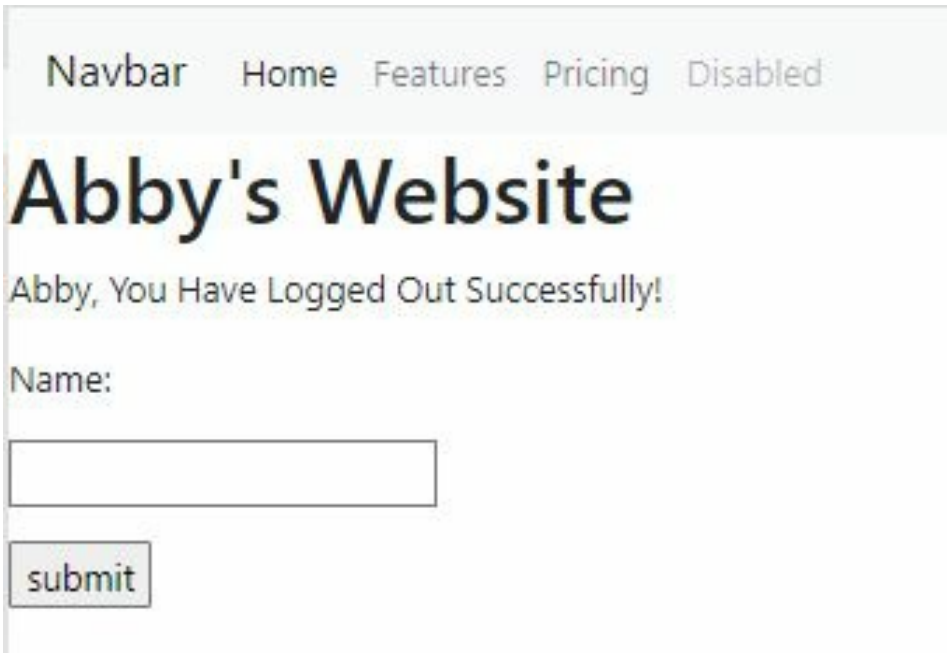
The problem we have now is that this message pops out whenever you type /log out, even if you had not been logged in before. In that case, we could check if we have a user in the session and only if we do will we say you've been logged out.

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.

