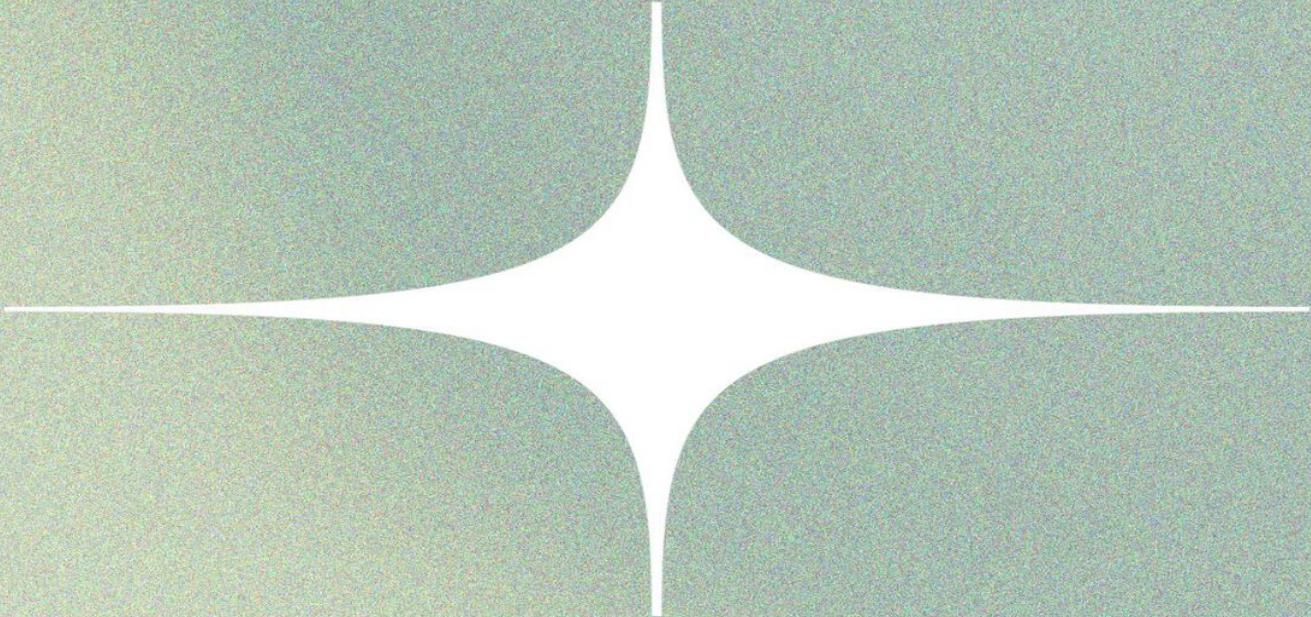
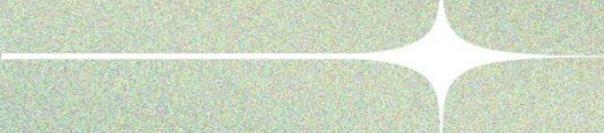




Python Web Applications with Flask



Jeffrey Leon Stroup



About the Authors

Thomas Carroll is a senior software engineer, programmer who works with tech enthusiasts, & passionate to learn more about programming and machine learning. After spending nearly a decade working for big companies , Jeffrey gained an in-depth knowledge of software systems and applications. As our society becomes increasingly reliant on technology, he believes that technology is at the very core of our life and is profoundly changing the way we live and work.

Table of Contents

Contents

[Table of Contents](#)

[Python Web Applications with Flask](#)

[PART 1: Flask Setup & Installation](#)

[CHAPTER 1: Introduction to Web development using Flask](#)

[CHAPTER 2: Install Flask in Windows](#)

[Install Virtual Environment](#)

[Install Flask on Windows or Linux](#)

[PART 2: Flask Quick Start](#)

[CHAPTER 1: Creating first simple application](#)

[Building a webpage using python.](#)

[CHAPTER 2: Run a Flask Application](#)

[Run Flask application Syntax](#)

[Run a Flask Application](#)

[Run the app in the debugger](#)

[CHAPTER 3: Flask App Routing](#)

[CHAPTER 4: Flask – HTTP Method](#)

[Flask HTTP Methods](#)

[GET Method in Flask](#)

[Example of HTTP GET in Flask](#)

[POST Method in Flask](#)

[Example of HTTP POST in Flask](#)

[CHAPTER 5: Flask - Variable Rule](#)

[Dynamic URLs Variable In Flask](#)

[Simple flask program](#)

[String Variable in Flask](#)

[Integer Variable in Flask](#)

[Float Variable in Flask](#)

[CHAPTER 6: Redirecting to URL in Flask](#)

[Redirect to a URL in Flask](#)

[Syntax of Redirect in Flask](#)

[How To Redirect To Url in Flask](#)

[url_for\(\) Function in Flask](#)

[CHAPTER 7: Python Flask - Redirect and Errors](#)

[Syntax of Redirect](#)

[Import the redirect attribute](#)

[Flasks Errors](#)

[Syntax of abort\(\) method](#)

[Example to demonstrate abort](#)

[CHAPTER 8: Change Port in Flask app](#)

[CHAPTER 9: Changing Host IP Address in Flask](#)

[Changing the IP address in a Flask application using the "host" parameter](#)

[Changing IP from the command line while deploying the Flask app](#)

[PART 3: Serve Templates and Static Files in Flask](#)

[CHAPTER 1: Flask Rendering Templates](#)

[Rendering a Template in a Flask Application](#)

[Setting up the Virtual Environment](#)

[Creating Templates in a Flask Application](#)

[Adding Routes and Rendering Templates](#)

[Templating With Jinja2 in Flask](#)

[Flask - Jinja Template Inheritance Example](#)

[If statement in HTML Template in Python Flask](#)

[CHAPTER 2: CSRF Protection in Flask](#)

[What is CSRF?](#)

[Solution for Preventing CSRF Attacks](#)

[Example of CSRF Protection in Flask](#)

[CHAPTER 3: Templating With Jinja2 in Flask](#)

[Templating with Jinja2 in Flask](#)

[Main Python File](#)

[Jinja Template Variables](#)

[Syntax of Jinja Template Variables](#)

[Jinja Template if Statements](#)

[Syntax of Jinja Template if Statements](#)

[Jinja Template for Loop](#)

[Syntax of Jinja Template for Loops](#)

[Jinja Template Inheritance](#)

[Syntax of Jinja Template Inheritance](#)

[Jinja Template url_for Function](#)

[Syntax of Jinja Template url_for Function](#)

[CHAPTER 4: Placeholders in jinja2 Template](#)

[Template Variables in Jinja2](#)

[Syntax of Template Variables in Jinja2](#)

[Example](#)

[Conditionals and Looping in Jinja2](#)

[Syntax of Conditionals and Looping](#)

[Template Inheritance in Jinja2](#)

[Syntax of Jinja extend block](#)

[CHAPTER 5: Serve static files in Flask](#)

[Serving Static Files in Flask](#)

[HTML File](#)

[Serve CSS file in Flask](#)

[Serve JavaScript file in Flask](#)

[Serve Media files in Flask \(Image, Video, Audio\)](#)

[Images](#)

[Video Files](#)

[Audio Files](#)

[Complete Flask Code](#)

[CHAPTER 6: Uploading and Downloading Files in Flask](#)

[Uploading and Downloading Files in Flask](#)

[Templates File](#)

[app.py](#)

[Complete Code](#)

[CHAPTER 7: Upload File in Python-Flask](#)

[Stepwise Implementation](#)

[CHAPTER 8: Upload Multiple files with Flask](#)

[Stepwise Implementation](#)

[CHAPTER 9: Flask – Message Flashing](#)

[What is Message Flashing](#)

[app.py File](#)

[Templates File](#)

[CHAPTER 10: Create Contact Us using WTForms in Flask](#)

[Advantages of WT-FORM:](#)

[Installation](#)

[Stepwise Implementation](#)

[Adding Bootstrap](#)

[CHAPTER 11: Sending Emails Using API in Flask-Mail](#)

[PART 4: User Registration, Login, and Logout in Flask](#)

[CHAPTER 1: Add Authentication to Your App with Flask-Login](#)

[Stepwise Implementation](#)

[Complete Code](#)

[CHAPTER 2: Add User and Display Current Username in Flask](#)

[Display Username on Multiple Pages using Flask](#)

[Templates Files](#)

[app.py](#)

[Complete Code](#)

[CHAPTER 3: Password Hashing with Bcrypt in Flask](#)

[Stepwise Implement with Bcrypt in Flask](#)

[Complete Code](#)

[CHAPTER 4: Role Based Access Control](#)

[Creating the Flask Application](#)

[CHAPTER 5: Use Flask-Session in Python Flask](#)

[Flask Session –](#)

[Installation](#)

[Configuring Session in Flask](#)

[Remember User After Login](#)

[Complete Project –](#)

[Output –](#)

[You can also see your generated session.](#)

[CHAPTER 6: Using JWT for user authentication in Flask](#)

[CHAPTER 7: Flask Cookies](#)

[Setting Cookies in Flask:](#)

[Getting Cookies in Flask:](#)

[Login Application in Flask using cookies](#)

[Getting website Visitors counted through cookies](#)

[CHAPTER 8: Return a JSON response from a Flask API](#)

[PART 5: Define and Access the Database in Flask](#)

[CHAPTER 1: Connect Flask to a Database with Flask-SQLAlchemy](#)

[Installing Flask](#)

[Creating app.py](#)

[Setting Up SQLAlchemy](#)

[Creating Models](#)

[Creating the database](#)

[Making Migrations in database](#)

[Creating the Index Page Of the Application](#)

[Creating HTML page for form](#)

[Function to add data using the form to the database](#)

[Display data on Index Page](#)

[Deleting data from our database](#)

[CHAPTER 2: Build a Web App using Flask and SQLite in Python](#)

[Steps to Build an App Using Flask and SQLite](#)

[CHAPTER 3: Sending Data from a Flask app to MongoDB Database](#)

[Configuring MongoDB](#)

[Setup a Development Environment](#)

[Installing Dependencies for the Project](#)

[Creating a Flask App](#)

[Connecting Flask App to Database](#)

[Sending Data from Flask to MongoDB](#)

[CHAPTER 4: Build a Web App using Flask and SQLite in Python](#)

[Steps to Build an App Using Flask and SQLite](#)

[CHAPTER 5: Login and Registration Project Using Flask and MySQL](#)

[CHAPTER 6: Execute raw SQL in Flask-SQLAlchemy app](#)

[Installing requirements](#)

[Syntax](#)

[PART 6: Flask Deployment and Error Handling](#)

[CHAPTER 1: Subdomain in Flask](#)

[CHAPTER 2: Handling 404 Error in Flask](#)

[Automatically Redirecting to the Home page after 5 seconds](#)

[CHAPTER 3: Deploy Python Flask App on Heroku](#)

[CHAPTER 4: Deploy Machine Learning Model using Flask](#)

Python Web Applications with Flask

PART 1: Flask Setup & Installation

CHAPTER 1: Introduction to Web development using Flask

- [REDACTED]
- [REDACTED]

What is Flask?

Flask is an API of Python that allows us to build up web-applications. It was developed by Armin Ronacher. Flask's framework is more explicit than Django's framework and is also easier to learn because it has less base code to implement a simple web-Application. A Web-Application Framework or Web Framework is the collection of modules and libraries that helps the developer to write applications without writing the low-level codes such as protocols, thread management, etc. Flask is based on WSGI(Web Server Gateway Interface) toolkit and Jinja2 template engine.

Getting Started With Flask:

Python 2.6 or higher is required for the installation of the Flask. You can start by import Flask from the flask package on any python IDE. For installation on any environment, you can click on the installation link given below.

To test that if the installation is working, check out this code given below.

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/')

def student():

    return render_template('student.html')


@app.route('/result', methods = ['POST', 'GET'])

def result():

    if request.method == 'POST':
```

```
result = request.form

return render_template("result.html", result = result)

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

'/' URL is bound with `hello()` function. When the home page of the webserver is opened in the browser, the output of this function will be rendered accordingly.

The Flask application is started by calling the `run()` function. The method should be restarted manually for any change in the code. To overcome this, the debug support is enabled so as to track any error.

Routing:

Nowadays, the web frameworks provide routing technique so that user can remember the URLs. It is useful to access the web page directly without navigating from the Home page. It is done through the following `route()` decorator, to bind the URL to a function.

```
from flask import Flask

app = Flask(__name__)

# /login display login form

@app.route('/login', methods = ['GET', 'POST'])

# login function verify username and password

def login():

    error = None

    if request.method == 'POST':

        if request.form['username'] != 'admin' or \
           request.form['password'] != 'admin':
```

```

        error = 'Invalid username or password. Please try again !'

else:

    # flashes on successful login
    flash('You were successfully logged in')

    return redirect(url_for('index'))

return render_template('login.html', error = error)

```

If a user visits `http://localhost:5000/hello` URL, the output of the `hello_world()` function will be rendered in the browser.

The `add_url_rule()` function of an application object can also be used to bind URL with the function as in above example.

Using Variables in Flask:

The Variables in the flask is used to build a URL dynamically by adding the variable parts to the rule parameter. This variable part is marked as. It is passed as keyword argument. See the example below

```

from flask import Flask

app = Flask(__name__)

# routing the decorator function hello_name
@app.route('/hello/<name>')

def hello_name(name):

    return 'Hello %s!' % name

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

```

Save the above example as hello.py and run from power shell. Next, open the browser and enter the URL <http://localhost:5000/hello/GeeksforGeeks>.

Output:

Hello GeeksforGeeks!

In the above example, the parameter of route() decorator contains the variable part attached to the URL '/hello' as an argument. Hence, if URL like <http://localhost:5000/hello/GeeksforGeeks> is entered then 'GeeksforGeeks' will be passed to the hello() function as an argument.

In addition to the default string variable part, other data types like int, float, and path(for directory separator channel which can take slash) are also used. The URL rules of Flask are based on Werkzeug's routing module. This ensures that the URLs formed are unique and based on precedents laid down by Apache.

Examples:

```
from flask import Flask

app = Flask(__name__)

@app.route('/blog/<postID>')

def show_blog(postID):

    return 'Blog Number %d' % postID

@app.route('/rev/<revNo>')

def revision(revNo):

    return 'Revision Number %f' % revNo

if __name__ == '__main__':
    app.run()

# say the URL is http://localhost:5000/blog/555
```

Output :

Blog Number 555

Enter this URL in the browser ? http://localhost:5000/rev/1.1

Revision Number: 1.100000

Building URL in FLask:

Dynamic Building of the URL for a specific function is done using `url_for()` function. The function accepts the name of the function as first argument, and one or more keyword arguments. See this example

```
from flask import Flask, redirect, url_for

app = Flask(__name__)

@app.route('/admin') #decorator for route(argument) function

def hello_admin():
    #binding to hello_admin call
    return 'Hello Admin'

@app.route('/guest/<guest>')

def hello_guest(guest):
    #binding to hello_guest call
    return 'Hello %s as Guest' % guest

@app.route('/user/<name>')

def hello_user(name):
    if name =='admin': #dynamic binding of URL to function
        return redirect(url_for('hello_admin'))
    else:
        return redirect(url_for('hello_guest', guest = name))

if __name__ == '__main__':
```

```
app.run(debug = True)
```

To test this, save the above code and run through python shell and then open browser and enter the following URL:-

Input: http://localhost:5000/user/admin

Output: Hello Admin

Input: http://localhost:5000/user/ABC

Output: Hello ABC as Guest

The above code has a function named user(name), accepts the value through input URL. It checks that the received argument matches the 'admin' argument or not. If it matches, then the function hello_admin() is called else the hello_guest() is called.

Flask support various HTTP protocols for data retrieval from the specified URL, these can be defined as:-

Method	Description
GET	This is used to send the data in an without encryption of the form to the server.
HEAD	provides response body to the form
POST	Sends the form data to server. Data received by POST method is not cached by server.
PUT	Replaces current representation of target resource with URL.
DELETE	Deletes the target resource of a given URL

Handling Static Files :

A web application often requires a static file such as javascript or a CSS file to render the display of the web page in browser. Usually, the web server is configured to set them, but during development, these files are served as static folder in your package or next to the module. See the example in JavaScript given below:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index():

    return render_template("index.html")

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

The following HTML code:

This will be inside **templates** folder which will be sibling of the python file we wrote above

```
<html>

<head>
    <script type = "text/javascript"
        src = "{{ url_for('static', filename = 'hello.js') }}"
    ></script>
</head>

<body>
    <input type = "button" onclick = "sayHello()" value = "Say Hello" />
</body>

</html>
```

The JavaScript file for hello.js is:

*This code will be inside **static** folder which will be sibling of the templates folder*

```
function sayHello() {  
    alert("Hello World")  
}
```

The above hello.js file will be rendered accordingly to the HTML file.

Object Request of Data from a client's web page is send to the server as a global request object. It is then processed by importing the Flask module. These consist of attributes like Form(containing Key-Value Pair), Args(parsed URL after question mark(?)), Cookies(contain Cookie names and Values), Files(data pertaining to uploaded file) and Method(current request).

Cookies:

A Cookie is a form of text file which is stored on a client's computer, whose purpose is to remember and track data pertaining to client's usage in order to improve the website according to the user's experience and statistic of webpage.

The Request object contains cookie's attribute. It is the dictionary object of all the cookie variables and their corresponding values. It also contains expiry time of itself. In Flask, cookie are set on response object. See the example given below:-

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/')  
  
def index():  
  
    return render_template('index.html')
```

HTML code for index.html

```
<html>
```

```

<body>

    <form action = "/setcookie" method = "POST">
        <p><h3>Enter userID</h3></p>
        <p><input type = 'text' name = 'nm' /></p>
        <p><input type = 'submit' value = 'Login' /></p>
    </form>

</body>
</html>

```

Add this code to the python file defined above

```

@app.route('/setcookie', methods = ['POST', 'GET'])

def setcookie():

    if request.method == 'POST':

        user = request.form['nm']

        resp = make_response(render_template('cookie.html'))
        resp.set_cookie('userID', user)

    return resp

@app.route('/getcookie')

def getcookie():

    name = request.cookies.get('userID')

    return '<h1>welcome '+name+'</h1>'

```

HTML code for cookie.html

```
<html>
  <body>
    <a href="/getcookie">Click me to get Cookie</a>
  </body>
</html>
```

Run the above application and visit link on Browser <http://localhost:5000/>. The form is set to '/setcookie' and function set contains a Cookie name userID that will be rendered to another webpage. The 'cookie.html' contains hyperlink to another view function getcookie(), which displays the value in browser.

Sessions in Flask:

In Session, the data is stored on Server. It can be defined as a time interval in which the client logs into a server until the user logs out. The data in between them are held in a temporary folder on the Server. Each user is assigned with a specific **Session ID**. The Session object is a dictionary that contains the key-value pair of the variables associated with the session. A SECRET_KEY is used to store the encrypted data on the cookie.

For example:

```
Session[key] = value // stores the session value
Session.pop(key, None) // releases a session variable
```

Other Important Flask Functions:

redirect(): It is used to return the response of an object and redirects the user to another target location with specified status code.

Syntax: Flask.redirect(location, statuscode, response)

//location is used to redirect to the desired URL
//statuscode sends header value, default 302
//response is used to initiate response.

abort: It is used to handle the error in the code.

Syntax: Flask.abort(code)

The code parameter can take the following values to handle the error accordingly:

- **400** – For Bad Request
- **401** – For Unauthenticated
- **403** – For Forbidden request
- **404** – For Not Found
- **406** – For Not acceptable
- **425** – For Unsupported Media
- **429** – Too many Requests

File-Uploading in Flask:

File Uploading in Flask is very easy. It needs an HTML form with enctype attribute and URL handler, that fetches file and saves the object to the desired location. Files are temporary stored on server and then on the desired location.

The HTML Syntax that handle the uploading URL is :

form action="http://localhost:5000/uploader" method="POST" enctype="multipart/form-data"
and following python code of Flask is:

```
from flask import Flask, render_template, request

from werkzeug import secure_filename

app = Flask(__name__)

@app.route('/upload')

def upload_file():

    return render_template('upload.html')

@app.route('/uploader', methods = ['GET', 'POST'])

def upload_file():

    if request.method == 'POST':

        f = request.files['file']

        f.save(secure_filename(f.filename))

    return 'file uploaded successfully'
```

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

Sending Form Data to the HTML File of Server:

A Form in HTML is used to collect the information of required entries which are then forwarded and stored on the server. These can be requested to read or modify the form. The flask provides this facility by using the URL rule. In the given example below, the '/' URL renders a web page(student.html) which has a form. The data filled in it is posted to the '/result' URL which triggers the result() function. The results() function collects form data present in request.form in a dictionary object and sends it for rendering to result.html.

```
from flask import Flask, render_template, request
app = Flask(__name__)
@app.route('/')
def student():
    return render_template('student.html')

@app.route('/result', methods = ['POST', 'GET'])
def result():
    if request.method == 'POST':
        result = request.form
        return render_template("result.html", result = result)

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

<!doctype html>
```

```

<html>
  <body>

    <table border = 1>
      {% for key, value in result.items() %}

        <tr>
          <th> {{ key }} </th>
          <td> {{ value }} </td>
        </tr>

      {% endfor %}
    </table>

  </body>
</html>

```

```

<html>
  <body>

    <form action = "http://localhost:5000/result" method = "POST">
      <p>Name <input type = "text" name = "Name" /></p>
      <p>Physics <input type = "text" name = "Physics" /></p>
      <p>Chemistry <input type = "text" name = "chemistry" /></p>
      <p>Maths <input type = "text" name = "Maths" /></p>
      <p><input type = "submit" value = "submit" /></p>
    </form>
  </body>
</html>

```

<p>Name: <input type="text" value="ABC"/></p> <p>Physics: <input type="text" value="50"/></p> <p>Chemistry: <input type="text" value="60"/></p> <p>Maths: <input type="text" value="70"/></p> <p><input type="button" value="Submit"/></p>	<table border="1"> <tr> <td>Maths</td> <td>70</td> </tr> <tr> <td>Chemistry</td> <td>60</td> </tr> <tr> <td>Physics</td> <td>50</td> </tr> <tr> <td>Name</td> <td>ABC</td> </tr> </table>	Maths	70	Chemistry	60	Physics	50	Name	ABC
Maths	70								
Chemistry	60								
Physics	50								
Name	ABC								

Input values to the form HTML

Template as the output

Message Flashing:

It can be defined as a pop-up or a dialog box that appears on the web-page or like alert in JavaScript, which are used to inform the user. This in flask can be done by using the method `flash()` in Flask. It passes the message to the next template.

Syntax: `flash(message, category)`

message is actual text to be displayed and **category** is optional which is to render any error or info.

Example :

```
from flask import Flask

app = Flask(__name__)

# /login display login form

@app.route('/login', methods = ['GET', 'POST'])

# login function verify username and password

def login():

    error = None
```

```
if request.method == 'POST':  
    if request.form['username'] != 'admin' or \  
        request.form['password'] != 'admin':  
            error = 'Invalid username or password. Please try again !'  
    else:  
  
        # flashes on successful login  
        flash('You were successfully logged in')  
  
        return redirect(url_for('index'))  
  
return render_template('login.html', error = error)
```

CHAPTER 2: Install Flask in Windows

Flask is basically a Python module. It can work with Python only and it is a web-developing framework. It is a collection of libraries and modules. Frameworks are used for developing web platforms. Flask is such a type of web application framework. It is completely written in Python language. Unlike Django, it is only written in Python. As a new user, Flask is to be used. As it is easier to handle. As it is only written in Python, before installing Flask on the machine, Python should be installed previously.

Features of Python Flask:

- Flask is easy to use and easily understandable for new users in Web Framework.
- It can also be used as any third-party plugin extension.
- It is also used for prototyping purposes.

Install Virtual Environment

We use a module named virtualenv which is a tool to create isolated Python environments. virtualenv creates a folder that contains all the necessary executables to use the packages that a Python project would need.

```
pip install virtualenv
```

Create Python virtual environment

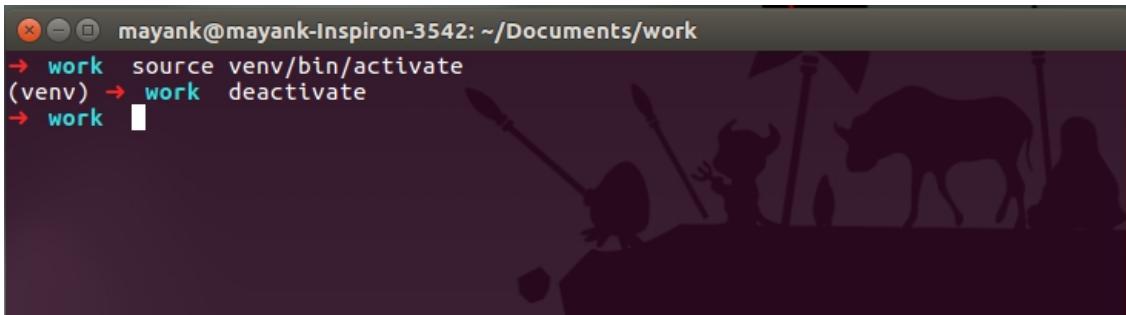
Go to the local directory where you want to create your Flask app.

```
virtualenv venv
```

Activate a virtual environment based on your OS

For windows > venv\Scripts\activate

For linux > source ./venv/bin/activate



The screenshot shows a terminal window with a dark background and light-colored text. The title bar reads "mayank@mayank-Inspiron-3542: ~/Documents/work". The command entered is "source venv/bin/activate". The output shows three arrows pointing right, followed by the text "(venv) work". Below this, another arrow points right to the word "deactivate". The prompt "work" is shown again at the bottom. In the background of the terminal window, there is a faint watermark or illustration of a scene from the movie "The Lion King".

Install Flask on Windows or Linux

Step 1: Make sure that Python PIP should be installed on your OS. You can check using the below command.

```
pip -V
```

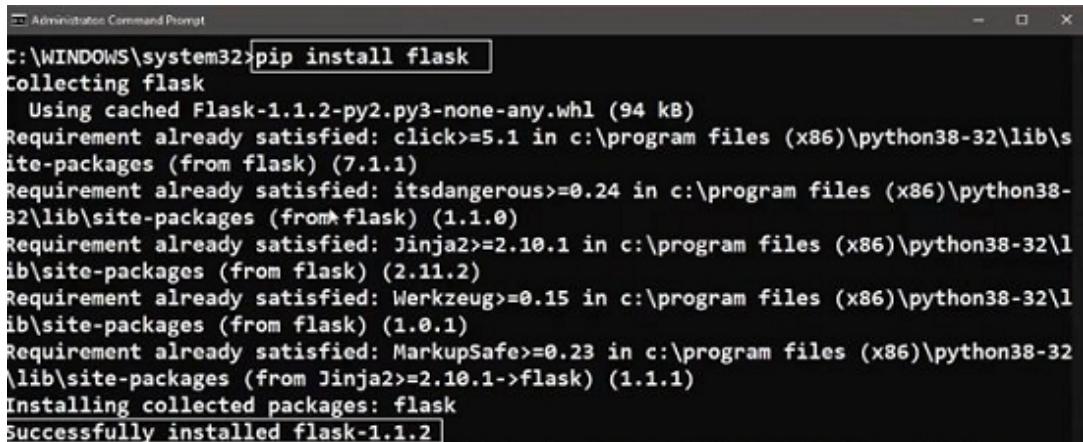
or

```
pip --version
```

Step 2: At first, open the command prompt in administrator mode. Then the following command should be run. This command will help to install Flask using Pip in Python and will take very less time to install. According to the machine configuration, a proper Flask version should be installed. Wait for some time till the process is completed. After completion of the process, Flask is completed successfully,

the message will be displayed. Hence Installation is successful.

```
pip install flask
```

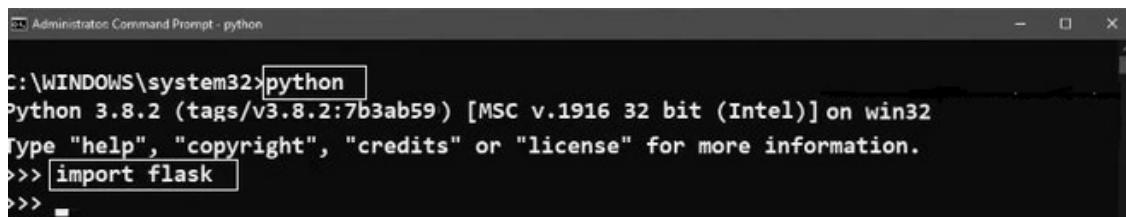


```
C:\WINDOWS\system32>pip install flask
Collecting flask
  Using cached Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Requirement already satisfied: click>=5.1 in c:\program files (x86)\python38-32\lib\site-packages (from flask) (7.1.1)
Requirement already satisfied: itsdangerous>=0.24 in c:\program files (x86)\python38-32\lib\site-packages (from flask) (1.1.0)
Requirement already satisfied: Jinja2>=2.10.1 in c:\program files (x86)\python38-32\lib\site-packages (from flask) (2.11.2)
Requirement already satisfied: Werkzeug>=0.15 in c:\program files (x86)\python38-32\lib\site-packages (from flask) (1.0.1)
Requirement already satisfied: MarkupSafe>=0.23 in c:\program files (x86)\python38-32\lib\site-packages (from Jinja2>=2.10.1->flask) (1.1.1)
Installing collected packages: flask
Successfully installed flask-1.1.2
```

Step 3: After that, also the following two commands should be run. These commands will start Flask in the command prompt. Hence, the process is completed successfully.

```
python
```

```
import flask
```



```
C:\WINDOWS\system32>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2019, 17:15:01) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>> =
```

PART 2: Flask Quick Start

CHAPTER 1: Creating first simple application

Building a webpage using python.

There are many modules or frameworks which allow building your webpage using python like a bottle, Django, Flask, etc. But the real popular ones are Flask and Django. Django is easy to use as compared to Flask but Flask provides you with the versatility to program with.

To understand what Flask is you have to understand a few general terms.

1. **WSGI** Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.
2. **Werkzeug** It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.
3. **jinja2** jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Flask is a web application framework written in Python. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

Installation:

We will require two packages to set up your environment. *virtualenv* for a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries and the next will be *Flask* itself.

virtualenv

```
pip install virtualenv
```

Create Python virtual environment

```
virtualenv venv
```

Activate virtual environment

```
windows > venv\Scripts\activate
```

```
linux > source ./venv/bin/activate
```

For windows, if this is your first time running the script, you might get an error like below:

```
venv\Scripts\activate : File C:\flask_project\venv\Scripts\Activate.ps1  
cannot be loaded because running scripts is disabled on this system. For  
more information, see about_Execution_Policies at  
https://go.microsoft.com/fwlink/?LinkID=135170.
```

```
At line:1 char:1
```

```
+ venv\Scripts\activate
```

```
    + FullyQualifiedErrorId : UnauthorizedAccess
```

This means that you don't have access to execute the scripts.

To solve this error, run the powershell as admin, when you right click on powershell icon, choose the option 'Run as administrator'. Now, the powershell will open in the admin mode.

Type the following command in Shell

```
set-executionpolicy remotesigned
```

Now, you will be prompted to change the execution policy. Please type A.
This means Yes to all.

Flask

```
pip install Flask
```

After completing the installation of the package, let's get our hands on the code.

- Python3

```
# Importing flask module in the project is mandatory
```

```
# An object of Flask class is our WSGI application.

from flask import Flask


# Flask constructor takes the name of
# current module (__name__) as argument.

app = Flask(__name__)

# The route() function of the Flask class is a decorator,
# which tells the application which URL should call
# the associated function.

@app.route('/')
# '/' URL is bound with hello_world() function.

def hello_world():

    return 'Hello World'

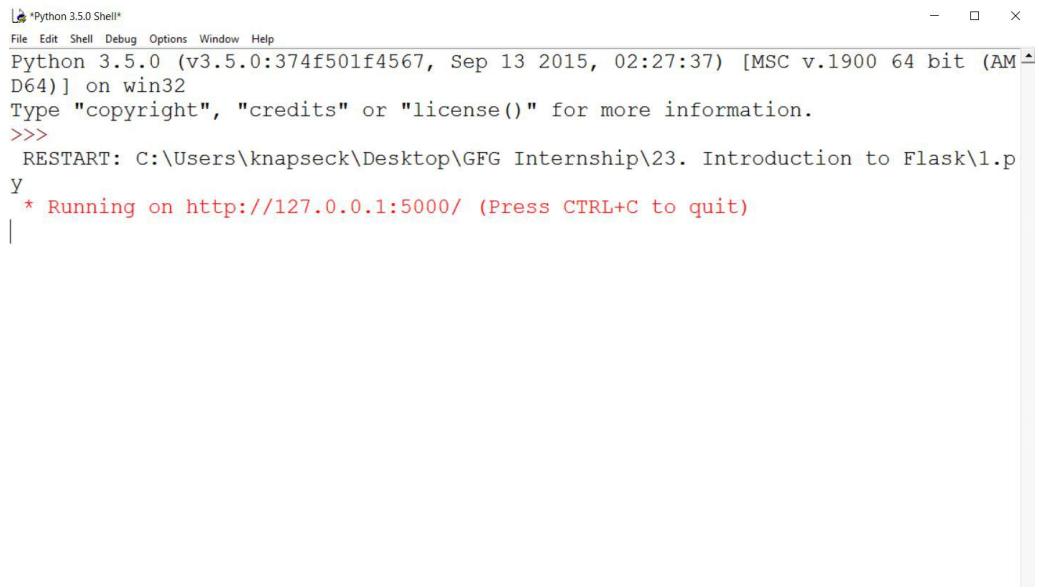
# main driver function

if __name__ == '__main__':

    # run() method of Flask class runs the application
    # on the local development server.

    app.run()
```

Save it in a file and then run the script we will be getting an output like this.



```
*Python 3.5.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\knapseck\Desktop\GFG Internship\23. Introduction to Flask\1.p
y
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
|
```

Then go to the URL given there you will see your first webpage displaying hello worldthereonyourlocalserver.

Digging further into the context, the **route()** decorator in Flask is used to bind a URL to a function. Now to extend this functionality our small web app is also equipped with another method **add_url_rule()** which is a function of an application object that is also available to bind a URL with a function as in the above example, route() is used.

Example:

```
def gfg():
    return 'geeksforgeeks'
app.add_url_rule('/', 'g2g', gfg)
```

Output:

```
geeksforgeeks
```

You can also add variables in your web app, well you might be thinking about how it'll help you, it'll help you to build a URL dynamically. So let's figure it out with an example.

- Python3

```
from flask import Flask

app = Flask(__name__)

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

def hello_name(name):

    return 'Hello %s!' % name

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

And go to the URL `http://127.0.0.1:5000/hello/geeksforgeeks` it'll give you the following output



Hello geeksforgeeks!

We can also use HTTP methods in Flask let's see how to do that. The HTTP protocol is the foundation of data communication on the world wide web. Different methods of data retrieval from specified URL are defined in this protocol. The methods are described down below.

GET : Sends data in simple or unencrypted form to the server.

HEAD : Sends data in simple or unencrypted form to the server without body.

HEAD : Sends form data to the server. Data is not cached.

PUT : Replaces target resource with the updated content.

DELETE : Deletes target resource provided as URL.

By default, the Flask route responds to the GET requests. However, this preference can be altered by providing methods argument to route() decorator. In order to demonstrate the use of the POST method in URL

routing, first, let us create an HTML form and use the POST method to send form data to a URL. Now let's create an HTML login page.

Below is the source code of the file:

- HTML

```
<html>
<body>

<form action = "http://localhost:5000/login" method = "post">

<p>Enter Name:</p>

<p><input type = "text" name = "nm" /></p>

<p><input type = "submit" value = "submit" /></p>

</form>

</body>

</html>
```

Now save this file HTML and try this python script to create the server.

- Python3

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

app = Flask(__name__)

@app.route('/success/<name>')
def success(name):
    return 'welcome %s' % name

@app.route('/login', methods=['POST', 'GET'])

def login():
    if request.method == 'POST':
        user = request.form['nm']
        return redirect(url_for('success', name=user))
    else:
        user = request.args.get('nm')
        return redirect(url_for('success', name=user))

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

After the development server starts running, open login.html in the browser, enter your name in the text field and click *submit* button. The output would be the following.



A screenshot of a web browser window. The address bar shows the URL `file:///C:/Users/knapseck/Desktop/webdev_practice/login.html`. The page content is a simple form with the following fields:

- A label "Enter Name:" followed by an input field containing the text "subhajit".
- A submit button labeled "submit".

The result will be something like this



And there's much more to Flask than this. If you are interested in this web framework of Python you can dig into the links provided down below for further information.

CHAPTER 2: Run a Flask Application

The backend server Flask was created fully in Python. It is a framework made up of Python modules and packages. With its characteristics, it is a lightweight Flask application that speeds up the development of backend apps. We will learn how to execute a Flask application in this tutorial.

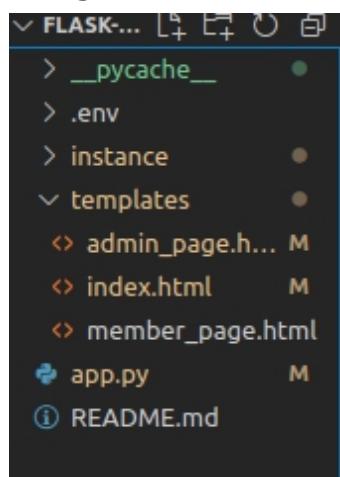
Run Flask application Syntax

We can run the Flask application using the below command.

```
flask -app <hello> run  
flask run  
python <app_name>.py
```

File Structure

Here, we are using the following folder and file.



Run a Flask Application

In this example, we have an application called **helloworld.py** below is the basic code for Flask.

- Python3

```

# import flask module

from flask import Flask


# instance of flask application

app = Flask(__name__)

# home route that returns below text when root url is accessed

@app.route("/")
def hello_world():

    return "<p>Hello, World!</p>"


if __name__ == '__main__':
    app.run()

```

Output:

Using **flask -app <app_name> run**

```

* Debugger PIN: 145-287-441
* ^C(flask_env) (base) gfg19473@GFG19473-DL0449:~/Flask_gfg/flask-rolebased$ flask --app app run
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production
* Running on http://127.0.0.1:5000
Press CTRL+C to quit

```

Using **flask run**

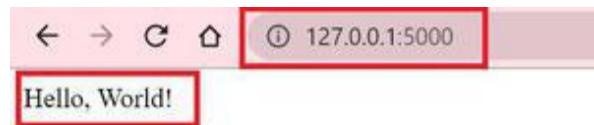
```

● (flask_env) (base) gfg19473@GFG19473-DL0449:~/Flask_gfg/flask-rolebased$ flask run
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a prod
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [13/Apr/2023 17:57:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [13/Apr/2023 17:57:50] "GET /favicon.ico HTTP/1.1" 404 -

```

Using the **python app_name.py**

```
((test) ) C:\Users\suraj\Desktop\flask\test>python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```



Run the app in the debugger

We will use the below command to run the flask application with debug mode as on. When debug mode is turned on, It allows developers to locate any possible error and as well the location of the error, by logging a traceback of the error.

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

```
((test) ) C:\Users\suraj\Desktop\flask\test>python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 115-625-310
127.0.0.1 - - [25/oct/2022 20:45:11] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [25/oct/2022 20:45:11] "GET /helloworld HTTP/1.1" 200 -
```

CHAPTER 3: Flask App Routing

App Routing means mapping the URLs to a specific function that will handle the logic for that URL. Modern web frameworks use more meaningful URLs to help users remember the URLs and make navigation simpler.

Example: In our application, the URL (“/”) is associated with the root URL. So if our site’s domain was www.example.org and we want to add routing to “www.example.org/hello”, we would use “/hello”.

To bind a function to an URL path we use the `app.route` decorator. In the below example, we have implemented the above routing in the flask.

- main.py

```
from flask import Flask

app = Flask(__name__)

# Pass the required route to the decorator.
@app.route("/hello")
def hello():
    return "Hello, Welcome to GeeksForGeeks"

@app.route("/")
def index():
    return "Homepage of GeeksForGeeks"

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

The hello function is now mapped with the “/hello” path and we get the output of the function rendered on the browser.

Step to run the application: Run the application using the following command.

```
python main.py
```

Output: Open the browser and visit `127.0.0.1:5000/hello`, you will see the following output.



Dynamic URLs - We can also build dynamic URLs by using variables in the URL. To add variables to URLs, use `<variable_name>` rule. The function then receives the `<variable_name>` as keyword argument.

Example: Consider the following example to demonstrate the dynamic URLs.

- main.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/user/<username>')
def show_user(username):
    # Greet the user
    return f'Hello {username} !'

# Pass the required route to the decorator.
```

```

@app.route("/hello")

def hello():

    return "Hello, Welcome to GeeksForGeeks"

@app.route("/")

def index():

    return "Homepage of GeeksForGeeks"

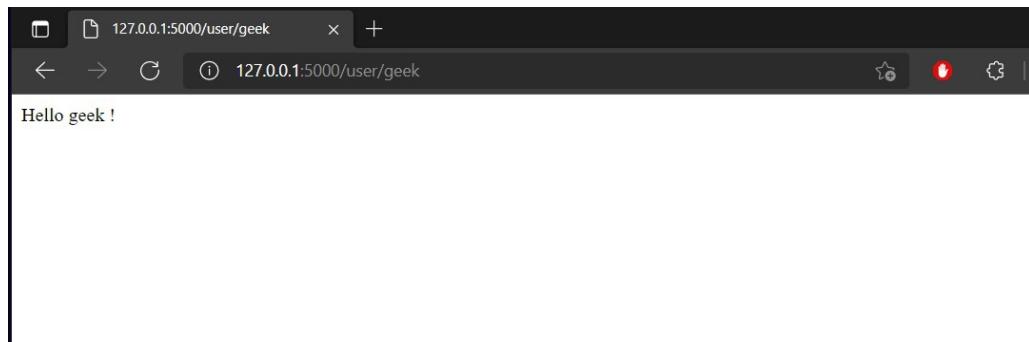
if __name__ == "__main__":
    app.run(debug=True)

```

Step to run the application: Run the application using the following command.

python main.py

Output: Open the browser and visit `127.0.0.1:5000/user/geek`, you will see the following output.



Additionally, we can also use a converter to convert the variable to a specific data type. By default, it is set to string values. To convert use `<converter:variable_name>` and following converter types are supported.

- **string:** It is the default type and it accepts any text without a slash.
- **int:** It accepts positive integers.
- **float:** It accepts positive floating-point values.
- **path:** It is like a string but also accepts slashes.
- **uuid:** It accepts UUID strings.

Example: Consider the following example to demonstrate the converter type.

- main.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/post/<int:id>')
def show_post(id):
    # Shows the post with given id.
    return f'This post has the id {id}'

@app.route('/user/<username>')
def show_user(username):
    # Greet the user
    return f'Hello {username} !'

# Pass the required route to the decorator.
@app.route("/hello")
def hello():
    return "Hello, Welcome to GeeksForGeeks"

@app.route("/")
def index():
    return "Homepage of GeeksForGeeks"

if __name__ == "__main__":
```

```
app.run(debug=True)
```

Step to run the application: Run the application using the following command.

```
python main.py
```

Output: Open the browser and visit `127.0.0.1:5000/post/13`, you will see the following output.



The add_url_rule() function - The URL mapping can also be done using the `add_url_rule()` function. This approach is mainly used in case we are importing the view function from another module. In fact, the `app.route` calls this function internally.

Syntax:

```
add_url_rule(<url rule>, <endpoint>, <view function>)
```

Example: In the below example, we will try to map the `show_user` view function using this approach.