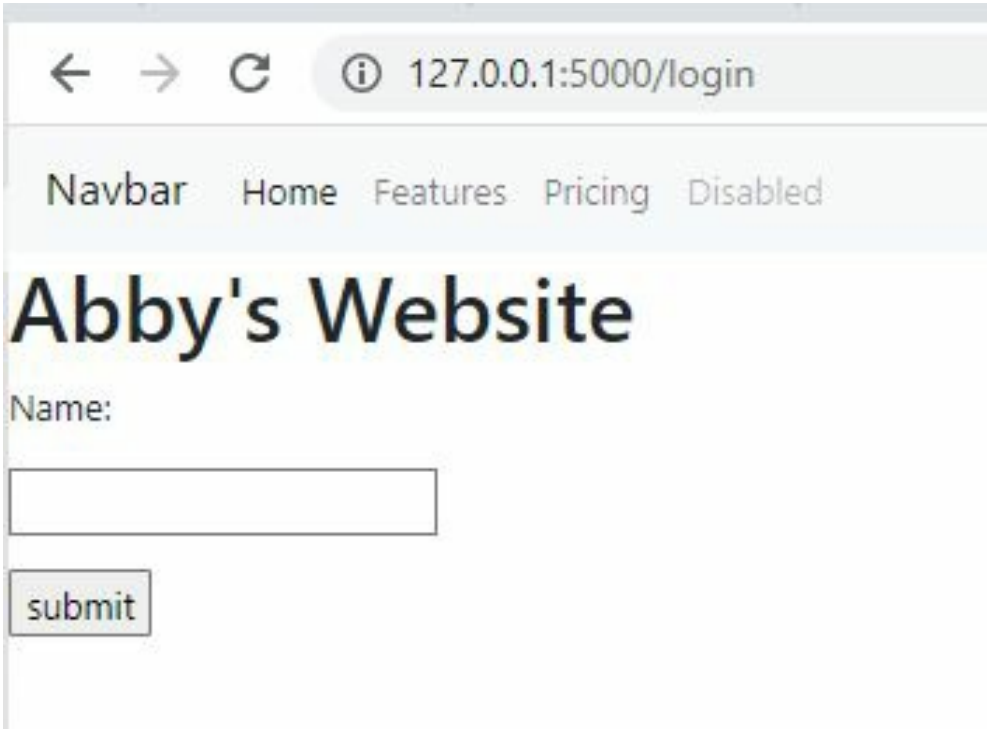
```
from flask import Flask, redirect, url_for, render_template, request, session, flash
from datetime import timedelta

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 %}
```

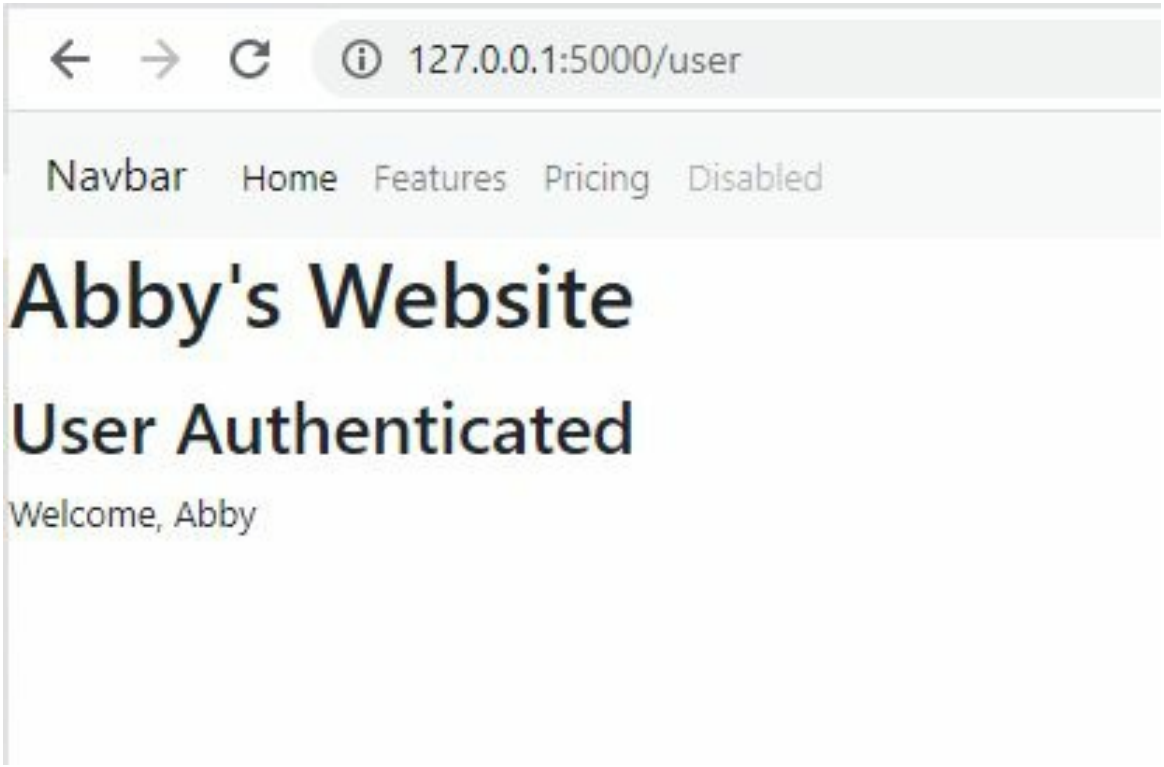
Now, we go to the user function in the app.py file and render the new template.