- `main.py`

```
from flask import Flask

app = Flask(__name__)

def show_user(username):
    # Greet the user
    return f'Hello {username} !'
```

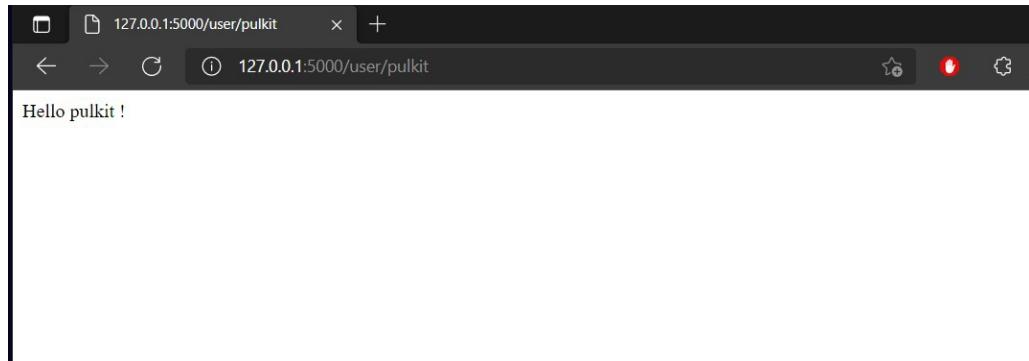
```
app.add_url_rule('/user/<username>', 'show_user', show_user)

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

Step to run the application: Run the application using the following command.

```
python main.py
```

Output: Open the browser and visit `127.0.0.1:5000/user/pulkit`, you will see the following output.



CHAPTER 4: Flask - HTTP Method

In this article, we will learn how to handle HTTP methods, such as GET and POST in Flask using Python. Here, we will understand the concept of HTTP, GET, and HTTP POST, and then we will see the example and implement each in Flask. Before starting let's understand the basic terminology:

- **GET**: to request data from the server.
- **POST**: to submit data to be processed to the server.
- **PUT**: A PUT request is used to modify the data on the server. It replaces the entire content at a particular location with data that is passed in the body payload. If there are no resources that match the request, it will generate one.
- **PATCH**: PATCH is similar to a PUT request, but the only difference is, it modifies a part of the data. It will only replace the content that you want to update.
- **DELETE**: A DELETE request is used to delete the data on the server at a specified location.

Flask HTTP Methods

In a Client-Server architecture, there is a set of rules, called a protocol, using which, we can allow the clients, to communicate with the server, and, vice-versa. Here, the Hyper Text Transfer Protocol is used, through which, communication is possible. For example, Our browser, passes our query, to the Google server, receiving which, the Google server, returns relevant suggestions. The commonly used HTTP methods, for this interconnection, are - **GET** and **POST**.

GET Method in Flask

The request we type, in the browser address bar, say: 'http://google.com' is called the Uniform Resource Locator. It mentions the address we are looking for, in this case, the Google landing(starting) page. The browser, sends a GET request, to the Google server, which returns the starting webpage, in response. The GET request is simply used, to fetch data from the server. It should not be used, to apply changes, to the server data.

Example of HTTP GET in Flask

Let us discuss, the execution of the GET method, using the Flask library. In this example, we will consider, a landing page, that gives us facts, about Math calculations, and, allows us to enter a number, and, return its square. Let us see the example:

Step 1: The ‘**squarenum.html**’ file, has a form tag, that allows the user to enter a number. The form data is sent, to the page mentioned, in the ‘action’ attribute. Here, the data is sent, to the same page, as indicated by '#'. Since we are using the GET method, the data will be appended, to the URL, in the name-value pair format. So, if we enter number 12, and, click on Submit button, then data will be appended, as ‘<http://localhost:5000/square?num=12&bttnum=Submit#>’

- HTML

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Square Of Number!</title>

</head>

<body>

<h1><i> Welcome to the Maths page!</i></h1>

<p>Logic shapes every choice of our daily lives.<br>

Logical thinking enables someone to learn and

make decisions that affect their way of life. !</p>

<form method="GET" action ="#">

    Enter a number :

    <input type="text" name="num" id="num"></input>

    <input type="submit" name="bttnum" id="bttnum"></input>

</form>
```

```
</body>  
</html>
```

Step 2: The **answer.html** code file looks as follows:

- HTML

```
<!DOCTYPE html>  
  
<html lang="en">  
  
<head>  
  
    <meta charset="UTF-8">  
  
    <title>Answer Page!</title>  
  
</head>  
  
<body>  
  
    <h1>Keep Learning Maths!</h1>  
  
    <h2>Square of number {{num}} is :{{squareofnum}}</h2>  
  
</body>  
  
</html>
```

Step 3: Here, we have written a view function, called ‘squarenumber’. The view function returns an HTTP response. The function is decorated with ‘app.route('/square’). This decorator, matches the incoming request URLs, to the view functions. Thus, incoming requests like ‘localhost:5000/’ will be mapped to the view squarenumber() function.

In this case, we have used a GET, hence we mention ‘methods=[‘GET’]’ in the app.route decorator. The HTML form will append the number, entered by the user, in the URL. Hence, we check if data is present. To do so, we have to use the if-elif-else logic. When data is not present, the value of argument “num” will be None. Here, we will display, the webpage to the user. If the user has not entered, any number, and right away clicked the Submit button, then, we have displayed the error message. We have to

use the Flask Jinja2 template, to pass the result, and, the number entered into the HTML file.

- Python3

```
# import the Flask library

from flask import Flask, render_template, request

# Create the Flask instance and pass the Flask
# constructor the path of the correct module

app = Flask(__name__)

# The URL 'localhost:5000/square' is mapped to
# view function 'squarenumber'

# The GET request will display the user to enter
# a number (coming from squarenum.html page)

@app.route('/', methods=['GET'])

def squarenumber():

    # If method is GET, check if number is entered
    # or user has just requested the page.

    # Calculate the square of number and pass it to
    # answermaths method

    if request.method == 'GET':

        # If 'num' is None, the user has requested page the first time

        if(request.args.get('num') == None):

            return render_template('squarenum.html')
```

```

# If user clicks on Submit button without
# entering number display error

elif(request.args.get('num') == ""):

    return "<html><body> <h1>Invalid number</h1></body></html>"

else:

    # User has entered a number

    # Fetch the number from args attribute of

    # request accessing its 'id' from HTML

        number = request.args.get('num')

        sq = int(number) * int(number)

        # pass the result to the answer HTML

        # page using Jinja2 template

            return render_template('answer.html',

```

```
# Start with flask web app with debug as
```

```
# True only if this is the starting page
```

```
if(__name__ == "__main__"):
```

```
    app.run(debug=True)
```

Output:

```

127.0.0.1 - - [12/Dec/2022 13:18:55] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:18:56] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:18:59] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:19:15] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:19:28] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:19:42] "POST / HTTP/1.1" 200 -

```



The user requests 'localhost:5000/square' and enters a number

On clicking the Submit button, one can notice, the value entered, appended, to the URL, and, the output displayed.



Data entered is appended in the URL and output is displayed.

POST Method in Flask

Suppose, we need to register our details, to a website, OR, upload our files, we will send data, from our browser(the client) to the desired server. The HTTP method, preferred here, is POST. The data sent, from HTML, is then saved, on the server side, post validation. The POST method should be used, when we need to change/add data, on the server side.

Example of HTTP POST in Flask

In this example, we will consider, the same landing page, giving us facts, about Math calculations, and, allowing us to enter a number, and, return its square. Let us see the example:

Step 1: The same HTML page, called '**squarenum.html**', is in the templates folder. At the backend side, we will write, appropriate logic, in a view function, to get the number, entered by the user, and, return the same, in the '**answer.html**' template. The frontend code file is as shown below:

- HTML

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Square Of Number!</title>
</head>
<body>
    <h1><i> Welcome to the Maths page!</i></h1>
    <p>Logic shapes every choice of our daily lives.<br>
        Logical thinking enables someone to learn and
        make decisions that affect their way of life. !</p>
    <form method="POST" action ="#">
        Enter a number :
        <input type="text" name="num" id="num"></input>
        <input type="submit" name="bttnum" id="bttnum"></input>
    </form>
</body>
</html>
```

Step 2: The view function `squarenumber()`, now also contains value POST in the ‘methods’ attribute, in the decorator. Thus, when the user requests the page, the first time, by calling “`http://localhost:5000/square`”, a GET request will be made. Here, the server will render, the corresponding “`squarenum.html`”, webpage. After clicking on Submit, on entering a number, the data is posted back, to the same webpage. Here, the POST method, sends data, in the message body, unlike GET, which appends data in the URL. In the view function,

the If-Else condition block, retrieves the number, by accessing the ‘form’ attribute, of the request. The square of the number is calculated, and, the value is passed to the same “answer.html” webpage, using the Jinja2 template.

- Python3

```
# import the Flask library

from flask import Flask, render_template, request


# Create the Flask instance and pass the Flask constructor the path of the correct module

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])

def squarenumber():

    # If method is POST, get the number entered by user

    # Calculate the square of number and pass it to answermaths

    if request.method == 'POST':

        if(request.form['num'] == ""):

            return "<html><body> <h1>Invalid number</h1></body></html>"

        else:

            number = request.form['num']

            sq = int(number) * int(number)

            return render_template('answer.html',

                                  squareofnum=sq, num=number)

    # If the method is GET, render the HTML page to the user

    if request.method == 'GET':
```

```

        return render_template("squarenum.html")

# Start with flask web app with debug as True only

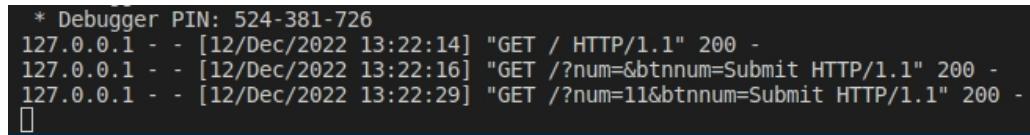
# if this is the starting page

if(__name__ == "__main__"):

    app.run(debug=True)

```

Output:



```

* Debugger PIN: 524-381-726
127.0.0.1 - - [12/Dec/2022 13:22:14] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:22:16] "GET /?num=&bttnum=Submit HTTP/1.1" 200 -
127.0.0.1 - - [12/Dec/2022 13:22:29] "GET /?num=11&bttnum=Submit HTTP/1.1" 200 -

```



The user enters a number when the page is rendered for URL 'localhost:5000/square'

On clicking the Submit button, the data is posted back, to the server, and, the result page is rendered. Please note, the value entered, is not visible in the URL now, as the method used, is POST.



CHAPTER 5: Flask - Variable Rule

In this article, we will discuss Flask-Variable Rule using Python. Flask is a microweb framework written in Python it is classified as a micro framework because it doesn't require particular tools or libraries, and it has no database abstraction layer form validation or any other components where pre-existing third-party libraries provide common functions it is designed to make getting started quick and easy with the ability to scale up complex application.

Flask -Variable Rules:

- By using variable rules we can create a dynamic URL by adding variable parts, to the rule parameter.
- We can define variable rule by <variable-name> using this syntax in our code.
- Variable is always passed as a keyword argument to the function with which the rule is properly associated.

Dynamic URLs Variable In Flask

The following converters are available in Flask-Variable Rules:

- **String** – It accepts any text without a slash(the default).
- **int** – accepts only integers. ex =23
- **float** – like int but for floating point values ex. = 23.9
- **path** – like the default but also accepts slashes.
- **any** – matches one of the items provided.
- **UUID** = accepts UUID strings.

Simple flask program

In this code, we import the first Flask and then we initialize the Flask function and then we return the simple welcome line we need to run the Flask run command in the terminal to execute our code.

- Python3

```
# first we import flask
```

```
from flask import Flask

# Initialize flask function

app = Flask(__name__)

@app.route('/')

def msg():

    return "Welcome To The GreeksForGreeks"

# we run code in debug mode

app.run(debug=True)
```

Output:



String Variable in Flask

In this code, we import the first Flask and then we initialize the Flask function after that we made a string function string is already defined in the code we don't need to specify the string and then we execute our code by a run in the terminal Flask run. and in URL we need to write **127.0.0.1:5000/vstring/<name>** to run our vstring function.

- Python3

```
# First we import flask
```

```
from flask import Flask

# initialize flask

app = Flask(__name__)

# Display first simple welcome msg

@app.route('/')

def msg():

    return "Welcome"

# We defined string function

@app.route('/vstring/<name>')

def string(name):

    return "My Name is %s" % name

# we run app debugging mode

app.run(debug=True)
```

Output :



Integer Variable in Flask

In this code, we import the first Flask, and then we initialize the Flask function after that we made an int function, int is not defined in a Flask so we need to first define it then we return the int function and run Flask run in terminal and for int page, we need to write function name in URL and also the integer value.

- Python3

```
# first we import flask

from flask import Flask

# Initialize flask function

app = Flask(__name__)

@app.route('/')

def msg():

    return "Welcome"

# define int function

@app.route('/vint/<int:age>')

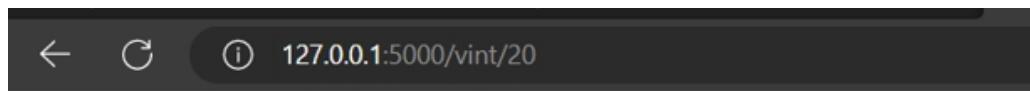
def vint(age):

    return "I am %d years old " % age

# we run our code in debug mode

app.run(debug=True)
```

Output:



I am 20 years old

Float Variable in Flask

In this code, we import the first Flask and then we initialize the Flask function after that we made a float function float is not defined in a Flask so we need to first define it then return the float function and run the Flask run in the terminal and for float page, we need to write function name in URL and also the floating value.

- Python3

```
# first we import flask

from flask import Flask

# Initialize flask function

app = Flask(__name__)

@app.route('/')

def msg():

    return "Welcome"

# define float function

@app.route('/vfloat/<float:balance>')

def vfloat(balance):

    return "My Account Balance %f" % balance
```

```
# we run our code in debugging mode  
app.run(debug=True)
```

Output:



A screenshot of a mobile browser interface. At the top, there are navigation icons: a left arrow, a refresh symbol, and a circular info icon followed by the URL "127.0.0.1:5000/vfloat/2000.2". Below the header, the main content area displays the text "My Account Balance 2000.200000".

My Account Balance 2000.200000

CHAPTER 6: Redirecting to URL in Flask

Flask is a backend server that is built entirely using Python. It is a framework that consists of Python packages and modules. It is lightweight which makes developing backend applications quicker with its features. In this article, we will learn to redirect a URL in the Flask web application.

Redirect to a URL in Flask

A redirect is used in the Flask class to send the user to a particular URL with the status code. conversely, this status code additionally identifies the issue. When we access a website, our browser sends a request to the server, and the server replies with what is known as the HTTP status code, which is a three-digit number.

Syntax of Redirect in Flask

Syntax: `flask.redirect(location,code=302)`

Parameters:

- **location(str):** the location which URL directs to.
- **code(int):** The status code for Redirect.
- **Code:** The default code is 302 which means that the move is only temporary.

Return: The response object and redirects the user to another target location with the specified code.

The different types of HTTP codes are:

Cod e	Status
300	Multiple_choices

Code	Status
301	Moved_permanently
302	Found
303	See_other
304	Not_modified
305	Use_proxy
306	Reserved
307	Temporary_redirect

How To Redirect To Url in Flask

In this example, we have created a sample flask application **app.py**. It has two routes, One is the base route \ and another is \helloworld route. Here when a user hits the base URL \ (**root**) it will redirect to \helloworld.

app.py

- Python3

```
# import flask module
from flask import Flask, redirect

# instance of flask application
app = Flask(__name__)

# home route that redirects to
```

```

# helloworld page

@app.route("/")
def home():

    return redirect("/helloworld")

# route that returns hello world text

@app.route("/helloworld")

def hello_world():

    return "<p>Hello, World from \
            redirected page.!</p>"


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

```

Output:

We hit the base URL in the browser as shown below. As soon we hit the URL flask application returns the redirect function and redirects to the given URL.

```

((test) ) C:\Users\suraj\Desktop\flask\test>python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 115-625-310
127.0.0.1 - - [25/Oct/2022 20:45:11] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [25/Oct/2022 20:45:11] "GET /helloworld HTTP/1.1" 200 -

```



Hello, World from redirected page..!

url_for() Function in Flask

Another method you can use when performing redirects in Flask is the `url_for()` function URL building. This function accepts the name of the function as the first argument, and the function named `user(name)` accepts the value through the input URL. It checks whether the received argument matches the 'admin' argument or not. If it matches, then the function `hello_admin()` is called else the `hello_guest()` is called.

- Python3

```
from flask import Flask, redirect, url_for

app = Flask(__name__)

# decorator for route(argument) function
@app.route('/admin')
# binding to hello_admin call
def hello_admin():

    return 'Hello Admin'

@app.route('/guest/<guest>')
# binding to hello_guest call
def hello_guest(guest):
```

```
return 'Hello %s as Guest' % guest

@app.route('/user/<name>')

def hello_user(name):

    # dynamic binding of URL to function

    if name == 'admin':

        return redirect(url_for('hello_admin'))

    else:

        return redirect(url_for('hello_guest'

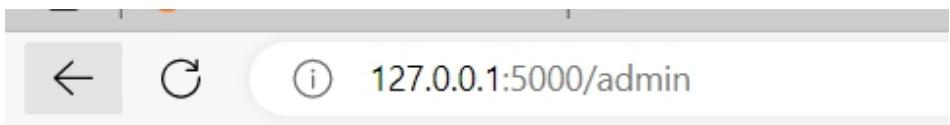
                               , guest=name))

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

Output:



hello_guest function



hello_user function

CHAPTER 7: Python Flask - Redirect and Errors

We'll discuss redirects and errors with Python Flask in this article. A redirect is used in the Flask class to send the user to a particular URL with the status code. conversely, this status code additionally identifies the issue. When we access a website, our browser sends a request to the server, and the server replies with what is known as the HTTP status code, which is a three-digit number, The different reasons for errors are:

- Unauthorized access or poor request.
- Unsupported media file types.
- Overload of the backend server.
- Internal hardware/connection error.

Syntax of Redirect

`flask.redirect(location,code=302)`

Parameters:

- **location(str):** the location which URL directs to.
- **code(int):** The status code for Redirect.
- **Code:** The default code is 302 which means that the move is only temporary.

Return: The response object and redirects the user to another target location with the specified code.

The different types of HTTP codes are:

Code	Status
300	Multiple_choices

Code	Status
301	Moved_permanently
302	Found
303	See_other
304	Not_modified
305	Use_proxy
306	Reserved
307	Temporary_redirect

Import the redirect attribute

We can redirect the URL using Flask by importing **redirect**. Let's see how to import redirects from Flask

- Python3

```
from flask import redirect
```

Example:

Let's take a simple example to redirect the URL using a flask named **app.py**.

- Python3

```
# importing redirect

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

```
# Initialize the flask application

app = Flask(__name__)

# It will load the form template which
# is in login.html

@app.route('/')

def index():

    return render_template("login.html")



@app.route('/success')

def success():

    return "logged in successfully"



# loggnig to the form with method POST or GET

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

def login():

    # if the method is POST and Username is admin then
    # it will redirects to success url.

    if request.method == "POST" and request.form["username"] == "admin":

        return redirect(url_for("success"))




    # if the method is GET or username is not admin,
    # then it redirects to index method.

    return redirect(url_for('index'))
```

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

login.html

Create a folder named templates. In the templates, creates a templates/**login.html** file. On this page, we will take input from the user of its username.

- HTML

```
<!DOCTYPE html>

<html lang="en">

<body>

    <form method="POST", action="\login">

        Username: <input name=username type=text></input><br>

        <button type="submit">Login</button>

    </form>

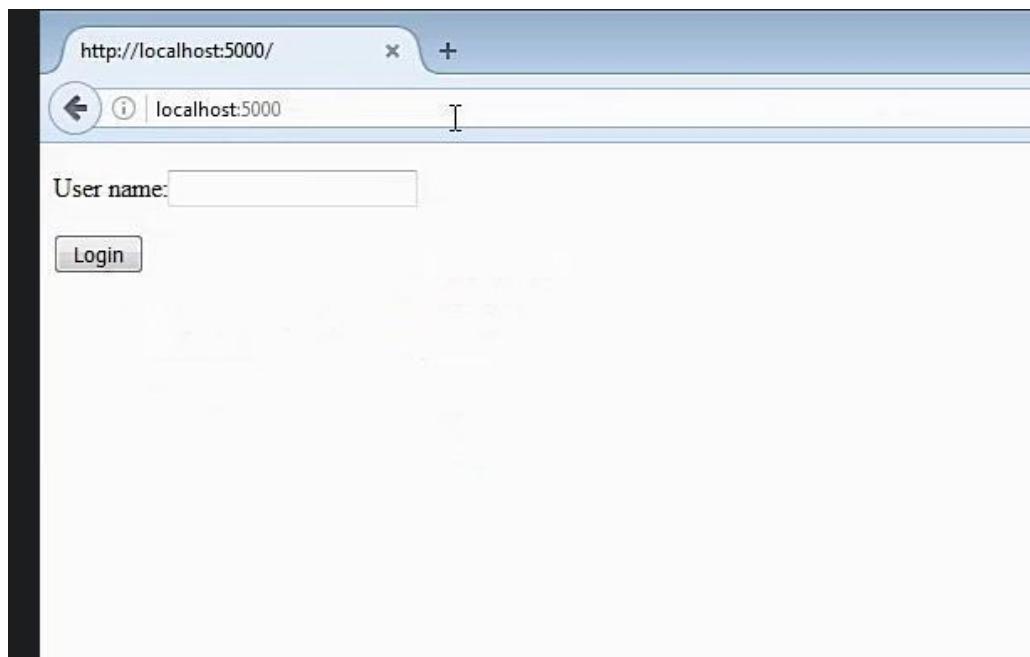
</body>

</html>
```

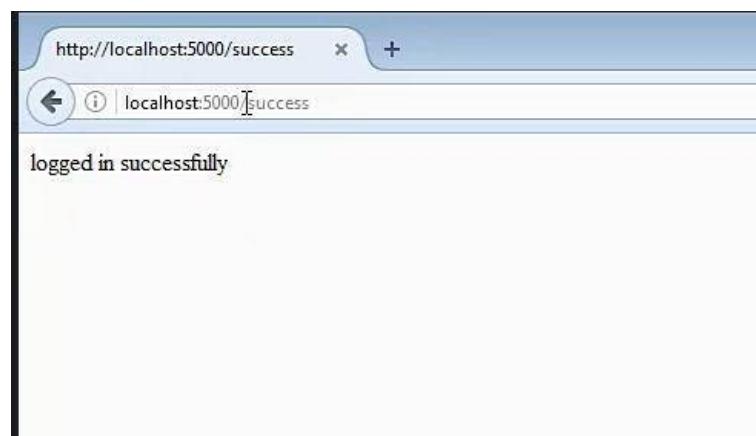
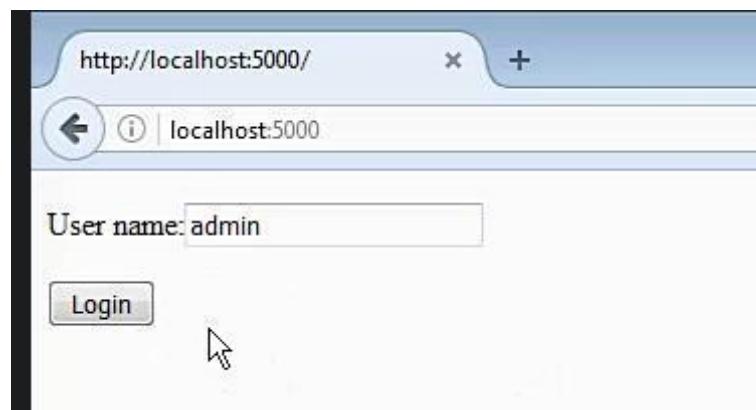
Now we will deploy our code.

```
python app.py
```

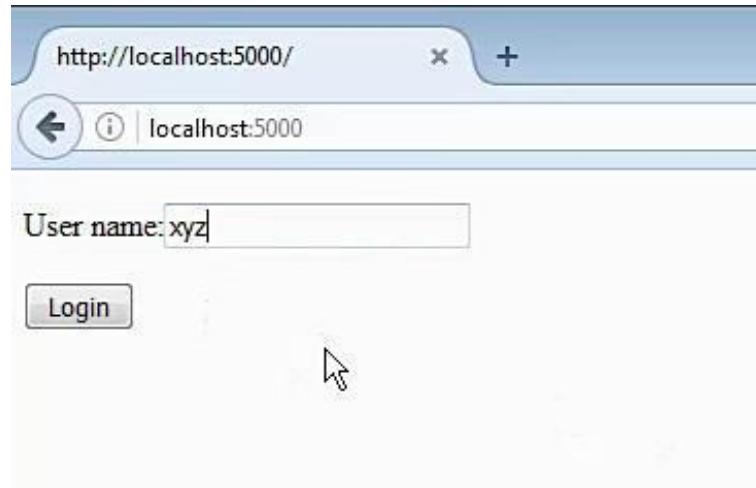
Output:



Case 1: If the Username is **admin** and the method is **POST** then it will redirect to the success URL and display logged in successfully.



Case 2: In the other case, if the Username is something like xyz or anything then it will redirect to the **index** method i.e., it will load the form again until the username is admin.



Flasks Errors

If there is an error in the address or if there is no such URL then Flask has an **abort()** function used to exit with an error code.

Syntax of abort() method

Syntax: `abort(code, message)`

- **code:** int, The code parameter takes any of the following values
- **message:** str, create your custom message Error.

The different types of errors we can abort in the application in your Flask.

Code	Error
400	Bad request
401	Unauthenticated
403	Forbidden
404	Not Found

Code	Error
406	Not Acceptable
415	Unsupported Media Type
429	Too Many Requests

Example to demonstrate abort

In this example, if the username starts with a number then an error code message will through, else on success “Good username” will be printed.

Example 1:

- Python3

```
# importing abort

from flask import Flask, abort

# Initialize the flask application

app = Flask(__name__)

@app.route('/<uname>')

def index(uname):

    if uname[0].isdigit():

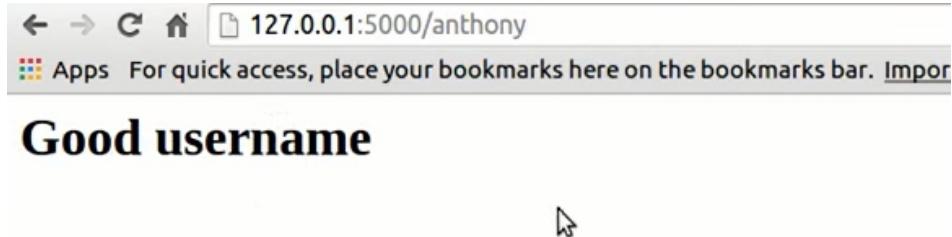
        abort(400)

    return '<h1>Good Username</h1>'
```

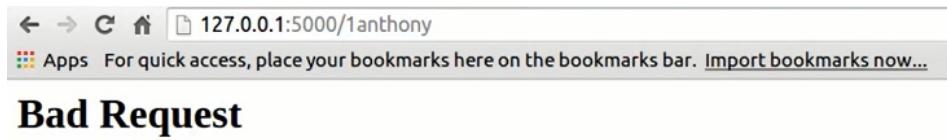
```
if __name__ == '__main__':
    app.run()
```

Output:

Case 1: If the username doesn't start with a number.



Case 2: If the username starts with a number.



Example 2:

In the above Python code, the status code to the abort() is **400**, so it raises a Bad Request Error. Let's try to change the code to **403**. If the username starts with a number then it raises a Forbidden Error.

- Python3

```
# importing abort

from flask import Flask, abort

# Initialize the flask application

app = Flask(__name__)

@app.route('/<uname>')

def index(uname):

    if uname[0].isdigit():

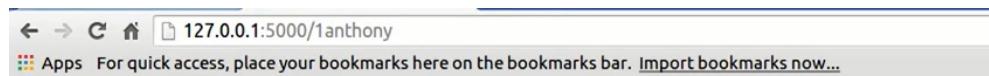
        abort(403)

    return '<h1>Good Username</h1>'

if __name__ == '__main__':

    app.run()
```

Output:



Forbidden

You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.

CHAPTER 8: Change Port in Flask app

In this article, we will learn to change the port of a [Flask](#) application. The default port for the Flask application is 5000. So we can access our application at the below URL.

`http://127.0.0.1:5000/`

We may want to change the port may be because the default port is already occupied. To do that we just need to provide the port while running the Flask application. We can use the below command to run the Flask application with a given port.

```
if __name__ == '__main__':
    app.run(debug=True, port=port_number)
```

In this example, we will be using a sample flask application that returns a text when we hit the root URL.

helloworld.py

- Python3

```
# import flask module

from flask import Flask


# instance of flask application

app = Flask(__name__)

# home route that returns below text

# when root url is accessed

@app.route("/")
```

```
def hello_world():

    return "<p>Hello, World!</p>"


if __name__ == '__main__':
    app.run(debug=True, port=8001)
```

Output:

We are running the flask application on **port 8001**.

```
((test)) C:\Users\suraj\Desktop\flask\test>python app.py
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:8001 (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-625-310
127.0.0.1 - - [25/Oct/2022 20:31:01] "GET / HTTP/1.1" 200 -
```



CHAPTER 9: Changing Host IP Address in Flask

In this article, we will cover how we can change the host IP address in Flask using Python. The host IP address is the network address that identifies a device on a network. In the context of a Flask application, the host IP address is the address that the application listens to for incoming requests. By default, Flask applications listen on the localhost address **127.0.0.1:5000**, which means they can only be accessed from the same machine that the application is running on. However, it is often useful to be able to access the application from other devices on the same network, or even from the internet. In these cases, the host IP address can be changed to allow the application to be accessed from other devices. This is done by specifying the host IP address in the app.run() function of the Flask application.

Changing the IP address in a Flask application using the “host” parameter

Here are the steps for changing the host IP address in a Flask application using the “host” parameter in the app.run() function. Open the Flask application in a text editor. Locate app.run() function in the main script file. Add the “host” parameter to the app.run() function, followed by the desired IP address, such as: app.run(host='192.168.0.105') this is the local address of your system. This will only allow the application to be accessed from the specified IP address.

- Python3

```
from flask import Flask

app = Flask(__name__)
```

```

@app.route('/')
def hello():

    return 'Hello, World! this application runing on 192.168.0.105'

if __name__ == '__main__':
    app.run(host='192.168.0.105')

```

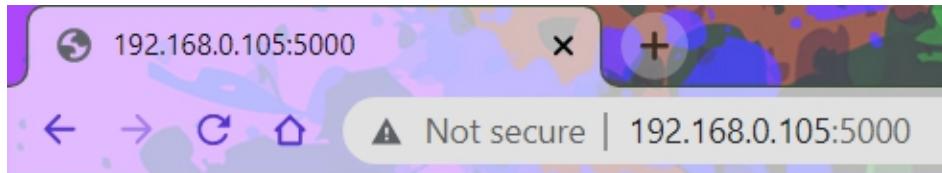
Output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Python + × ⓘ ^ ×

PS C:\Users\Lenovo\Desktop\Flask> & C:/Users/Lenovo/AppData/Local/Programs/Python/Python311/python.exe c:/Users/Lenovo/Desktop/p/Flask/host.py
* Serving Flask app 'host'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.0.107:5000
Press CTRL+C to quit

```



Hello, World! this application runing on 192.168.0.105

Changing IP from the command line while deploying the Flask app

Here, the `app.run()` function does not specify an IP address or a port, so it will use the defaults of `localhost` (`127.0.0.1`) and port `5000`. You can run this application by setting the `FLASK_APP` environment variable to the name of your application file and then using the `flask run` command and then you can change the IP address and port number while running the command

```
set FLASK_APP=app.py  
flask run  
flask run --host=192.168.0.105 --port=5000
```

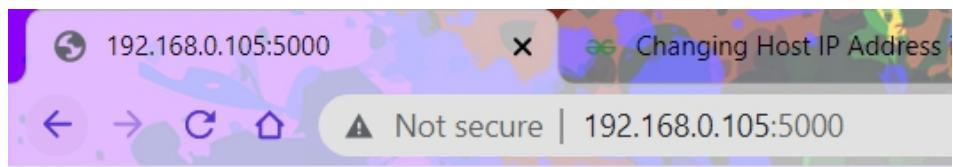
- Python3

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/')  
def hello():  
    return 'Hello, World!'  
  
if __name__ == '__main__':  
    app.run()
```

Output:

This will run the server on IP address 192.168.0.105 and port 5000.

```
C:\Users\Lenovo\Desktop\Flask>set FLASK_APP=app.py  
C:\Users\Lenovo\Desktop\Flask>flask run --host=192.168.0.105 --port=5000  
* Serving Flask app 'app.py'  
* Debug mode: off  
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.  
* Running on http://192.168.0.105:5000  
Press CTRL+C to quit
```



Hello, World!

PART 3: Serve Templates and Static Files in Flask

CHAPTER 1: Flask Rendering Templates

Flask is a backend web framework based on the Python programming language. It basically allows the creation of web applications in a Pythonic syntax and concepts. With Flask, we can use Python libraries and tools in our web applications. Using Flask we can set up a web server to load up some basic HTML templates along with Jinja2 templating syntax. In this article, we will see how we can render the HTML templates in Flask.

Rendering a Template in a Flask Application

Setting up Flask is quite easy. We can use a virtual environment to create an isolated environment for our project and then install the Python packages in that environment. After that, we set up the environment variables for running Flask on the local machine. This tutorial assumes that you have a Python environment configured, if not please follow through for setting up Python and pip on your system. Once you are done, you are ready to develop Flask applications.

Setting up the Virtual Environment

To set up a virtual environment, we can make use of the Python Package Manager “pip” to install the “virtualenv” package.

```
pip install virtualenv
```

This will install the package “virtualenv” on your machine. The pip command can be different on the version of your Python installed so please do look at the different syntax of the pip for your version here.

Creating Virtual Environment:

After the package has been installed we need to create a virtual environment in our project folder. So you can locate an empty folder where you want to create the Flask application or create an empty folder in your desired path. To create the environment we simply use the following command.

```
virtualenv venv
```

Here, venv is the name of the environment, after this command has been executed, you will see a folder named “venv” in the current folder. The name “venv” can be anything(“env”) you like but it is standard to reference a virtual environment at a production level.

Activating Virtual Environment:

Now after the virtual env has been set up and created, we can activate it by using the commands in CMD\Powershell or Terminal:

Note: You need to be in the same folder as the “venv” folder.

For Windows:

```
venv\Scripts\activate
```

For Linux/macOS:

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

This should activate the virtualenv with “(venv)” before the command prompt.

```

Volume in drive D has no label.
Volume Serial Number is D288-09FA

Directory of D:\meet\Code\test-server

10/26/2021  06:57 PM    <DIR>      .
10/26/2021  06:57 PM    <DIR>      ..
0 File(s)          0 bytes
2 Dir(s)  309,950,869,504 bytes free

D:\meet\Code\test-server>pip install virtualenv
Requirement already satisfied: virtualenv in d:\code\python-3.7.6\lib\site-packages (20.7.2)
Requirement already satisfied: distlib<1,>=0.3.1 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (0.3.2)
Requirement already satisfied: backports.entry-points-selectable>=1.0.4 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (1.1.0)
Requirement already satisfied: filelock<4,>=3.0.0 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (3.0.12)
Requirement already satisfied: platformdirs<3,>=2 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (2.2.0)
Requirement already satisfied: importlib-metadata>=0.12 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (4.6.4)
Requirement already satisfied: six<2,>=1.9.0 in d:\code\python-3.7.6\lib\site-packages (from virtualenv) (1.16.0)
Requirement already satisfied: typing-extensions>=3.6.4 in d:\code\python-3.7.6\lib\site-packages (from importlib-metadata) (3.10.0.0)
Requirement already satisfied: zipp>0.5 in d:\code\python-3.7.6\lib\site-packages (from importlib-metadata) (0.12->virtualenv) (3.5.0)
WARNING: You are using pip version 21.2.4; however, version 21.3 is available.
You should consider upgrading via the 'D:\Code\python-3.7.6\python.exe -m pip install --upgrade pip' command.

D:\meet\Code\test-server>virtualenv venv
created virtual environment CPython3.7.6.final.0-32 in 724ms
  creator CPython3Windows(dest=D:\meet\Code\test-server\venv, clear=False, no_vc
s_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle
, via=copy, app_data_dir=C:\Users\Admin\AppData\Local\pypa\virtualenv)
    added seed packages: pip==21.2.3, setuptools==57.4.0, wheel==0.37.0
  activators BashActivator, BatchActivator, FishActivator, PowerShellActivator, PythonActivator

D:\meet\Code\test-server>dir
 Volume in drive D has no label.
 Volume Serial Number is D288-09FA

 Directory of D:\meet\Code\test-server

10/26/2021  06:59 PM    <DIR>      .
10/26/2021  06:59 PM    <DIR>      ..
10/26/2021  06:59 PM    <DIR>      venv
          0 File(s)          0 bytes
          3 Dir(s)  309,939,281,920 bytes free

D:\meet\Code\test-server>venv\Scripts\activate

(venv) D:\meet\Code\test-server>

```

Screenshot of the entire virtualenv setup

As we can see we have successfully created the virtualenv in Windows Operating System, in Linux/macOS the process is quite similar. The (venv) is indicating the current instance of the terminal/CMD is in a virtual environment, anything installed in the current instance of a terminal using pip will be stored in the venv folder without affecting the entire system.

Installing Flask:

After the virtual environment has been set up, we can simply install Flask with the following command:

`pip install flask`

This should install the actual Flask Python package in the virtual environment.

Adding Flask to Environment Variables: We need to create an app for Flask to set it as the starting point of our application. We can achieve this by creating a file called “**server.py**” You can call this anything you like, but keep it consistent with other Flask projects you create. Inside the server.py paste the following code:

- Python

```
from flask import Flask

app = Flask(__name__)

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

This is the code for actually running and creating the Flask app. This is so-called the entry point of a Flask web server. As you can see we are importing the Flask module and instantiating with the current file name in “Flask(__name__)”. Hence after the check, we are running a function called run().

After this, we need to set the file as the Flask app to the environment variable.

For Windows:

```
set FLASK_APP=server
```

For Linux/macOS:

```
export FLASK_APP=server
```

Now, this will set up the Flask starting point to that file we created, so once we start the server the Flask server will find the way to the file “server.py”

To run the server, enter the command :

```
flask run
```

This will run the server and how smartly it detected the server.py file as our actual flask app. If you go to the URL “<http://localhost:5000>”, you would see nothing than a Not Found message this is because we have not configured our web server to serve anything just yet. You can press **CTRL + C** to stop the server

```
<venv> D:\meet\Code\test-server>pip install flask
Collecting flask
  Using cached Flask-2.0.2-py3-none-any.whl (95 kB)
Collecting click>=7.1.2
  Using cached click-8.0.3-py3-none-any.whl (97 kB)
Collecting itsdangerous>=2.0
  Using cached itsdangerous-2.0.1-py3-none-any.whl (18 kB)
Collecting Werkzeug>=2.0
  Using cached Werkzeug-2.0.2-py3-none-any.whl (288 kB)
Collecting Jinja2>=3.0
  Using cached Jinja2-3.0.2-py3-none-any.whl (133 kB)
Collecting importlib-metadata
  Using cached importlib_metadata-4.8.1-py3-none-any.whl (17 kB)
Collecting colorama
  Using cached colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Using cached MarkupSafe-2.0.1-cp37-cp37m-win32.whl (14 kB)
Collecting zipp>=0.5
  Using cached zipp-3.6.0-py3-none-any.whl (5.3 kB)
Collecting typing-extensions>=3.6.4
  Using cached typing_extensions-3.10.0.2-py3-none-any.whl (26 kB)
Installing collected packages: zipp, typing-extensions, MarkupSafe, importlib-metadata, colorama, Werkzeug, Jinja2, itsdangerous, click, flask
Successfully installed Jinja2-3.0.2 MarkupSafe-2.0.1 Werkzeug-2.0.2 click-8.0.3 colorama-0.4.4 flask-2.0.2 importlib-metadata-4.8.1 itsdangerous-2.0.1 typing-extensions-3.10.0.2 zipp-3.6.0
WARNING: You are using pip version 21.2.3; however, version 21.3.1 is available.
You should consider upgrading via the 'D:\meet\Code\test-server\venv\Scripts\python.exe -m pip install --upgrade pip' command.

<venv> D:\meet\Code\test-server>set FLASK_APP=server
<venv> D:\meet\Code\test-server>flask run
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask set up for webserver

Creating Templates in a Flask Application

Now, we can move on to the goal of this article i.e. to render the template. To do that we need to first create the templates, you can use any HTML template but for simplicity, I am going with a basic HTML template. Before that, **create a folder called “templates” in the current folder**. Inside this “**templates**” folder, all of the templates will be residing. Now let us create a basic HTML template: This template

must have some Jinja blocks that can be optionally replaced later. We start with a single block called the body.

templates\index.html

- HTML

```
<!DOCTYPE html>
<html>

<head>
    <title>FlaskTest</title>
</head>

<body>
    <h2>Welcome To GFG</h2>
    <h4>Flask: Rendering Templates</h4>
    <!-- this section can be replaced by a child document -->
    {% block body %}

    <p>This is a Flask application.</p>

    {% endblock %}

</body>

</html>
```

Adding Routes and Rendering Templates

A route is a mapping of a URL with a function or any other piece of code to be rendered on the webserver. In the Flask, we use the function decorate @app.route to indicate that the function is bound with the URL provided in the parameter of the route function.

Creating the basic route: In this case, we are binding the **URL** “/” which is the base URL for the server with the **function “index”**, you can call it whatever you like but it makes more sense to call it index here. The function simply returns something here it calls the function render_template. The render_template finds the app by default in the templates folder. So, we just need to provide the name of the template instead of the entire path to the template. The index function renders a template index.html and hence we see the result in the browser. Now, we need a way to actually link the template with a specific route or URL. This means whenever the user goes to a specific URL then a specific template should be rendered or generated. Now, we need to change the “server.py” with the following:

- Python

```
from flask import Flask, render_template

app = Flask(__name__)

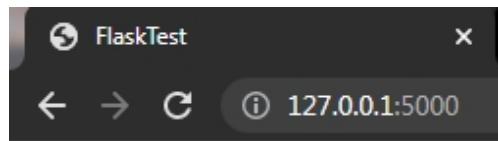
@app.route("/")
def index():

    return render_template("index.html")

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

Output:

We have imported the `render_template` function from the Flask module and added a route.



render the basic template

Templating With Jinja2 in Flask

Now, we'll create a new route for demonstrating the usage of the Jinja template. We need to add the route, so just add one more chunk of the code to the "server.py" file"

- Python

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

Now, this might look pretty easy to understand, we are simply creating a route "/<name>" which will be bound to the `welcome` function. The "<name>" is standing for anything after the "/". So we take that as the parameter to our function and pass it to the `render_template` function as `name`. So, after passing the variable `name` in the `render_template` function, it would be accessible in the template for us to render that

variable. You can even perform an operation on the variable and then parse it.