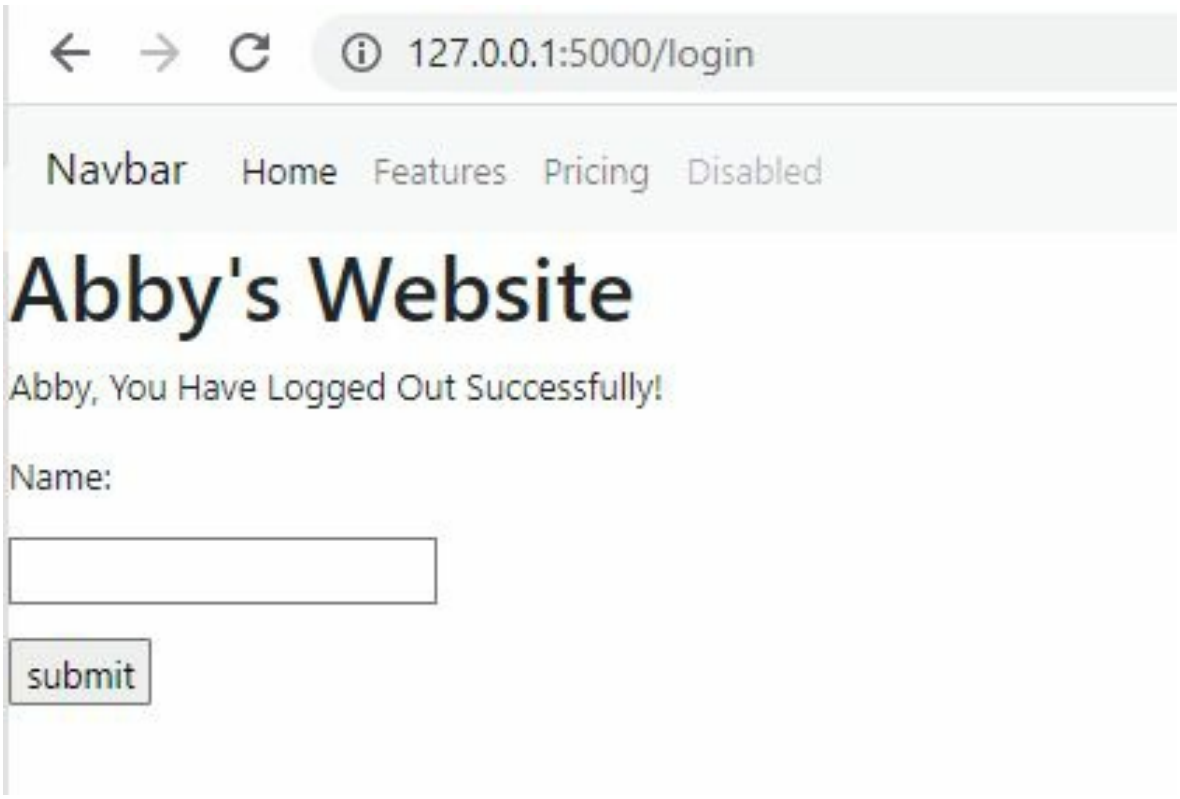
We can also flash a new message after running the log-in function. We could also flash “You are not logged in” when the person tries to enter the /user page from the url.

```
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.

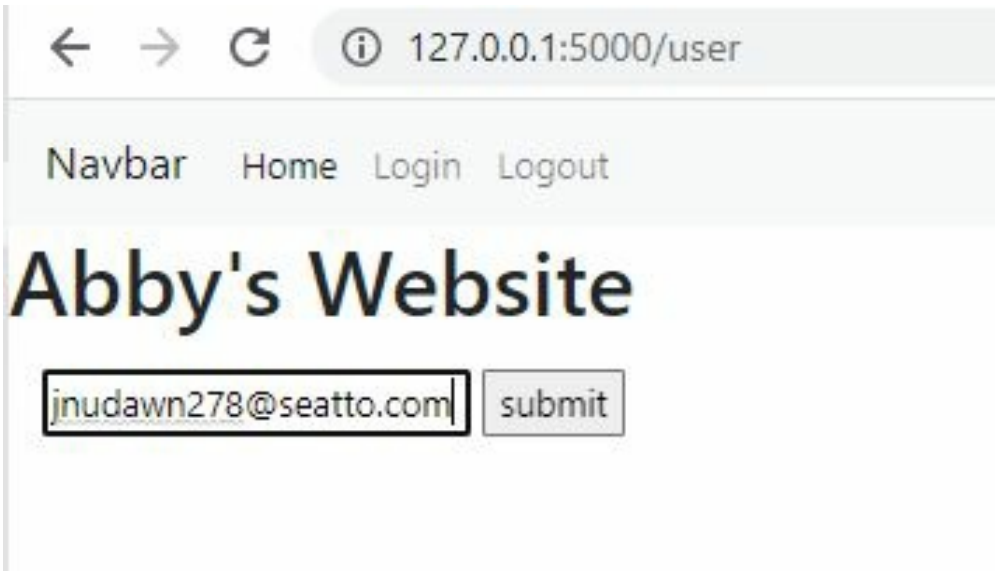
```
{% extends "base.html" %}
{% block title %}User{% endblock %}
{% block content %}
{% with messages = get_flashed_messages() %}
{% if messages %}
{% for message in messages %}
<p>{{ msg }}</p>
{% endfor %}
{% endif %}
{% endwith %}
<form action="#" method="POST">
   <input type="email" name="email"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