No, we need to create another template called “welcome.html” inside the template folder. This file should contain the following markup

- **HTML**

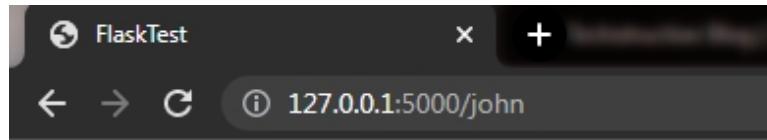
```
<!DOCTYPE html>

<html>

<head>
    <title>FlaskTest</title>
</head>

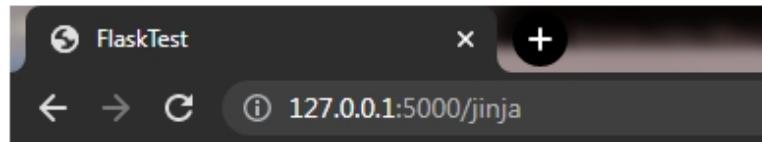
<body>
    <h2>Welcome To GFG</h2>
    <h3>Welcome, {{name}}</h3>
</body>

</html>
```



Welcome To GFG

Welcome, john



Welcome To GFG

Welcome, jinja

Using Jinja template

Flask - Jinja Template Inheritance Example

Now, we need a way to actually inherit some templates instead of reusing them, we can do that by creating the blocks in Jinja. They allow us to create a template block and we can use them in other templates with the name given to the block.

So, let us re-use our "index.html" and create a block in there. To do that we use "{% block <name> %}" (where name = 'body') to start the block, this will take everything above it and store it in a virtual block of template, to end the block you simply use "{% endblock %}" this will copy everything below it.

templates/index.html

- HTML

```

<!DOCTYPE html>

<html>
<head>
<title>FlaskTest</title>
</head>
<body>
<h2>Welcome To GFG</h2>
<h4>Flask: Rendering Templates</h4>
<a href="{{ url_for('home') }}>Home</a>
<a href="{{ url_for('index') }}>Index</a>
{% block body %}

<p>This is a Flask application.</p>

{% endblock %}
</body>
</html>

```

So, here we are not including the `<p>` tags as everything below the `{% endblock %}` and everything above the `{% block body %}` tag is copied. We are also using absolute URLs. The URLs are dynamic and quite easy to understand. We enclose them in “`{{ }}`” as part of the Jinja2 syntax. The `url_for` function reverses the entire URL for us, we just have to pass the name of the function as a string as a parameter to the function.

Now, we'll create another template to reuse this created block “body”, let's create the template “`home.html`” with the following contents:

templates/home.html

- HTML

```
{% extends 'index.html' %}

{% block body %}

<p> This is a home page</p>

{% endblock %}
```

This looks like a two-liner but will also **extend** (*not include*) the index.html. This is by using the **{% extends <file.html> %}** tags, they parse the block into the mentioned template. After this, we can add the things we want. If you use the **include** tag it will not put the replacement paragraph in the correct place on the index.html page. It will create an invalid HTML file, but since the browser is very forgiving you will not notice unless you look at the source generated. The body text must be properly nested.

Finally, the piece left here is the route to home.html, so let's create that as well. Let's add another route to the "server.py" file"

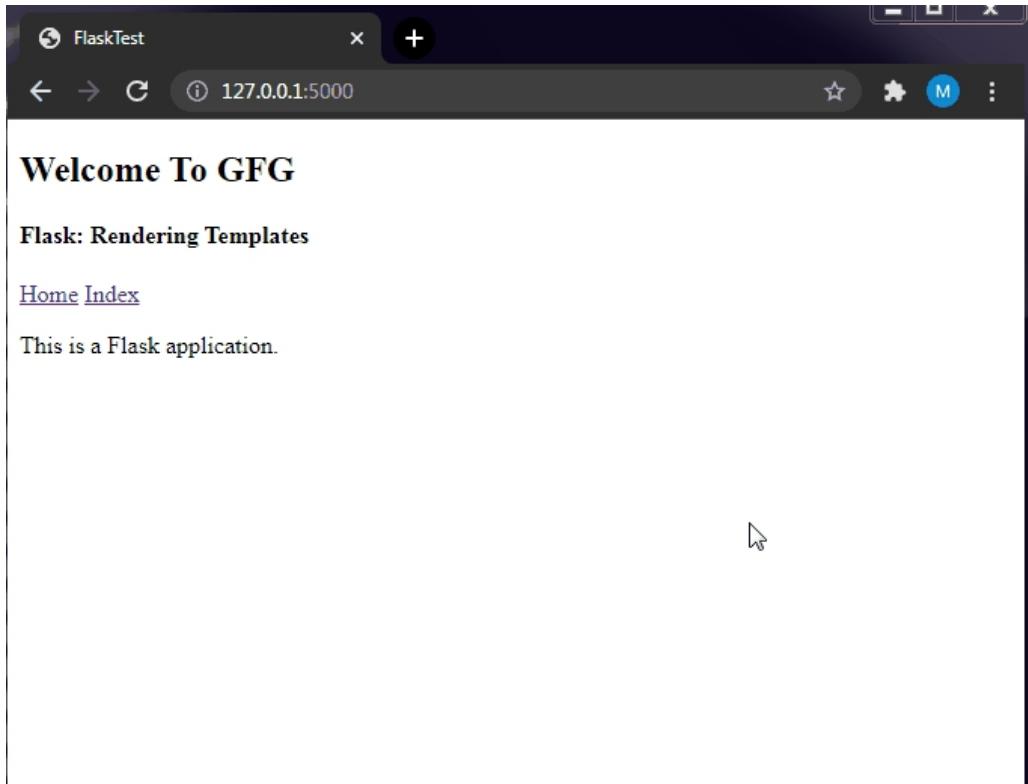
- Python

```
@app.route("/home")

def home():

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

So, this is a route bound to the "/home" URL with the home function that renders the template "home.html" that we created just right now.



Demonstrating block and URLs

As we can see the URL generated is dynamic, otherwise, we would have to hardcode both the template page paths. And also the block is working and inheriting the template as provided in the base templates. Open the page source in the browser to check it is properly formed HTML.

```
<!DOCTYPE html>
<html>
<head>
<title>FlaskTest</title>
</head>
<body>
<h2>Welcome To GFG</h2>
<h4>Flask: Rendering Templates</h4>
<a href="/home">Home</a>
<a href="/">Index</a>
<a href="/about">About</a>
<a href="/documentation">Documentation</a>
```

```
<p> This is a home page</p>
<p>must use extends not include</p>
</body>
</html>
```

Inducing Logic in Templates: We can use for loops if conditions in templates. this is such a great feature to leverage on. We can create some great dynamic templates without much of a hassle. Let us create a list in Python and try to render that on an HTML template.

Using for loops in templates: For that, we will create another route, this time at “/about”, this route will bind to the function about that renders the template “about.html” but we will add some more things before returning from the function. We will create a list of some dummy strings and then parse them to the render_template function.

- Python

```
@app.route("/about")

def about():

    sites = ['twitter', 'facebook', 'instagram', 'whatsapp']

    return render_template("about.html", sites=sites)
```

So, we have created the route at “/about” bound to the about function. Inside that function, we are first creating the list “Sites” with some dummy strings and finally while returning, we parse them to the render_template function as sites, you can call anything you like but remember to use that name in the templates. Now, to create the templates, we’ll create the template “about.html” with the following contents:

templates/about.html

- HTML

```
{% extends 'index.html' %}

{% block body %}

<ul>

    {% for social in sites %}

        <li>{{ social }}</li>

    {% endfor %}

</ul>

{% endblock %}
```

We can use for loops in templates enclosed in “{ % % }” we can call them in a regular Pythonic way. The sites are the variable(list) that we parsed in the route function. We can again use the iterator as a variable enclosed in “{{ }}”. This is like joining the puzzle pieces, the values of variables are accessed with “{{ }}”, and any other structures or blocks are enclosed in “{ % % }”.

Now to make it more accessible you can add its URL to the index.html like so:

- **HTML**

```
<!DOCTYPE html>

<html>

    <head>
        <title>FlaskTest</title>
    </head>

    <body>
        <h2>Welcome To GFG</h2>
        <h4>Flask: Rendering Templates</h4>
        <a href="{{ url_for('home') }}">Home</a>
```

```

<a href="{{ url_for('index') }}>Index</a>
<a href="{{ url_for('about') }}>About</a>
{% block body %}

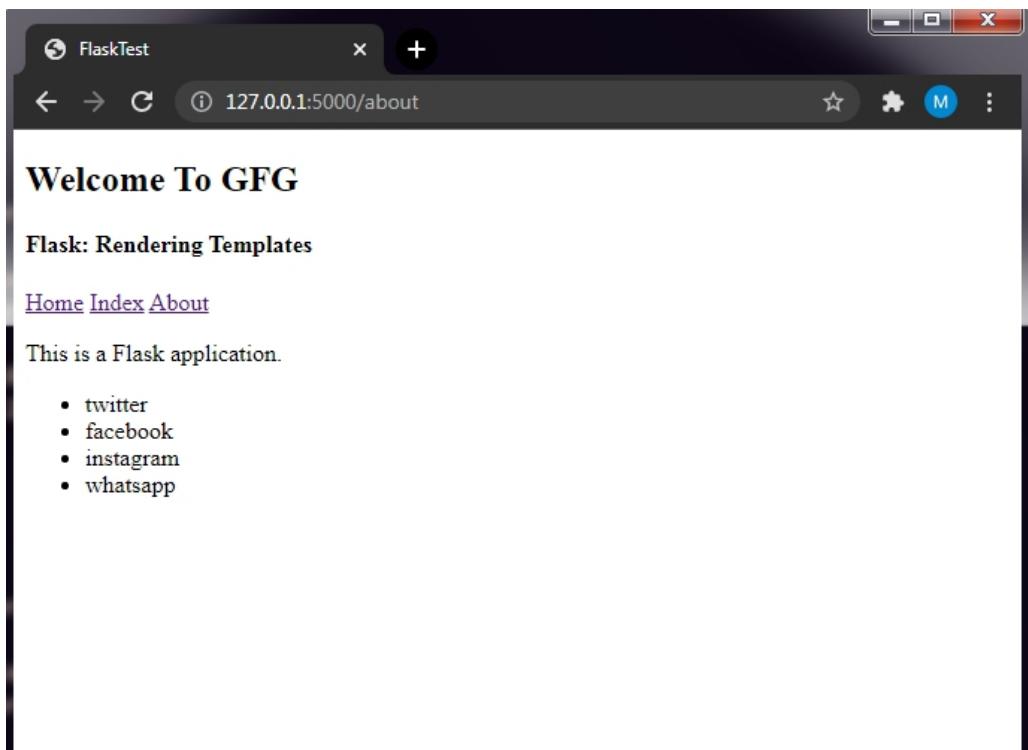
<p>This is a Flask application.</p>

{% endblock %}
</body>

</html>

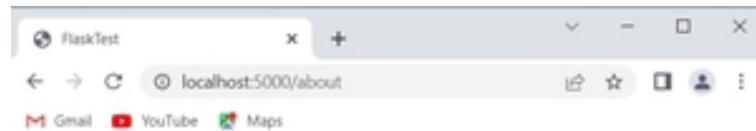
```

This is not mandatory but it creates an accessible link for ease.



Demonstrating for loop-in templates

As we can see it has dynamically created all the lists in the template. This can be used for fetching the data from the database if the app is production ready. Also, it can be used to create certain repetitive tasks or data which is very hard to do them manually.



Welcome To GFG

Flask: Rendering Templates

[Home](#) [Index](#) [About](#) [Documentation](#)

- twitter
- facebook
- instagram
- whatsapp
- documentation

Corrected extends file

This correctly defined extends file removed the placeholder paragraph and replaces it in the body of the HTML.

If statement in HTML Template in Python Flask

We can even use if-else conditions in flask templates. Similar to the syntax for the for loops we can leverage that to create dynamic templates. Let's see an example of a role for a website.

Let's build the route for the section contact. This URL is "contact/<role>", which is bound to the function contact which renders a template called "contacts.html". this takes in the argument as role. Now we can see some changes here, this is just semantic changes nothing new, we can use the variable person as a different name in the template which was assigned as the values of the role.

- Python

```
@app.route("/contact/<role>")  
def contact(role):
```

```
return render_template("contact.html", person=role)
```

So, this creates the route as desired and parses the variable role as a person to the template. Now let us create the template.

template/contact.html

- HTML

```
{% extends 'index.html' %}

{% block body %}

{% if person == "admin" %}

<p> Admin Section </p>




{% elif person == "maintainer" %}

<p> App Source Page for Maintainer</p>




{% elif person == "member" %}

<p> Hope you are enjoying our services</p>



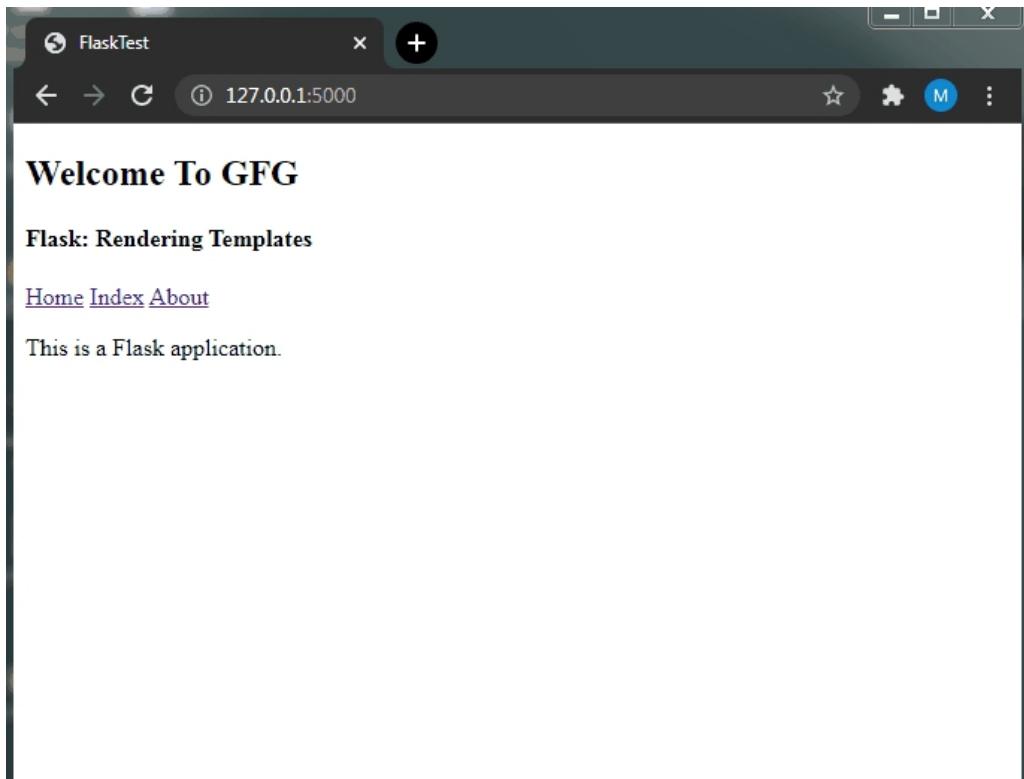

{% else %}
```

```
<p> Hello, {{ person }}</p>
```

```
{% endif %}
```

```
{% endblock %}
```

So, in the template, we are checking the value of the variable person which is obtained from the URL and parsed from the render_template function. The if-else syntax is similar to Python with just “{ % % }” enclosed. The code is quite self-explanatory as we create if-elif and else ladder, checking for a value and creating the HTML elements as per the requirement.



So, we can see that the template is rendering the contents as per the role variable passed in the URL. Don't try to create a URL link for this as it would not work since we need to enter the role variable manually. There needs to be some workaround done to use it.

So that was about using and rendering the templates in Flask. We have leveraged the Jinja templating syntax with Python to create some

dynamic templates.

CHAPTER 2: CSRF Protection in Flask

Let's see how you can manually protect your data using CSRF protection by doing a mini-project in Flask. In this, we will have to create a webpage containing 2 forms using Python one of them is having protection. By creating forms like these we can easily see the results and advantages of using CSRF protection for our application.

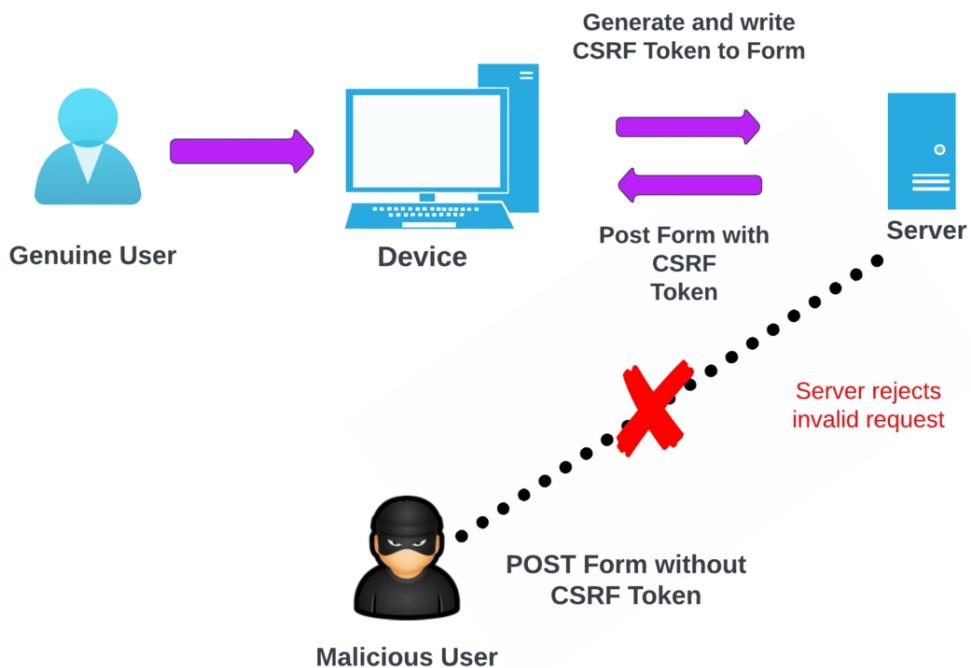
What is CSRF?

Cross-Site Request Forgery(CSRF) is a weighty exposure that results from weak gathering administration. If that requests shipped by an application aren't rare, it's likely for an aggressor to art a certain request and transmits that to a consumer. If the consumer communicates accompanying the workout request, and gatherings aren't controlled correctly, an aggressor grant permission within financial means to acquire the gathering similarity of that consumer and complete activity requests on their side.

Solution for Preventing CSRF Attacks

Cross-Site Request Forgery (CSRF) attacks are comparably smooth to diminish. One of the plainest habits to manage this is through the use of CSRF tokens, which are uncommon principles dynamically created by a server-side request and shipped to the customer. Since these principles are rare for each request, and uniformly changeful, it is almost hopeless for a raider to pre-generate the URLs/requests for an attack.

recreate image



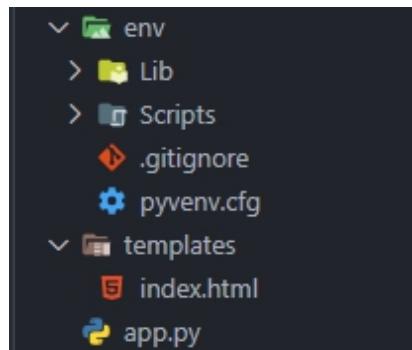
Example of CSRF Protection in Flask

Step 1: Create a Virtual environment for our application and install the following packages.

Step 2: Installing Packages.

pip install flask, flask-wtf

Step 3: You should have to create a folder structure like this.



Step 4: app.py

In Flask, we are having generally 2 ways to create a form one by using `FlaskForm` and another by creating forms manually. `FlaskForm` processes

the request that already getting CSRF Protection. Csrf requires a secret key by default, it uses the Flask app's Secret Key. If you like to set up a separate token then you can use `WTF_CSRF_SECRET_KEY` instead of using a flask app's secret key. While using `FlaskForm`, you will have to render the forms CSRF field.n You can disable the CSRF Protection in all views by default, then set `WTF_CSRF_CHECK_DEFAULT` to False in the `app.py` file.

- Python3

```
from flask import Flask, render_template, request

from flask_wtf import CSRFProtect

app = Flask(__name__)

app.secret_key = b'_53oi3uriq9pifpff;apl'

csrf = CSRFProtect(app)

@app.route("/protected_form", methods=['GET', 'POST'])

def protected_form():

    if request.method == 'POST':

        name = request.form['Name']

        return ('Hello ' + name + '!!!')

    return render_template('index.html')

@app.route("/unprotected_form", methods=['GET', 'POST'])

def unprotected_form():

    if request.method == 'POST':

        name = request.form['Name']
```

```
    return ('Hello ' + name + '!!!')

return render_template('index.html')

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

Step 5: templates/index.html

A simple HTML page is set up for the app to show the unprotected and protected submission of the form.

- HTML

```
<html>
<head></head>
<body>

<form action="{{ url_for('protected_form') }}" method="POST">

    <label for="Name">Your Name Please ? </label>

    <input type="text" name="Name">

    <input type="hidden" name="csrf_token" value = "{{ csrf_token() }}" />

    <button type="submit">Submit</button>

</form>

<form action="{{ url_for('unprotected_form') }}" method="POST">

    <label for="Name">Your Name Please ? </label>

    <input type="text" name="Name">
```

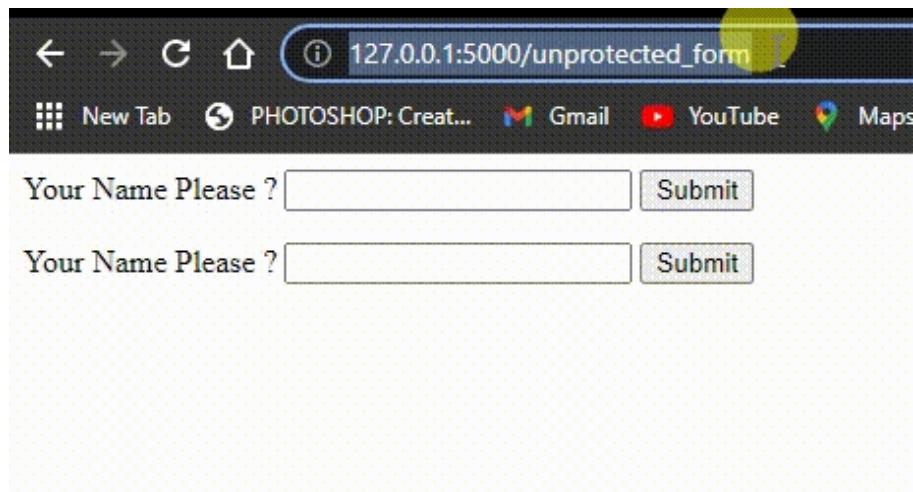
```
<button type="submit">Submit</button>  
</form>  
  
</body>  
</html>
```

Step 6: Now run it to see the webpage and perform the practice.

python app.py

Output:

Visit '**127.0.0.1:5000/protected_form**' and try submitting both forms and one by one you should get the following outputs. While submitting the first form we applied the token inside the form so that it checks the token if it presents it serves the request else it generates an error.



CHAPTER 3: Templating With Jinja2 in Flask

Flask is a lightweight WSGI framework that is built on Python programming. WSGI simply means Web Server Gateway Interface. Flask is widely used as a backend to develop a fully-fledged Website. And to make a sure website, templating is very important. Flask is supported by inbuilt template support named Jinja2. Jinja2 is one of the most used Web template engines for Python. This Web template engine is a fast, expressive, extensible templating engine. Jinja2 extensively helps to write Python code within the HTML file. Further, it also includes:

- Async support for generating templates that automatically handle sync and async functions without extra syntax.
- Template inheritance and inclusion.
- The Template engine makes debugging easier.
- Support of both High-level and Low-level API support.

Install the required package

To install the Jinja2 package in Python, check your latest pip version and stay updated. Install Jinja2 using the following command:

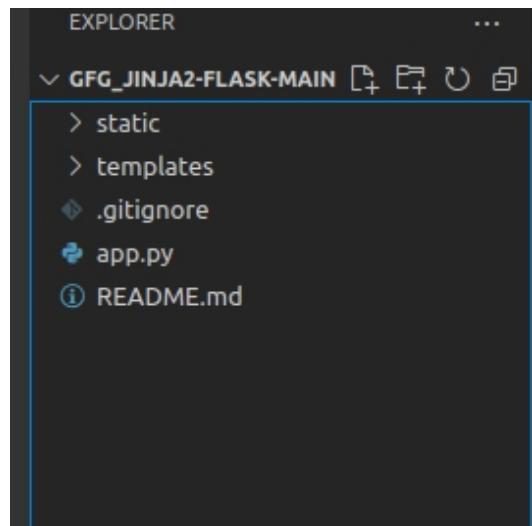
```
pip install Jinja2
```

But since we are dealing with the Templating with Jinja2 in Flask, there is no need to separately install Jinja2. When you install the Flask framework, the Jinja2 comes installed with it.

```
pip install flask
```

Templating with Jinja2 in Flask

Before we proceed with the coding part, this is how our project directory should look like:



Main Python File

Here is the common app.py file that interfaces with all the HTML files.

- Python3

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

app = Flask(__name__)

@app.route("/")
def home():

    return render_template("index.html")

@app.route("/default")
def default():

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

```
@app.route("/variable")

def var():

    user = "Geeksforgeeks"

    return render_template("variable_example.html", name=user)

@app.route("/if")

def ifelse():

    user = "Practice GeeksforGeeks"

    return render_template("if_example.html", name=user)

@app.route("/for")

def for_loop():

    list_of_courses = ['Java', 'Python', 'C++', 'MATLAB']

    return render_template("for_example.html", courses=list_of_courses)

@app.route("/choice/<pick>")

def choice(pick):

    if pick == 'variable':

        return redirect(url_for('var'))

    if pick == 'if':

        return redirect(url_for('ifelse'))

    if pick == 'for':

        return redirect(url_for('for_loop'))
```

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

Jinja Template Variables

To declare the variable using Jinja Template we use **{{variable_name}}** within the HTML file. As a result, the variable will be displayed on the Website.

Syntax of Jinja Template Variables

```
{{any_variable_name}}
```

variable_example.html

- HTML

```
<html>
    <head>
        <title>Variable Example</title>
    </head>
    <body>
        <h3>Hello {{name}}</h3>
    </body>
</html>
```

Output

Jinja Template if Statements

Just like declaring the variable in the Jinja2 template, if conditions have almost similar syntax. But here we specify the beginning and end of the if block.

Syntax of Jinja Template if Statements

```
{% if conditions %}
```

```
...
```

```
...
```

```
...
```

```
{% endif %}
```

Our app.py will stay the same. Just only one change instead of Geeksforgeeks tries giving Geeks so that we can verify the else block.

if_example.html

- HTML

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>If example</title>  
  </head>
```

```
<body>

{% if(name == "Geeksforgeeks") %}

    <h3> Welcome </h3>

{% else %}

    <h3> Unknown name entered: {{name}} </h3>

{% endif %}

</body>

</html>
```

Output:

Unknown name entered: Practice GeeksforGeeks

Jinja Template for Loop

Jinja for loop syntax is similar to the if statements, the only difference is for loop requires sequence to loop through.

Syntax of Jinja Template for Loops

```
{% for variable_name in sequence%}

...
...
...

{% endfor %}
```

for_example.html

- HTML

```
<!DOCTYPE html>

<html>

    <head>
        <title>For Example</title>
    </head>

    <body>
        <h2> Geeksforgeeks Available Course </h2>
        {% for course in courses%}
            <h4> {{course}} </h4>
        {% endfor %}
    </body>
</html>
```

Output:

Geeksforgeeks Available Course

Java

Python

C++

MATLAB

Jinja Template Inheritance

If you closely check the project files, you will find the index.html and layout.html. In this example, we gonna take look into Template

Inheritance. In most of the websites, if you notice, the footer and header remain the same, which means they share similar formats. In this example, layout.html will contain the default design that is common to all the pages, but here we will keep it specifically for index.html to understand how it works.

The syntax for layout.html contains the default text, along with the block contain, that will be inherited by other HTML files. You can think of layout.html as the parent and index.html as a child.

Syntax of Jinja Template Inheritance

layout.html

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

```
{% endblock %}
```

index.html

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

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

```
....
```

```
{% endblock %}
```

Example

In **layout.html** we define the top block and specify a template to insert block content that acts as parent HTML files.

- HTML

```
<!DOCTYPE html>

<html>

    <head>
        <title>Jinja2 and Flask</title>
    </head>
    <body>
        <h1>Welcome to Geeksforgeeks</h1>
        <h4>A Computer Science portal for geeks.</h4>
        {% block content %}
    
```

```
{% endblock %}

</body>

</html>
```

In **index.html** using the layout.html as the parent file, we shall derive all its content and add the block content to the existing HTML file.

Note: No need to define the HTML, head, and body tag in the child HTML file.

- HTML

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

{% block content %}

<ul>

    <li><a href="default"> Check Layout(Inheritance) </a></li>

    <li><a href="/variable"> Try Variable Example </a></li>

    <li><a href="/if"> Try If-else Example </a></li>

    <li><a href="/for"> Try For Example </a></li>

    <li><a href="/url"> Try URL Example </a></li>

</ul>

{% endblock %}
```

Output:
layout.html

Welcome to Geeksforgeeks

A Computer Science portal for geeks.

index.html

Welcome to Geeksforgeeks

A Computer Science portal for geeks.

- [Check Layout\(Inheritance\)](#)
- [Try Variable Example](#)
- [Try If-else Example](#)
- [Try For Example](#)
- [Try URL Example](#)

Jinja Template url_for Function

To build a dynamic website you need multiple re-direction within the website. url_for function is a very handy method that helps in re-direction from one page to another. url_for is also used to link HTML templates with static CSS or JavaScript files.

In our example since we have multiple choice for example, i.e., variable, if and for. Using url_for, we can create a custom function in which the user can alter the URL to get the specific result. For example, we shall define a function inside app.py and in example 2 we will take link HTML with CSS.

Syntax of Jinja Template url_for Function

```
url_for(function_name)
```

Example 1:

In the below example, if the user enters choice/<his choice> then it will redirect to that HTML file. Make sure redirect and url_for are imported.

- Python3

```
@app.route("/choice/<pick>")

def choice(pick):

    if pick == 'variable':

        return redirect(url_for('var'))

    if pick == 'if':

        return redirect(url_for('ifelse'))

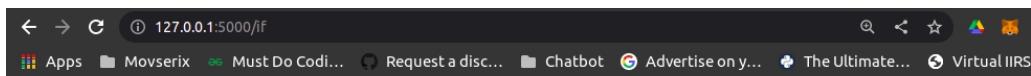
    if pick == 'for':

        return redirect(url_for('for_loop'))
```

Output:



- [Check Layout\(Inheritance\)](#)
- [Try Variable Example](#)
- [Try If-else Example](#)
- [Try For Example](#)
- [Try URL Example](#)



Unknown name entered: Practice GeeksforGeeks

Example 2:

In example 1 we used `url_for` inside a Python file. Now we shall use `url_for` inside the `layout.html` (parent file) HTML file, it will follow the variable define syntax i.e., to be enclosed within `{}{}`. Just like templates, create a static file for CSS.

```
{% url_for('static', filename='<path of the file>') %}
```

- **HTML**

```
<!DOCTYPE html>

<html>
  <head>
    <title>Template with Jinja2 and Flask</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}"/>
  </head>
  <body>
    <h1>Welcome to Geeksforgeeks</h1>
    <h4>A Computer Science portal for geeks.</h4>
    {% block content %}
    {% endblock %}
  </body>
</html>
```

```
</body>  
</html>
```

Output

Welcome to Geeksforgeeks

A Computer Science portal for geeks.

- [Check Layout\(Inheritance\)](#)
- [Try Variable Example](#)
- [Try If-else Example](#)
- [Try For Example](#)
- [Try URL Example](#)

CHAPTER 4: Placeholders in jinja2 Template

Web pages use HTML for the things that users see or interact with. But how do we show things from an external source or a controlling programming language like Python? To achieve this templating engine like Jinja2 is used. Jinja2 is a templating engine in which placeholders in the template allow writing code similar to Python syntax which after passing data renders the final document. In this article we will cover some of the points as mentioned below:

- Template Variables
- Template if Statements
- Template for Loops
- Template Inheritance

Let's start by creating a virtual environment. It's always a good idea to work in a virtual environment as it will not cause changes to the global system environment. For using Jinja2 in Python, we need to install the Jinja2 library.

```
pip install Jinja2
```

Template Variables in Jinja2

Jinja2 is a Python library that allows us to build expressive and extensible templates. It has special placeholders to serve dynamic data. A Jinja template file is a text file that does not have a particular extension.

Syntax of Template Variables in Jinja2

For a placeholder, we have the following syntax in Jinja2.

```
{variable_name}
```

Example

In an HTML file called **index_template.html**, write the following code.

- HTML

```
<!-- index_template.html -->

Hello {{pl_name}}! Your email is: {{pl_email}}
```

app.py

We open this HTML file in Python and read its content to a variable called `content`. Pass the content to `Template`, and store it in the `template` variable. Now, we will pass the name and email to render and replace the placeholders `{{pl_name}}` and `{{pl_email}}` respectively, by using `template.render`; and store this in `rendered_form`.

- Python3

```
# app.py

# import Template from jinja2 for passing the content
from jinja2 import Template

# variables that contain placeholder data
name = 'John'
email = 'you@example.co'
```

```

# Create one external form_template html page and read it

File = open('index_template.html', 'r')

content = File.read()

File.close()

# Render the template and pass the variables

template = Template(content)

rendered_form = template.render(pl_name=name, pl_email=email)

# save the txt file in the form.html

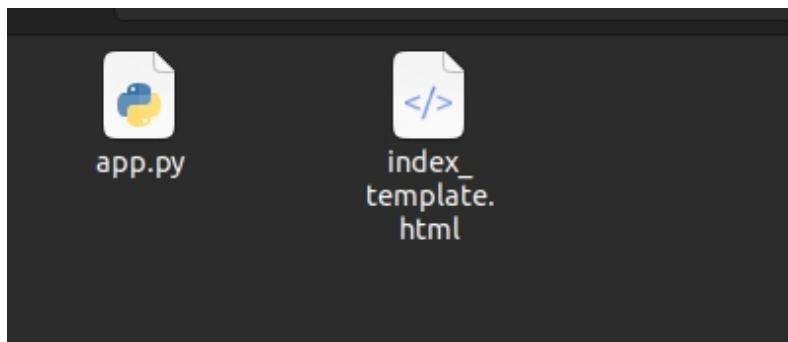
output = open('index.html', 'w')

output.write(rendered_form)

output.close()

```

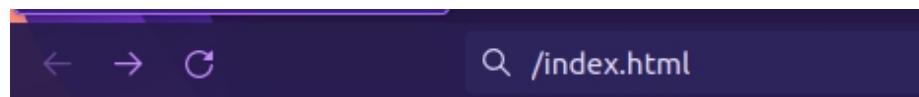
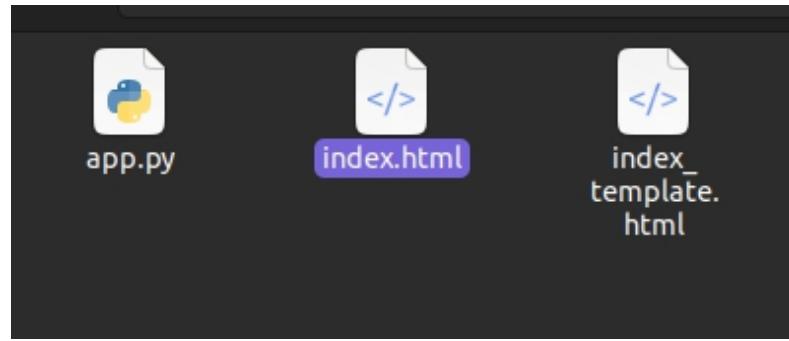
The index.html file is created in the variable output. Write the content to this HTML file using output.write(rendered_form). Below are the two files before running the Python program.



Now, run app.py using the following command:

```
python app.py
```

A new file is created named *index.html*. Open it and see the code. The placeholder text is changed to the values that we passed.



Hello John! Your email is: you@example.co

Conditionals and Looping in Jinja2

Jinja in-line conditionals are started with a curly brace and a % symbol, like `{% if condition %}` and closed with `{% endif %}`. You can optionally include both `{% elif %}` and `{% else %}` tags, and for loop, we use `{% for index in numbers %}` and end with `{% endfor %}`.

Syntax of Conditionals and Looping

For conditions, we have the following syntax in Jinja2.

For loop	If condition
<code>{% for i in numbers %} {% endfor %}</code>	<code>{% if i % 2 == 0 %} {% endif %}</code>

Example

A list can also be passed using Jinja. To iterate through the list and for using conditions, similar to Python we use loop and if-condition. Let's pass a list of numbers as well:

- Python3

```
# app.py

# import Template from jinja2 for passing the content
from jinja2 import Template

# variables that contain placeholder data
name = 'John'
email = 'you@example.co'

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Create one external form_template.html page and read it
File = open('index_template.html', 'r')
content = File.read()
File.close()

# Render the template and pass the variables
template = Template(content)

rendered_form = template.render(pl_name=name,
                                pl_email=email, numbers=numbers)

# save the txt file in the form.html
output = open('index.html', 'w')
output.write(rendered_form)
output.close()
```

index_template.html

Here we will iterate the number and print the even number from the list.

- HTML

```
<!-- index_template.html -->

Hello {{pl_name}}! Your email is: {{pl_email}}

<br>

Even numbers:

{% for i in numbers %}

  {% if i%2==0 %}

    {{i}}

  {% endif %}

{% endfor %}
```

Output:

```
Hello John! Your email is: you@example.co
Even numbers: 2 4 6 8
```

Template Inheritance in Jinja2

Template inheritance is a very good feature of Jinja templating. All that is needed is to add a `{% extend %}` tag. The home page `{% extends "base.html" %}` inherits everything from the base template.

Syntax of Jinja extend block

For Inherit the base page, we have the following syntax in Jinja2.

```
{% block content %}  
<Code>  
{% endblock %}  
{% extends "base.html" %}  
{% block content %}  
    <Code>  
{% endblock %}
```

Example

Here, we want to use the same HTML content across pages like a Website name or a search bar without repeating the HTML code. For this Jinja2 has a feature called template inheritance. Suppose we need this heading and search bar on every page without repeating the code:

My Blog

base.html: This is the code of the website name and search bar.

- HTML

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

<h1>My Blog</h1>

<input type="search">

<button>Search</button>

<!-- Child page code goes between this -->

{% block content %}{% endblock %}

<!-- You can continue base.html code after this if you want -->
```

```
<br><br>
```

Let's include this in our **index_template.html**. In the child template or the page, you want to include the website name and search bar.

- HTML

```
<!-- index_template.html -->

<!-- include base.html -->
{% extends "base.html" %}

<!-- Write any code only in this block -->
{% block content %}

Hello {{pl_name}}! Your email is: {{pl_email}}

<br>

Even numbers:

{% for i in numbers %}
  {% if i%2==0 %}
    {{i}}
  {% endif %}
{% endfor %}

<!-- end the block -->
{% endblock %}
```

Output:



Search /index.html

My Blog

Search

Hello John! Your email is: you@example.co
Even numbers: 2 4 6 8

CHAPTER 5: Serve static files in Flask

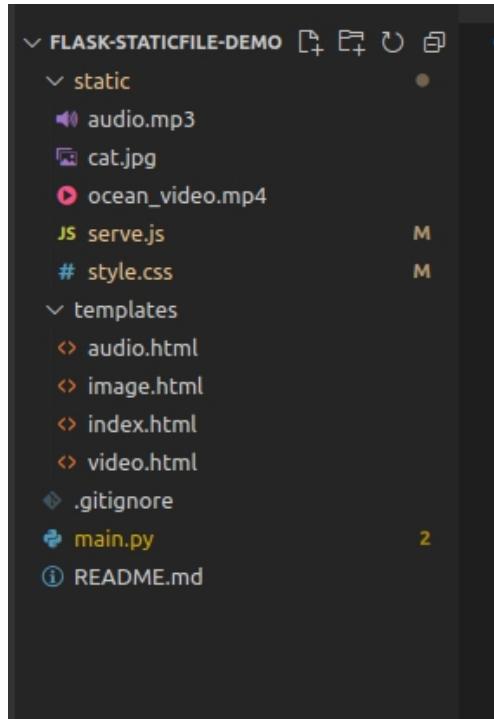
Flask is a lightweight Web Server Gateway Interface or WSGI framework for web applications written in Python. It is designed to make web application development fast and easy and can scale to complex applications. This article describes and demonstrates how to serve various static files in Flask.

Serving Static Files in Flask

Let's configure the virtual environment first. Although this step is optional, we always recommend using a dedicated development environment for each project. This can be achieved in a Python virtual environment.

Now that we have created our Flask app, let's see how to serve static files using the Flask app we just created. First, static files are files served by a web server and do not change over time like CSS and Javascript files used in web applications to improve user experience. Below you will find a demonstration of various static files served by the Flask app.

File Structure



HTML File

Serving HTML files using Flask is fairly simple just create a templates folder in the project root directory and create the HTML files, as **templates/index.html**. Here, we are passing text, and with the help of **Jinja {{message}}**, we are printing text that is present in the variable.

- HTML

```
<html>
<head>
    <title>Flask Static Demo</title>
</head>
<body>
    <h1>{{message}}</h1>
</body>
</html>
```

main.py

In main.py we render the HTML file when we run it, we are using the `render_template()` function provided by Flask to render the HTML file. The final code looks like this:

- Python3

```
from flask import Flask

from flask import render_template


# creates a Flask application

app = Flask(__name__)




@app.route("/")
def hello():

    message = "Hello, World"

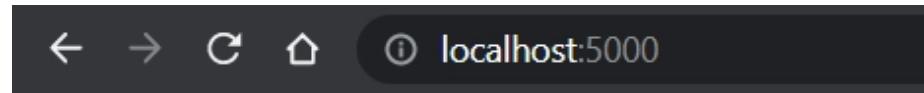
    return render_template('index.html',
                          message=message)


# run the application

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

Output:

The Flask is up and running on localhost port `http://127.0.0.1:5000/`



Hello, World

Serve CSS file in Flask

Now serving a CSS file is the same as an HTML file but instead of /templates folder, we create a static folder in the root directory and add all CSS files to it, For simplicity, we have used a very simple CSS file.

- CSS

```
h1{  
    color: red;  
    font-size: 36px;  
}
```

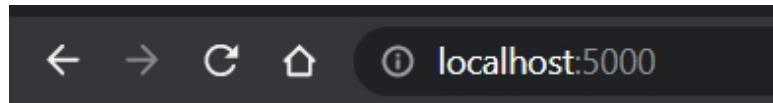
Now, let us link it with the HTML template file using the link tag referring to the CSS file in the static folder.

- HTML

```
<html>  
    <head>  
        <title>Flask Static Demo</title>  
        <link rel="stylesheet" href="/static/style.css" />
```

```
</head>  
  
<body>  
  
    <h1>{{ message }}</h1>  
  
</body>  
  
</html>
```

Output:



Hello, World

Serve JavaScript file in Flask

To serve Javascript it is the same as a CSS file create a javascript file in the static folder.

- Javascript

```
document.write("This is a Javascript static file")
```

Now link it with the HTML and run the Flask app.

- HTML

```
<html>  
  
<head>  
  
    <title>Flask Static Demo</title>  
  
    <link rel="stylesheet" href="/static/style.css" />
```

```
</head>

<body>

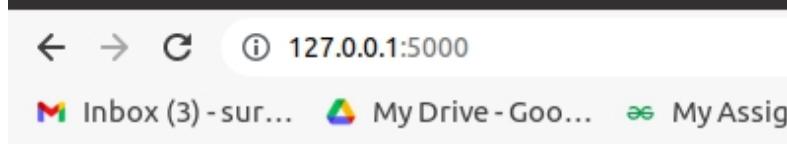
<h1>{{message}}</h1>

<script src="/static/serve.js" charset="utf-8"></script>

</body>

</html>
```

Output:



Hello, World

This is a Javascript static file

Serve Media files in Flask (Image, Video, Audio)

You can also use Flask to serve media files such as images, videos, audio files, text files, and PDFs. You can use the same /static folder that you used for CSS and Javascript to serve these kinds of files.

Place all media files in a static folder and associate them with their respective HTML files as shown below. Once all template files have been processed, create routes in main.py for all static files you want to render.

Images

Create an image.html file in the templates folder and add the following code to the main.py and image.html respectively.

- Python3

```
# Images

@app.route("/image")

def serve_image():

    message = "Image Route"

    return render_template('image.html', message=message)
```

templates/images.html

- HTML

```
<html>

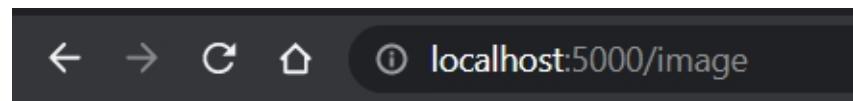
<head>
    <title>Flask Static Demo</title>
    <link rel="stylesheet" href="/static/style.css" />
</head>

<body>
    <h1>{{message}}</h1>

    <script src="/static/serve.js" charset="utf-8"></script>
</body>
</html>
```

Output:



Video Files

To serve a video file, create a video.html file in your templates folder and add the following code to your main.py and video.html files.

- Python3

```
# video

@app.route("/video")

def serve_video():

    message = "Video Route"
```

```
return render_template('video.html', message=message)
```

templates/video.html

As you see the mp4 video file is been served by Flask over localhost.

- HTML

```
<html>

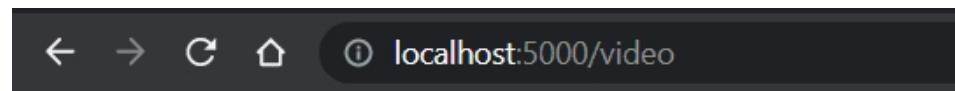
    <head>
        <title>Flask Static Demo</title>
        <link rel="stylesheet" href="/static/style.css" />
    </head>

    <body>
        <h1>{{message}}</h1>

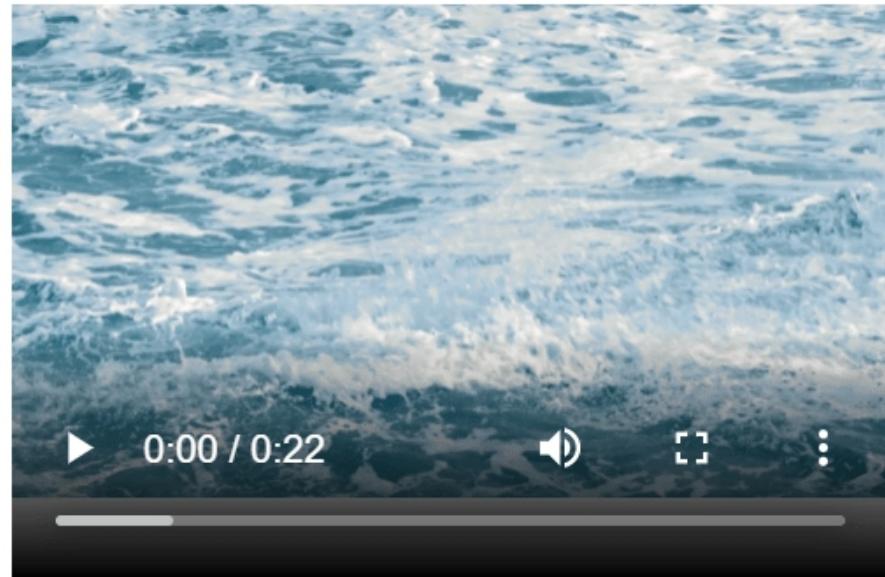
        <video width="320" height="240" controls>
            <source src="/static/ocean_video.mp4" type="video/mp4" />
        </video>

        <script src="/static/serve.js" charset="utf-8"></script>
    </body>
</html>
```

Output:



Video Route



Audio Files

Respectively an audio file can be served by creating an **audio.html** template file and adding the following code to the **main.py**.

- Python3

```
# audio

@app.route("/audio")

def serve_audio():

    message = "Audio Route"

    return render_template('audio.html', message=message)
```

templates/audio.html

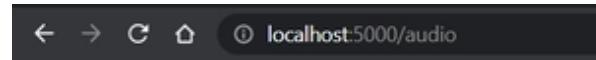
- HTML

```
<html>
  <head>
    <title>Flask Static Demo</title>
    <link rel="stylesheet" href="/static/style.css" />
  </head>
  <body>
    <h1>{{ message }}</h1>

    <audio controls>
      <source src="/static/audio.mp3" />
    </audio>

    <script src="/static/serve.js" charset="utf-8"></script>
  </body>
</html>
```

Output:



Audio Route

II 0:02 / 1:42 - 🔊 :

Complete Flask Code

For simplicity, we have created a simple Flask application for a better understanding of how to serve static files in Flask.

- Python3

```
from flask import Flask

from flask import render_template

# creates a Flask application
app = Flask(__name__)

@app.route("/")
def hello():

    message = "Hello, World"

    return render_template('index.html', message=message)

@app.route("/video")
def serve_video():
```

```

message = "Video Route"

return render_template('video.html', message=message)

@app.route("/audio")

def serve_audio():

    message = "Audio Route"

    return render_template('audio.html', message=message)

@app.route("/image")

def serve_image():

    message = "Image Route"

    return render_template('image.html', message=message)

# run the application

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

```

Let's test the Flask app by running it, to run the app just run the **python main.py** which will serve output as shown above:

```
(venv) PS C:\Users\hariv\PycharmProjects\static_file> python main.py
 * Serving Flask app 'main'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 136-919-433
```

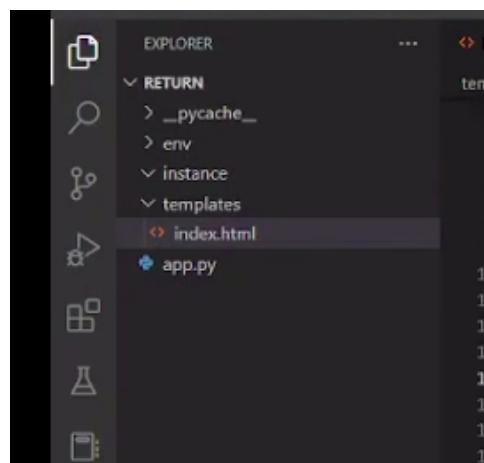
CHAPTER 6: Uploading and Downloading Files in Flask

This article will go over how to upload and download files using a Flask database using Python. Basically, we have a section for uploading files where we can upload files that will automatically save in our database. When we upload a file and submit it, a message stating that your file has been uploaded and displaying the file name on the screen appears. When we view our Flask SQLite database, we can see that our file has been automatically saved in the database. We can also download our files by using the /download /id number using the link. Consequently, we shall comprehend each phase of this method in this post.

Uploading and Downloading Files in Flask

For our upload and return files with the database in a Flask, first, we create a templates folder for making choose file and submit button in HTML file form so let's get started. To upload and download files with the database in Flask, first we need to download SQLite DB browser to save our data in SQLite.

File structure



Templates File

In the templates file, we will create only one HTML file which will operate our all frontend part code.

index.html

In the index.html file, we make the first heading using h1 for showing exactly what we are doing and after that, we write one form in which we declare the method POST action by using URL '/' and after that, we initialize data type by using Enctype and we take the multipart/form-data in this so our all file will safely save in database and after that, we write code for simple input button and also submit button for showing all these functionalities clearly we are using some CSS in style tag for making our frontend part beautiful

- **HTML**

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>File Upload Example</title>
    <style>
        .ok{
            font-size: 20px;
        }
        .op{
            font-size: 20px;
            margin-left: -70px;
            font-weight: bold;
        }
    </style>

```

```

background-color: yellow;
border-radius: 5px;
cursor: pointer;
}

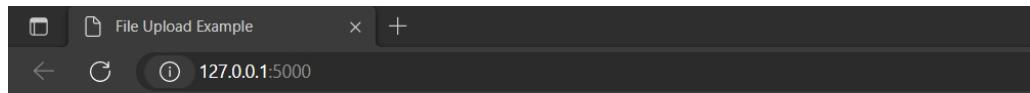
</style>
</head>
<body>

<div class="center">
    <h1> Uploading and Returning Files With a Database in Flask </h1>
    <form method="POST" action="/" enctype="multipart/form-data">
        <input class="ok" type="file" name="file">
        <button class="op">Submit</button>
    </form>
</div>

</body>
</html>

```

Output:



Uploading and Returning Files With a Database in Flask

No file chosen

app.py

After writing code for templates we create an app.py file outside of the templates folder and create app,.py in which we will write our main code of uploading and returning files with a database in a Flask in Python language.

Step 1: Import all libraries

In the app.py file we will write our main part of the code through these all operations will operate easily first in app.py we need to import all important libraries which are important for doing upload and returning files with database operations first we import io BytesIO this module will convert our all pdf files binary in below you can see what type of output will show in the database when we upload any pdf in the database and we import render_template for rendering templates and after that, we are importing send_file module which will help us to send the file to database and after that, we are importing flask_sqlalchemy for our SQL data

- Python

```
from io import BytesIO  
  
from flask import Flask, render_template, request, send_file  
  
from flask_sqlalchemy import SQLAlchemy
```

Step 2: Create a database

After importing all libraries we create an SQL database for uploading and returning our file we initialize the Flask function and after that, we make a database sqlite:///db.sqlite3 to save our uploading files and we create one DB as SQLAlchemy saving database

- Python3

```
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite3'

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

Step 3: Create a table for the database

after creating the database we need to make a table database for which we create one ID which we make the primary key after making the id we create the filename and in the filename we initialize a varchar limit of 50 and after that, we create one data which is a blob which means we can see our file by click on the blob and for making all these functionalities we write following lines of code in our app.py file

- Python3

```
class Upload(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    filename = db.Column(db.String(50))

    data = db.Column(db.LargeBinary)
```

Step 4: Create an index function

We create one index function in which we set the first path '/' and after that, we pass the request for method functionality in method functionality we create one var upload in which we initialize the filename and read the file if we upload a pdf so it will read it in binary form after writing this we create DB session in our database and also we commit our DB session after creating and after that we return file name with flashing message uploaded 'filename' and we return our render template on index,.html file

- Python3

```

@app.route('/', methods=['GET', 'POST'])

def index():

    if request.method == 'POST':

        file = request.files['file']

        upload = Upload(filename=file.filename, data=file.read())

        db.session.add(upload)

        db.session.commit()

        return f'Uploaded: {file.filename}'

    return render_template('index.html')

```

After writing the code we need to create db.sqlite in our system to create a database we need to run the following command in the terminal:

```

python

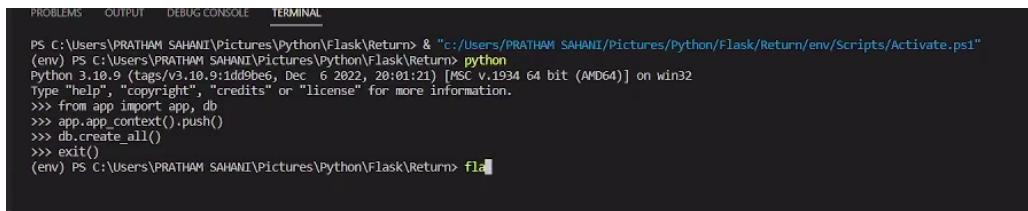
from app import app, db

app.app_context().push()

db.create_all()

exit()

```



The screenshot shows a terminal window with the following text:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\PRATHAM SAHANI\Pictures\Python\Flask\Return & "c:/Users/PRATHAM SAHANI/Pictures/Python/Flask/Return/env/Scripts/Activate.ps1"
(env) PS C:\Users\PRATHAM SAHANI\Pictures\Python\Flask\Return> python
Python 3.10.9 (tags/v3.10.9:1dd9be6, Dec  6 2022, 20:01:21) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from app import app, db
>>> app.app_context().push()
>>> db.create_all()
>>> exit()
(env) PS C:\Users\PRATHAM SAHANI\Pictures\Python\Flask\Return> file

```

by running the above command in the terminal we can create a database in which our file will save after uploading and submitting the database and we can see our database in the instance folder automatically created for running how to run these commands watch a video which is attached below.

Step 5: Create a download function for downloading the file.

After uploading our file we need to make a download function for download our uploading file for download file we make one upload variable in which we add an upload query and we filter our files by id as we set every id unique so every file has a unique id and after that, we return our file by using send_file and also for download pdf file we convert it into binary to our data file by using BytesIO module and after we can download our file by using the following link

http://127.0.0.1:5000/download/id_number

- Python

```
@app.route('/download/<upload_id>')

def download(upload_id):

    upload = Upload.query.filter_by(id=upload_id).first()

    return send_file(BytesIO(upload.data), download_name=upload.filename,
as_attachment=True )
```

Complete Code

- Python3

```
# import all libraires

from io import BytesIO

from flask import Flask, render_template, request, send_file

from flask_sqlalchemy import SQLAlchemy

# Initialize flask and create sqlite database

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite3'
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# create datatable

class Upload(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    filename = db.Column(db.String(50))

    data = db.Column(db.LargeBinary)

# Create index function for upload and return files

@app.route('/', methods=['GET', 'POST'])

def index():

    if request.method == 'POST':

        file = request.files['file']

        upload = Upload(filename=file.filename, data=file.read())

        db.session.add(upload)

        db.session.commit()

        return f'Uploaded: {file.filename}'

    return render_template('index.html')

# create download function for download files

@app.route('/download/<upload_id>')

def download(upload_id):

    upload = Upload.query.filter_by(id=upload_id).first()

    return send_file(BytesIO(upload.data),
```

```
download_name=upload.filename, as_attachment=True)
```

To run the above code we need to run the following command in the terminal:

```
flask run
```

Output:

After running these commands in the terminal when we upload files like pdf and images so following output will show in the database when we blob SQLite database.

Image upload

When we upload images so following image type interface will display on the screen when we blob our database.

The screenshot shows the DB Browser for SQLite application interface. The title bar reads "DB Browser for SQLite - C:\Users\PRATHAM SAHANI\Pictures\Python\Flask\Return\instance\db.sqlite3". The menu bar includes File, Edit, View, Tools, and Help. The toolbar has buttons for New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach Database. The main window shows a table named "upload" with columns id, filename, and data. There is one row with id 1, filename "download.jpg", and data type BLOB. To the right of the table, there is a preview area showing the "GeeksforGeeks" logo. Below the preview, it says "Type of data currently in cell: JPEG Image 268x188 pixel(s), 5.02 KiB". At the bottom, there is a "Remote" section with an "Identity" dropdown set to "Select an identity to connect" and buttons for DBHub.io, Local, and Current Database.

Database for image

Pdf upload

When we upload any pdf file so database converts our pdf file into the binary form so the following type of interface will show when we upload any pdf file in the database and the blob.

The screenshot shows the DB Browser for SQLite interface. At the top, there's a menu bar with File, Edit, View, Tools, Help. Below it is a toolbar with New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, Attach Database, and Close Database. The main area has tabs for Database Structure, Browse Data, Edit Pragmas, Execute, and Edit Database Cell. The 'Edit Database Cell' tab is selected. A table named 'upload' is displayed with three columns: Id, filename, and data. There are two rows in the table. Row 1 has Id 1, filename 'download.jpg', and data type BLOB. Row 2 has Id 2, filename 'PRATHAM...', and data type BLOB. In the data column for Row 1, there is a large binary dump of a PDF file. A tooltip above the dump reads: 'Type of data currently in cell: Binary' and '85982 byte(s)'. The bottom right corner of the window says 'Remote'.

Database of pdf file in binary form

CHAPTER 7: Upload File in Python-Flask

File uploading is a typical task in web apps. Taking care of file upload in Flask is simple all we need is to have an HTML form with the encryption set to multipart/form information to publish the file into the URL. The server-side flask script brings the file from the request object utilizing the `request.files[]` Object. On effectively uploading the file, it is saved to the ideal location on the server.

Install the Flask by writing the command in terminal:

```
pip install flask
```

Stepwise Implementation

Step 1: A new folder “file uploading” should be created. Create the folders “templates” and “main.py” in that folder, which will store our HTML files and serve as the location for our Python code.

Step 2: For the front end, we must first develop an HTML file where the user can select a file and upload it by clicking the upload buttons. The user will click the submit button after choosing the file from their local computer in order to transmit it to the server.

Index.html

- HTML

```
<html>
<head>
    <title>upload the file : GFG</title>
</head>
<body>

    <form action = "/success" method = "post" enctype="multipart/form-data">
        <input type="file" name="file" />
    </form>
</body>
</html>
```

```
<input type = "submit" value="Upload">  
</form>  
</body>  
</html>
```

Step 3: We must make another HTML file just for acknowledgement. Create a file inside the templates folder called “Acknowledgement.html” to do this. This will only be triggered if the file upload went smoothly. Here, the user will receive a confirmation.

Acknowledgement.html

- HTML

```
<html>  
  <head>  
    <title>success</title>  
  </head>  
  <body>  
    <p>File uploaded successfully</p>  
    <p>File Name: {{name}}</p>  
  </body>  
</html>
```

Step 4: Now inside the ‘main.py’ write the following codes. The name of the objective file can be obtained by using the following code and then we will save the uploaded file to the root directory.

main.py

- Python3

```
from distutils.log import debug

from fileinput import filename

from flask import *

app = Flask(__name__)

@app.route('/')

def main():

    return render_template("index.html")



@app.route('/success', methods = ['POST'])

def success():

    if request.method == 'POST':

        f = request.files['file']

        f.save(f.filename)

        return render_template("Acknowledgement.html", name = f.filename)



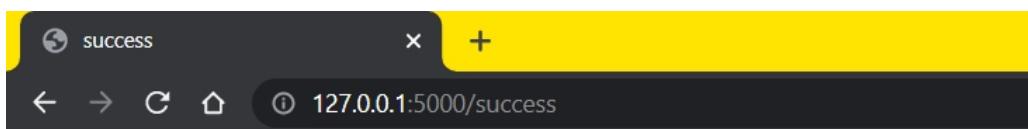
if __name__ == '__main__':

    app.run(debug=True)
```

Output:

Run the following command in your terminal.

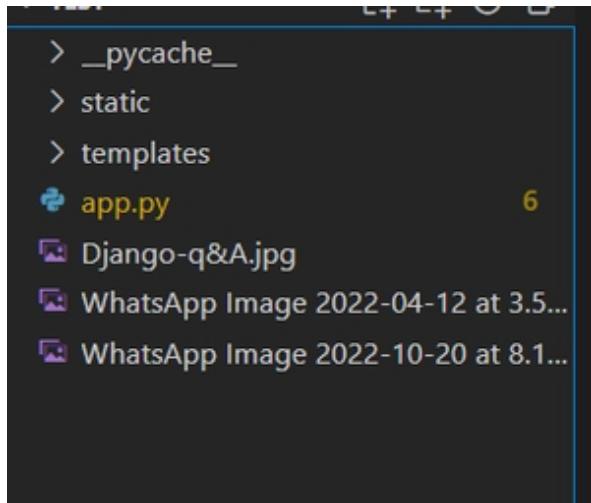
```
python main.py
```



File uploaded successfully

File Name: GFG_Image.jpeg

Step 5: Now, to check if it is correctly working or not go to the folder where '*main.py*'. Check in that folder you will find the files there.



File structure

CHAPTER 8: Upload Multiple files with Flask

In online apps, uploading files is a common task. Simple HTML forms with encryption set to multipart/form information are all that is required to publish a file into a URL when using Flask for file upload. The file is obtained from the request object by the server-side flask script using the request. In this article, we will look at how to upload multiple files with Python. It allows the user to select multiple files at once and upload all files to the server. Before proceeding, Install the Flask by writing the command in the terminal:

```
pip install flask
```

Stepwise Implementation

Step 1: Create a new project folder **Upload**. Inside this folder create **main.py**, and create folder **templates**.

Step 2: Create a simple HTML page **index.html** to select multiple files and submit them to upload files on the server. Here, the HTML file contains a form to select and upload files using the POST method. The **enctype** attribute plays an important role here. It specifies how the form data should be encoded when submitting it to the server. we are uploading files that's why we should set the attribute value to **multipart/form-data**.

- HTML

```
<html>
<head>
    <title>Upload Multiple files : GFG</title>
</head>
```

```

<body>

    <form action = "/upload" method="POST" enctype="multipart/form-data">

        <input type="file" name="file" multiple />

        <input type = "submit" value="Upload">

    </form>

</body>

</html>

```

Step 3: Now inside the **main.py**. Here the list of the file object is collected and then we will save the uploaded files one by one to the root directory using the loop and file.save() function.

- Python3

```

from flask import *

app = Flask(__name__)

@app.route('/')

def main():

    return render_template("index.html")



@app.route('/upload', methods=['POST'])

def upload():

    if request.method == 'POST':


        # Get the list of files from webpage

```

```
files = request.files.getlist("file")

# Iterate for each file in the files List, and Save them

for file in files:

    file.save(file.filename)

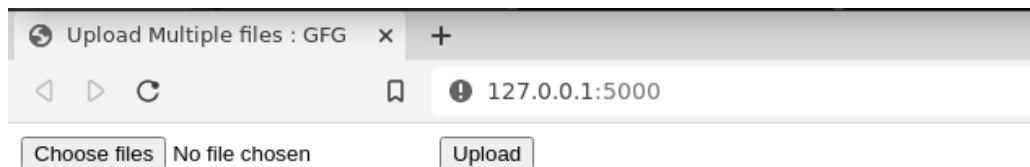
return "<h1>Files Uploaded Successfully.!</h1>"


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

Output:

Run the following command in Terminal

```
python main.py
```



Index Page

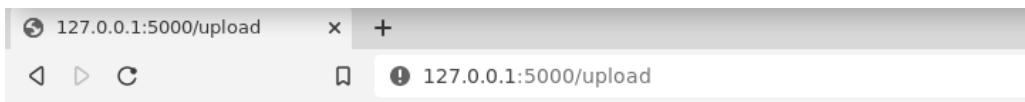
Select multiple files from your folder.

Open Files

	Name	Location	Size	Type	Accessed
Recent	index.png	Desktop	11.4 kB	Image	19:58
Home	pp3.png	Pictures	124.6 kB	Image	16:20
Desktop	pp2.png	Pictures	532.0 kB	Image	16:20
Documents	pp1.png	Pictures	964.1 kB	Image	16:20
Downloads	student.png	Home	38.4 kB	Image	16:16
Music	register.png	Home	30.7 kB	Image	16:16
Pictures	faculty.png	Home	49.2 kB	Image	16:10
Videos	login.png	Home	22.2 kB	Image	16:10
	manage.py	projects/beyond	662 bytes	Text	13:07
	requirements.txt	...upload_multiple_file_flask	5 bytes	Text	13:01
	upload_file.py	projects/flask_test	1.5 kB	Text	12:53
	1.png	Pictures	33.6 kB	Image	12:50

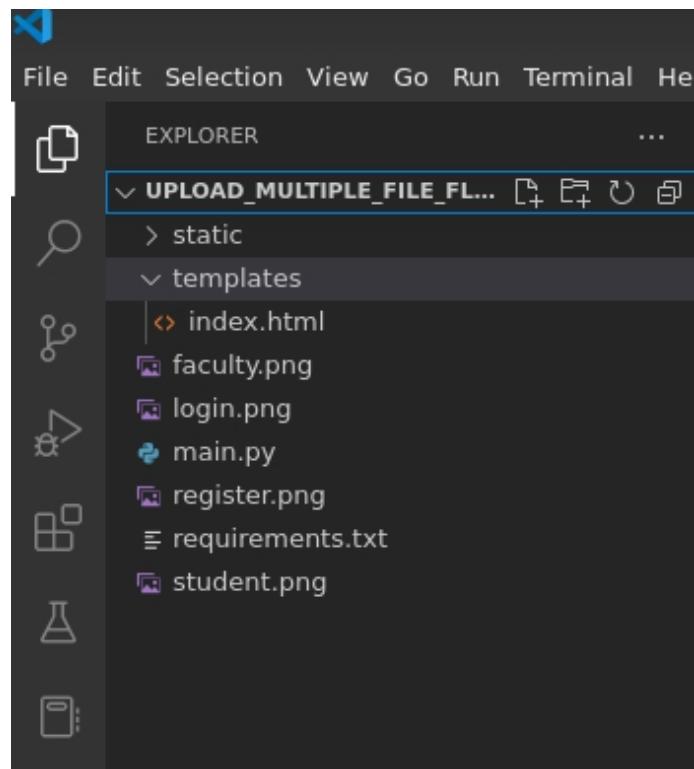
Select Multiple Images

After submitting, a Success message will displayed.



Files Uploaded

Here we can see that four *.png images are uploaded to the root directory of a project.



Verification

CHAPTER 9: Flask - Message Flashing

In this article, we will discuss Flask – Message Flashing. As we know best Graphical User Interface which provides feedback to a user when users interact, as an example, we can know that desktop applications use the messages box or JS for an alert purpose. generating like same informative message is easy to display in the Flask web application. The flashing message of the Flask web application makes flask easier and more useful for users.

What is Message Flashing

Message Flashing means when we click on any button(any website) and the website showing immediately any massage that can be an error, alert, or any type of massage that's call massage Flashing and when that happens in a flask or we are creating any website which shows an alert, error type massage than its call Flash-Message Flashing.

A Flask contains a **flash()** function. which passes a request to the next user. which generally is a template we are using flash methods in the above template in which the **message** parameter is the real message to be flashed, and another one is the **category** parameter which is optional, it will happen “error” or “warning”.

```
flash(message, category)
```

To understand flashing message and know about more flashing messages, we start to write code for flashing massages we take an example of a login system so let's write code for flashing message. Create a Virtual Environment.

app.py File

Here, we create an app.py file in which we write our main code of Flask.

In app.py first, we import flask then we initialize the flask function and create a secret key which will help us when we forgot our password then we made two functions one home() for index.html which will help us to call and display the home page when we run flask using flask run command and another one for login login.html we joint both files by

using form action and after that, we use if and else condition for flashing message (error or warning) and also for redirect on profile page by filling right password and also right Email ID and after that in if and else condition we create password GFG we can set any password it up to you and also write error flashing message which will show when we fill the wrong password and click on submit button for login and if we fill right password and email id it will redirect us on profile page were also showing one welcome flashing message which we create using python code in app.py file.

That error message which is showing in the above images when we enter the wrong password is called a flashing message also if we fill right password and email id and click on submit button then it will redirect us to the profile page which will also show a welcome flashing message so this way we can display flashing message in our display screen. for getting the code and take help completing this flashing message click here. It will also help to know how to show flashing messages.

- Python3

```
from flask import *

# Initialize Flask function
app = Flask(__name__)
app.secret_key = "GeeksForGeeks"

# home function for index.html
@app.route("/index")
def home():

    return render_template("index.html")

# row function for profile.html
```

```

@app.route("/profile")

def row():

    return render_template("profile.html")



# write if and else condition if we provide write password then he will redirect
# us in profile page otherwise he will redirect us on same page with
# flashing message Invalid Password

@app.route("/login", methods=['GET', 'POST'])

def login():

    error = None

    if request.method == "POST":

        if request.form['pass'] != "GFG":

            error = "Invalid Password"

        else:

            flash("You are successfully login into the Flask Application")

            return redirect(url_for('row'))


    return render_template("login.html", error=error)






# execute command with debug function

if __name__ == '__main__':

    app.run(debug=True)

```

Templates File

index.html

In this code, we are writing some flashing (error code) massage command and we connect this page to login by using the URL “/login” in the app.py file it means when we click on the URL ” login “redirects us to the login page. where we can log in. and we can enter our profile by filling right password which we will set in the below code of the app.py file.

- HTML

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Home</title>

</head>

<body>

    {% with messages = get_flashed_messages()%}

    {% if messages%}

        {% for message in messages%}

            <p>{{message}}</p>

        {%endfor%}

        {%endif%}

        {%endwith%}

    <h3>Welcome to the GeeksForGeeks</h3>

    <a href="{{url_for('login')}}">login</a>

</body>
```

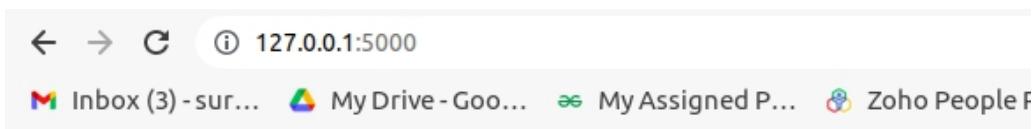
```
</html>
```

When we run the following command in our terminal.

```
flask run
```

Output:

This will show on our display screen and it also shows our flask is successfully running.



Welcome to the GeeksForGeeks

[login](#)

index.html

login.html

In the login page, we create simple input for the password and email which is required to fill .and also write three lines of python code for showing a flashing message when we enter the wrong password and try to log in and also set both inputs in from tag in which we create one method post and action “/login” which means when we click on previous page login link so it will redirect us on this login form page and we write for one button in which we set the value to submit for submitting our login form after filling Email and password.

- HTML

```
<!DOCTYPE html>
```

```
<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Login</title>

</head>

<body>

    {% if error%}

        <p><strong>{{error}}</strong></p>

    {% endif%}

    <form method="post" action="/login">

        <table>

            <tr>

                <td>Email</td>

                <td><input type="email" name="email"></td>

            </tr>

            <tr>

                <td>Password</td>

                <td><input type="password" name="pass"></td>

            </tr>

            <tr>

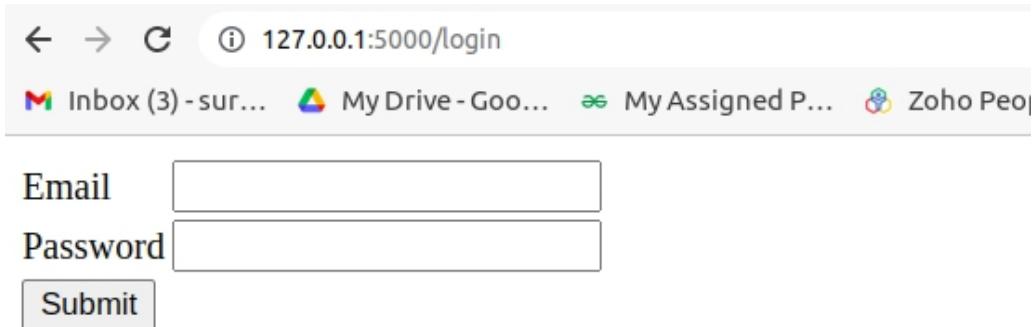
                <td><input type="submit" value="Submit"></td>

            </tr>

    
```

```
</table>  
</form>  
</body>  
  
</html>
```

Output:



The screenshot shows a web browser window with the URL `127.0.0.1:5000/login`. The page displays a simple login form. At the top, there are navigation icons (back, forward, search) and a refresh button. Below the address bar, there are several links: `Inbox (3) - sur...`, `My Drive - Goo...`, `My Assigned P...`, and `Zoho Peo...`. The main content area contains two input fields: one labeled `Email` and another labeled `Password`. Below these fields is a `Submit` button.

login.html

profile.html

In this code, we made a simple message that will show after successful login. it will also show us the welcome flashing message which we create in the app.py file by using the flask function.

• HTML

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

```

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Profile</title>

</head>

<body>

    <div class="container">

        <p><strong>Welcome to your GFG Profile Page</strong></p>

    </div>

</body>

</html>

```

Final Output of unsuccessful Flashing Message

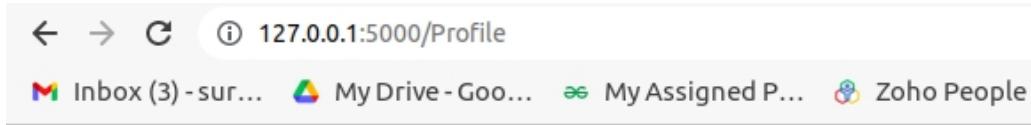
If we write the wrong Password and press submit button so it will show a flashing message (error or warning) like the below output image **Invalid Password** is called **Flashing Message**. and if we fill right password and email it will redirect us to the profile page which will also show one flashing welcome message on our display screen.



Invalid Password

Email	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="Submit"/>	

The output of successful Flashing Message



profile.html

CHAPTER 10: Create Contact Us using WTForms in Flask

WTForms is a library designed to make the processing of forms easier to manage. It handles the data submitted by the browser very easily. In this article, we will discuss how to create a contact us form using WTForms.

Advantages of WT-FORM:

1. We don't have to worry about validators.
2. Avoidance of Cross-Site Request Forgery (CSRF).
3. WTForms come as classes, so all the good come's from an object form.
4. No need to create any <label> or <input> elements manually using HTML.

Installation

Use the Terminal to install Flask-WTF.

```
pip install Flask-WTF
```

Stepwise Implementation

Step 1: Create a class having all elements that you want in your Form in the **main.py**.

- Python3

```
from flask_wtf import FlaskForm  
  
from wtforms import StringField, validators, PasswordField, SubmitField  
  
from wtforms.validators import DataRequired, Email  
  
import email_validator
```

```
class contactForm(FlaskForm):  
  
    name = StringField(label='Name', validators=[DataRequired()])  
  
    email = StringField(label='Email', validators=[  
        DataRequired(), Email(granular_message=True)])  
  
    message= StringField(label='Message')  
  
    submit = SubmitField(label="Log In")
```

Step 2: Create the object of the form and pass the object as a parameter in the render_template

- Python3

```
@app.route("/", methods=["GET", "POST"])  
  
def home():  
  
    cform = contactForm()  
  
    return render_template("contact.html", form=cform)
```

Step 3: Add CSRF protection. Add a secret key.

```
app.secret_key = "any-string-you-want-just-keep-it-secret"
```

Step 4: Add the fields in the contact.html HTML FILE.

`{{ form.csrf_token }}` is used to provide csrf protection.

- HTML

```
<!DOCTYPE HTML>  
  
<html>  
  <head>
```

```
<title>Contact</title>

</head>

<body>

    <div class="container">

        <h1>Contact Us</h1>

        <form method="POST" action="{{ url_for('home') }}>
            {{ form.csrf_token }}

            <p>
                {{ form.name.label }}
                <br>
                {{ form.name }}
            </p>

            <p>
                {{ form.email.label }}
                <br>
                {{ form.email(size=30) }}
            </p>

            <p>
                {{ form.message.label }}
                <br>
                {{ form.message }}
            </p>
    
```

```
    {{ form.submit }}
```

```
</form>
```

```
</div>
```

```
</body>
```

```
</html>
```

Step 5: Validating the Form and receiving the data.

- Python3

```
@app.route("/", methods=["GET", "POST"])
```

```
def home():
```

```
    cform = contactForm()
```

```
    if cform.validate_on_submit():
```

```
        print(f"Name:{cform.name.data},
```

```
             E-mail:{cform.email.data},
```

```
             message:{cform.message.data}")
```

```
    else:
```

```
        print("Invalid Credentials")
```

```
    return render_template("contact.html", form=cform)
```

Complete Code:

- Python3

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

```
from flask_wtf import FlaskForm
```

```
from wtforms import StringField, validators, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email
import email_validator

app = Flask(__name__)

app.secret_key = "any-string-you-want-just-keep-it-secret"

class contactForm(FlaskForm):
    name = StringField(label='Name', validators=[DataRequired()])
    email = StringField(
        label='Email', validators=[DataRequired(), Email(granular_message=True)])
    message = StringField(label='Message')
    submit = SubmitField(label="Log In")

@app.route("/", methods=["GET", "POST"])

def home():
    cform=contactForm()

    if cform.validate_on_submit():
        print(f"Name:{cform.name.data}, E-mail:{cform.email.data},
              message:{cform.message.data}")

    return render_template("contact.html",form=cform)

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

Output:

Contact Us

Name

Email

Message

Name:Rahul Singh, E-mail:rahuls@gmail.com, message:This is Sample gfg Output!!!

Adding Bootstrap

We can also add the bootstrap to the above form to make it look interactive. For this, we will use the Flask-Bootstrap library. To install this module type the below command in the terminal.

pip install Flask-Bootstrap

Step 1: Create base.html

-

```
<!DOCTYPE html>
```

```
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>{% block title %} {% endblock %}</title>  
</head>  
<body>  
    {% block content %} {% endblock %}  
</body>  
</html>
```

Step 2: Modify contact.html to
with single line {{ wtf.quick_form(form) }}

- HTML

```
{% extends 'bootstrap/base.html' %}  
{% import "bootstrap/wtf.html" as wtf %}  
  
{% block title %}  
Contact Us  
{% endblock %}  
  
{% block content %}  


# Contact Us



{{ wtf.quick_form(form) }}

{% endblock %}|>
```

Step 3: MODIFY main.py

It is very simple to modify the .py file. We just have to import the module and add the below line into the code

```
Bootstrap(app)
```

- Python3

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

from flask_wtf import FlaskForm

from wtforms import StringField, validators, PasswordField, SubmitField

from wtforms.validators import DataRequired, Email

from flask_bootstrap import Bootstrap

import email_validator

app = Flask(__name__)

Bootstrap(app)

app.secret_key = "any-string-you-want-just-keep-it-secret"

class contactForm(FlaskForm):

    name = StringField(label='Name', validators=[DataRequired()])

    email = StringField(label='Email', validators=[DataRequired(),
Email(granular_message=True)])

    message = StringField(label='Message')

    submit = SubmitField(label="Log In")

@app.route("/", methods=["GET", "POST"])
```

```

def home():

    cform=contactForm()

    if cform.validate_on_submit():

        print(f"Name:{cform.name.data}, E-mail:{cform.email.data}, message:
{cform.message.data}")

    return render_template("contact.html",form=cform)

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

```

Output:

Contact Us

Name

Email

Message

CHAPTER 11: Sending Emails Using API in Flask-Mail

Python, being a powerful language don't need any external library to import and offers a native library to send emails- "SMTP lib". "smtplib" creates a Simple Mail Transfer Protocol client session object which is used to send emails to any valid email id on the internet. This article revolves around how we can send bulk customised emails to a group of people with the help of Flask.

Installation :

Three packages are required for flask mail to work, Install them using pip,

1) virtualenv:

```
pip install virtualenv
```

2) Flask:

```
pip install Flask
```

3) Flask-Mail :

```
pip install Flask-Mail
```

After installing the packages, we have to use **virtualenv** (optional)

1) Create a virtualenv

Open cmd

Go to the folder you want to use for your project.

Write the following code:

```
python3 -m venv env (macOS/Linux)
```

```
py -m venv env (Windows)
```

Here **env** is name of your environment.

2) Activate the environment

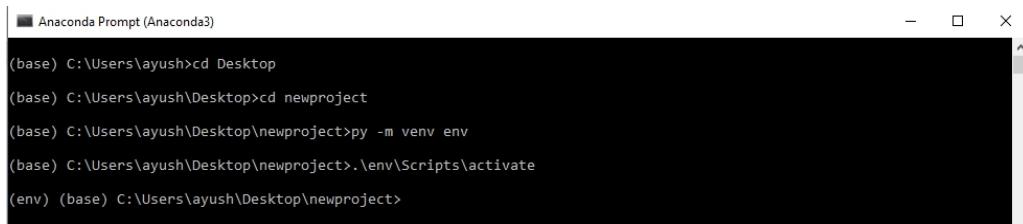
On windows :

```
.\env\Scripts\activate
```

On macOS/ Linux:

```
source env/bin/activate
```

3) Make sure you get the (env) in the beginning displayed in picture below :



```
Anaconda Prompt (Anaconda3)
(base) C:\Users\ayush>cd Desktop
(base) C:\Users\ayush\Desktop>cd newproject
(base) C:\Users\ayush\Desktop\newproject>py -m venv env
(base) C:\Users\ayush\Desktop\newproject>.\env\Scripts\activate
(env) (base) C:\Users\ayush\Desktop\newproject>
```

Configuring Flask-Mail

Flask-Mail is configured through the standard Flask config API. These are the available options (each is explained later in the documentation):

- 1) **MAIL_SERVER** : Name/IP address of the email server.
- 2) **MAIL_PORT** : Port number of server used.
- 3) **MAIL_USE_TLS** : Enable/disable Transport Security Layer encryption.
- 4) **MAIL_USE_SSL** : Enable/disable Secure Sockets Layer encryption
- 5) **MAIL_DEBUG** : Debug support. The default is Flask application's debug status.
- 6) **MAIL_USERNAME** : Username of the sender
- 7) **MAIL_PASSWORD** : The password of the corresponding Username of the sender.
- 8) **MAIL_ASCII_ATTACHMENTS** : If set to true, attached filenames converted to ASCII.
- 9) **MAIL_DEFAULT_SENDER** : sets default sender
- 10) **MAIL_SUPPRESS_SEND** : Sending suppressed if app.testing set to true
- 11) **MAIL_MAX_EMAILS** : Sets maximum mails to be sent

Note : Not all of the configuration is to be set.

Sending Emails using Flask-Mail

Classes in Flask-Mail:

Mail Class : Manages email-messaging requirements
Message Class: encapsulates an email message

Let's get our hands on the code.

- Python3

```
# importing libraries

from flask import Flask

from flask_mail import Mail, Message


app = Flask(__name__)

mail = Mail(app) # instantiate the mail class


# configuration of mail

app.config['MAIL_SERVER']='smtp.gmail.com'

app.config['MAIL_PORT'] = 465

app.config['MAIL_USERNAME'] = 'yourId@gmail.com'

app.config['MAIL_PASSWORD'] = '*****'

app.config['MAIL_USE_TLS'] = False

app.config['MAIL_USE_SSL'] = True

mail = Mail(app)


# message object mapped to a particular URL '/'

@app.route("/")

def index():

    msg = Message(

        'Hello',

        sender ='yourId@gmail.com',

        recipients = ['receiver'sid@gmail.com']

    )

    msg.body = 'Hello Flask message sent from Flask-Mail'

    mail.send(msg)
```

```
    return 'Sent'

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

Save it in a file and then run the script in Python Shell or CMD & Visit <http://localhost:5000/>.

Note :

Due to Google's built-in security features, Gmail service may block this login attempt. You may have to decrease the security level. Visit <https://myaccount.google.com/lesssecureapps?pli=1> to decrease security.

PART 4: User Registration, Login, and Logout in Flask

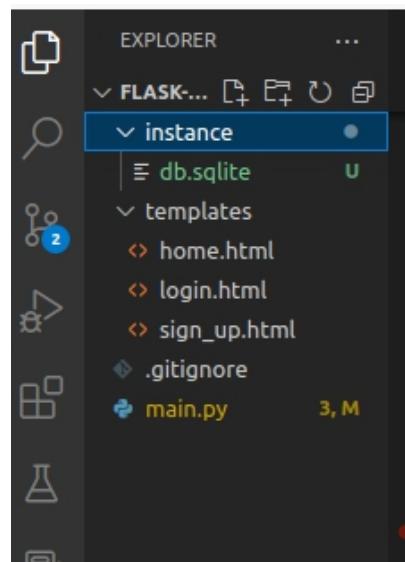
CHAPTER 1: Add Authentication to Your App with Flask-Login

Whether it is building a simple blog or a social media site, ensuring user sessions are working correctly can be tricky. Fortunately, Flask-Login provides a simplified way of managing users, which includes easily logging in and out users, as well as restricting certain pages to authenticated users. In this article, we will look at how we can add Authentication to Your App with Flask-Login in Flask using Python. To start, install flask, flask-login, and flask-sqlalchemy:

- Flask-Login helps us manage user sessions
- Flask-SQLAlchemy helps us store our user's data, such as their username and password

```
pip install flask flask-login flask-sqlalchemy
```

File structure



Stepwise Implementation

Step 1: Import the necessary modules.