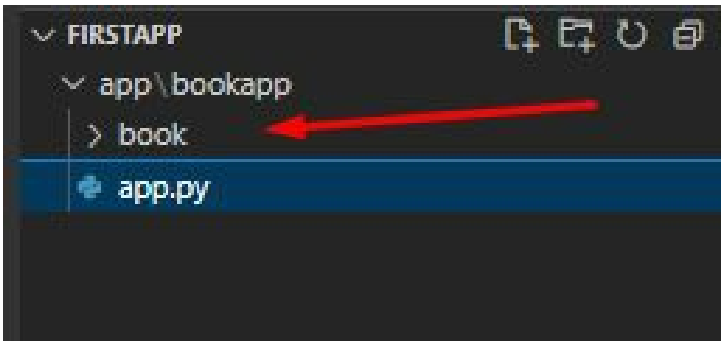
You can install everything with pip by running the following command:

```
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 a 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

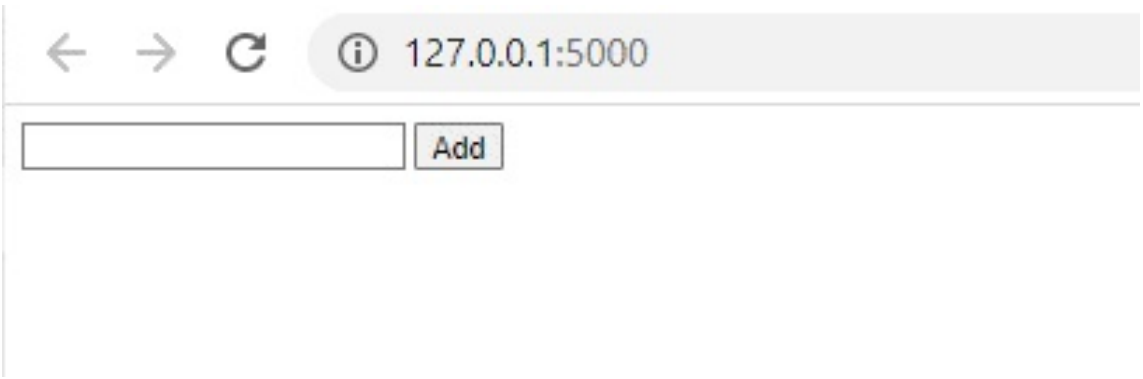
A submit button with the word "Add" on it.

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")
```

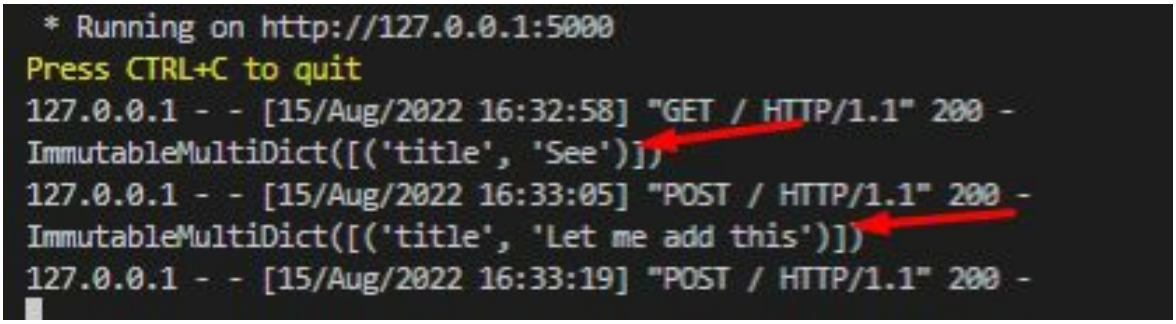
Here's what we did:

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 and 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 the "title" field from our form. 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.