We first import the classes we need from Flask, Flask-SQLAlchemy, and Flask-Login. We then create our flask application, indicate what database Flask-SQLAlchemy should connect to, and initialize the Flask-SQLAlchemy extension. We also need to specify a secret key, which can be any random string of characters, and is necessary as Flask-Login requires it to sign session cookies for protection again data tampering. Next, we initialize the *LoginManager* class from Flask-Login, to be able to log in and out users.

- Python3

```
from flask import Flask

from flask_sqlalchemy import SQLAlchemy

from flask_login import LoginManager


# Create a flask application

app = Flask(__name__)

# Tells flask-sqlalchemy what database to connect to

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///db.sqlite"

# Enter a secret key

app.config["SECRET_KEY"] = "ENTER YOUR SECRET KEY"

# Initialize flask-sqlalchemy extension

db = SQLAlchemy()


# LoginManager is needed for our application

# to be able to log in and out users

login_manager = LoginManager()

login_manager.init_app(app)
```

Step 2: Create a User Model & Database

To be able to store users' information such as their username and password, we need to create a table with Flask-SQLAlchemy, this is done by creating a model that represents the information we want to store. In this case, we first create a Users class and make it a subclass of db.Model to make it a model with the help of Flask-SQLAlchemy. We also make the Users class a subclass of UserMixin, which will help to implement properties such as is_authenticated to the Users class. We will also need to create columns within the user model, to store individual attributes, such as the user's username. When creating a new column, we need to specify the datatype such as db.Integer and db.String as well. When creating columns, we also need to specify keywords such as unique = True, if we want to ensure values in the column are unique, nullable = False, which indicates that the column's values cannot be NULL, and primary_key = True, which indicates that the row can be identified by that primary_key index. Next, the db.create_all method is used to create the table schema in the database.

- Python3

```
# Create user model

class Users(UserMixin, db.Model):

    id = db.Column(db.Integer, primary_key=True)

    username = db.Column(db.String(250), unique=True,
                        nullable=False)

    password = db.Column(db.String(250),
                        nullable=False)

# Initialize app with extension

db.init_app(app)

# Create database within app context
```

```
with app.app_context():
    db.create_all()
```

Step 3: Adding a user loader

Before implementing the functionality for authenticating the user, we need to specify a function that Flask-Login can use to retrieve a user object given a user id. This functionality is already implemented by Flask-SQLAlchemy, we simply need to query and use the `get` method with the user id as the argument.

- Python3

```
# Creates a user loader callback that returns the user object given an id
@login_manager.user_loader
def loader_user(user_id):
    return Users.query.get(user_id)
```

Step 4: Registering new accounts with Flask-Login

Add the following code to a file name `sign_up.html` in a folder called `templates`. To allow the user to register an account, we need to create the HTML. This will need to contain a form that allows the user to enter their details, such as their username and chosen password.

- HTML

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Sign Up</title>
<style>
  h1 {
    color: green;
  }
</style>
</head>
<body>
  <nav>
    <ul>
      <li><a href="/login">Login</a></li>
      <li><a href="/register">Create account</a></li>
    </ul>
  </nav>
  <h1>Create an account</h1>
  <form action="#" method="post">
    <label for="username">Username:</label>
    <input type="text" name="username" />
    <label for="password">Password:</label>
    <input type="password" name="password" />
    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

Create a route that renders the template, and creates the user account if they make a POST request.

We create a new route with Flask by using the **@app.route decorator**. The @app.route decorator allows us to specify the route it accepts, and the methods it should accept. By default, it only accepts requests using the GET method, but when the form is submitted it is done using a POST request, so we'll need to make POST an accepted method for the route as well. Within the register function that is called whenever the user visits that route, we can check if the method used was a POST request using the request variable that Flask provides and that needs to be imported. If a post request was made, this indicates the user is trying to register a new account, so we create a new user using the Users model, with the username and password set to whatever the user entered, which we can get by using **request.form.get**. Lastly, we add the user object that was created to the session and commit the changes made. Once the user account has been created, we redirect them to a route with a callback function called "login", which we will create in a moment. Ensure that you also import the redirect and url_for functions from Flask.

- Python3

```
@app.route('/register', methods=["GET", "POST"])

def register():

    # If the user made a POST request, create a new user

    if request.method == "POST":

        user = Users(username=request.form.get("username"),

                     password=request.form.get("password"))

        # Add the user to the database

        db.session.add(user)

        # Commit the changes made

        db.session.commit()

        # Once user account created, redirect them

        # to login route (created later on)
```

```
    return redirect(url_for("login"))

# Renders sign_up template if user made a GET request

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

Step 5: Allowing users to log in with Flask-Login

Like with creating the registered route, we first need a way for the user to log in through an HTML form. Add the following code to a file named login.html in the same templates folder.

- HTML

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>Login</title>

<style>

h1{

color: green;

}

</style>

</head>

<body>

<nav>

<ul>
```

```

<li><a href="/login">Login</a></li>

<li><a href="/register">Create account</a></li>

</ul>

</nav>

<h1>Login to your account</h1>

<form action="#" method="post">

    <label for="username">Username:</label>

    <input type="text" name="username" />

    <label for="password">Password:</label>

    <input type="password" name="password" />

    <button type="submit">Submit</button>

</form>

</body>

</html>

```

Add the functionality to log in to the user within a login function for the /login route.

With the login route, we do the same thing of checking if the user made a POST request. If they did, we filter the users within the database for a user with the same username as the one being submitted. Next, we check if that user has the same password as the password the user entered in the form. If they are the same, we log-in to the user by using the *login_user* function provided by Flask-Login. We can then redirect the user back to a route with a function called “home”, which we will create in a moment. If the user didn’t make a *POST* request, and instead a *GET* request, then we’ll render the login template.

- Python3

```
@app.route("/login", methods=["GET", "POST"])
```

```

def login():

    # If a post request was made, find the user by

    # filtering for the username

    if request.method == "POST":

        user = Users.query.filter_by(
            username=request.form.get("username")).first()

        # Check if the password entered is the

        # same as the user's password

        if user.password == request.form.get("password"):

            # Use the login_user method to log in the user

            login_user(user)

            return redirect(url_for("home"))

        # Redirect the user back to the home

        # (we'll create the home route in a moment)

    return render_template("login.html")

```

Step 6: Conditionally rendering HTML based on the user's authentication status with Flask-Login

When using Flask, it uses Jinja to parse the templates. Jinja is a templating engine that allows us to add code, such as if-else statements within our HTML, we can then use it to conditionally render certain elements depending on the user's authentication status for example the current_user variable is exported by Flask-Login, and we can use it within the Jinja template to conditionally render HTML based on the user's authentication status.

- HTML

```

<!DOCTYPE html>

<html lang="en">

```

```

<head>

    <meta charset="UTF-8" />

    <meta http-equiv="X-UA-Compatible" content="IE=edge" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Home</title>

</head>

<body>

    <nav>

        <ul>

            <li><a href="/login">Login</a></li>

            <li><a href="/register">Create account</a></li>

        </ul>

    </nav>

    {% if current_user.is_authenticated %}

        <h1>You are logged in</h1>

    {% else %}

        <h1>You are not logged in</h1>

    {% endif %}

</body>

</html>

```

Add the functionality to render the homepage when the user visits the “/” route.

This will then render the template of home.html whenever the user visits the “/” route. After running the code in main.py, navigate to <http://127.0.0.1:5000/>

- Python3

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

Step 7: Adding Logout Functionality

Here, we will **update the home.html** template to the following to add a logout link, and this will give the homepage a link to log out the user if they are currently logged in.

- HTML

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Home</title>
        <style>
            h1 {
                color: green;
            }
        </style>
    </head>
    <body>
        <nav>
```

```
<ul>
    <li><a href="/login">Login</a></li>
    <li><a href="/register">Create account</a></li>
</ul>
</nav>

{% if current_user.is_authenticated %}

<h1>You are logged in</h1>

<a href="/logout">Logout</a>

{% else %}

<h1>You are not logged in</h1>

{% endif %}

</body>

</html>
```

Complete Code

Add the logout functionality and code initializer.

- Python3

```
from flask import Flask, render_template, request, url_for, redirect
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///db.sqlite"
app.config["SECRET_KEY"] = "abc"
```

```
db = SQLAlchemy()

login_manager = LoginManager()
login_manager.init_app(app)

class Users(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(250), unique=True, nullable=False)
    password = db.Column(db.String(250), nullable=False)

db.init_app(app)

with app.app_context():
    db.create_all()

@login_manager.user_loader
def loader_user(user_id):
    return Users.query.get(user_id)

@app.route('/register', methods=["GET", "POST"])
def register():

    if request.method == "POST":
        user = Users(username=request.form.get("username"),
```

```
        password=request.form.get("password"))

    db.session.add(user)

    db.session.commit()

    return redirect(url_for("login"))

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

```
@app.route("/login", methods=["GET", "POST"])

def login():

    if request.method == "POST":

        user = Users.query.filter_by(
            username=request.form.get("username")).first()

        if user.password == request.form.get("password"):

            login_user(user)

            return redirect(url_for("home"))

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

```
@app.route("/logout")

def logout():

    logout_user()

    return redirect(url_for("home"))
```

```
@app.route("/")

def home():

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

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

Output:

To test the application, we would navigate to the /register route, create an account and we'll be redirected to the /login route. From there, we can log in, and we can verify that we have been logged in by the conditional HTML rendering.

- [Login](#)
- [Create account](#)

You are not logged in

Now, whenever the user is logged in, they can log out by clicking the link within the homepage, which will logout the user, by using the `logout_user` function provided by Flask-Login.

- [Login](#)
- [Create account](#)

You are logged in

[Logout](#)

CHAPTER 2: Add User and Display Current Username in Flask

In this article, we'll talk about how to add a User and Display the Current Username on a Flask website. When we log in using our username and password, we will be taken to the profile page where we can see the welcome message and the username we created during registration. When additional users register using the login credentials we will use here, their names will also appear on the profile page screen. Python code will be connected to a MySQL database to preserve the user login and registration credentials. From there, we can observe how many users have registered and edited their information using phpmyadmin. For making our project we install flask first and create a virtual environment.

Display Username on Multiple Pages using Flask

Templates Files

In the templates folder we basically made three files one for register, another one for login, and at last one for the user so first we write code for register.html

register.html

This HTML file contains a straightforward registration form that asks for three inputs: username, email address, and password. Once these fields have been completed, click the register button to see a flashing message stating that the form has been successfully submitted and that the registration information has been safely saved in the MySQL database. For flashing message, we are using Jinja2 in an HTML file, so we can now log in using our credentials. If we registered using the same email address, the flash email id will also exist.

- HTML

```
<html>  
<head>
```

```
<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>User Registration Form</title>

</head>

<style>

.hi{
    color: green;
}

.ok{
    display: block;
    margin-left: 80px;
    margin-top: -15px;
    border: 1px solid black;
}

.gfg{
    margin-left: 30px;
    font-weight: bold;
}

.gf{
    margin-left: 10px;
    font-weight: bold;
}

.btn{
    margin-top: 20px;
    width: 80px;
    height: 25px;
    background-color: orangered;
}
```

```
        color: white;  
    }  
  
.y{  
    color: gray;  
}  
  
</style>  
  
<body>  
  
<div class="container">  
  
    <h2 class="hi" > GFG User Registration </h2>  
  
    <h4 class="y" >Note : fill following details !</h4>  
  
    <form action="{{ url_for('register') }}" method="post">  
  
        {% if message is defined and message %}  
  
            <div class="alert alert-warning"> <strong> {{ message }} ???? </strong>  
        </div>  
  
        {% endif %}  
  
        <br>  
  
        <div class="form-group">  
  
            <label class="gfg">Name:</label>  
  
            <input class="ok" type="text" class="form-control" id="name" name="name"  
placeholder="Enter name" name="name">  
  
        </div>  
  
        <div class="form-group">  
  
            <label class="gfg">Email:</label>  
  
            <input class="ok" type="email" class="form-control" id="email" name="email"  
placeholder="Enter email" name="email">  
  
        </div>  
  
        <div class="form-group">
```

```

<label class="gf">Password:</label>

<input class="ok" type="password" class="form-control" id="password"
name="password" placeholder="Enter password" name="pswd">

</div>

<button class="btn" type="submit" class="btn btn-primary">Register</button>

<p class="bottom">Already have an account? <a class="bottom" href="
{url_for('login')}"> Login here</a></p>

</form>

</div>

</body>

</html>

```

Output:

Login here' is present."/>

Note : fill following details !

Name:	Enter name
Email:	Enter email
Password:	Enter password

Register

Already have an account? [Login here](#)

User Registration page

login.html

In login.html, we have created two straightforward inputs: a username and a password that we successfully registered. If we enter the correct email address and password, it will direct us to the user/login profile

page, where we have used the URL for the function to write the file function that we want to display following successful registration.

- **HTML**

```
<html>
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>User Login Form</title>

</head>
<style>
.gfg{
    display: block;
    margin-left: 70px;
    margin-top: -15px;
}

.ok{
    margin-left: 20px;
    font-weight: bold;
}

.btn{
    margin-top: 20px;
    width: 80px;
    height: 25px;
}
```

```
background-color: gray;  
color: white;  
  
}  
.user{  
color: green;  
}  
  
</style>  
<body>  
<div class="container">  
    <h2 class="user"> GFG User Login</h2>  
    <form action="{{ url_for('login') }}" method="post">  
        {% if message is defined and message %}  
            <div class="alert alert-warning"> <strong> {{ message }} ????</strong></div>  
        {% endif %}  
        <br>  
        <div class="form-group">  
            <label class="ok">Email:</label>  
            <input class="gfg" type="email" class="form-control" id="email" name="email" placeholder="Enter email" name="email">  
        </div>  
        <div class="form-group">  
            <label class="pop"> <strong> Password:</strong></label>  
            <input class="gfg" type="password" class="form-control" id="password" name="password" placeholder="Enter password" name="pswd">  
        </div>
```

```

<button class="btn" type="submit" class="btn btn-primary">Login</button>

    <p class="bottom">Don't have an account? <a class="bottom" href="
{{url_for('register')}}> Register here</a></p>

</form>

</div>

</body>

</html>

```

Output:



Welcome To GFG Login 😊

Username:	<input type="text"/>
Password:	<input type="password"/>
<input type="button" value="Login"/>	

User Login.html

user.html

After a successful login, we put a few lines of code to greet the user in these files. We also add a second session. The name code we use during registration means that when we log in, our name will also appear on the screen. Additionally, a button for logging out will appear on the screen; by clicking this button, we can log out and must log in again.

- HTML

```

<html>
<head>
<meta charset="utf-8">

```

```
<meta name="viewport" content="width=device-width, initial-scale=1">

<title>User Account</title>

</head>

<style>

.gfg{
    font-size: 25px;
    color: red;
    font-style: italic;
}

</style>

<body>

<div class="container">

    <div class="row">

        <h2>User Profile</h2>

    </div>

    <br>

    <div class="row">

        Logged in : <strong class="gfg"> {{session.name}} ???? </strong>| <a href="{{url_for('logout')}}"> Logout</a>

    </div>

    <br><br>

    <div class="row">

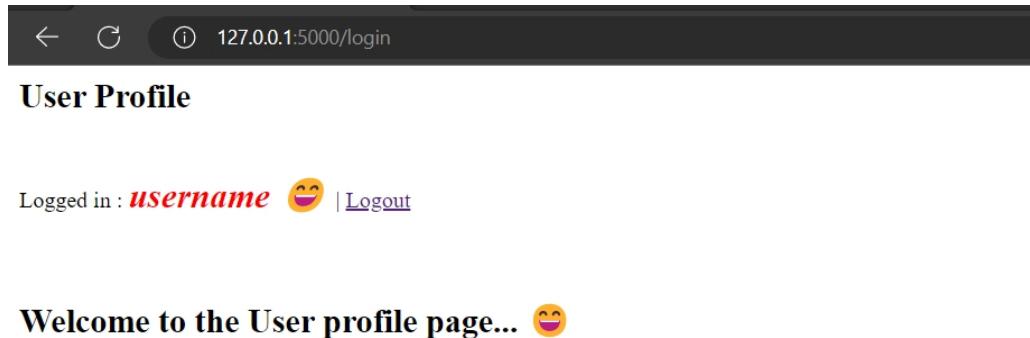
        <h2>Welcome to the User profile page... ????</h2>

    </div>

</div>
```

```
</body>  
</html>
```

Output:



Username display on screen

app.py

Step 1: Import all library

First, the python code is written in a file called app.py. We import all the libraries required for running our application, connecting to MySQL, and performing admin login from the database into this file. Following the import of re(regular expression), which will read the data from our MySQL database, the Python code database, MySQL DB, is used to construct the data for our database. We initialize the flask function and generate a secret key for our flask after importing all modules. The database name, email address, and password are then added to the database.

- Python3

```
# Import all important libraries  
  
from flask import *  
  
from flask_mysqldb import MySQL
```

```
import MySQLdb.cursors

import re

# initialize first flask

app = Flask(__name__)

app.secret_key = 'GeeksForGeeks'

# Set MySQL data

app.config['MYSQL_HOST'] = 'localhost'

app.config['MYSQL_USER'] = 'root'

app.config['MYSQL_PASSWORD'] = ""

app.config['MYSQL_DB'] = 'user-table'

mysql = MySQL(app)
```

Step 2: login and logout functions

Then, we develop a functional login() and develop a session for login and registration for a system that also obtains our data from MySQL. A successfully registered message will also appear on the login page when we successfully register on the register page. In this function, we pass the request to the login form by entering our name, password, and email when we click on enter. It will automatically save on our PHPMyAdmin by MySQL data.

- Python3

```
# Make login function for login and also make
# session for login and registration system
# and also fetch the data from MySQL
```

```
@app.route('/')
@app.route('/login', methods=['GET', 'POST'])

def login():

    message = ""

    if request.method == 'POST' and 'email' in

        request.form and 'password' in request.form:

            email = request.form['email']

            password = request.form['password']

            cursor = mysql.connection.cursor

                (MySQLdb.cursors.DictCursor)

            cursor.execute(
                'SELECT * FROM user WHERE email = % s AND password = % s',
                (email, password,))

            user = cursor.fetchone()

            if user:

                session['loggedin'] = True

                session['userid'] = user['userid']

                session['name'] = user['name']

                session['email'] = user['email']

                message = 'Logged in successfully !'

                return render_template('user.html',
                                      message=message)

            else:

                message = 'Please enter correct email / password !'

    return render_template('login.html', message=message)
```

```
# Make function for logout session

@app.route('/logout')

def logout():

    session.pop('loggedin', None)

    session.pop('userid', None)

    session.pop('email', None)

    return redirect(url_for('login'))
```

Step 3: User Registration

On the login screen, we can log in by providing our email address and password. There are also more flashing notifications, such as “User already exists” if we attempt to register again using the same email address. With the same email address, we are able to create two registered accounts. The username we specified on the registration page will also show up on the profile once we successfully log in. For this instance, we typed “GFG.” “Welcome GFG” will appear once we have successfully logged in. That clarifies the entire code as well as its intended use.

- Python3

```
# Make a register session for registration

# session and also connect to Mysql to code for access

# login and for completing our login

# session and making some flashing massage for error

@app.route('/register', methods=['GET', 'POST'])

def register():

    message = ""

    if request.method == 'POST' and 'name' in

        request.form and 'password' in request.form
```

```
        and 'email' in request.form:

userName = request.form['name']

password = request.form['password']

email = request.form['email']

cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)

cursor.execute('SELECT * FROM user WHERE email = % s',
               (email,))

account = cursor.fetchone()

if account:

    message = 'Account already exists !'

elif not re.match(r'[^@]+@[^@]+\.[^@]+', email):

    message = 'Invalid email address !'

elif not userName or not password or not email:

    message = 'Please fill out the form !'

else:

    cursor.execute(
        'INSERT INTO user VALUES (NULL, % s, % s, % s)',
        (userName, email, password,))

    mysql.connection.commit()

    message = 'You have successfully registered !'

elif request.method == 'POST':

    message = 'Please fill out the form !'

return render_template('register.html', message=message)
```

Complete Code

- Python3

```
# Import all important libraries

from flask import *

from flask_mysqldb import MySQL

import MySQLdb.cursors

import re

# initialize first flask

app = Flask(__name__)

app.secret_key = 'GeeksForGeeks'

# Set MySQL data

app.config['MYSQL_HOST'] = 'localhost'

app.config['MYSQL_USER'] = 'root'

app.config['MYSQL_PASSWORD'] = ""

app.config['MYSQL_DB'] = 'user-table'

mysql = MySQL(app)

@app.route('/')

@app.route('/login', methods=['GET', 'POST'])

def login():

    message = "
```

```
if request.method == 'POST' and 'email' in
    request.form and 'password' in request.form:
        email = request.form['email']

        password = request.form['password']

        cursor = mysql.connection.cursor
            (MySQLdb.cursors.DictCursor)

        cursor.execute(
            'SELECT * FROM user WHERE email = % s AND password = % s',
            (email, password,))

        user = cursor.fetchone()

        if user:
            session['loggedin'] = True
            session['userid'] = user['userid']
            session['name'] = user['name']
            session['email'] = user['email']

            message = 'Logged in successfully !'

            return render_template('user.html',
                message=message)

        else:
            message = 'Please enter correct email / password !'

            return render_template('login.html',
                message=message)

# Make function for logout session
@app.route('/logout')
```

```
def logout():

    session.pop('loggedin', None)
    session.pop('userid', None)
    session.pop('email', None)

    return redirect(url_for('login'))


@app.route('/register', methods=['GET', 'POST'])

def register():

    message = ""

    if request.method == 'POST' and 'name' in request.form

        and 'password' in request.form and 'email' in request.form:

        userName = request.form['name']

        password = request.form['password']

        email = request.form['email']

        cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)

        cursor.execute('SELECT * FROM user WHERE email = % s', (email,))

        account = cursor.fetchone()

        if account:

            message = 'Account already exists !'

        elif not re.match(r'[^@]+@[^@]+\.[^@]+', email):

            message = 'Invalid email address !'

        elif not userName or not password or not email:

            message = 'Please fill out the form !'

    else:
```

```

cursor.execute(
    'INSERT INTO user VALUES (NULL, % s, % s, % s)',
    (userName, email, password, ))
mysql.connection.commit()

message = 'You have successfully registered !'

elif request.method == 'POST':
    message = 'Please fill out the form !'

return render_template('register.html', message=message)

# run code in debug mode

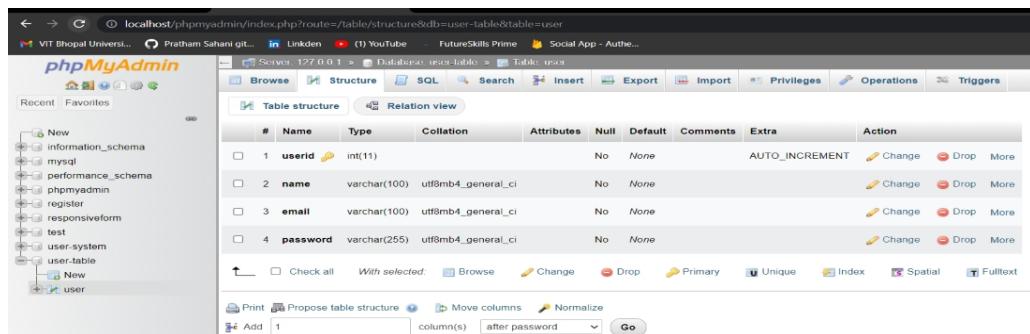
if __name__ == "__main__":
    app.run(debug=True)

```

After writing whole open your terminal and run the following command
 python app.py

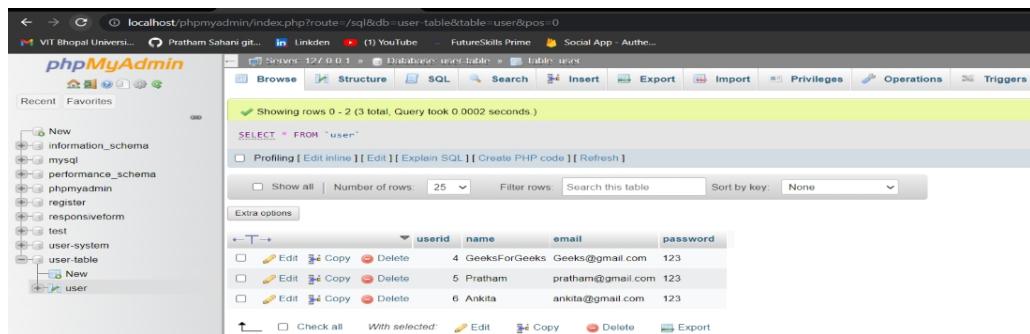
Database Output:

After registering multiple users these outputs will show in your database by watching the video you can understand how the username will display on the screen and how multiple users can register and login.



The screenshot shows the phpMyAdmin interface for a MySQL database named 'user-table'. The left sidebar lists databases like 'information_schema', 'mysql', 'performance_schema', 'phpmyadmin', 'register', 'responsiviform', 'test', 'user-system', and 'user-table'. The 'user-table' database is selected. The main panel displays the 'user' table structure with four columns: 'userid' (int(11)), 'name' (varchar(100)), 'email' (varchar(100)), and 'password' (varchar(255)). The 'password' column has a 'Collation' of 'utf8mb4_general_ci'. The 'Attributes' column shows 'No' for all columns, and 'Default' shows 'None' for 'name', 'email', and 'password', while 'userid' has 'AUTO_INCREMENT'. The 'Extra' column shows 'None' for all. Action buttons for each column include 'Change', 'Drop', and 'More'.

When we register multiple users then these types of interfaces will show on our PHP admin panel.



The screenshot shows the phpMyAdmin interface for a MySQL database. The left sidebar lists databases: New, information_schema, mysql, performance_schema, phpmyadmin, register, responsiveform, test, user-system, user-table, and user. The user-table database is selected. The main area displays the 'user' table with the following data:

	userid	name	email	password
<input type="checkbox"/>	4	GeeksForGeeks	Geeks@gmail.com	123
<input type="checkbox"/>	5	Pratham	pratham@gmail.com	123
<input type="checkbox"/>	6	Ankita	ankita@gmail.com	123

Below the table are standard MySQL management buttons: Edit, Copy, Delete, Check all, With selected, and Export.

CHAPTER 3: Password Hashing with Bcrypt in Flask

In this article, we will use Password Hashing with Bcrypt in Flask using Python. Password **hashing** is the process of converting a plaintext password into a hashed or encrypted format that cannot be easily reverse-engineered to reveal the original password. Bcrypt is a popular hashing algorithm used to hash passwords. It is a password-hashing function that is based on the Blowfish cipher and is designed to be slow and computationally expensive, making it more difficult for attackers to guess or crack passwords.

Key Terminologies:

- **Password Hashing:** The process of converting a plaintext password into a hashed or encrypted format.
- **Bcrypt:** A password-hashing function based on the Blowfish cipher.
- **Salt:** Random data that is used as additional input to a one-way function that hashes a password or passphrase.
- **Hashing Algorithm:** A mathematical function that converts a plaintext password into a fixed-length hash value.
- **Iterations:** The number of times a password is hashed using the bcrypt algorithm.

Stepwise Implement with Bcrypt in Flask

Step 1: Install Flask-Bcrypt

To use Bcrypt in Flask, we need to install the Flask-Bcrypt extension. We can install it using pip.

```
pip install flask-bcrypt
```

Step 2: Import Flask-Bcrypt

We need to import the Bcrypt module from Flask-Bcrypt in our Flask app.

- Python3

```
from flask_bcrypt import Bcrypt
```

Step 3: Create a Bcrypt Object

We need to create a Bcrypt object and pass our Flask app as an argument.

- Python3

```
bcrypt = Bcrypt(app)
```

Step 4: Hash a Password

We need to decode the hashed password using Python `decode('utf-8')` as the `generate_password_hash()` function returns a bytes object. We can hash a password using the **`generate_password_hash()`** function of the Bcrypt object.

- Python3

```
hashed_password = bcrypt.generate_password_hash  
('password').decode('utf-8')
```

Step 5: Verify a Password

The `check_password_hash()` function returns True if the password matches the hashed password, otherwise, it returns False. We can verify a password using the **`check_password_hash()`** function of the Bcrypt object.

- Python3

```
is_valid = bcrypt.check_password_hash(hashed_password, 'password')
```

Complete Code

Here is an example of how to implement Bcrypt in a Flask app.

- Python3

```
from flask import Flask

from flask_bcrypt import Bcrypt

app = Flask(__name__)

bcrypt = Bcrypt(app)

@app.route('/')

def index():

    password = 'password'

    hashed_password = bcrypt.generate_password_hash

        (password).decode('utf-8')

    is_valid = bcrypt.check_password_hash

        (hashed_password, password)

    return f"Password: {password}<br>Hashed Password:

        {hashed_password}<br>Is Valid: {is_valid}"

if __name__ == '__main__':

    app.run()
```

Output:

When we run the Flask app, we will see the following output.

```
← → C ⌂ ⓘ 127.0.0.1:5000
Password: password
Hashed Password: $2b$12$11qL2x0qmmwOW/I4yuzcmuL0mzGK8H1/e02WQ5Ha/2I1HuiBO6kc2
Is Valid: True
```

Output

CHAPTER 4: Role Based Access Control

Flask is a micro-framework written in Python. It is used to create web applications using Python. Role-based access control means certain users can access only certain pages. For instance, a normal visitor should not be able to access the privileges of an administrator. In this article, we will see how to implement this type of access with the help of the **flask-security** library where a Student can access one page, a Staff can access two, a Teacher can access three, and an Admin accesses four pages.

Note: For storing users' details we are going to use flask-sqlalchemy and db-browser for performing database actions. You can find detailed tutorial here.

Creating the Flask Application

Step 1: Create a Python virtual environment.

To avoid any changes in the system environment, it is better to work in a virtual environment.

Step 2: Install the required libraries

```
pip install flask flask-security flask-wtf==1.0.1 flask_sqlalchemy email-validator
```

Step 3: Initialize the flask app.

Import Flask from the flask library and pass `_name_` to Flask. Store this in a variable.

- Python3

```
# import Flask from flask
from flask import Flask
# pass current module (_name_) as argument
# this will initialize the instance
```

```
app = Flask(__name__)
```

Step 4: Configure some settings that are required for running the app. To do this we use **app.config['_____']**. Using it we can set some important things without which the app might not work.

- **SQLALCHEMY_DATABASE_URI** is the path to the database. SECRET_KEY is used for securely signing the session cookie.
- **SECURITY_PASSWORD_SALT** is only used if the password hash type is set to something other than plain text.
- **SECURITY_REGISTERABLE** allows the application to accept new user registrations. SECURITY_SEND_REGISTER_EMAIL specifies whether the registration email is sent.

Some of these are not used in our demo, but they are required to mention explicitly.

- Python3

```
# path to sqlite database
# this will create the db file in instance
# if database not present already

app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///g4g.sqlite3"

# needed for session cookies

app.config['SECRET_KEY'] = 'MY_SECRET'

# hashes the password and then stores in the database

app.config['SECURITY_PASSWORD_SALT'] = "MY_SECRET"

# allows new registrations to application

app.config['SECURITY_REGISTERABLE'] = True

# to send automatic registration email to user

app.config['SECURITY_SEND_REGISTER_EMAIL'] = False
```

Step 5: Import SQLAlchemy for database

Because we are using SQLAlchemy for database operations, we need to import and initialize it into the app using db.init_app(app). The app_context() keeps track of the application-level data during a request

- Python3

```
# import SQLAlchemy for database operations  
# and store the instance in 'db'  
  
from flask_sqlalchemy import SQLAlchemy  
  
db = SQLAlchemy()  
  
db.init_app(app)  
  
  
# runs the app instance  
  
app.app_context().push()
```

Step 6: Create DB Models

For storing the user session information, the **flask-security** library is used. Here to store information of users **UserMixin** is used by importing from the library. Similarly, to store information about the roles of users, **RoleMixin** is used. Both are passed to the database tables' classes.

Here we have created a **user** table for users containing id, email, password, and active status. The **role** is a table that contains the roles created with id and role name. The **roles_users** table contains the information about which user has what roles. It is dependent on the user and role table for user_id and role_id, therefore they are referenced from ForeignKeys.

Then we are creating all those tables using db.create_all() this will make sure that the tables are created in the database for the first time. Keeping or removing it afterward will not affect the app unless a change is made to the structure of the database code.

- Python3

```
# import UserMixin, RoleMixin

from flask_security import UserMixin, RoleMixin


# create table in database for assigning roles

roles_users = db.Table('roles_users',
    db.Column('user_id', db.Integer(), db.ForeignKey('user.id')),
    db.Column('role_id', db.Integer(), db.ForeignKey('role.id'))


# create table in database for storing users

class User(db.Model, UserMixin):

    __tablename__ = 'user'

    id = db.Column(db.Integer, autoincrement=True, primary_key=True)

    email = db.Column(db.String, unique=True)

    password = db.Column(db.String(255), nullable=False, server_default="")

    active = db.Column(db.Boolean())

    # backreferences the user_id from roles_users table

    roles = db.relationship('Role', secondary=roles_users, backref='roled')


# create table in database for storing roles

class Role(db.Model, RoleMixin):

    __tablename__ = 'role'

    id = db.Column(db.Integer(), primary_key=True)

    name = db.Column(db.String(80), unique=True)


# creates all database tables
```

```
@app.before_first_request  
  
def create_tables():  
  
    db.create_all()
```

Step 7: Define User and Role in Database

We need to pass this database information to `flask_security` so as to make the connection between those. For that, we

import `SQLAlchemySessionUserDatastore` and pass the table containing users and then the roles. This datastore is then passed to `Security` which binds the current instance of the app with the data. We also import `LoginManager` and `login_manager` which will maintain the information for the active session. `login_user` assigns the user as a current user for the session.

- Python3

```
# import required libraries from flask_login and flask_security  
  
from flask_login import LoginManager, login_manager, login_user  
  
from flask_security import Security, SQLAlchemySessionUserDatastore  
  
  
# load users, roles for a session  
  
user_datastore = SQLAlchemySessionUserDatastore(db.session, User, Role)  
  
security = Security(app, user_datastore)
```

Step 8: Create a Home Route

The home page of our web app is at the '/' route. So, every time the '/' is routed, the code in `index.html` file will be rendered using a module `render_template`. Below the `@app.route()` decorator, the function needs to be defined so, that code is executed from that function.

- Python3

```

# import the required libraries

from flask import render_template, redirect, url_for

# '/' URL is bound with index() function.

@app.route('/')

# defining function index which returns the rendered html code

# for our home page

def index():

    return render_template("index.html")

```

index.html

Below is the HTML code for index.html. Some logic is applied using the Jinja2 templating engine which behaves similarly to python. The current_user variable stores the information of the currently logged-in user. So, here `{% if current_user.is_authenticated %}` means if a user is logged in show that code: `{% current_user.email %}` is a variable containing the email of the current user. Because the user can have many roles so roles are a list, that's why a for loop is used. Otherwise, the code in `{% else %}` part is rendered, finally getting out of the if block with `{% endif %}`.

- HTML

```

<!-- index.html -->

<!-- links to the pages -->

<a href="/teachers">View all Teachers</a> (Access: Admin)<br><br>
<a href="/staff">View all Staff</a> (Access: Admin, Teacher)<br><br>
<a href="/students">View all Students</a> (Access: Admin, Teacher, Staff)<br><br>
<a href="/mydetails">View My Details</a> (Access: Admin, Teacher, Staff, Student)

```

```
<br><br>

<!-- Show only if user is logged in -->

{% if current_user.is_authenticated %}

    <!-- Show current users email -->

    <b>Current user</b>: {{current_user.email}}

    <!-- Current users roles -->

    | <b>Role</b>: {% for role in current_user.roles%}

        {{role.name}}


    {% endfor %} <br><br>

    <!-- link for logging out -->

    <a href="/logout">Logout</a>

<!-- Show if user is not logged in -->

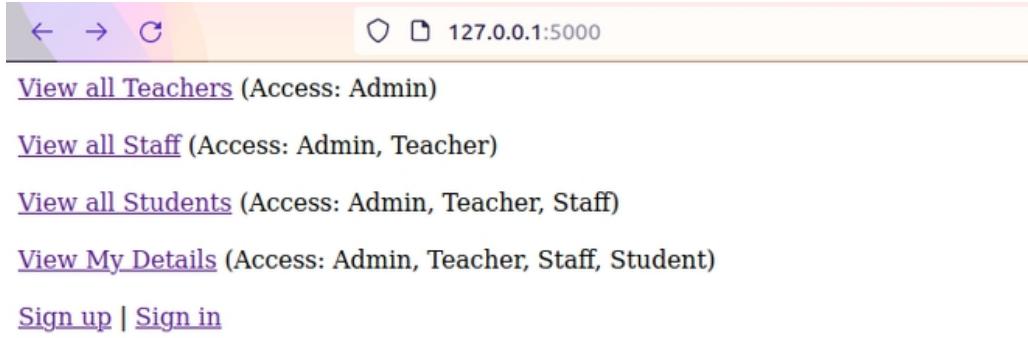
{% else %}

    <a href="/signup">Sign up</a> | <a href="/signin">Sign in</a>

{% endif %}

<br><br>
```

Output:



Step 9: Create Signup Route

If the user visits the '/signup' route the request is GET, so the else part will render this HTML page, and if the form is submitted the code in if the condition that is POST method is executed.

The data in the HTML form is requested using the request module. First, we check if the user already exists in the database by querying for the user using the email provided and passing the msg according to that. If not then we add the user and append the chosen role to the roles_users DB table, for this we query for the role using the id that we will get from options in the radio button, this will return an object containing all the column attributes of that role, in this case, the id and name of the role. And then log the user in for the user's current session with **login_user(user)**.

- Python3

```
# import 'request' to request data from html

from flask import request

# signup page

@app.route('/signup', methods=['GET', 'POST'])

def signup():

    msg=""
```

```
# if the form is submitted

if request.method == 'POST':

# check if user already exists

    user = User.query.filter_by(email=request.form['email']).first()

    msg=""

# if user already exists render the msg

    if user:

        msg="User already exist"

        # render signup.html if user exists

        return render_template('signup.html', msg=msg)

# if user doesn't exist

# store the user to database

    user = User(email=request.form['email'], active=1,
password=request.form['password'])

    # store the role

    role = Role.query.filter_by(id=request.form['options']).first()

    user.roles.append(role)

# commit the changes to database

db.session.add(user)

db.session.commit()

# login the user to the app

# this user is current user

login_user(user)

# redirect to index page
```

```
return redirect(url_for('index'))  
  
# case other than submitting form, like loading the page itself  
else:  
  
    return render_template("signup.html", msg=msg)
```

signup.html page:

Here in the form, the action is '#' which means after submitting the form the current page itself is loaded. The method in the form is *POST* because we are creating a new entry in the database. There are fields for email, password, and choosing a role with a radio button. In the radio button, the *value* should be different because that's the main differentiator of the chosen role.

Also, an if condition is applied using Jinja2, `{% if %}` which checks that if a user is logged in, and if not `{%else %}` then only shows the form otherwise just shows the already logged-in message.

- HTML

```
<!-- signup.html -->  
  
<h2>Sign up</h2>  
  
<!-- Show only if user is logged in -->  
{% if current_user.is_authenticated %}  
  
    You are already logged in.  
  
<!-- Show if user is NOT logged in -->  
{% else %}  
  
{{ msg }}<br>
```

```

<!-- Form for signup -->

<form action="#" method="POST" id="signup-form">

    <label>Email Address </label>

    <input type="text" name="email" required /><br><br>

    <label>Password </label>

    <input type="password" name="password" required/><br><br>

    <!-- Options to choose role -->

    <!-- Give the role ids in the value -->

    <input type="radio" name="options" id="option1" value=1 required> Admin
</input>

    <input type="radio" name="options" id="option2" value=2> Teacher </input>

    <input type="radio" name="options" id="option3" value=3> Staff </input>

    <input type="radio" name="options" id="option3" value=4> Student </input>
<br>

    <br>

    <button type="submit">Submit</button><br><br>

    <!-- Link for signin -->

    <span>Already have an account?</span>

    <a href="/signin">Sign in</a>

</form>

<!-- End the if block -->

{%- endif %}

```

Output:

← → ⌂ 127.0.0.1:5000/signup

Sign up

Email Address student@abc.com

Password

Admin Teacher Staff Student

Already have an account? [Sign in](#)

Step 10: Create Signin Route

As you might have noticed we are using two methods GET, and POST. That is because we want to know if the user has just loaded the page (GET) or submitted the form (POST). Then we check if the user exists by querying the database. If the user exists then we see if the password matches. If both are validated the user is logged in using **login_user(user)**. Otherwise, the msg is passed to HTML accordingly i.e., if the password is wrong msg is set to "Wrong password" and if the user doesn't exist then the msg is set to "User doesn't exist".

- Python3

```
# signin page

@app.route('/signin', methods=['GET', 'POST'])

def signin():

    msg=""

    if request.method == 'POST':

        # search user in database

        user = User.query.filter_by(email=request.form['email']).first()

        # if exist check password
```

```

if user:

    if user.password == request.form['password']:

        # if password matches, login the user

        login_user(user)

        return redirect(url_for('index'))

    # if password doesn't match

else:

    msg="Wrong password"

# if user does not exist

else:

    msg="User doesn't exist"

return render_template('signin.html', msg=msg)

else:

    return render_template("signin.html", msg=msg)

```

signin.html

Similar to the signup page, check if a user is already logged in, if not then show the form asking for email and password. The form method should be POST. Ask in the form for, email and password. We can also show links for sign-up optionally.

- HTML

```
<!-- signin.html -->
```

```
<h2>Sign in</h2>
```

```

<!-- Show only if user is logged in -->
{% if current_user.is_authenticated %}

    You are already logged in.

<!-- Show if user is NOT logged in -->
{% else %}

<!-- msg that was passed while rendering template -->
{{ msg }}<br>

<form action="#" method="POST" id="signin-form">

    <label>Email Address </label>

    <input type="text" name="email" required /><br><br>

    <label>Password </label>

    <input type="password" name="password" required/><br><br>

    <input class="btn btn-primary" type="submit" value="Submit"><br><br>

    <span>Don't have an account?</span>

    <a href="/signup">Sign up</a>

</form>

{% endif %}

```

Output:



Step 11: Create a Teacher Route

We are passing the users with the role of Teacher to the HTML template. On the home page if we click any link then it will load the same page if the user is not signed in. If the user is signed in we want to give Role Based Access so that the user with the role:

- **Students** can access *View My Details* page.
- **Staff** can access *View My Details* and *View all Students* pages.
- **The teacher** can access *View My Details*, *View all Students*, and *View all Staff* pages.
- **Admin** can access *View My Details*, *View all Students*, *View all Staff*, and *View all Teachers* pages.

We need to import, **roles_accepted**: this will check the database for the role of the user and if it matches the specified roles then only the user is given access to that page. The teacher's page can be accessed by Admin only using `@roles_accepted('Admin')`.

- Python3

```
# to implement role based access
# import roles_accepted from flask_security
from flask_security import roles_accepted

# for teachers page
```

```
@app.route('/teachers')

# only Admin can access the page

@roles_accepted('Admin')

def teachers():

    teachers = []

    # query for role Teacher that is role_id=2

    role_teachers = db.session.query(roles_users).filter_by(role_id=2)

    # query for the users' details using user_id

    for teacher in role_teachers:

        user = User.query.filter_by(id=teacher.user_id).first()

        teachers.append(user)

    # return the teachers list

    return render_template("teachers.html", teachers=teachers)
```

teachers.html

The `teachers` passed in the `render_template` is a list of objects, containing all the columns of the `user` table, so we're using Python for loop in jinja2 to show the elements in the list in HTML ordered list tag.

- HTML

```
<!-- teachers.html -->

<h3>Teachers</h3>

<!-- list that shows all teachers' email -->

<ol>

    {% for teacher in teachers %}
```

```
<li>  
{{teacher.email}}  
</li>  
{% endfor %}  
</ol>
```

Output:



Teachers

1. teacher@abc.com
2. teacher1@abc.com
3. teacher3@any.in

Step 12: Create staff, student, and mydetail Routes

Similarly, routes for other pages are created by adding the roles to the decorator **@roles_accepted()**.

- Python3

```
# for staff page  
  
@app.route('/staff')  
  
# only Admin and Teacher can access the page  
  
@roles_accepted('Admin', 'Teacher')  
  
def staff():  
  
    staff = []  
  
    role_staff = db.session.query(roles_users).filter_by(role_id=3)
```

```

for staf in role_staff:

    user = User.query.filter_by(id=staf.user_id).first()

    staff.append(user)

return render_template("staff.html", staff=staff)

# for student page

@app.route('/students')

# only Admin, Teacher and Staff can access the page

@roles_accepted('Admin', 'Teacher', 'Staff')

def students():

    students = []

    role_students = db.session.query(roles_users).filter_by(role_id=4)

    for student in role_students:

        user = User.query.filter_by(id=student.user_id).first()

        students.append(user)

    return render_template("students.html", students=students)

# for details page

@app.route('/mydetails')

# Admin, Teacher, Staff and Student can access the page

@roles_accepted('Admin', 'Teacher', 'Staff', 'Student')

def mydetails():

    return render_template("mydetails.html")

```

staff.html

Here, we are iterating all the staff and extracting their email IDs.

- HTML

```
<! staff.html -->

<h3>Staff</h3>

<ol>

{% for staf in staff %}

<li>

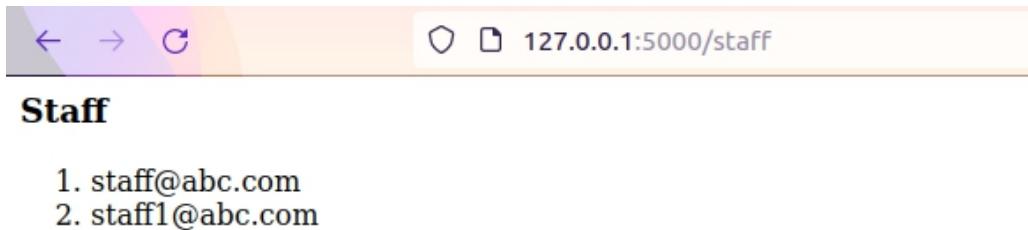
{{staf.email}}

</li>

{% endfor %}

</ol>
```

Output:



student.html

Here, we are iterating all the students and extracting their email IDs.

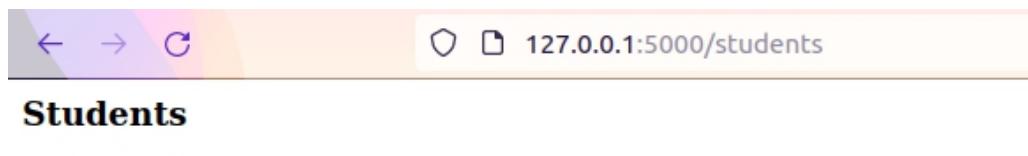
- HTML

```
<!-- students.html -->

<h3>Students</h3>

<ol>
  {% for student in students %}
    <li>
      {{student.email}}
    </li>
  {% endfor %}
</ol>
```

Output:



mydetails.html

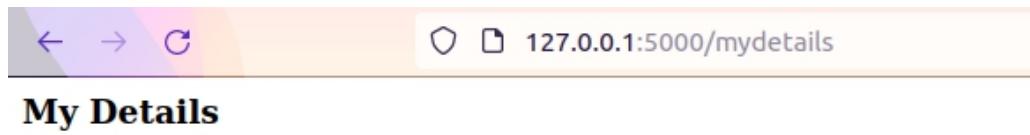
Similar to the index page, to show the role use a for loop from Jinja2, because a user can have more than one role i.e., current_user.roles is a list of roles that were queried from the database.

- HTML

```
<!-- mydetails.html -->
```

```
<h3>My Details</h3><br>
<b>My email</b>: {{current_user.email}}
| <b>Role</b>: {% for role in current_user.roles%}
    {{role.name}}
{% endfor %} <br><br>
```

Output:



Step 13: Finally, Add code Initializer.

Here, debug is set to True. When in a development environment. It can be set to False when the app is ready for production.

- Python3

```
#for running the app
if __name__ == "__main__":
    app.run(debug = True)
```

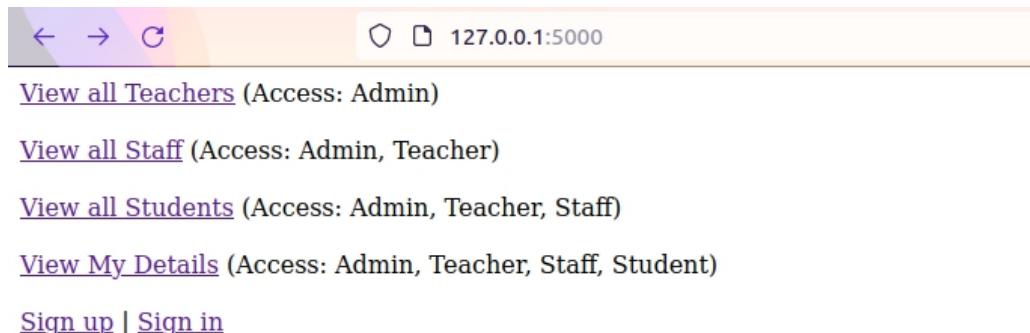
Now test your app by running the below command in the terminal.

python app.py

Go to:

<http://127.0.0.1:5000>

Output:



CHAPTER 5: Use Flask-Session in Python Flask

Flask Session -

- Flask-Session is an extension for Flask that supports **Server-side Session** to your application.
- The Session is the **time between** the **client logs in** to the server and **logs out** of the server.
- The data that is required to be saved in the Session is stored in a **temporary directory on the server**.
- The data in the Session is stored on the **top of cookies** and signed by **the server cryptographically**.
- **Each client** will have their **own session** where their own data will be stored in their session.

Uses of Session

- Remember each user when they log in
- Store User-specific website settings (theme)
- Store E-Commerce site user items in the cart

This article assumes you are familiar with flask basics. Checkout – [Flask – \(Creating first simple application\)](#) to learn how to make a simple web application in flask.

Installation

Install the extension with the following command

```
$ easy_install Flask-Session
```

Alternatively, if you have pip installed

```
$ pip install Flask-Session
```

Configuring Session in Flask

- The Session instance is not used for direct access. You should always use `flask_session`.

- The First line (session) from the flask is in such a way that each of us as a user gets our own version of the session.

- Python3

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

```
from flask_session import Session
```

This is specific to the flask_session library only

- **SESSION_PERMANENT = False** - So this session has a default time limit of some number of minutes or hours or days after which it will expire.
- **SESSION_TYPE = “filesystem”** - It will store in the hard drive (these files are stored under a /flask_session folder in your config directory.) or any online ide account, and it is an alternative to using a Database or something else like that.

- Python3

```
app = Flask(__name__)

app.config["SESSION_PERMANENT"] = False

app.config["SESSION_TYPE"] = "filesystem"

Session(app)
```

Remember User After Login

So we will start making two basic pages and their route called **index.html** and **login.html**

- **login.html** contains a form in which the user can fill their name and submit
- **index.html** is the main page

- Python3

```
@app.route("/")
def index():

    return render_template('index.html')

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

def login():

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

- We need to record the username in the session when they submit the form
- And we are using a dictionary in python where “name” is the key = request.form.get(“name”) is a value

- Python3

```
@app.route("/login", methods=["POST", "GET"])

def login():

    # if form is submitted

    if request.method == "POST":

        # record the user name

        session["name"] = request.form.get("name")

        # redirect to the main page

        return redirect("/")

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

- After storing the user name we need to check whenever user lands on the index page that any session with that user name exists or not.
- If the user name doesn't exist then redirect to the login page.

- Python3

```
@app.route("/")
def index():

    # check if the users exist or not

    if not session.get("name"):

        # if not there in the session then redirect to the login page

        return redirect("/login")

    return render_template('index.html')
```

- After successfully remember the **user we also need a way to logout the users.**
- So whenever the user **clicks logout** change the user **value to none** and redirect them to the **index page**.

- Python3

```
@app.route("/logout")

def logout():

    session["name"] = None

    return redirect("/")
```

Complete Project -

- Python3

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

# The Session instance is not used for direct access, you should always use flask.session

from flask_session import Session


app = Flask(__name__)

app.config["SESSION_PERMANENT"] = False

app.config["SESSION_TYPE"] = "filesystem"

Session(app)


@app.route("/")
def index():

    if not session.get("name"):

        return redirect("/login")

    return render_template('index.html')


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

def login():

    if request.method == "POST":

        session["name"] = request.form.get("name")

        return redirect("/")

    return render_template("login.html")


@app.route("/logout")
```

```
def logout():

    session["name"] = None

    return redirect("/")
```

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

index.html

- We can also use **session.name** to excess the value from the **session**.

- **HTML**

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

{% block y %}

{% if session.name %}

    You are Register {{ session.name }} <a href="/logout">logout</a>.

{% else %}

    You are not Register. <a href="/login">login</a>.

{% endif %}

{% endblock %}
```

login.html

- **HTML**

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

{% block y %}

<h1> REGISTER </h1>

<form action="/login" method="POST">

    <input placeholder="Name" autocomplete="off" type="text" name="name">

    <input type="submit" name="Register">

</form>

{% endblock %}
```

layout.html

- **HTML**

```
<!DOCTYPE html>

<html lang="en">

    <head>

        <meta name="viewport" content="initial-scale=1, width=device-width">

        <title> flask </title>

    </head>

    <body>

        {% block y %}{% endblock %}

    </body>

</html>
```

Output -

login.html

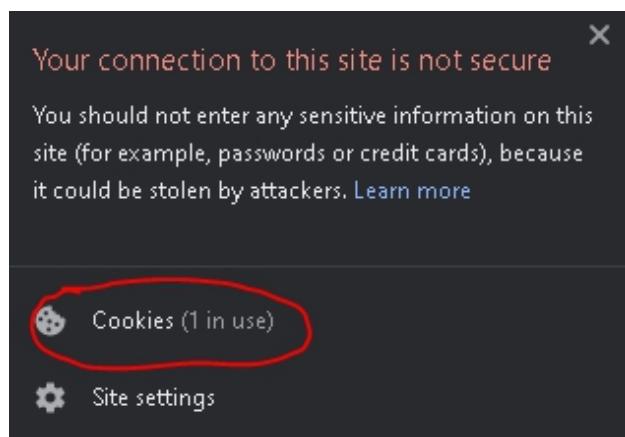
REGISTER

index.html

You are Register Rahul [logout](#).

You can also see your generated session.

```
▼ Request Headers    view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,be;q=0.8,bg;q=0.7
Cache-Control: max-age=0
Connection: keep-alive
Cookie: session=8c5d157b-2d19-478e-bab0-3171aac1ab99
```



CHAPTER 6: Using JWT for user authentication in Flask

Pre-requisite: Basic knowledge about [JSON Web Token \(JWT\)](#).

I will be assuming you have the basic knowledge of JWT and how JWT works. If not, then I suggest reading the linked Geeksforgeeks article. Let's jump right into the setup. Ofcourse, you need **python3** installed on your system. Now, follow along with me. I will be using a **virtual environment** where I will install the libraries which is undoubtedly the best way of doing any kind of development.

- First create a folder named **flask project** and change directory to it. If you are on linux then type the following in your terminal.

```
mkdir "flask project" && cd "flask project"
```

- Now, create a virtual environment. If you are on linux then type the following in your terminal.

```
python3 -m venv env
```

Note: If you get any error then that means **venv** isn't installed in your system. To install it, type *sudo apt install python3-venv* in your terminal and then you are good to go. If you are on windows then use something like *virtualenv* to make a virtual environment.

This will create a folder named **venv** in the **flask project** which will contain the project specific libraries.

- Now create a file named *requirements.txt* and add the following lines in it.

```
Flask-RESTful==0.3.8
```

```
PyJWT==1.7.1
```

```
Flask-SQLAlchemy==2.4.1
```

- Now, lets install these libraries for this project. To do so, first we need to activate the virtual environment. To do so, type the

following in your terminal.

```
source env/bin/activate
```

Note: If you are on windows then it would be *Scripts* instead of *bin*. Now, its time to install the libraries. To do so, again type the following in your terminal.

```
pip install -r requirements.txt
```

Now, we are done with the setup part. Lets now start writing the actual code. Before beginning with the code, I would like to make something clear. I would be writing the entire code in a single file, i.e. the database models and the routes all together, which is not a good practice and definitely not manageable for larger projects. Try keeping creating separate python files or modules for routes and database models. With that cleared out, lets directly jump into the writing the actual code. I will be adding inline comments explaining every part of the code.

Create a python file called *app.py* and type the following code in it.

- Python3

```
# flask imports

from flask import Flask, request, jsonify, make_response

from flask_sqlalchemy import SQLAlchemy

import uuid # for public id

from werkzeug.security import generate_password_hash, check_password_hash

# imports for PyJWT authentication

import jwt

from datetime import datetime, timedelta

from functools import wraps


# creates Flask object

app = Flask(__name__)

# configuration
```

```
# NEVER HARDCODE YOUR CONFIGURATION IN YOUR CODE
# INSTEAD CREATE A .env FILE AND STORE IN IT

app.config['SECRET_KEY'] = 'your secret key'

# database name

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///Database.db'

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

# creates SQLALCHEMY object

db = SQLAlchemy(app)

# Database ORMs

class User(db.Model):

    id = db.Column(db.Integer, primary_key = True)

    public_id = db.Column(db.String(50), unique = True)

    name = db.Column(db.String(100))

    email = db.Column(db.String(70), unique = True)

    password = db.Column(db.String(80))

# decorator for verifying the JWT

def token_required(f):

    @wraps(f)

    def decorated(*args, **kwargs):

        token = None

        # jwt is passed in the request header

        if 'x-access-token' in request.headers:

            token = request.headers['x-access-token']

        # return 401 if token is not passed

        if not token:
```

```
        return jsonify({'message' : 'Token is missing !!'}), 401

try:
    # decoding the payload to fetch the stored details

    data = jwt.decode(token, app.config['SECRET_KEY'])

    current_user = User.query\
        .filter_by(public_id = data['public_id'])\
        .first()

except:
    return jsonify({
        'message' : 'Token is invalid !!'
    }), 401

# returns the current logged in users context to the routes

return f(current_user, *args, **kwargs)

return decorated

# User Database Route

# this route sends back list of users

@app.route('/user', methods =['GET'])

@token_required

def get_all_users(current_user):

    # querying the database

    # for all the entries in it

    users = User.query.all()

    # converting the query objects

    # to list of jsons

    output = []
```

```
for user in users:
    # appending the user data json
    # to the response list
    output.append({
        'public_id': user.public_id,
        'name' : user.name,
        'email' : user.email
    })

return jsonify({'users': output})

# route for logging user in
@app.route('/login', methods =['POST'])

def login():
    # creates dictionary of form data
    auth = request.form

    if not auth or not auth.get('email') or not auth.get('password'):
        # returns 401 if any email or / and password is missing
        return make_response(
            'Could not verify',
            401,
            {'WWW-Authenticate' : 'Basic realm ="Login required !!"'}
        )

    user = User.query\
        .filter_by(email = auth.get('email'))\
        .first()
```

```
if not user:  
    # returns 401 if user does not exist  
  
    return make_response(  
        'Could not verify',  
        401,  
        {'WWW-Authenticate' : 'Basic realm ="User does not exist !!!'}  
    )  
  
if check_password_hash(user.password, auth.get('password')):  
    # generates the JWT Token  
  
    token = jwt.encode({  
        'public_id': user.public_id,  
        'exp' : datetime.utcnow() + timedelta(minutes = 30)  
    }, app.config['SECRET_KEY'])  
  
    return make_response(jsonify({'token' : token.decode('UTF-8')}), 201)  
  
# returns 403 if password is wrong  
  
return make_response(  
    'Could not verify',  
    403,  
    {'WWW-Authenticate' : 'Basic realm ="Wrong Password !!!'}  
)  
  
# signup route  
@app.route('/signup', methods =['POST'])  
  
def signup():  
    # creates a dictionary of the form data  
  
    data = request.form
```

```
# gets name, email and password

name, email = data.get('name'), data.get('email')

password = data.get('password')


# checking for existing user

user = User.query\

    .filter_by(email = email)\

    .first()

if not user:

    # database ORM object

    user = User(


        public_id = str(uuid.uuid4()),

        name = name,

        email = email,


        password = generate_password_hash(password)

    )

    # insert user

    db.session.add(user)

    db.session.commit()


    return make_response('Successfully registered.', 201)

else:

    # returns 202 if user already exists

    return make_response('User already exists. Please Log in.', 202)


if __name__ == "__main__":
    # setting debug to True enables hot reload
    # and also provides a debugger shell
```

```
# if you hit an error while running the server  
  
app.run(debug = True)
```

Now, our code is ready. We now need to create the database first and then the table *User* from the ORM (Object Relational Mapping). To do so, first start the `python3` interpreter in your terminal. You can do that by typing `python3` in your terminal and that should do the trick for you.

Next you need to type the following in your `python3` interpreter:

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

So, what this does is first it imports the database object and then calls the `create_all()` function to create all the tables from the ORM. It should look something like this.

```
Python 3.8.2 (default, Apr 27 2020, 15:53:34)  
[GCC 9.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> from app import db  
>>> db.create_all()  
>>> █
```

Now that our actual code is ready, let's test it out. I recommend using [postman](#) for testing out the APIs. You can use something like CURL but I will be using postman for this tutorial.

To start testing our api, first we need to run our API. To do so, open up a terminal window and type the following in it.

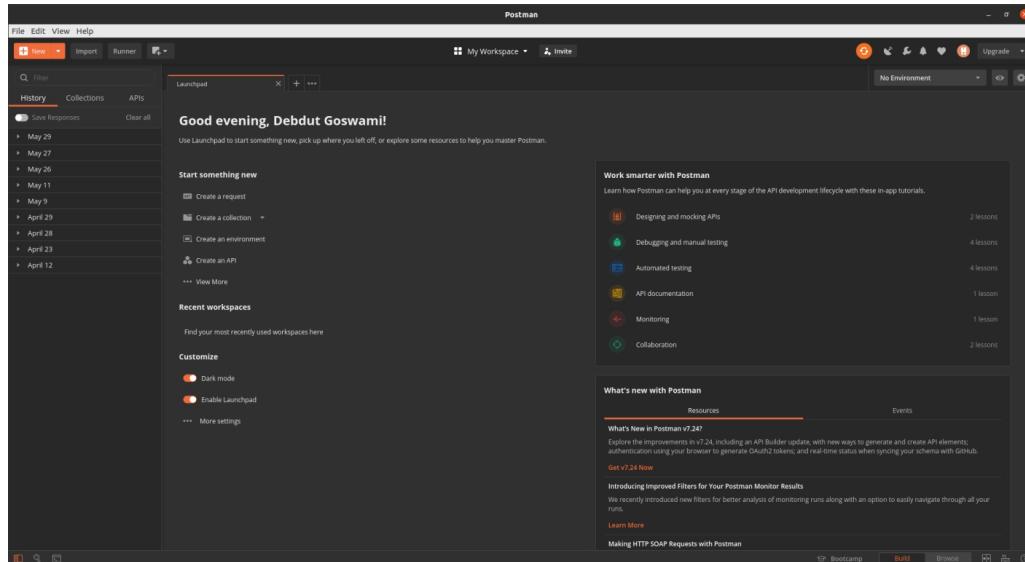
```
python app.py
```

You should see an output like this

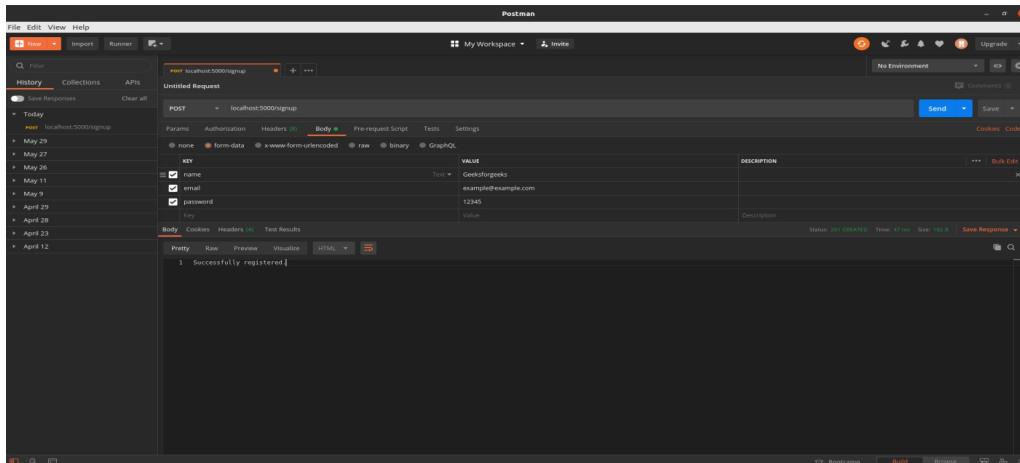
```
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
    Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 793-651-024
```

If you get any error then make sure all your syntax and indentation are correct. You can see that our api is running on <http://localhost:5000/>. Copy this url. We will use this url along with the routes to test the api.

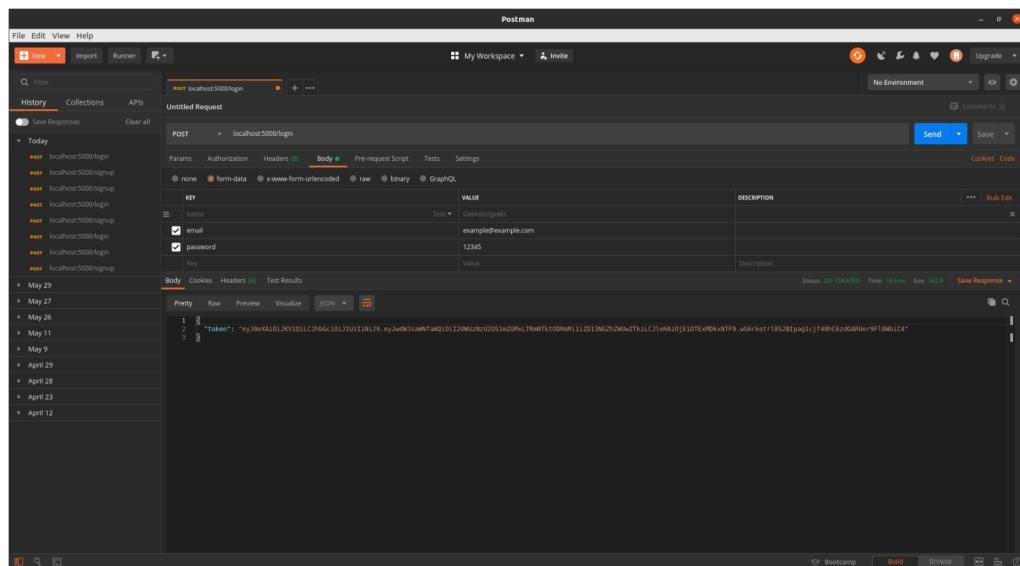
Now, open up *Postman*. You should be greeted with the following screen.



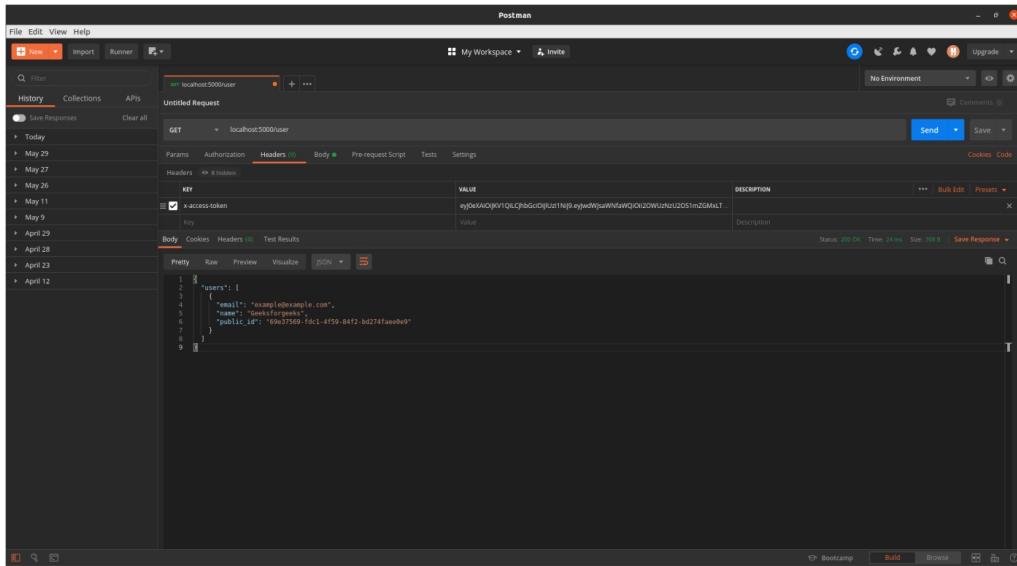
Now, click on the + sign and enter the url `localhost:5000/signup` change request type to *POST*, then select *Body* and then *form-data* and enter the data as key-value pair and then click on *Send* and you should get a response. It should look something like this.



So, we are registered. Now lets login. To do that just change the endpoint to `/login` and untick the `Name` field and click on `Send`. You should get a JWT as a response. Note down that JWT. That will be our token and we will need to send that token along with every subsequent requests. This token will identify us as logged in.



The JSON contains the token. Note it down. Next try to fetch the list of users. To do that, change the endpoint to `/user` and then in the headers section, add a field as `x-access-token` and add the JWT token in the value and click on *Send*. You will get the list of users as JSON.



So, this is how you can perform authentication with JWT in Flask. I recommend you to practice more with JWTs and user authentication to get your concepts more clear.

CHAPTER 7: Flask Cookies

Flask is a lightweight, web development framework built using python language. Generally, for building websites we use HTML, CSS and JavaScript but in flask the python scripting language is used for developing the web-applications.

To know more about Flask and how to run an application in Flask: Flask - First Application Creation

What are Cookies?

Technically, cookies track user activity to save user information in the browser as key-value pairs, which can then be accessed whenever necessary by the developers to make a website easier to use. These enhances the personal user experience on a particular website by remembering your logins, your preferences and much more.

For running a flask application, we need to have some prerequisites like installing the flask.

Prerequisites:

Use the upgraded version of pip by below command in your terminal. In this article, I am using Visual Studio Code to run my flask applications.

```
Python -m pip install --upgrade pip
```

```
Python -m pip install flask
```

Setting Cookies in Flask:

set_cookie() method: Using this method we can generate cookies in any application code. The syntax for this cookies setting method:
Response.set_cookie(key, value = "", max_age = None, expires = None, path = '/', domain = None,
secure = None, httponly = False)

Parameters:

- key – Name of the cookie to be set.
- value – Value of the cookie to be set.
- max_age – should be a few seconds, None (default) if the cookie should last as long as the client's browser session.

- expires – should be a datetime object or UNIX timestamp.
- domain – To set a cross-domain cookie.
- path – limits the cookie to given path, (default) it will span the whole domain.

Example:

- Python3

```
from flask import Flask, request, make_response

app = Flask(__name__)

# Using set_cookie( ) method to set the key-value pairs below.

@app.route('/setcookie')
def setcookie():

    # Initializing response object

    resp = make_response('Setting the cookie')
    resp.set_cookie('GFG','ComputerScience Portal')

    return resp

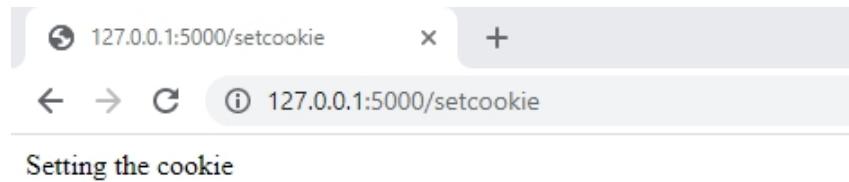
app.run()
```

Running the code in Visual Studio Code application.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL  
  
C:\Users\admin\Desktop>python flask-cookies.py  
* Serving Flask app "flask-cookies" (lazy loading)  
* Environment: production  
WARNING: This is a development server. Do not use it in a production deployment.  
Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)  
127.0.0.1 - - [15/Dec/2022 22:10:51] "GET /setcookie HTTP/1.1" 200 -
```

My Visual Studio Code terminal

Output: Go to the above-mentioned url in the terminal -For Example -
`http://127.0.0.1:5000/route-name`. Here the route-name is setcookie.



Output

Getting Cookies in Flask: `cookies.get()`

This `get()` method retrieves the cookie value stored from the user's web browser through the `request` object.

- Python3

```
from flask import Flask, request, make_response  
  
app = Flask(__name__)
```

```
# getting cookie from the previous set_cookie code

@app.route('/getcookie')

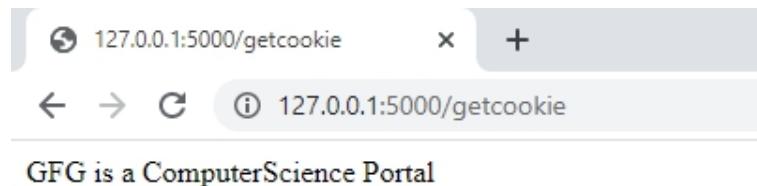
def getcookie():

    GFG = request.cookies.get('GFG')

    return 'GFG is a '+ GFG

app.run()
```

Output:



Login Application in Flask using cookies

Let's develop a simple login page with Flask using cookies. First, we are creating the main python file – app.py in our code editor. Next, we will create the UI of our web page which is Login.html where the user can enter his username and password. In this app.py, we are storing the username as cookie to know which user logged in to the website. In the below code, we are requesting the stored cookie from the browser and displaying it on the next page which routes to user details page.

app.py

- Python3

```
from flask import Flask, request, make_response, render_template

app = Flask(__name__)

@app.route('/', methods = ['GET'])

def Login():

    return render_template('Login.html')

@app.route('/details', methods = ['GET','POST'])

def login():

    if request.method == 'POST':

        name = request.form['username']

        output = 'Hi, Welcome '+name+ ""

        resp = make_response(output)

        resp.set_cookie('username', name)

    return resp

app.run(debug=True)
```

Login.html

- HTML

```
<!DOCTYPE html>
```

```
<html>

    <head>
        <title>Login</title>
    </head>

    <body>

        <form method="post" action="/details">

            <label for="username">Username</label>

            <input type="text" name="username" id="username"/>

            <br/>

            <br>

            <label for="password">Password</label>

            <input type="password" name="password" id="password"/>

            <br/>

            <br>

            <input type="submit" name="submit" id="submit" value="Login"/>

        </form>

    </body>

</html>
```

Output:

A screenshot of a web browser window titled "Login". The address bar shows the URL "127.0.0.1:5000". The page contains two input fields labeled "Username" and "Password", and a "Login" button.

Username

Password

Login

Login Page

A screenshot of a web browser window titled "Login". The address bar shows the URL "127.0.0.1:5000". The page contains two input fields labeled "Username" and "Password", and a "Login" button. The "Username" field has the value "Greeshma" and the "Password" field has the value "*****".

Username

Password

Login

User Logging

A screenshot of the Chrome DevTools Application tab. The left sidebar shows "Storage" with sections for "Local Storage", "Session Storage", "IndexedDB", "Web SQL", and "Cookies". Under "Cookies", there is one entry for the domain "http://127.0.0.1:5000" with the name "username" and value "Greeshma".

Name	Value	Domain	Path	Expires / Max-Age	Size	Http...	Sec...	Same...	Part...	Pr...
username	Greeshma	127.0.0.1	/	Session	16					Me...

User Logged In and Cookie Tracker

From the above image, the user can also see the cookies of a website. Here, ‘username’ is the key and its value is ‘Greeshma’ which reminds us that cookies are generally key-value pairs. To see the cookies in your browser, click the last 3 dots on the browser’s right corner>> More Tools>>Developer Tools>>Application window.

Getting website Visitors counted through cookies

In the below code, we want to know the number of visitors visiting our website. We are first retrieving the visitor’s count by the usage of cookies. But there is no variable named visitors count that we created previously. As this key(visitors count) is not present in the dictionary, it will take the value of 0 that is specified in the second parameter as per the dictionary collection in python. Hence, for the first-time visitors count=0, then incrementing the count according to the user’s visit to the website. The make_response() gets the response object and is used for setting the cookie.

- Python3

```
from flask import Flask, request, make_response

app = Flask(__name__)

app.config['DEBUG'] = True


@app.route('/')

def vistors_count():

    # Converting str to int

    count = int(request.cookies.get('visitors count', 0))

    # Getting the key-visitors count value as 0
```

```
count = count+1

output = 'You visited this page for '+str(count) + ' times'

resp = make_response(output)

resp.set_cookie('visitors count', str(count))

return resp
```

```
@app.route('/get')

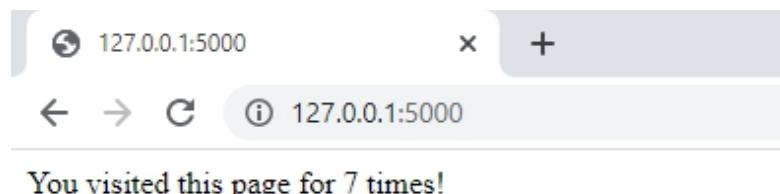
def get_vistors_count():

    count = request.cookies.get('visitors count')

    return count

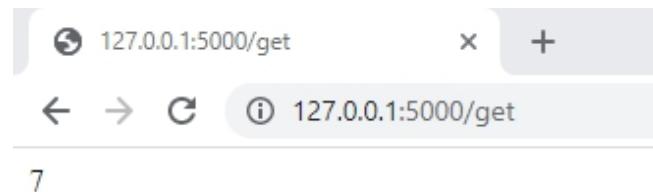
app.run()
```

Output: Url – http://127.0.0.1:5000



Output

Url for below output- http://127.0.0.1:5000/get



Visitors count output using cookies

In the above output screenshot, the value of the website visitors count is retrieved using `request.cookies.get()` method.

Cookies Tracking in Browser -

A screenshot of the Chrome DevTools Application tab. The left sidebar shows "Storage" with "Cookies" expanded, listing "http://127.0.0.1:5000" under it. The main panel displays a table of cookies. One cookie is listed: Name is "visitors count", Value is "3", Domain is "127.0.0.1", Path is "/", Expires is "Session", Size is "15", and HttpOnly is checked. The "Security" column shows "Med...".

Cookie Tracker for visitors count application

The flask cookies can be secured by putting the secure parameter in `response.set_cookie('key', 'value', secure = True)` and it is the best-recommended practice to secure cookies on the internet.

CHAPTER 8: Return a JSON response from a Flask API

Flask is one of the most widely used python micro-frameworks to design a REST API. In this article, we are going to learn how to create a simple REST API that returns a simple JSON object, with the help of a flask.

Prerequisites: [Introduction to REST API](#)

What is a REST API?

REST stands for Representational State Transfer and is an architectural style used in modern web development. It defines a set of rules/constraints for a web application to send and receive data. In this article, we will build a REST API in Python using the Flask framework. Flask is a popular micro framework for building web applications.

Approaches: We are going to write a simple flask API that returns a JSON response using two approaches:

1. Using Flask jsonify object.
2. Using the flask_restful library with Flask.

Libraries Required:

- Install the python *Flask* library using the following command:

```
pip install Flask
```

- Install the *flask-restful* library using the following command:

```
pip install Flask-RESTful
```

Approach 1: Using Flask jsonify object - In this approach, we are going to return a JSON response using the flask jsonify method. We are not going to use the flask-restful library in this method.

- Create a new python file named ‘main.py’.
- import Flask, jsonify, and request from the flask framework.
- Register the web app into an app variable using the following syntax.

```
app = Flask(__name__)
```

- Create a new function named ‘ReturnJSON’. This function is going to return the sample JSON response.
- Route the ‘ReturnJSON’ function to your desired URL using the following syntax.

```
@app.route('/path_of_the_response', methods = ['GET'])
```

```
def ReturnJSON():
```

```
    pass
```

- Inside the ‘ReturnJSON’ function if the request method is ‘GET’ then create a python dictionary with the two elements message.
- Jsonify the python dictionary and return it.
- Build the flask application using the following command.

```
if __name__=='__main__':
```

```
    app.run(debug=True)
```

- Run the ‘main.py’ file in the terminal or the IDE.

Code:

- Python3

```
from flask import Flask,jsonify,request
```

```
app = Flask(__name__)
```

```
@app.route('/returnjson', methods = ['GET'])
```

```
def ReturnJSON():
```

```
    if(request.method == 'GET'):
```

```
        data = {
```

```
            "Modules" : 15,
```

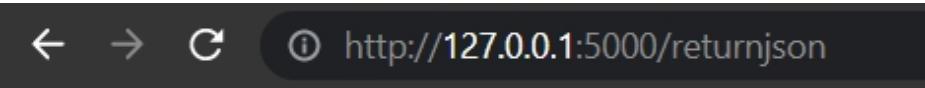
```
            "Subject" : "Data Structures and Algorithms",
```

```
}

return jsonify(data)

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

Output:



```
{
  "Modules": 15,
  "Subject": "Data Structures and Algorithms"
}
```

Approach 2: Using the flask_restful library with Flask - In this approach, we are going to create a simple JSON response with the help of the flask-restful library. The steps are discussed below:

- Create a new python file named 'main.py'.
- Import Flask from the flask framework.
- Import API and Resource from the 'flask_restful' library.
- Register the web app into an app variable using the following syntax.

```
app = Flask(__name__)

api = Api(app)
```

- Create a resource class named ‘ReturnJSON’.
- Inside the resource, the class creates a ‘get’ method.
- Return a dictionary with the simple JSON response from the ‘get’ method.
- Add the resource class to the API using the add_resource method.
- Build the flask application using the following command.

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

- Run the ‘main.py’ file in the terminal or the IDE.

Code:

- Python3

```
from flask import Flask
from flask_restful import Api, Resource

app = Flask(__name__)

api = Api(app)

class returnjson(Resource):
    def get(self):
        data={
            "Modules": 15,
            "Subject": "Data Structures and Algorithms"
        }
        return data

api.add_resource(returnjson,'/returnjson')
```

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

Output:



A screenshot of a terminal window. At the top, there are navigation icons: a left arrow, a right arrow, a refresh symbol, and a link icon followed by the URL "http://127.0.0.1:5000/returnjson". The main area of the terminal displays the following JSON output:

```
{
  "Modules": 15,
  "Subject": "Data Structures and Algorithms"
}
```

PART 5: Define and Access the Database in Flask

CHAPTER 1: Connect Flask to a Database with Flask-SQLAlchemy

Flask is a micro web framework written in python. Micro-framework is normally a framework with little to no dependencies on external libraries. Though being a micro framework almost everything can be implemented using python libraries and other dependencies when and as required.

In this article, we will be building a Flask application that **takes data** in a form from the user and then **displays** it on another page on the website. We can also **delete** the data. We won't focus on the front-end part rather we will be just coding the backend for the web application.

Installing Flask

In any directory where you feel comfortable create a folder and open the command line in the directory. Create a python virtual environment using the command below.

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

Once the command is done running activate the virtual environment using the command below.

```
<name>\scripts\activate
```

Now, install Flask using pip(package installer for python). Simply run the command below.

```
pip install Flask
```

Creating app.py

Once the installation is done create a file name app.py and open it in your favorite editor. To check whether Flask has been properly installed you can run the following code.

- Python

```
from flask import Flask

app = Flask(__name__)

"""If everything works fine you will get a
message that Flask is working on the first
page of the application

"""

@app.route('/')
def check():

    return 'Flask is working'

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

Output:

```
Flask is working
```

Setting Up SQLAlchemy

Now, let's move on to creating a **database for our application**. For the purpose of this article, we will be using SQLAlchemy a database toolkit, and an ORM(Object Relational Mapper). We will be using pip again to install SQLAlchemy. The command is as follows,

```
pip install flask-sqlalchemy
```

In your app.py file import SQLAlchemy as shown in the below code. We also need to add a configuration setting to our application so that we can use SQLite database in our application. We also need to create an SQLAlchemy database instance which is as simple as creating an object.

- Python

```
from flask import Flask

from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.debug = True

# adding configuration for using a sqlite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'

# Creating an SQLAlchemy instance
db = SQLAlchemy(app)

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

Creating Models

In sqlalchemy we use classes to create our database structure. In our application, we will create a Profile table that will be responsible for holding the user's id, first name, last name, and age.

- Python

```
from flask import Flask, request, redirect

from flask.templating import render_template

from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.debug = True

# adding configuration for using a sqlite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'

# Creating an SQLAlchemy instance
db = SQLAlchemy(app)

# Models

class Profile(db.Model):

    # Id : Field which stores unique id for every row in
    # database table.

    # first_name: Used to store the first name if the user

    # last_name: Used to store last name of the user

    # Age: Used to store the age of the user

    id = db.Column(db.Integer, primary_key=True)
```

```

first_name = db.Column(db.String(20), unique=False, nullable=False)

last_name = db.Column(db.String(20), unique=False, nullable=False)

age = db.Column(db.Integer, nullable=False)

# repr method represents how one object of this datatable

# will look like

def __repr__(self):

    return f"Name : {self.first_name}, Age: {self.age}"

if __name__ == '__main__':
    app.run()

```

The table below explains some of the keywords used in the model class.

Column	used to create a new column in the database table
Integer	An integer data field
primary_key	If set to True for a field ensures that the field can be used to uniquely identify objects of the data table.
String	An string data field. String(<maximum length>)
unique	If set to True it ensures that every data in that field is unique.
nullable	If set to False it ensures that the data in the field cannot be null.
__repr__	Function used to represent objects of the data table.

Creating the database

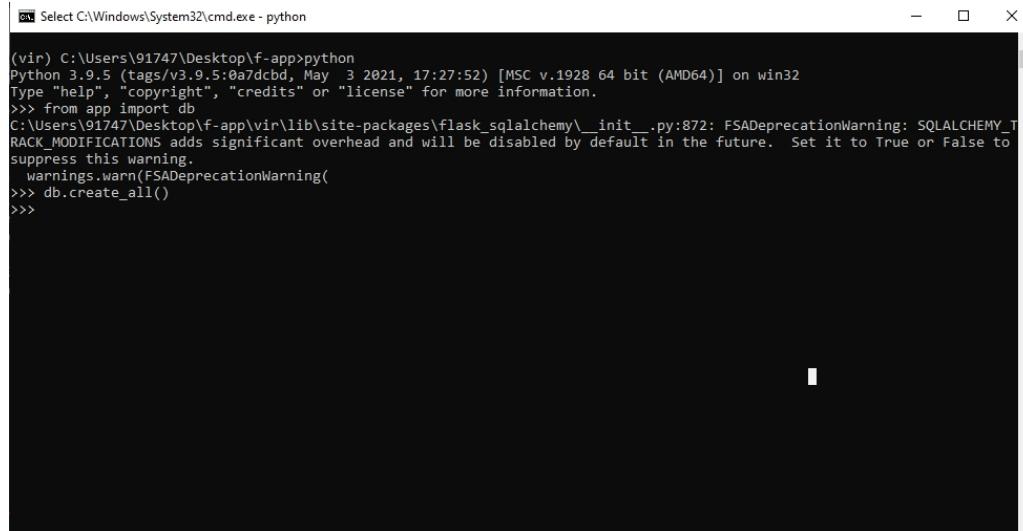
In the command line which is navigated to the project directory and virtual environment running, we need to run the following commands.