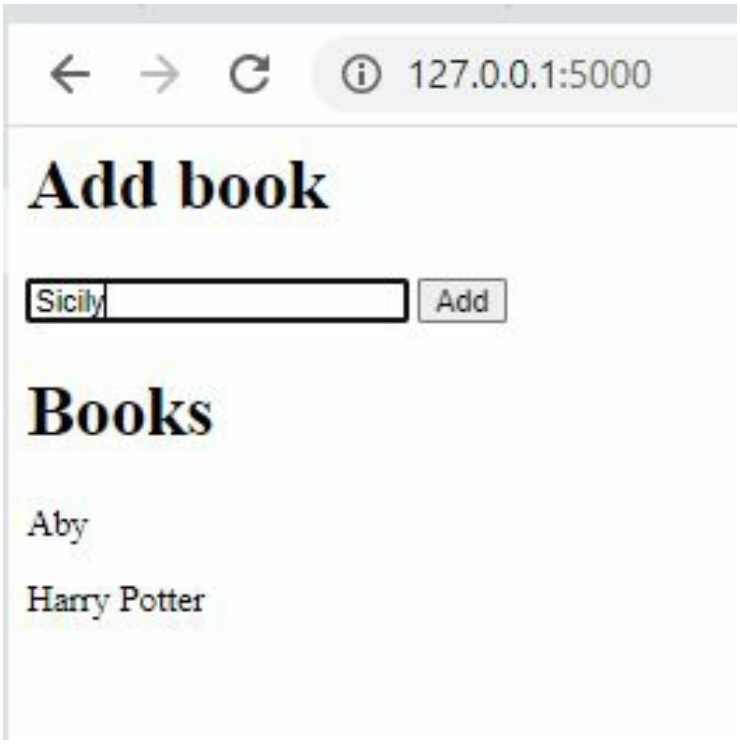
```
books = Book.query.all()
return render_template("index.html", books=books)
```

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.

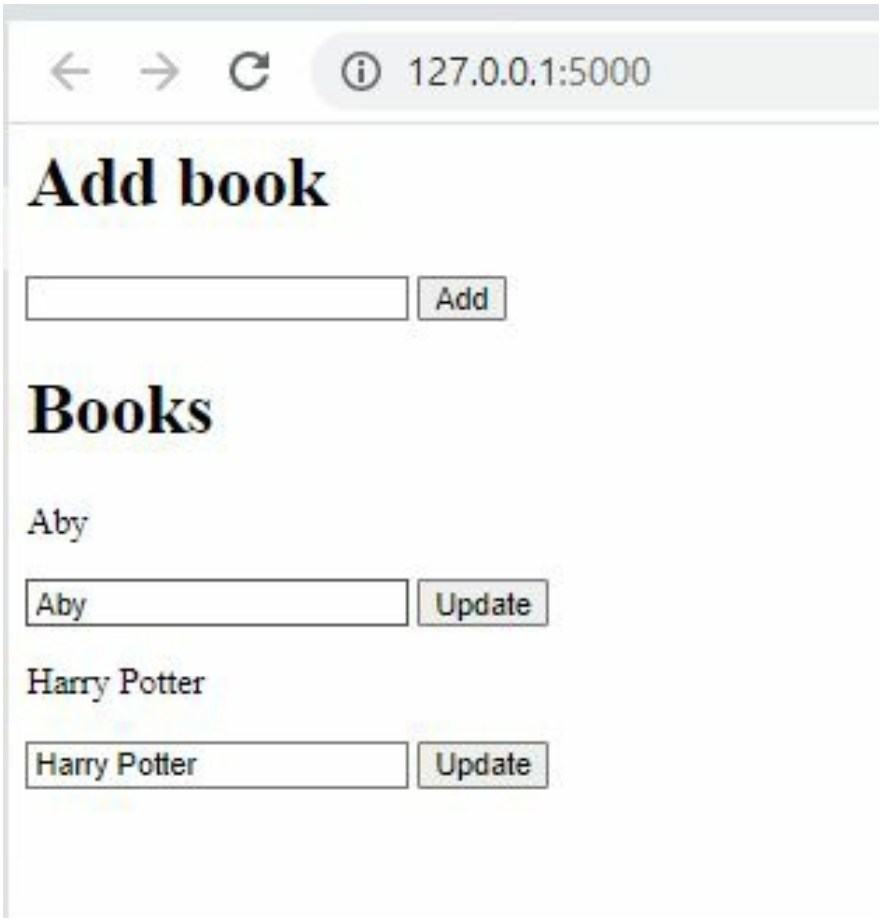
Open your app.py file and add a redirect to the imports list.