```
python
```

The above command will initiate a python bash in your command line where you can use further lines of code to create your data table according to your model class in your database.

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

After the commands, the response would look like something in the picture and in your project directory you will notice a new file named '**site.db**'.

A screenshot of a Windows Command Prompt window titled "Select C:\Windows\System32\cmd.exe - python". The window shows the following Python session:

```
(vir) C:\Users\91747\Desktop\f-app>python  
Python 3.9.5 (tags/v3.9.5:0a7dcbf, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> from app import db  
C:\Users\91747\Desktop\f-app\vir\lib\site-packages\flask_sqlalchemy\_init_.py:872: FSADeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future. Set it to True or False to suppress this warning.  
    warnings.warn(FSADeprecationWarning(  
>>> db.create_all()  
>>>
```

Making Migrations in database

Install Flask-Migrate using pip

```
pip install Flask-Migrate
```

Now, in your app.py add two lines, the code being as follows,

- Python

```
# Import for Migrations

from flask_migrate import Migrate, migrate

# Settings for migrations

migrate = Migrate(app, db)
```

Now to create migrations we run the following commands one after the other.

`flask db init`

```
(venv) nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/PycharmProjects/gfg$ flask db init
Set it to True or False to suppress this warning.
warnings.warn(FSDeprecationWarning)
Creating directory /home/nikhil/PycharmProjects/gfg/migrations ... done
Creating directory /home/nikhil/PycharmProjects/gfg/migrations/versions ... done
Generating /home/nikhil/PycharmProjects/gfg/migrations/alembic.ini ... done
Generating /home/nikhil/PycharmProjects/gfg/migrations/env.py ... done
Generating /home/nikhil/PycharmProjects/gfg/migrations/script.py.mako ... done
Generating /home/nikhil/PycharmProjects/gfg/migrations/README ... done
Please edit configuration/connection/logging settings in '/home/nikhil/PycharmProjects/gfg/migrations/alembic.ini' before proceeding.
```

`flask db init`

`flask db migrate -m "Initial migration"`

```
(venv) nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/PycharmProjects/gfg$ flask db migrate -m "Initial migration"
/home/nikhil/PycharmProjects/gfg/venv/lib/python3.8/site-packages/flask_sqlalchemy/_init_.py:872: FSDeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future
Set it to True or False to suppress this warning.
warnings.warn(FSDeprecationWarning)
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.autogenerate.compare] Detected added table 'profile'
Generating /home/nikhil/PycharmProjects/gfg/migrations/versions/e3d5ebd20914_initial_migration.py ... done
```

`flask db migrate -m "Initial migration"`

`flask db upgrade`

```
(venv) nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/PycharmProjects/gfg$ flask db upgrade
/home/nikhil/PycharmProjects/gfg/venv/lib/python3.8/site-packages/flask_sqlalchemy/_init_.py:872: FSDeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled by default in the future
Set it to True or False to suppress this warning.
warnings.warn(FSDeprecationWarning)
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.runtime.migration] Running upgrade -> e3d5ebd20914, Initial migration
(venv) nikhil@nikhil-Lenovo-ideapad-330-15IKB:~/PycharmProjects/gfg$
```

`flask db upgrade`

Now we have successfully created the data table in our database.

Creating the Index Page Of the Application

Before moving forward and building our form let's create an index page for our website. The HTML file is always stored inside a folder in the parent directory of the application named '**templates**'. Inside the templates folder create a file named index.html and paste the below code

for now. We will go back to adding more code into our index file as we move on.

- HTML

```
<html>
  <head>
    <title>Index Page</title>
  </head>
  <body>
    <h3>Profiles</h3>
  </body>
</html>
```

In the app.py add a small function that will render an HTML page at a specific route specified in app.route.

- Python

```
from flask import Flask, request, redirect
from flask.templating import render_template
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.debug = True

# adding configuration for using a sqlite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
```

```

# Creating an SQLAlchemy instance

db = SQLAlchemy(app)

# Models

class Profile(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    first_name = db.Column(db.String(20), unique=False, nullable=False)

    last_name = db.Column(db.String(20), unique=False, nullable=False)

    age = db.Column(db.Integer, nullable=False)

    def __repr__(self):

        return f"Name : {self.first_name}, Age: {self.age}"

# function to render index page

@app.route('/')

def index():

    return render_template('index.html')

if __name__ == '__main__':

    app.run()

```

To test whether everything is working fine you can run your application using the command

`python app.py`

The command will set up a local server at `http://localhost:5000`.

Output:

Profiles

[ADD](#)

Id First Name Last Name Age #

Creating HTML page for form

We will be creating an HTML page in which our form will be rendered. Create an HTML file named add_profile in your templates folder. The HTML code is as follows. The important points in the code will be **highlighted as you read on**.

- HTML

```
<!DOCTYPE html>

<html>
  <head>
    <title>Add Profile</title>
  </head>
  <body>
    <h3>Profile form</h3>
    <form action="/add" method="POST">
      <label>First Name</label>
      <input type="text" name="first_name" placeholder="first name...">
      <label>Last Name</label>
```

```
<input type="text" name= "last_name" placeholder="last name...">  
<label>Age</label>  
  
<input type="number" name="age" placeholder="age..">  
  
<button type="submit">Add</button>  
  
</form>  
  
</body>  
  
</html>
```

Adding a function in our application to render the form page

In our app.py file, we will add the following function. At route or site path 'http://localhost:5000/add_data' the page will be rendered.

- Python

```
@app.route('/add_data')  
  
def add_data():  
  
    return render_template('add_profile.html')
```

To check whether the code is working fine or not, you can run the following command to start the local server.

```
python app.py
```

Now, visit http://localhost:5000/add_data and you will be able to see the form.

Output:

Profile form

First Name Last Name Age

Function to add data using the form to the database

To add data to the database we will be using the “**POST**” method. POST is used to send data to a server to create/update a resource. In flask where we specify our route that is app.route we can also specify the HTTP methods there. Then inside the function, we create variables to store data and use **request objects** to procure data from the form.

Note: The name used in the input tags in the HTML file has to be the same one that is being used in this function,
For example,

```
<input type="number" name="age" placeholder="age..">  
"age" should also be used in the python function as,  
age = request.form.get("age")
```

Then we move on to create an object of the Profile class and store it in our database using database sessions.

- Python

```
# function to add profiles  
@app.route('/add', methods=["POST"])  
  
def profile():  
  
    # In this function we will input data from the  
    # form page and store it in our database.  
    # Remember that inside the get the name should
```

```

# exactly be the same as that in the html

# input fields

first_name = request.form.get("first_name")

last_name = request.form.get("last_name")

age = request.form.get("age")

# create an object of the Profile class of models

# and store data as a row in our datatable

if first_name != "" and last_name != "" and age is not None:

    p = Profile(first_name=first_name, last_name=last_name, age=age)

    db.session.add(p)

    db.session.commit()

    return redirect('/')

else:

    return redirect('/')

```

Once the function is executed it redirects us back to the index page of the application.

Display data on Index Page

On our index page now, we will be displaying all the data that has been stored in our data table. We will be using '**Profile.query.all()**' to query all the objects of the Profile class and then use **Jinja templating language** to display it dynamically on our index HTML file.
 Update your index file as follows. The delete function will be written later on in this article. For now, we will query all the data from the data table and display it on our home page.

- HTML

```

<!DOCTYPE html>

<html>
  <head>
    <title>Index Page</title>
  </head>
  <body>
    <h3>Profiles</h3>

    <a href="/add_data">ADD</a>
    <br>
    <table>
      <thead>
        <th>Id</th>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Age</th>
        <th>#</th>
      </thead>
      {% for data in profiles %}
        <tbody>
          <td>{{data.id}}</td>
          <td>{{data.first_name}}</td>
          <td>{{data.last_name}}</td>
          <td>{{data.age}}</td>
          <td><a href="/delete/{{data.id}}" type="button">Delete</a></td>
        </tbody>
      {% endfor%}
    </table>

```

```
</body>  
</html>
```

We loop through every object in profiles that we pass down to our template in our index function and print all its data in a tabular form. The index function in our app.py is updated as follows.

- Python

```
@app.route('/')  
  
def index():  
  
    # Query all data and then pass it to the template  
  
    profiles = Profile.query.all()  
  
    return render_template('index.html', profiles=profiles)
```

Deleting data from our database

To delete data we have already used an anchor tag in our table and now we will just be associating a function with it.

- Python

```
@app.route('/delete/<int:id>')  
  
def erase(id):  
  
    # Deletes the data on the basis of unique id and  
    # redirects to home page  
  
    data = Profile.query.get(id)  
  
    db.session.delete(data)
```

```
    db.session.commit()  
  
    return redirect('/')
```

The function queries data on the basis of id and then deletes it from our database.

Complete Code

The entire code for app.py, index.html, and add-profile.html has been given.

app.py

- Python

```
from flask import Flask, request, redirect  
  
from flask.templating import render_template  
  
from flask_sqlalchemy import SQLAlchemy  
  
from flask_migrate import Migrate, migrate  
  
  
app = Flask(__name__)  
  
app.debug = True  
  
  
# adding configuration for using a sqlite database  
  
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'  
  
  
# Creating an SQLAlchemy instance  
  
db = SQLAlchemy(app)  
  
  
# Settings for migrations  
  
migrate = Migrate(app, db)
```

```
# Models

class Profile(db.Model):

    # Id : Field which stores unique id for every row in
    # database table.

    # first_name: Used to store the first name if the user

    # last_name: Used to store last name of the user

    # Age: Used to store the age of the user

    id = db.Column(db.Integer, primary_key=True)

    first_name = db.Column(db.String(20), unique=False, nullable=False)

    last_name = db.Column(db.String(20), unique=False, nullable=False)

    age = db.Column(db.Integer, nullable=False)

    # repr method represents how one object of this datatable
    # will look like

    def __repr__(self):

        return f"Name : {self.first_name}, Age: {self.age}"


# function to render index page

@app.route('/')

def index():

    profiles = Profile.query.all()

    return render_template('index.html', profiles=profiles)


@app.route('/add_data')

def add_data():

    return render_template('add_profile.html')
```

```
# function to add profiles

@app.route('/add', methods=["POST"])

def profile():

    # In this function we will input data from the

    # form page and store it in our database. Remember

    # that inside the get the name should exactly be the same

    # as that in the html input fields

    first_name = request.form.get("first_name")

    last_name = request.form.get("last_name")

    age = request.form.get("age")

    # create an object of the Profile class of models and

    # store data as a row in our datatable

    if first_name != "" and last_name != "" and age is not None:

        p = Profile(first_name=first_name, last_name=last_name, age=age)

        db.session.add(p)

        db.session.commit()

        return redirect('/')

    else:

        return redirect('/')

@app.route('/delete/<int:id>')

def erase(id):

    # deletes the data on the basis of unique id and

    # directs to home page
```

```
data = Profile.query.get(id)

db.session.delete(data)

db.session.commit()

return redirect('/')

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

index.html

- HTML

```
<!DOCTYPE html>

<html>

    <head>
        <title>Index Page</title>
    </head>

    <body>
        <h3>Profiles</h3>

        <a href="/add_data">ADD</a>

        <br>

        <table>
            <thead>
                <th>Id</th>
                <th>First Name</th>
                <th>Last Name</th>
                <th>Age</th>
```

```

<th>#</th>
</thead>

{% for data in profiles %}

<tbody>

<td>{{data.id}}</td>
<td>{{data.first_name}}</td>
<td>{{data.last_name}}</td>
<td>{{data.age}}</td>

<td><a href="/delete/{{data.id}}" type="button">Delete</a></td>

</tbody>

{% endfor%}

</table>

</body>

</html>

```

add_profile.html

- HTML

```

<!DOCTYPE html>

<html>

<head>
    <title>Add Profile</title>
</head>

<body>
    <h3>Profile form</h3>

    <form action="/add" method="POST">
        <label>First Name</label>

```

```
<input type="text" name="first_name" placeholder="first name...">  
<label>Last Name</label>  
  
<input type="text" name= "last_name" placeholder="last name...">  
  
<label>Age</label>  
  
<input type="number" name="age" placeholder="age..">  
  
<button type="submit">Add</button>  
  
</form>  
  
</body>  
  
</html>
```

Output:

Profiles

[ADD](#)

Id First Name Last Name Age #

CHAPTER 2: Build a Web App using Flask and SQLite in Python

Python-based Flask is a microweb framework. Typically, a micro-framework has little to no dependencies on outside frameworks. Despite being a micro framework, practically everything may be developed when and as needed utilizing Python libraries and other dependencies. In this post, we'll develop a Flask application that collects user input in a form and shows it on an additional web page using SQLite in Python.

Package Required

Install flask to proceed with the Front End of the Web App.

```
pip install flask
```

```
pip install db-sqlite3
```

Steps to Build an App Using Flask and SQLite

Step 1: Create Virtual Environment

Step 2: Install the required modules inside Virtual Environment.

Step 3: Build a Front End of the Web App.

- **index.html**

The index.html file will contain two buttons, one button to check all the participant's lists (taken from the database). And the other button to create a new entry.

- **HTML**

```
<!DOCTYPE html>  
  
<html>  
  
  <head>  
    <title>Flask and SQLite </title>
```

```

</head>

<body>

    <h1>Build Web App Using Flask and SQLite</h1>

    <button class="btn" type="button" onclick="window.location.href='{{ url_for('join') }}';">Fill form to get updates</button><br/>

    <button class="btn" type="button" onclick="window.location.href='{{ url_for('participants') }}';">Check participant list</button>

</body>

</html>

```

- **join.html**

In the join.html, create a simple form that takes Name, Email, City, Country and Phone as the input to store in the database. By the POST method, receive the form request of all the columns and commit the changes in the database after inserting the details in the table.

- **HTML**

```

<!DOCTYPE html>

<html>

    <head>

        <title>Flask and SQLite </title>

    </head>

    <body>

        <form method="POST">

            <label>Enter Name:</label>

            <input type="name" name="name" placeholder="Enter your name" required>
<br/>

            <label>Enter Email:</label>

```

```

        <input type="email" name="email" placeholder="Enter your email" required>
<br/>

        <label>Enter City:</label>

        <input type="name" name="city" placeholder="Enter your City name" required>
<br/>

        <label>Enter Country:</label>

        <input type="name" name="country" placeholder="Enter the Country name"
required><br/>

        <label>Enter phone num:</label>

        <input type="name" name="phone" placeholder="Your Phone Number"
required><br/>

        <input type = "submit" value = "submit"/><br/>

</form>

</body>

</html>

```

participants.html

Use table tag and assign the heading using `<th>` tag. To auto increment, the table row on the new entry, use a For loop jinja template. Inside For loop add `<tr>` and `<td>` tags.

- HTML

```

<!DOCTYPE html>

<html>

    <head>
        <title>Flask and SQLite </title>
    </head>
    <style>

```

```

table, th, td {
    border:1px solid black;
}

</style>

<body>

    <table style="width:100%">

        <tr>

            <th>Name</th>
            <th>Email</th>
            <th>City</th>
            <th>Country</th>
            <th>Phone Number</th>

        </tr>

        {%for participant in data%}

            <tr>

                <td>{{participant[0]}}</td>
                <td>{{participant[1]}}</td>
                <td>{{participant[2]}}</td>
                <td>{{participant[3]}}</td>
                <td>{{participant[4]}}</td>

            </tr>

        {%endfor%}

    </table>

</body>

</html>

```

Step 4: Create app.py

Create a new file named app.py and build a Front End of the Web App by rendering HTML templates. From here we shall go function by function explanation as in points:

- To insert the data into the database, we first need to create a new database table. The column to be inserted in the database is Name, Email, City, Country, and Phone Number.
 - The basic syntax to start with sqlite3 is to first connect to the database. `sqlite3.connect("database.db")` will create a new database. The next step is to create a new table, but it will first check if the table already exists or not.
 - One button in the index.html prompts to the participant's list, and thus using the existing database select * from the table and display it using a Python template i.e., Jinja template to run through the loop within HTML. In the following code, we have created a table tag, inside the table tag for every new insertion in the database, we add a Loop Jinja Template to auto increment the new table row.
 - In the participants function, we use select all columns from the table name, we use `fetchall()` method you retrieve the data.

- Python3

```
from flask import Flask, render_template, request  
  
import sqlite3  
  
app = Flask(__name__)  
  
@app.route('/')  
  
@app.route('/home')  
  
def index():  
  
    return render_template('index.html')
```

```
connect = sqlite3.connect('database.db')
connect.execute(
    'CREATE TABLE IF NOT EXISTS PARTICIPANTS (name TEXT, \
email TEXT, city TEXT, country TEXT, phone TEXT)')

@app.route('/join', methods=['GET', 'POST'])

def join():

    if request.method == 'POST':

        name = request.form['name']

        email = request.form['email']

        city = request.form['city']

        country = request.form['country']

        phone = request.form['phone']

    with sqlite3.connect("database.db") as users:

        cursor = users.cursor()

        cursor.execute("INSERT INTO PARTICIPANTS \
(name,email,city,country,phone) VALUES (?,?,?,?,?)",
                    (name, email, city, country, phone))

        users.commit()

    return render_template("index.html")

else:

    return render_template('join.html')

@app.route('/participants')
```

```

def participants():

    connect = sqlite3.connect('database.db')

    cursor = connect.cursor()

    cursor.execute('SELECT * FROM PARTICIPANTS')


    data = cursor.fetchall()

    return render_template("participants.html", data=data)

if __name__ == '__main__':
    app.run(debug=False)

```

Output:

For route: <http://127.0.0.1:5000/>

Build Web App Using Flask and SQLite

[Fill form to get updates](#)
[Check participant list](#)

For route: <http://127.0.0.1:5000/join>

Here we are adding two new data to the database.

Enter Name:

Enter Email:

Enter City:

Enter Country:

Enter phone num:

data 1

Enter Name:

Enter Email:

Enter City:

Enter Country:

Enter phone num:

data 2

For route: <http://127.0.0.1:5000/participants>

Name	Email	City	Country	Phone Number
Tarun R Jain	tarun@gmail.com	Bengaluru	India	1111111111
Rahul	rahul@gmail.com	Bengaluru	India	0000000000

CHAPTER 3: Sending Data from a Flask app to MongoDB Database

This article covers how we can configure a MongoDB database with a Flask app and store some data in the database after configuring it. Before directly moving to the configuration phase here is a short overview of all tools and software we will use.

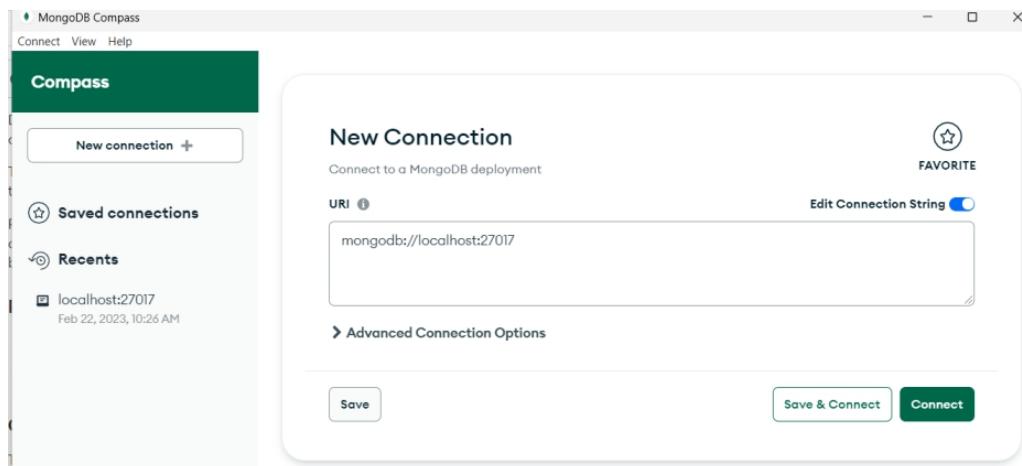
MongoDB is an open-source database that stores data in JSON-like documents. It is classified as a NoSQL database because it is based on different fundamentals as compared to a SQL relational database.

Prerequisites

- A decent understanding of Python and a machine with Python installed.
- Understanding of basic concepts of Flask.
- MongoDB is installed on your local machine if not you can refer to this article.

Configuring MongoDB

Till this step, you have a local machine with MongoDB installed on it now run the MongoDB compass and the following screen will appear. You can edit the settings or just click connect and MongoDB will be running on your local machine.



Setup a Development Environment

Let's configure the virtual environment for development, this step can be skipped but it is always recommended to use a dedicated development environment for each project to avoid dependency clash, this can be achieved using a Python virtual environment.

```
# Create gfg folder
```

```
$ mkdir gfg
```

```
# Move to gfg folder
```

```
$ cd gfg
```

```
PS C:\Users\hariv> mkdir gfg

    Directory: C:\Users\hariv

Mode           LastWriteTime         Length Name
----           -----          ----- 
d--- 12-03-2023 11:54 PM            0   gfg

PS C:\Users\hariv> cd .\gfg\
PS C:\Users\hariv\gfg>
```

We created a folder named `gfg` for the project, you can name it anything you want and cd (change directory) to go into your newly created directory then run the following command that will create a virtual environment for your project.

```
$ python -m venv venv
```

Now to use the virtual environment we need to first activate it, this can be done by executing the activated binary file.

```
$ .\venv\Scripts\activate # for Windows OS
```

```
$ source venv/bin/activate # for Linux OS
```

```
PS C:\Users\hariv\gfg> python -m venv venv
PS C:\Users\hariv\gfg>
PS C:\Users\hariv\gfg> .\venv\Scripts\activate
(venv) PS C:\Users\hariv\gfg> |
```

Installing Dependencies for the Project

We are done configuring a development environment now let us install the tools that we will be using including Flask, `pymongo` which provide the interface for Python-based apps to the MongoDB database.

```
$ pip install Flask pymongo
```

```
(venv) PS C:\Users\hariv\gfg> pip install Flask pymongo
Collecting Flask
  Using cached Flask-2.2.2-py3-none-anywheel-101.tgz
```

Next, we'll connect a FLask app to MongoDB and send some data into it.

Creating a Flask App

We are done with the database setup and installing the required libraries now we will create a Flask app and connect it to the MongoDB we created and insert some user data into it. First, let's create a Flask app as follows. Create a `main.py` file in the project directory and add the following code to the file.

- Python3

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():

    return 'Hello, World!'

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

Over here we have created a starter Flask App with a single route that returns a string “Hello, World!”, Now test this thing out by running the app as shown below:

Output:

```
(venv) PS C:\Users\hariv\gfg> python main.py
* Serving Flask app 'main'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

The app is up and running now let us test the output at `http://127.0.0.1:5000` ,



Hello, World!

Connecting Flask App to Database

We have got a running starter Flask App with some basic code let's connect it to the MongoDB database using the following script.

- Python3

```
from flask import Flask, request
```

```
from pymongo import MongoClient
```

```
# Flask app object
```

```
app = Flask(__name__)
```

```
# Set up MongoDB connection
```

```
client = MongoClient('mongodb://localhost:27017/')

db = client['demo']

collection = db['data']
```

The above script will connect to the MongoDB database that we configured earlier now we need a post route to add some data in the database which can be done as follow:

- Python3

```
@app.route('/add_data', methods=['POST'])

def add_data():

    # Get data from request

    data = request.json

    # Insert data into MongoDB

    collection.insert_one(data)

    return 'Data added to MongoDB'
```

Here we created a route named `/add_data` which when invoked with a post request reads the JSON data from the body and inserts it into the database.

For reference here is the overall code that I used for the demo.

- Python3

```
from flask import Flask, request

from pymongo import MongoClient
```

```
app = Flask(__name__)

# root route
@app.route('/')

def hello_world():

    return 'Hello, World!'

# Set up MongoDB connection and collection
client = MongoClient('mongodb://localhost:27017/')

# Create database named demo if they don't exist already
db = client['demo']

# Create collection named data if it doesn't exist already
collection = db['data']

# Add data to MongoDB route
@app.route('/add_data', methods=['POST'])

def add_data():

    # Get data from request
    data = request.json

    # Insert data into MongoDB
    collection.insert_one(data)

    return 'Data added to MongoDB'
```

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

Sending Data from Flask to MongoDB

Now let us test the entire script and if we can insert some data into the database using it, First run the Flask App shown before then make a POST request to `/add_data` a route using a tool like Postman.

The screenshot shows a Postman interface with a POST request to `http://127.0.0.1:5000/add_data`. The Body tab is selected, showing a JSON payload:

```
1 { "data": "gfg" }
```

The response body shows the message `Data added to MongoDB`.

The response above looks fine let us check the MongoDB database if there is any data inserted or not.

The screenshot shows the MongoDB Compass interface. A database named `demo` is selected, and a collection named `data` is shown. A document is selected, displaying its contents:

```
_id: ObjectId('640e1d373e3221b39807ae4e')
data: "gfg"
```

As you can see a database named `demo` is created with a collection of `'data'` with a single document that we just inserted using Flask.

CHAPTER 4: Build a Web App using Flask and SQLite in Python

Python-based Flask is a microweb framework. Typically, a micro-framework has little to no dependencies on outside frameworks. Despite being a micro framework, practically everything may be developed when and as needed utilizing Python libraries and other dependencies. In this post, we'll develop a Flask application that collects user input in a form and shows it on an additional web page using SQLite in Python.

Package Required

Install flask to proceed with the Front End of the Web App.

```
pip install flask
```

```
pip install db-sqlite3
```

Steps to Build an App Using Flask and SQLite

Step 1: Create Virtual Environment

Step 2: Install the required modules inside Virtual Environment.

Step 3: Build a Front End of the Web App.

index.html

The index.html file will contain two buttons, one button to check all the participant's lists (taken from the database). And the other button to create a new entry.

- HTML

```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
<title>Flask and SQLite </title>
```

```

</head>

<body>

    <h1>Build Web App Using Flask and SQLite</h1>

    <button class="btn" type="button" onclick="window.location.href='{{ url_for('join') }}';">Fill form to get updates</button><br/>

    <button class="btn" type="button" onclick="window.location.href='{{ url_for('participants') }}';">Check participant list</button>

</body>

</html>

```

join.html

In the join.html, create a simple form that takes Name, Email, City, Country and Phone as the input to store in the database. By the POST method, receive the form request of all the columns and commit the changes in the database after inserting the details in the table.

- HTML

```

<!DOCTYPE html>

<html>

    <head>
        <title>Flask and SQLite </title>
    </head>

    <body>

        <form method="POST">

            <label>Enter Name:</label>

            <input type="name" name="name" placeholder="Enter your name" required>
<br/>

            <label>Enter Email:</label>

```

```

        <input type="email" name="email" placeholder="Enter your email" required>
<br/>

        <label>Enter City:</label>

        <input type="name" name="city" placeholder="Enter your City name" required>
<br/>

        <label>Enter Country:</label>

        <input type="name" name="country" placeholder="Enter the Country name"
required><br/>

        <label>Enter phone num:</label>

        <input type="name" name="phone" placeholder="Your Phone Number"
required><br/>

        <input type = "submit" value = "submit"/><br/>

    </form>

</body>

</html>

```

participants.html

Use table tag and assign the heading using `<th>` tag. To auto increment, the table row on the new entry, use a For loop jinja template. Inside For loop add `<tr>` and `<td>` tags.

- HTML

```

<!DOCTYPE html>

<html>

    <head>

        <title>Flask and SQLite </title>

    </head>

    <style>

        table, th, td {

```

```
border:1px solid black;  
}  
  
</style>  
  
<body>  
  
    <table style="width:100%">  
  
        <tr>  
            <th>Name</th>  
            <th>Email</th>  
            <th>City</th>  
            <th>Country</th>  
            <th>Phone Number</th>  
        </tr>  
  
        {%for participant in data%}  
            <tr>  
                <td>{{participant[0]}}</td>  
                <td>{{participant[1]}}</td>  
                <td>{{participant[2]}}</td>  
                <td>{{participant[3]}}</td>  
                <td>{{participant[4]}}</td>  
            </tr>  
        {%endfor%}  
    </table>  
    </body>  
</html>
```

Step 4: Create app.py

Create a new file named app.py and build a Front End of the Web App by rendering HTML templates. From here we shall go function by function explanation as in points:

- To insert the data into the database, we first need to create a new database table. The column to be inserted in the database is Name, Email, City, Country, and Phone Number.
 - The basic syntax to start with sqlite3 is to first connect to the database. `sqlite3.connect("database.db")` will create a new database. The next step is to create a new table, but it will first check if the table already exists or not.
 - One button in the index.html prompts to the participant's list, and thus using the existing database select * from the table and display it using a Python template i.e., Jinja template to run through the loop within HTML. In the following code, we have created a table tag, inside the table tag for every new insertion in the database, we add a Loop Jinja Template to auto increment the new table row.
 - In the participants function, we use select all columns from the table name, we use `fetchall()` method you retrieve the data.

- Python3

```
from flask import Flask, render_template, request  
  
import sqlite3  
  
app = Flask(__name__)  
  
@app.route('/')  
  
@app.route('/home')  
  
def index():  
  
    return render_template('index.html')
```

```
connect = sqlite3.connect('database.db')
connect.execute(
    'CREATE TABLE IF NOT EXISTS PARTICIPANTS (name TEXT, \
email TEXT, city TEXT, country TEXT, phone TEXT)')

@app.route('/join', methods=['GET', 'POST'])

def join():

    if request.method == 'POST':

        name = request.form['name']

        email = request.form['email']

        city = request.form['city']

        country = request.form['country']

        phone = request.form['phone']

    with sqlite3.connect("database.db") as users:

        cursor = users.cursor()

        cursor.execute("INSERT INTO PARTICIPANTS \
(name,email,city,country,phone) VALUES (?,?,?,?,?)",
                    (name, email, city, country, phone))

        users.commit()

    return render_template("index.html")

else:

    return render_template('join.html')

@app.route('/participants')
```

```
def participants():

    connect = sqlite3.connect('database.db')

    cursor = connect.cursor()

    cursor.execute('SELECT * FROM PARTICIPANTS')


    data = cursor.fetchall()

    return render_template("participants.html", data=data)

if __name__ == '__main__':
    app.run(debug=False)
```

Output:

For route: <http://127.0.0.1:5000/>

Build Web App Using Flask and SQLite

[Fill form to get updates](#)
[Check participant list](#)

For route: <http://127.0.0.1:5000/join>

Here we are adding two new data to the database.

Enter Name:

Enter Email:

Enter City:

Enter Country:

Enter phone num:

data 1

Enter Name:

Enter Email:

Enter City:

Enter Country:

Enter phone num:

data 2

For route: <http://127.0.0.1:5000/participants>

Name	Email	City	Country	Phone Number
Tarun R Jain	tarun@gmail.com	Bengaluru	India	1111111111
Rahul	rahul@gmail.com	Bengaluru	India	0000000000

CHAPTER 5: Login and Registration Project Using Flask and MySQL

Project Title: Login and registration Project using Flask framework and MySQL Workbench. **Type of Application (Category):** Web application. **Introduction:** A framework is a code library that makes a developer's life easier when building web applications by providing reusable code for common operations. There are a number of frameworks for Python, including Flask, Tornado, Pyramid, and Django. Flask is a lightweight web application framework. It is classified as a micro-framework because it does not require particular tools or libraries. **Pre-requisite:** Knowledge of Python, MySQL Workbench and basics of Flask Framework. Python and MySQL Workbench should be installed in the system. Visual studio code or Spyder or any code editor to work on the application. **Technologies used in the project:** Flask framework, MySQL Workbench. **Implementation of the Project:**

(1) Creating Environment

Step-1: Create an environment. Create a project folder and a venv folder within.

py -3 -m venv venv

Step-2: Activate the environment.

venv\Scripts\activate

Step-3: Install Flask.

pip install Flask

(2) MySQL Workbench

Step-1: Install MySQL workbench. Link to install

: <https://dev.mysql.com/downloads/workbench/> Know more about it

: <https://www.mysql.com/products/workbench/>

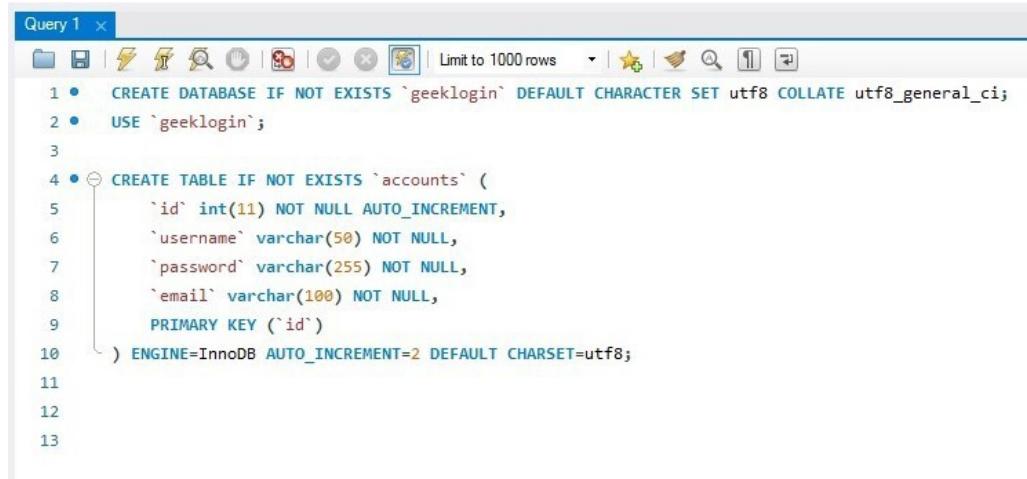
Step-2: Install 'mysqlbd' module in your venv.

pip install flask-mysqldb

Step-3: Open MySQL workbench.

Step-4: Write the following code. The above SQL statement will create our database **geeklogin** with the table **accounts**.

Step-5: Execute the query.



```
Query 1 ×
CREATE DATABASE IF NOT EXISTS `geeklogin` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
USE `geeklogin`;
CREATE TABLE IF NOT EXISTS `accounts` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `username` varchar(50) NOT NULL,
    `password` varchar(255) NOT NULL,
    `email` varchar(100) NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

(3) Creating Project

Step-1: Create an empty folder '**login**'.

Step-2: Now open your code editor and open this 'login' folder.

Step-3: Create '**app.py**' folder and write the code given below.

- Python3

```
# Store this code in 'app.py' file

from flask import Flask, render_template, request, redirect, url_for, session
from flask_mysqldb import MySQL
import MySQLdb.cursors
import re

app = Flask(__name__)
```

```
app.secret_key = 'your secret key'

app.config['MYSQL_HOST'] = 'localhost'
app.config['MYSQL_USER'] = 'root'
app.config['MYSQL_PASSWORD'] = 'your password'
app.config['MYSQL_DB'] = 'geeklogin'

mysql = MySQL(app)

@app.route('/')
@app.route('/login', methods=['GET', 'POST'])

def login():

    msg = ""

    if request.method == 'POST' and 'username' in request.form and 'password' in request.form:

        username = request.form['username']

        password = request.form['password']

        cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)

        cursor.execute('SELECT * FROM accounts WHERE username = % s AND password = % s',
        (username, password,))

        account = cursor.fetchone()

        if account:

            session['loggedin'] = True

            session['id'] = account['id']

            session['username'] = account['username']

            msg = 'Logged in successfully !'

    return render_template('index.html', msg = msg)
```

```

        else:

            msg = 'Incorrect username / password !'

        return render_template('login.html', msg = msg)

@app.route('/logout')

def logout():

    session.pop('loggedin', None)
    session.pop('id', None)
    session.pop('username', None)

    return redirect(url_for('login'))

@app.route('/register', methods =['GET', 'POST'])

def register():

    msg = ""

    if request.method == 'POST' and 'username' in request.form and 'password' in request.form and 'email' in request.form :

        username = request.form['username']

        password = request.form['password']

        email = request.form['email']

        cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)

        cursor.execute('SELECT * FROM accounts WHERE username = % s', (username,))

        account = cursor.fetchone()

        if account:

            msg = 'Account already exists !'

        elif not re.match(r'[^@]+@[^@]+\.[^@]+', email):

            msg = 'Invalid email address !'

        elif not re.match(r'[A-Za-z0-9]+', username):

```

```

msg = 'Username must contain only characters and numbers !'

elif not username or not password or not email:

    msg = 'Please fill out the form !'

else:

    cursor.execute('INSERT INTO accounts VALUES (NULL, % s, % s, % s)', (username,
password, email,))

    mysql.connection.commit()

    msg = 'You have successfully registered !'

elif request.method == 'POST':

    msg = 'Please fill out the form !'

return render_template('register.html', msg = msg)

```

- Step-4:** Create the folder '**templates**'. create the file 'login.html', 'register.html', 'index.html' inside the 'templates' folder.
- Step-5:** Open '**login.html**' file and write the code given below. In 'login.html', we have two fields i.e. username and password. When user enters correct username and password, it will route you to index page otherwise 'Incorrect username/password' is displayed.

- html

```

<!-- Store this code in 'login.html' file inside the 'templates' folder -->

<html>
    <head>
        <meta charset="UTF-8">
        <title> Login </title>
        <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    </head>
    <body><br><br><br><br><br>
        <div align="center">

```

```

<div align="center" class="border">
    <div class="header">
        <h1 class="word">Login</h1>
    </div><br><br><br>
    <h2 class="word">
        <form action="{{ url_for('login') }}" method="post">
            <div class="msg">{{ msg }}</div>
            <input id="username" name="username" type="text" placeholder="Enter Your Username" class="textbox"/><br><br>
            <input id="password" name="password" type="password" placeholder="Enter Your Password" class="textbox"/><br><br><br>
            <input type="submit" class="btn" value="Sign In"><br><br>
        </form>
    </h2>
    <p class="bottom">Don't have an account? <a class="bottom" href="{{ url_for('register') }}> Sign Up here</a></p>
</div>
</div>
</body>
</html>

```

Step-6: Open ‘**register.html**’ file and write the code given below. In ‘register.html’, we have three fields i.e. username, password and email. When user enters all the information, it stored the data in the database and ‘Registration successful’ is displayed.

- html

```
<!-- Store this code in 'register.html' file inside the 'templates' folder -->
```

```
<html>
```

```
<head>

    <meta charset="UTF-8">

    <title> Register </title>

    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">

</head>

<body><br><br><br><br><br>

<div align="center">

    <div align="center" class="border">

        <div class="header">

            <h1 class="word">Register</h1>

        </div><br><br><br>

        <h2 class="word">

            <form action="{{ url_for('register') }}" method="post">

                <div class="msg">{{ msg }}</div>

                <input id="username" name="username" type="text"
placeholder="Enter Your Username" class="textbox"/><br><br>

                <input id="password" name="password" type="password"
placeholder="Enter Your Password" class="textbox"/><br><br>

                <input id="email" name="email" type="text" placeholder="Enter Your
Email ID" class="textbox"/><br><br>

                <input type="submit" class="btn" value="Sign Up"><br>

            </form>

        </h2>

        <p class="bottom">Already have an account? <a class="bottom" href="{{ url_for('login') }}> Sign In here</a></p>

    </div>

</div>

</body>
```

```
</html>
```

Step-7: Open ‘index.html’ file and write the code given below. This page is displayed when login is successful and username is also displayed. The logout functionality is also included in this page. When user logs out, it moves to fresh login page again.

- html

```
<!-- Store this code in 'index.html' file inside the 'templates' folder-->
```

```
<html>
  <head>
    <meta charset="UTF-8">
    <title> Index </title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body><br><br><br><br><br>
    <div align="center">
      <div align="center" class="border">
        <div class="header">
          <h1 class="word">Index</h1>
        </div><br><br>
        <h1 class="bottom">
          Hi {{session.username}}!!<br><br> Welcome to the index
          page...
        </h1><br><br>
        <a href="{{ url_for('logout') }}" class="btn">Logout</a>
      </div>
    </div>
  </body>
```

```
</html>
```

Step-8: Create the folder '**static**'. create the file '**style.css**' inside the 'static' folder and paste the given CSS code.

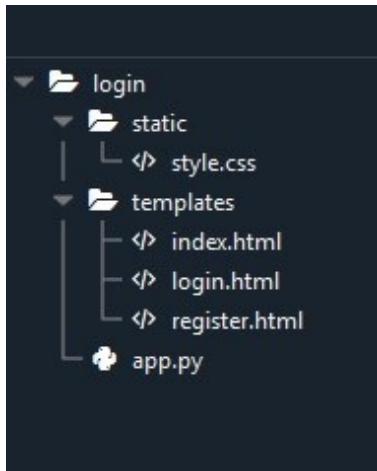
- **CSS**

```
/* Store this code in 'style.css' file inside the 'static' folder*/
```

```
.header{  
    padding: 5px 120px;  
    width: 150px;  
    height: 70px;  
    background-color: #236B8E;  
}  
  
.border{  
    padding: 80px 50px;  
    width: 400px;  
    height: 450px;  
    border: 1px solid #236B8E;  
    border-radius: 0px;  
    background-color: #9AC0CD;  
}  
  
.btn {  
    padding: 10px 40px;  
    background-color: #236B8E;  
    color: #FFFFFF;  
    font-style: oblique;
```

```
    font-weight: bold;  
    border-radius: 10px;  
}  
  
.textbox{  
    padding: 10px 40px;  
    background-color: #236B8E;  
    text-color: #FFFFFF;  
    border-radius: 10px;  
}  
  
::placeholder {  
    color: #FFFFFF;  
    opacity: 1;  
    font-style: oblique;  
    font-weight: bold;  
}  
  
.word{  
    color: #FFFFFF;  
    font-style: oblique;  
    font-weight: bold;  
}  
  
.bottom{  
    color: #236B8E;  
    font-style: oblique;  
    font-weight: bold;  
}
```

Step-9: The project structure will look like this.



(4) Run the Project

Step-1: Run the server.

Step-2: Browse the URL 'localhost:5000'.

Step-3: The output web page will be displayed.

(5) Testing of the Application

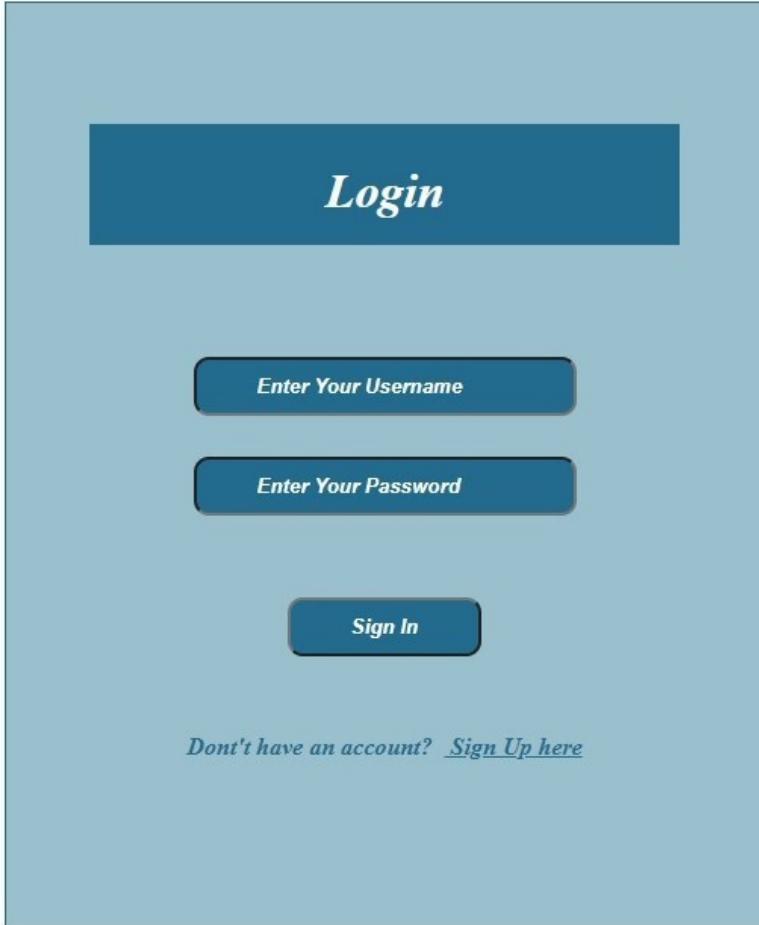
Step-1: If you are new user, go to sign up page and fill the details.