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

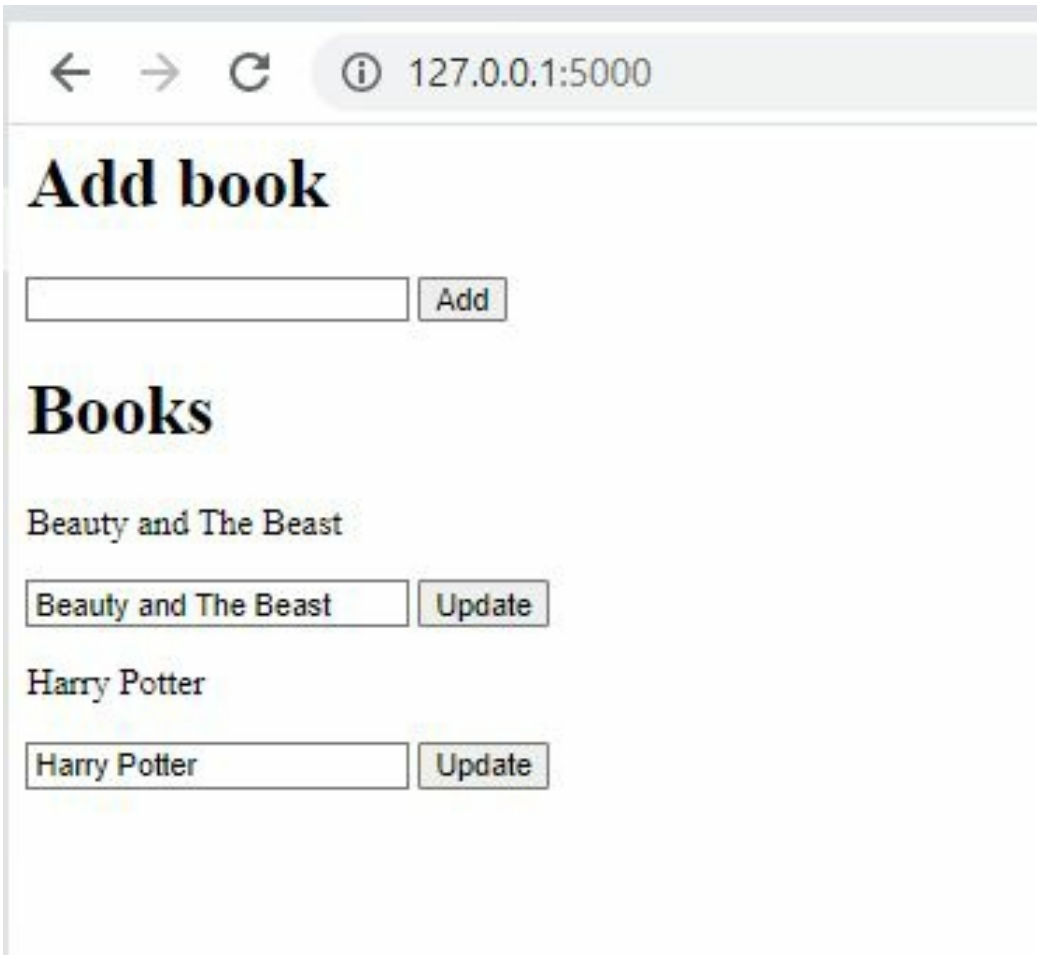
```
@app.route("/update", methods=["POST"])
def update():
    newtitle = request.form.get("newtitle")
    oldtitle = request.form.get("oldtitle")
    book = Book.query.filter_by(title=oldtitle).first()
```

```
    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:



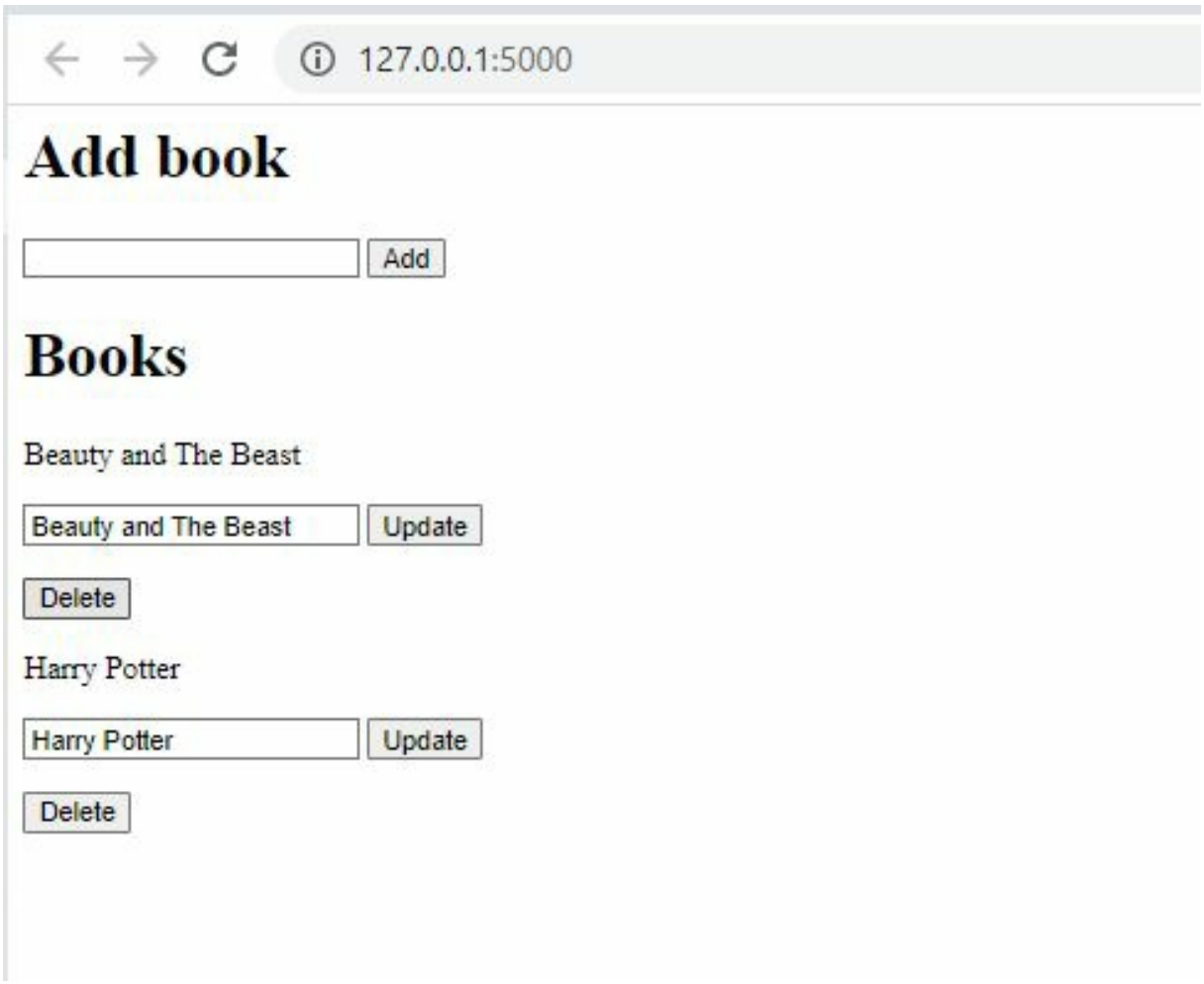
Well done. You have seen how we handle the CRU with Flask. Now, let us give the user the power to Delete books that they no longer want to see.

Deleting books from our database

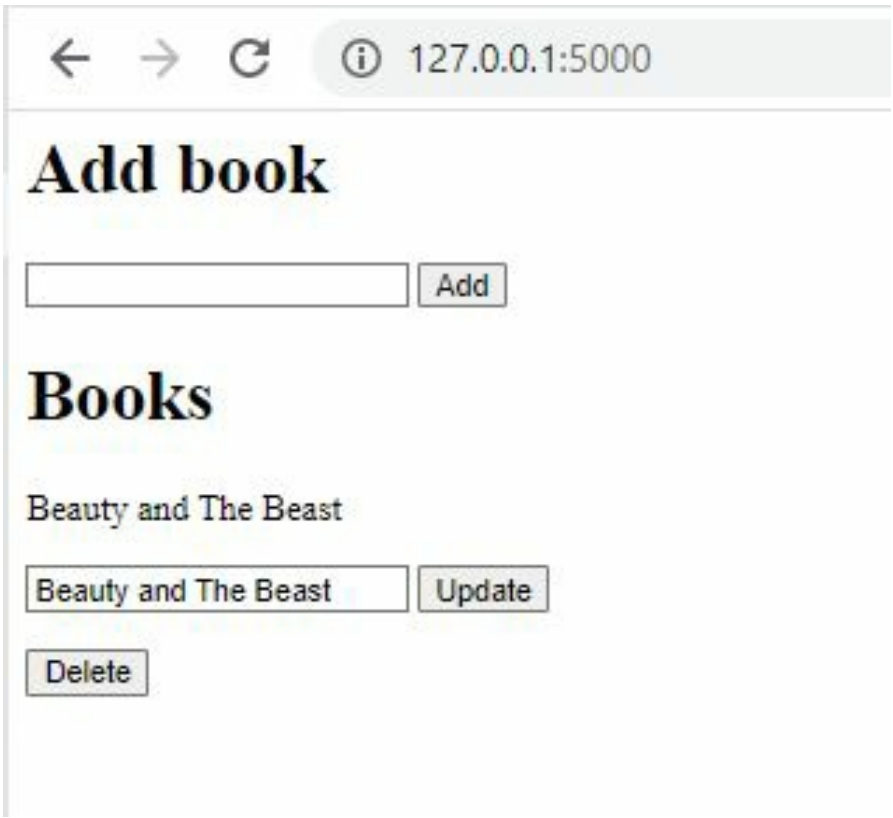
This feature is not completely different from how we created the Update feature. In this case, we don't need to call the old title. Open the index.html file and create another form with the for loop.

```
<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

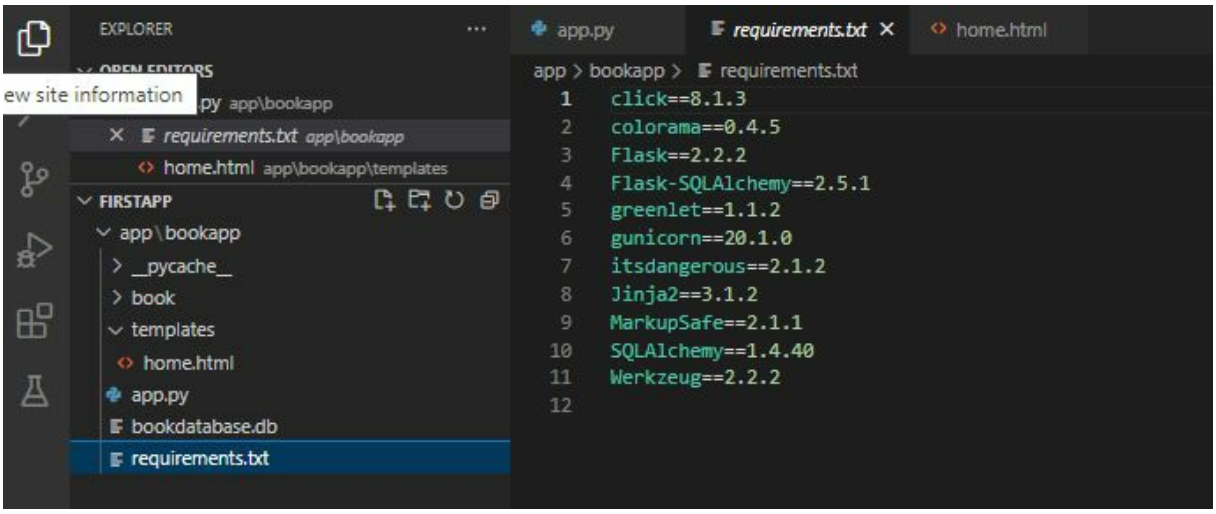
It is easy to get this by installing with pip:

```
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.