Step-2: After registration, go to login page. Enter your username and password and sign in.

Step-3: If your login is successful, you will be moved to index page and your name will be displayed.

Output:

Login page:



Registration page:

Register

Enter Your Username

Enter Your Password

Enter Your Email ID

Sign Up

Already have an account? [Sign In here](#)

If registration successful:

Register

You have successfully registered!

Enter Your Username

Enter Your Password

Enter Your Email ID

Sign Up

Already have an account? [Sign In here](#)

Before registration, Database table:

Query 1 accounts

1 • SELECT * FROM geeklogin.accounts;

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

	id	username	password	email
*	NULL	NULL	NULL	NULL

After registration, Database table:

Query 1 accounts

1 • SELECT * FROM geeklogin.accounts;

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

	id	username	password	email
▶	1	user	user@123	user@gmail.com
*	NULL	NULL	NULL	NULL

If login successful, Indexpage is displayed:

Index

Hi user!!

Welcome to the index page...

[*Logout*](#)

If Login fails:

Log in

Incorrect username/password!

Enter Your Username

Enter Your Password

Sign In

Dont't have an account? [Sign Up here](#)

CHAPTER 6: Execute raw SQL in Flask-SQLAlchemy app

In this article, we are going to see how to execute raw SQL in Flask-SQLAlchemy using Python.

Installing requirements

Install the Flask and Flask-SQLAlchemy libraries using pip

```
pip install Flask
```

```
pip install flask_sqlalchemy
```

Syntax

To run raw SQL queries, we first create a flask-SQLAlchemy engine object using which we can connect to the database and execute the SQL queries. The syntax is -

`flask_sqlalchemy.SQLAlchemy.engine.execute(statement)`

Executes a SQL expression construct or string statement within the current transaction.

Parameters:

- *statement: SQL expression*

Returns:

- *sqlalchemy.engine.result.ResultProxy*

Example 1

- Python

```
# IMPORT REQUIRED LIBRARIES
```

```
from flask import Flask, request

from flask_sqlalchemy import SQLAlchemy

# CREATE THE FLASK APP

app = Flask(__name__)

# ADD THE DATABASE CONNECTION TO THE FLASK APP

db = SQLAlchemy(app)

db_cred = {

    'user': 'root',      # DATABASE USER

    'pass': 'password',  # DATABASE PASSWORD

    'host': '127.0.0.1', # DATABASE HOSTNAME

    'name': 'Geeks4Geeks' # DATABASE NAME

}

app.config['SQLALCHEMY_DATABASE_URI'] = f"mysql+pymysql://{db_cred['user']}:{db_cred['pass']}@{db_cred['host']}\\n{db_cred['name']}"

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# CREATE A users TABLE USING RAW SQL QUERY

db.engine.execute(
    """
CREATE TABLE users (
    email VARCHAR(50),
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    passwd VARCHAR(50)
    """
)
```

```
);

...
)

# INSERT TEMP VALUES IN THE users TABLE USING RAW SQL QUERY

db.engine.execute(
    ...

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('john.doe@zmail.com', 'John', 'Doe', 'john@123');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('john.doe@zmail.com', 'John', 'Doe', 'johndoe@777');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('noah.emma@wmail.com', 'Emma', 'Noah', 'emaaa!00');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('emma@tmail.com', 'Emma', 'Noah', 'whrfc2bfh904');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('noah.emma@wmail.com', 'Emma', 'Noah', 'emaaa!00');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('liam.olivia@wmail.com', 'Liam', 'Olivia', 'lolivia#900');

    INSERT INTO users(email, first_name, last_name, passwd) VALUES
    ('liam.olivia@wmail.com', 'Liam', 'Olivia', 'lolivia$345');

    ...
)

# VIEW THE RECORDS INSERTED

for record in db.engine.execute('SELECT * FROM users'):

    print(record)
```

```
# RUN THE APP

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

Output:

```
('john.doe@zmail.com', 'John', 'Doe', 'john@123')
('noah.emma@wmail.com', 'Emma', 'Noah', 'emaaa!00')
('liam.olivia@wmail.com', 'Liam', 'Olivia', 'lolivia#900')
('john.doe@zmail.com', 'John', 'Doe', 'johndoe@777')
('emma@tmail.com', 'Emma', 'Noah', 'emaaa!00')
('liam.olivia@wmail.com', 'Liam', 'Olivia', 'lolivia$345')
('emma@tmail.com', 'Emma', 'Noah', 'whrfc2bfh904')
('adam@hmail.com', 'Adam', 'Plum', 'plum45')
('adam@hmail.com', 'Adam', 'Plum', 'plum45')
('adam@hmail.com', 'Adam', 'Plum', 'plum45')
* Serving Flask app "flask_sqlalchemy_query" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

In this example, we created a simple flask app that does not have any route but instead runs raw SQL queries. We have created the SQLAlchemy connection and then executed 3 different raw SQL queries. The first query creates the user's table. The second query inserts some sample records in the table. The third query fetches all the records and displays them in the terminal.

In all three cases, we have used the **db.engine.execute()** method. The db.engine provides an SQLAlchemy engine connection and the execute method takes in a SQL query to execute the request.

Example 2

In this example, we have created 2 different routes to work with. These routes will act as an API where we can send a POST request with a query key in the body. The value for this query key will be the raw SQL query that we need to execute. The get_results API will be used to fetch the records that we get from the SELECT query. The execute_query API is used to execute raw SQL queries and will return the response message if the query is successfully executed or not.

- Python

```
# IMPORT REQUIRED LIBRARIES

from flask import Flask, request

from flask_sqlalchemy import SQLAlchemy


# CREATE THE FLASK APP

app = Flask(__name__)

# ADD THE DATABASE CONNECTION TO THE FLASK APP

db = SQLAlchemy(app)

db_cred = {

    'user': 'root',      # DATABASE USER

    'pass': 'password',  # DATABASE PASSWORD

    'host': '127.0.0.1', # DATABASE HOSTNAME

    'name': 'Geeks4Geeks' # DATABASE NAME

}

app.config['SQLALCHEMY_DATABASE_URI'] = f"mysql+pymysql://{db_cred['user']}:{db_cred['pass']}@{db_cred['host']}\\n{db_cred['name']}"

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False


# APP ROUTE TO GET RESULTS FOR SELECT QUERY

@app.route('/get_results', methods=['POST'])

def get_results():

    # GET THE SQLALCHEMY RESULTPROXY OBJECT
```

```
result = db.engine.execute(request.get_json()['query'])

response = {}

i = 1

# ITERATE OVER EACH RECORD IN RESULT AND ADD IT
# IN A PYTHON DICT OBJECT

for each in result:

    response.update({f'Record {i}': list(each)})

    i+= 1


return response

# APP ROUTE TO RUN RAW SQL QUERIES

@app.route('/execute_query', methods=['POST'])

def execute_query():

    try:

        db.engine.execute(request.get_json()['query'])

    except:

        return {"message": "Request could not be completed."}

    return {"message": "Query executed successfully."}

# RUN THE APP

if __name__ == '__main__':

    app.run()
```

Output:

We will test the routes through POSTMAN. Following are the 3 cases that are tested using POSTMAN.

1. Running a SELECT query to fetch all the records through the get_results API

The screenshot shows the Postman interface with a successful API call. The URL is `http://127.0.0.1:5000/get_results`. The Body tab contains a raw JSON response with two records. The response is displayed in Pretty mode.

```
1 "Record 1": [
2     "john.doe@zmail.com",
3     "John",
4     "Doe",
5     "john@123",
6     "New user created",
7     "Sat, 06 Nov 2021 12:51:33 GMT"
8 ],
9 "Record 2": [
10     "noah.emma@wmail.com",
11     "Emma",
12     "Noah",
13     "emaaa!00",
14     "New user created",
15     "Sat, 06 Nov 2021 12:53:13 GMT"
16 ],
17 ]
```

2. Next, we will test the execute_query API for a valid INSERT query

POST http://127.0.0.1:5000/execute_query

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body (raw) "query": "INSERT INTO users(email, first_name, last_name, passwd) VALUES ('adam@hmail.com', 'Adam', 'Plum', 'plum45');"

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON

Status: 200 OK Time: 2.28 s Size: 19c

1 "message": "Query executed successfully."

3. Lastly, we will put any random query and see if we get any error message

POST http://127.0.0.1:5000/execute_query

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body (raw) "query": "THIS IS JUST ANYTHING;"

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON

1 "message": "Request could not be completed."

PART 6: Flask Deployment and Error Handling

CHAPTER 1: Subdomain in Flask

Prerequisite: [Introduction to Flask](#)

In this article, we will learn how to setup subdomains in Flask. But first, let's go through the basic like what is DNS and subdomains.

Domain Name System (DNS):

The Domain Name System (DNS) is a hierarchical and decentralized naming system for computers, services, or other resources connected to the Internet or a private network. Most prominently, it translates more readily memorized domain names to the numerical IP addresses needed for locating and identifying computer services and devices with the underlying network protocols.

DNS is basically using words (Domain Names) in place of numbers (IP addresses) to locate something. For example, 127.0.0.1 is used to point the local computer address, *localhost*.

Subdomain:

A subdomain is a domain that is part of a larger domain. Basically, it's a sort of child domain which means it is a part of some parent domain. For example, practice.geeksforgeeks.org and write.geeksforgeeks.org are subdomains of the geeksforgeeks.org domain, which in turn is a subdomain of the org top-level domain (TLD).

These are different from the path defined after TLD as in geeksforgeeks.org/basic/.

Further, we will discuss how to set endpoints in your web application using Python's micro-framework, Flask.

Adding alternate domain name for local IP -

Prior to the coding part, we got to setup *hosts* file in order to provide alternate names to local IP so that we are able to test our app locally. Edit this file with root privileges.

Linux: /etc/hosts

Windows: C:\Windows\System32\Drivers\etc\hosts

Add these lines to set up alternate domain names.

127.0.0.1 vibhu.gfg

127.0.0.1 practice.vibhu.gfg

In this example, we're considering vibhu.gfg as our domain name, with gfg being the TLD. practice would be a subdomain we're targeting to set in our web app.

Setting up the Server -

In the app's configuration SERVER_NAME is set to the domain name, along with the port number we intend to run our app on. The default port, flask uses is 5000, so we take it as it is.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():

    return "Welcome to GeeksForGeeks !"

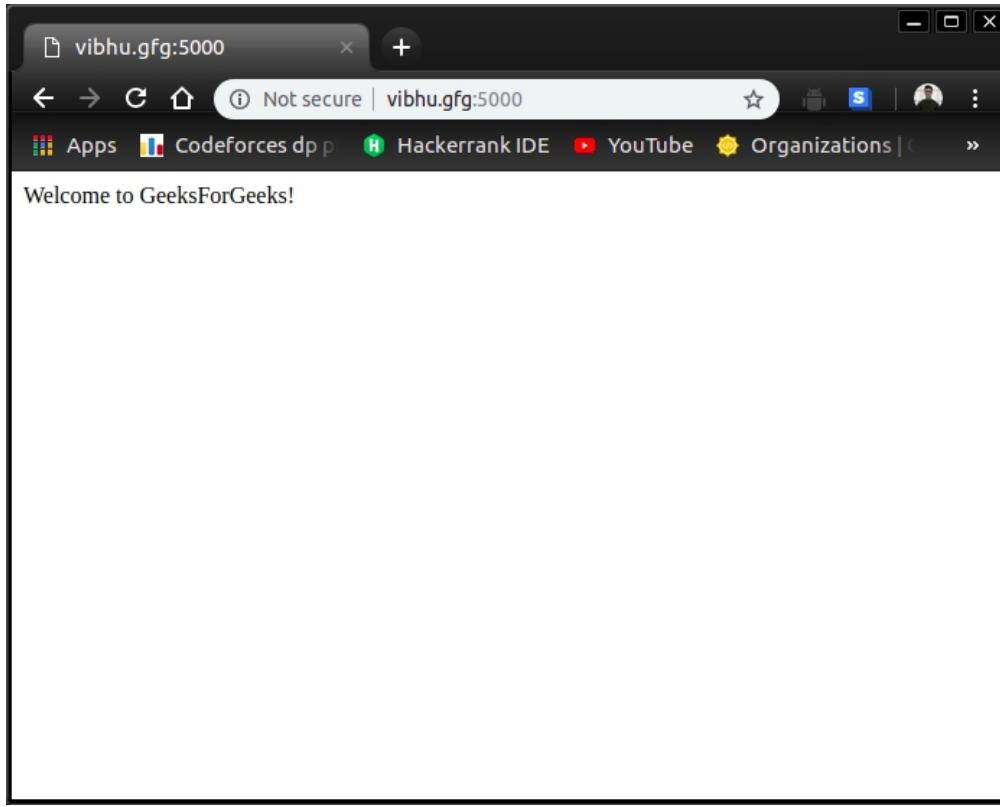
if __name__ == "__main__":
    website_url = 'vibhu.gfg:5000'
    app.config['SERVER_NAME'] = website_url
    app.run()
```

Output:

Run the app and notice the link on which the app is running.

```
vibhu@potterhead:~/Desktop/GfG$ python3 flask_code.py
 * Serving Flask app "flask_code" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://vibhu.gfg:5000/ (Press CTRL+C to quit)
```

Test the link on your browser.



Adding Several Endpoints -

1. **basic:** An endpoint with extension to the path on the main domain.
2. **practice:** An endpoint serving on the `practice` subdomain.
3. **courses:** An endpoint with extension on to the path on the practice subdomain.

Subdomains in Flask are set using the `subdomain` parameter in the `app.route` decorator.

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/')
```

```
def home():
```

```
return "Welcome to GeeksForGeeks !"

@app.route('/basic/')

def basic():

    return "Basic Category Articles " \
           "listed on this page."

@app.route('/', subdomain ='practice')

def practice():

    return "Coding Practice Page"

@app.route('/courses/', subdomain ='practice')

def courses():

    return "Courses listed " \
           "under practice subdomain."

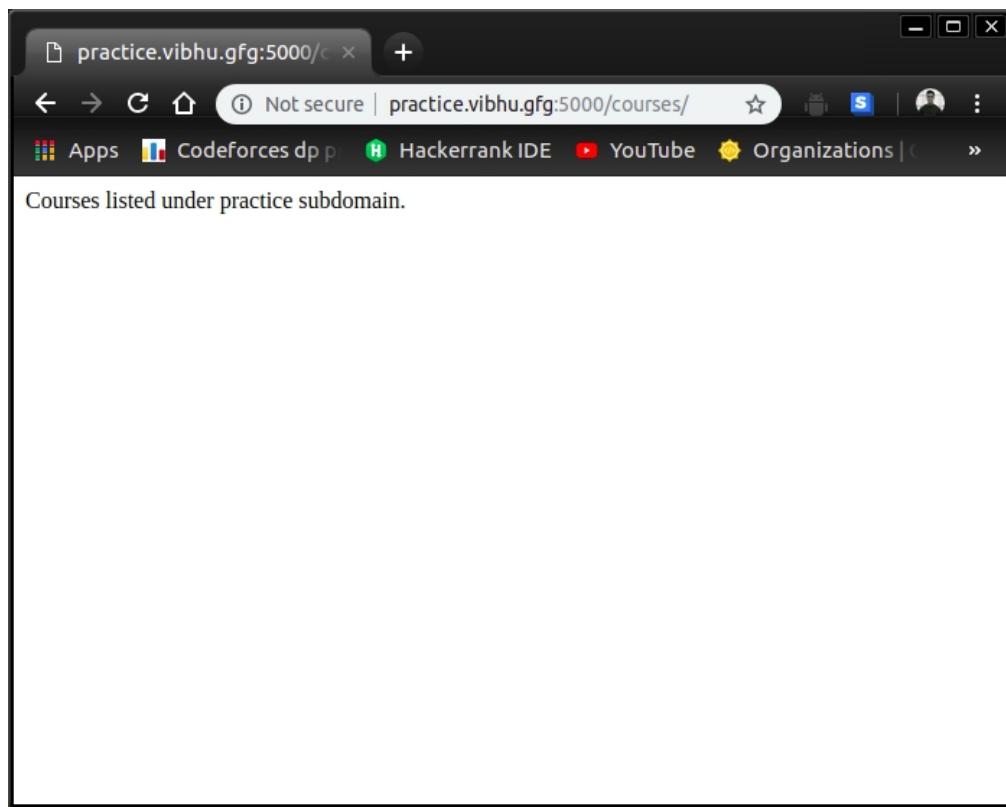
if __name__ == "__main__":

    website_url = 'vibhu.gfg:5000'

    app.config['SERVER_NAME'] = website_url

    app.run()
```

Output:



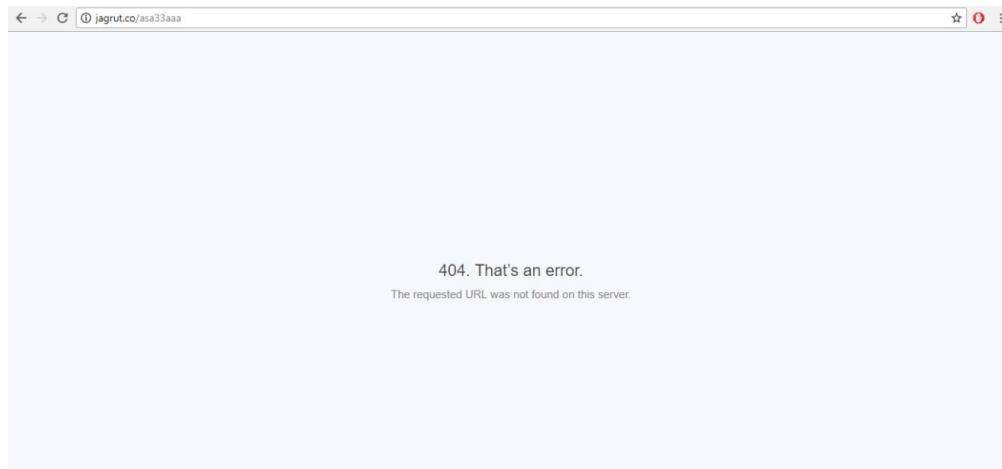
CHAPTER 2: Handling 404 Error in Flask

Prerequisite: Creating simple application in Flask

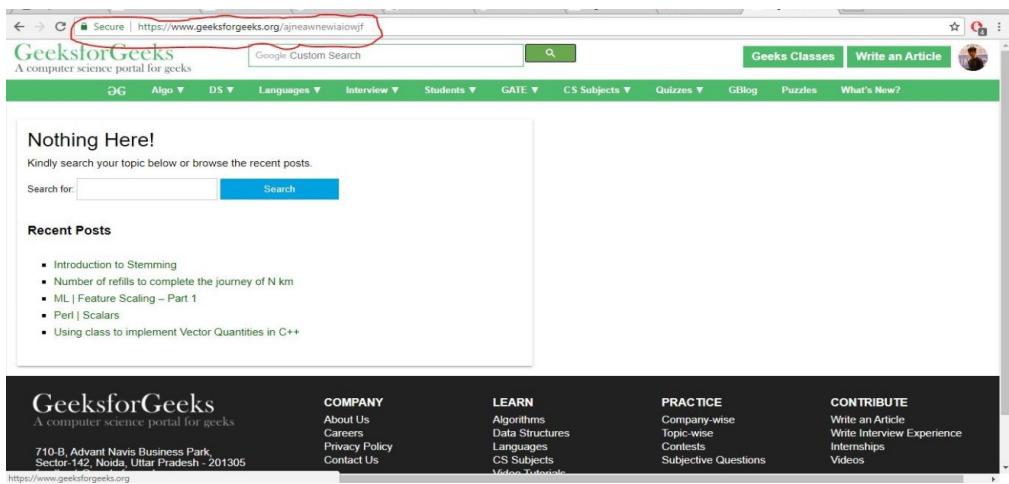
A 404 Error is showed whenever a page is not found. Maybe the owner changed its URL and forgot to change the link or maybe they deleted the page itself. Every site needs a Custom Error page to avoid the user to see the default Ugly Error page.

GeeksforGeeks also has a customized error page. If we type a URL like www.geeksforgeeks.org/ajneawnewiaiowjf

Default 404 Error



GeeksForGeeks Customized Error Page



It will show an Error 404 page since this URL doesn't exist. But an error page provides a beautiful layout, helps the user to go back, or even takes them to the homepage after a specific time interval. That is why Custom Error pages are necessary for every website.

Flask provides us with a way to handle the error and return our Custom Error page.

For this, we need to download and import flask. Download the flask through the following commands on CMD.

```
pip install flask
```

Using app.py as our Python file to manage templates, 404.html be the file we will return in the case of a 404 error and header.html be the file with header and navbar of a website.

app.py

Flask allows us to make a python file to define all routes and functions. In app.py we have defined the route to the main page ('/') and error handler function which is a flask function and we passed 404 error as a parameter.

```
from flask import Flask, render_template
```

```
app = Flask(__name__)
```

```
# app name
```

```
@app.errorhandler(404)
```

```
# inbuilt function which takes error as parameter

def not_found(e):

    # defining function

    return render_template("404.html")
```

The above python program will return 404.html file whenever the user opens a broken link.

404.html

The following code exports header and navbar from header.html. Both files should be stored in templates folder according to the flask.

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

<!-- Exports header and navbar from header.html

or any file you want--&gt;

{% block title %}Page Not Found{% endblock %}

{% block body %}

&lt;h1&gt;Oops! Looks like the page doesn't exist anymore&lt;/h1&gt;

&lt;a href="{{ url_for('index') }}"&gt;&lt;p&gt;Click Here&lt;/p&gt;To go to the Home Page&lt;/p&gt;

<!-- {{ url_for('index') }} is a var which returns url of index.html--&gt;

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

Automatically Redirecting to the Home page after 5 seconds

The app.py code for this example stays the same as above. The following code Shows the Custom 404 Error page and starts a countdown of 5 seconds. After 5 seconds are completed, it redirects the user back to the homepage.

404.html

The following code exports header and navbar from header.html.
Both files should be stored in the templates folder according to the flask.
After 5 seconds, the user will get redirected to the Home Page
Automatically.

```
<html>
<head>
<title>Page Not Found</title>

<script language="JavaScript" type="text/javascript">

var seconds =6;
// countdown timer. took 6 because page takes approx 1 sec to load

var url="{!!url_for(index)}";
// variable for index.html url

function redirect(){
if (seconds <=0){

// redirect to new url after counter down.

window.location = url;

} else {
seconds--;

document.getElementById("pageInfo").innerHTML="Redirecting to Home Page after "
+seconds+" seconds."

setTimeout("redirect()", 1000)

}
}

</script>
```

```

</head>

{% extends "header.html" %}

//exporting navbar and header from header.html

{% block body %}

<body onload="redirect()">

<p id="pageInfo"></p>

{% endblock %}

</body>

</html>

```

Sample header.html

This is a sample header.html which includes a navbar just like shown in the image.

It's made up of bootstrap. You can also make one of your own.
For this one, refer the bootstrap documentation.

```

<!DOCTYPE html>

<html>
<head>

<!-- LINKING ALL SCRIPTS/CSS REQUIRED FOR NAVBAR -->

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
css/bootstrap.min.css"
integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E26
3XmFcJISAwGgFAW/dAiS6JXm" crossorigin="anonymous">

```

```
<title>Flask</title>

</head>

<body>

<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity=
"sha384-KJ3o2DKtlkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"
crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd
/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K
/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQxSfFWpi1MquVdAyjUar5+76PVCmYI"
crossorigin="anonymous"></script>

<header>

<!-- Starting header -->

<nav class="navbar navbar-expand-lg navbar-light bg-light">

<a class="navbar-brand" href="#">Navbar</a>

<!-- bootstrap classes for navbar -->

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target=
"#navbarSupportedContent" aria-controls="navbarSupportedContent"
aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>
```

```
<div class="collapse navbar-collapse" id="navbarSupportedContent">

  <ul class="navbar-nav mr-auto">

    <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="#">Link</a>

    </li>

    <li class="nav-item dropdown">

      <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">

        Dropdown

      </a>

      <div class="dropdown-menu" aria-labelledby="navbarDropdown">

        <a class="dropdown-item" href="#">Action</a>

        <a class="dropdown-item" href="#">Another action</a>

        <div class="dropdown-divider"></div>

        <a class="dropdown-item" href="#">Something else here</a>

      </div>

    </li>

    <li class="nav-item">

      <a class="nav-link disabled" href="#">Disabled</a>

    </li>

  </ul>

  <form class="form-inline my-2 my-lg-0">
```

```
<input class="form-control mr-sm-2" type="search"
placeholder="Search" aria-label="Search">

<button class="btn btn-outline-success my-2 my-sm-0"
type="submit">Search</button>

</form>

</div>

</nav>

</head>

<body >

{block body}

{endblock}

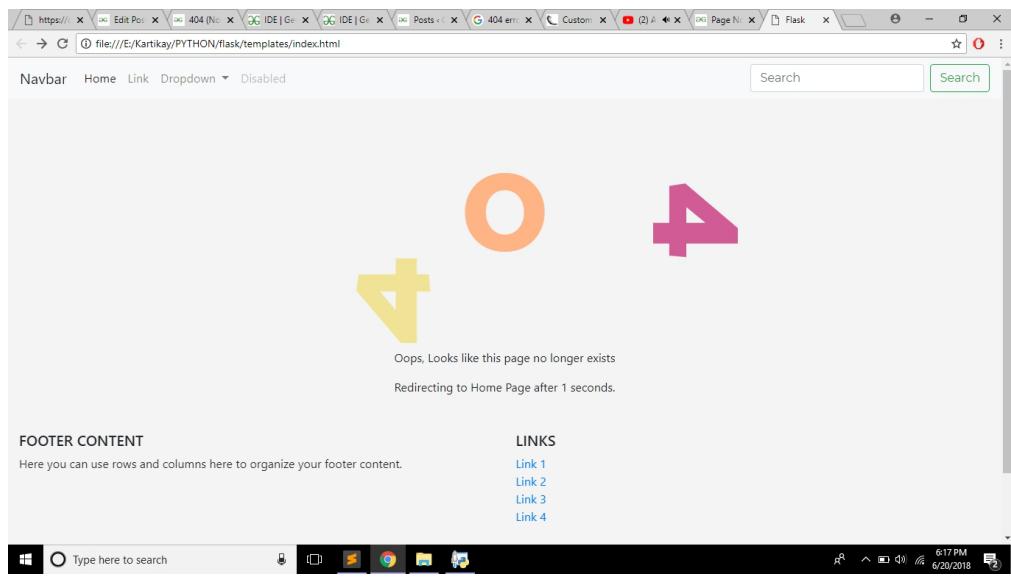
</body>

</html>
```

Output:

The output will be a custom error page with header.html that the user exported.

The following is an example output with my custom header, footer, and 404.html file.



CHAPTER 3: Deploy Python Flask App on Heroku

Flask is a web application framework written in Python. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects. This article revolves around **how to deploy a flask app on Heroku**. To demonstrate this, we are first going to create a sample application for a better understanding of the process.

Prerequisites

- Python
- pip
- Heroku CLI
- Git

Deploying Flask App on Heroku

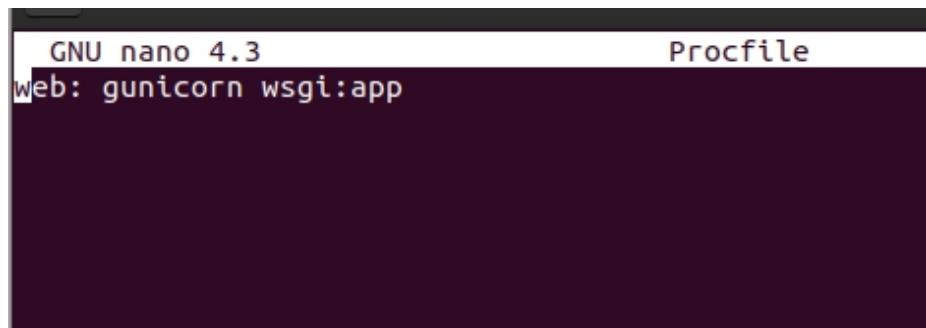
Let's create a simple flask application first and then it can be deployed to heroku. Create a folder named “**eFlask**” and open the command line and cd inside the “**eFlask**” directory. Follow the following steps to create the sample application for this tutorial.

STEP 1 : Create a virtual environment with pipenv and install **Flask** and **Gunicorn** .

```
$ pipenv install flask gunicorn
```

STEP 2 : Create a “**Procfile**” and write the following code.

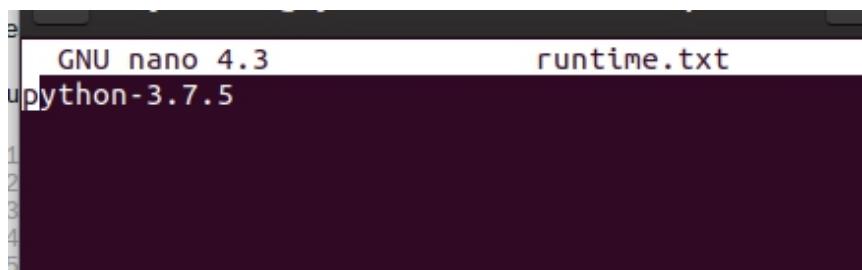
```
$ touch Procfile
```



```
GNU nano 4.3          Procfile
web: gunicorn wsgi:app
```

STEP 3 : Create “**runtime.txt**” and write the following code.

```
$ touch runtime.txt
```



```
GNU nano 4.3          runtime.txt
python-3.7.5
```

STEP 4 : Create a folder named “**app**” and enter the folder.

```
$ mkdir app
$ cd app
```

STEP 5 : Create a python file, “**main.py**” and enter the sample code.

```
touch main.py
```

```
• Python3
```

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route("/")
```

```
def home_view():
```

```
return "<h1>Welcome to Geeks for Geeks</h1>"
```

STEP 6 :Get back to the previous directory “eflask”.Create a file “**wsgi.py**” and insert the following code.

```
$ cd ../  
$ touch wsgi.py
```

- Python3

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

STEP 7 : Run the virtual environment.

```
$ pipenv shell
```

STEP 8 : Initialize an empty repo, add the files in the repo and commit all the changes.

```
$ git init  
$ git add .  
$ git commit -m "Initial Commit"
```

STEP 9 : Login to heroku CLI using

```
heroku login
```

Now, Create a unique name for your Web app.

```
$ heroku create eflask-app
```

```
Create Mode 100044 wsgi.py  
(eflask--Z9sd7Cc) ayushman@ayushman-Lenovo-G50-80:~/Documents/PROJECT_TEST/eflask$ heroku create eflask-app  
Creating ⬤eflask-app... done  
https://eflask-app.herokuapp.com/ | https://git.heroku.com/eflask-app.git
```

STEP 10 : Push your code from local to the heroku remote.

```
$ git push heroku master
```

```
(eflask-z9sD7Cc) ayushman@ayushman-Lenovo-G50-80:~/Documents/PROJECT_TEST/eflask$ git push heroku master
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 2.73 KiB | 932.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: !     Python has released a security update! Please consider upgrading to python-3.7.6
remote:     Learn More: https://devcenter.heroku.com/articles/python-runtimes
remote: -----> Installing python-3.7.5
remote: -----> Installing pip
remote: -----> Installing dependencies with Pipenv 2018.5.18...
remote:     Installing dependencies from Pipfile.lock (d977f2)...
remote: -----> Installing SQLite3
remote: SQLite3 successfully installed.
remote: -----> Discovering process types
remote:     Procfile declares types -> web
remote:
remote: -----> Compressing...
remote: Done: 60.3M
remote: -----> Launching...
remote: Released v3
remote: https://eflask-app.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/eflask-app.git
 * [new branch]      master -> master
```

Finally, web app will be deployed on <http://eflask-app.herokuapp.com>.



CHAPTER 4: Deploy Machine Learning Model using Flask

Machine learning is a process that is widely used for prediction. Number of algorithms are available in various libraries which can be used for prediction. In this article, we are going to build a prediction model on historical data using different machine learning algorithms and classifiers, plot the results, and calculate the accuracy of the model on the testing data.

Building/Training a model using various algorithms on a large dataset is one part of the data. But using these models within the different applications is the second part of deploying machine learning in the real world.

To put it to use in order to predict the new data, we have to deploy it over the internet so that the outside world can use it. In this article, we will talk about how we have trained a machine learning model and created a web application on it using Flask.

We have to install many required libraries which will be used in this model. Use pip command to install all the libraries.

```
pip install pandas  
pip install numpy  
pip install sklearn
```

Decision Tree is a well-known supervised machine learning algorithm because it is easy to use, resilient and flexible. I have implemented the algorithm on Adult dataset from the UCI machine learning repository.

Note: One can [get the custom dataset from here](#).

Getting the dataset is not the end. We have to preprocess the data, which means we need to clean the dataset. Cleaning of the dataset includes different types of processes like removing missing values, filling NA values, etc.

Example

• | Python3

```
# importing the dataset

import pandas

import numpy

from sklearn import preprocessing

df = pandas.read_csv('adult.csv')

df.head()
```

Output:

age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States <=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States <=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States <=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States <=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba <=50K

Preprocessing the dataset: It consists of 14 attributes and a class label telling whether the income of the individual is less than or more than 50K a year. These attributes range from the age of the person and the working-class label to relationship status and the race the person belongs to. The information about all the attributes can be found here. At first, we find and remove any missing values from the data. We have replaced the missing values with the mode value in that column. There are many other ways to replace missing values but for this type of dataset, it seemed most optimal.

• | Python3

```
df = df.drop(['fnlwgt', 'educational-num'], axis=1)
```

```

col_names = df.columns

for c in col_names:
    df = df.replace("?", numpy.NaN)

df = df.apply(lambda x: x.fillna(x.value_counts().index[0]))

```

The machine learning algorithm cannot process categorical data values. It can only process numerical values.

To fit the data into the prediction model, we need to convert categorical values to numerical ones. Before that, we will evaluate if any transformation on categorical columns is necessary.

Discretization is a common way to make categorical data more tidy and meaningful. We have applied discretization on column marital_status where they are narrowed down to only to values married or not married. Later, we will apply a label encoder in the remaining data columns. Also, there are two redundant columns {'education', 'educational-num'}. Therefore, we have removed one of them.

- Python3

```

df.replace(['Divorced', 'Married-AF-spouse',
           'Married-civ-spouse', 'Married-spouse-absent',
           'Never-married', 'Separated', 'Widowed'],
           ['divorced', 'married', 'married', 'married',
            'not married', 'not married', 'not married'], inplace=True)

category_col = ['workclass', 'race', 'education', 'marital-status', 'occupation',
                'relationship', 'gender', 'native-country', 'income']

labelEncoder = preprocessing.LabelEncoder()

mapping_dict = {}

for col in category_col:

```

```

df[col] = labelEncoder.fit_transform(df[col])

le_name_mapping = dict(zip(labelEncoder.classes_,
                           labelEncoder.transform(labelEncoder.classes_)))

mapping_dict[col] = le_name_mapping

print(mapping_dict)

```

Output :

```

{'workclass': {'?': 0, 'Federal-gov': 1, 'Local-gov': 2, 'Never-worked': 3,
  'Private': 4, 'Self-emp-inc': 5, 'Self-emp-not-inc': 6, 'State-gov': 7, 'Without-pay': 8}, 'race': {'Amer-Indian-Eskimo': 0, 'Asian-Pac-Islander': 1, 'Black': 2, 'Other': 3, 'White': 4}, 'education': {'10th': 0, '11th': 1, '12th': 2, '1st-4th': 3, '5th-6th': 4, '7th-8th': 5, '9th': 6, 'Assoc-acdm': 7, 'Assoc-voc': 8, 'Bachelors': 9, 'Doctorate': 10, 'HS-grad': 11, 'Masters': 12, 'Preschool': 13, 'Prof-school': 14, 'Some-college': 15}, 'marital-status': {'Divorced': 0, 'Married-AF-spouse': 1, 'Married-civ-spouse': 2, 'Married-spouse-absent': 3, 'Never-married': 4, 'Separated': 5, 'Widowed': 6}, 'occupation': {'?': 0, 'Adm-clerical': 1, 'Armed-Forces': 2, 'Craft-repair': 3, 'Exec-managerial': 4, 'Farming-fishing': 5, 'Handlers-cleaners': 6, 'Machine-op-inspect': 7, 'Other-service': 8, 'Priv-house-serv': 9, 'Prof-specialty': 10, 'Protective-serv': 11, 'Sales': 12, 'Tech-support': 13, 'Transport-moving': 14}, 'relationship': {'Husband': 0, 'Not-in-family': 1, 'Other-relative': 2, 'Own-child': 3, 'Unmarried': 4, 'Wife': 5}, 'gender': {'Female': 0, 'Male': 1}, 'native-country': {'?': 0, 'Cambodia': 1, 'Canada': 2, 'China': 3, 'Columbia': 4, 'Cuba': 5, 'Dominican-Republic': 6, 'Ecuador': 7, 'El-Salvador': 8, 'England': 9, 'France': 10, 'Germany': 11, 'Greece': 12, 'Guatemala': 13, 'Haiti': 14, 'Holand-Netherlands': 15, 'Honduras': 16, 'Hong': 17, 'Hungary': 18, 'India': 19, 'Iran': 20, 'Ireland': 21, 'Italy': 22, 'Jamaica': 23, 'Japan': 24, 'Laos': 25, 'Mexico': 26, 'Nicaragua': 27, 'Outlying-US(Guam-USVI-etc)': 28, 'Peru': 29, 'Philippines': 30, 'Poland': 31, 'Portugal': 32, 'Puerto-Rico': 33, 'Scotland': 34, 'South': 35, 'Taiwan': 36, 'Thailand': 37, 'Trinadad&Tobago': 38, 'United-States': 39, 'Vietnam': 40, 'Yugoslavia': 41}, 'income': {'50K': 1}}

```

Fitting the model: After pre-processing the data, the data is ready to be fed to the machine learning algorithm. We then slice the data separating the labels with the attributes. Now, we split the dataset into two halves, one for training and one for testing. This is achieved using `train_test_split()` function of `sklearn`.

- Python3

```
from sklearn.model_selection import train_test_split  
  
from sklearn.tree import DecisionTreeClassifier  
  
from sklearn.metrics import accuracy_score  
  
  
X = df.values[:, 0:12]  
  
Y = df.values[:, 12]
```

We have used decision tree classifier here as a predicting model. We fed the training part of the data to train the model.

Once training is done, we test what is the accuracy of the model by providing testing part of the data to the model.

With this, we achieve an accuracy of 84% approximately. Now in order to use this model with new unknown data, we need to save the model so that we can predict the values later. For this, we make use of *pickle* in Python, which is a powerful algorithm for serializing and de-serializing a Python object structure.

- Python3

```
X_train, X_test, y_train, y_test = train_test_split(  
    X, Y, test_size = 0.3, random_state = 100)  
  
  
dt_clf_gini = DecisionTreeClassifier(criterion = "gini",  
    random_state = 100,  
    max_depth = 5,
```

```
min_samples_leaf = 5)
```

```
dt_clf_gini.fit(X_train, y_train)

y_pred_gini = dt_clf_gini.predict(X_test)

print ("Decision Tree using Gini Index\nAccuracy is ",

accuracy_score(y_test, y_pred_gini)*100 )
```

Output :

Decision Tree using Gini Index

Accuracy is 83.13031016480704

Now, Flask is a Python-based micro framework used for developing small-scale websites. Flask is very easy to make Restful APIs using python. As of now, we have developed a model i.e model.pkl , which can predict a class of the data based on various attributes of the data. The class label is *Salary >=50K or <50K*.

Now we will design a web application where the user will input all the attribute values and the data will be given to the model, based on the training given to the model, the model will predict what should be the salary of the person whose details have been fed.

HTML Form: We first need to collect the data(new attribute values) to predict the income from various attributes and then use the decision tree model we build above to predict whether the income is more than 50K or less. Therefore, in order to collect the data, we create an HTML form which would contain all the different options to select from each attribute. Here, we have created a simple form using HTML only. If you want to make the form more interactive you can do so as well.

- HTML

```
<html>
<body>
<h3>Income Prediction Form</h3>
```

```
<div>

<form action="/result" method="POST">

    <label for="age">Age</label>

    <input type="text" id="age" name="age">

    <br>

    <label for="w_class">Working Class</label>

    <select id="w_class" name="w_class">

        <option value="0">Federal-gov</option>

        <option value="1">Local-gov</option>

        <option value="2">Never-worked</option>

        <option value="3">Private</option>

        <option value="4">Self-emp-inc</option>

        <option value="5">Self-emp-not-inc</option>

        <option value="6">State-gov</option>

        <option value="7">Without-pay</option>

    </select>

    <br>

    <label for="edu">Education</label>

    <select id="edu" name="edu">

        <option value="0">10th</option>

        <option value="1">11th</option>

        <option value="2">12th</option>

        <option value="3">1st-4th</option>

        <option value="4">5th-6th</option>

        <option value="5">7th-8th</option>

        <option value="6">9th</option>

    </select>

</div>
```

```
<option value="7">Assoc-acdm</option>
<option value="8">Assoc-voc</option>
<option value="9">Bachelors</option>
<option value="10">Doctorate</option>
<option value="11">HS-grad</option>
<option value="12">Masters</option>
<option value="13">Preschool</option>
<option value="14">Prof-school</option>
<option value="15">16 - Some-college</option>
</select>
<br>
<label for="marital_stat">Marital Status</label>
<select id="marital_stat" name="marital_stat">
<option value="0">divorced</option>
<option value="1">married</option>
<option value="2">not married</option>
</select>
<br>
<label for="occup">Occupation</label>
<select id="occup" name="occup">
<option value="0">Adm-clerical</option>
<option value="1">Armed-Forces</option>
<option value="2">Craft-repair</option>
<option value="3">Exec-managerial</option>
<option value="4">Farming-fishing</option>
<option value="5">Handlers-cleaners</option>
```

```
<option value="6">Machine-op-inspect</option>
<option value="7">Other-service</option>
<option value="8">Priv-house-serv</option>
<option value="9">Prof-specialty</option>
<option value="10">Protective-serv</option>
<option value="11">Sales</option>
<option value="12">Tech-support</option>
<option value="13">Transport-moving</option>
</select>
<br>
<label for="relation">Relationship</label>
<select id="relation" name="relation">
<option value="0">Husband</option>
<option value="1">Not-in-family</option>
<option value="2">Other-relative</option>
<option value="3">Own-child</option>
<option value="4">Unmarried</option>
<option value="5">Wife</option>
</select>
<br>
<label for="race">Race</label>
<select id="race" name="race">
<option value="0">Amer Indian Eskimo</option>
<option value="1">Asian Pac Islander</option>
<option value="2">Black</option>
<option value="3">Other</option>
```

```
<option value="4">White</option>
</select>
<br>

<label for="gender">Gender</label>

<select id="gender" name="gender">
    <option value="0">Female</option>
    <option value="1">Male</option>
</select>
<br>

<label for="c_gain">Capital Gain </label>

<input type="text" id="c_gain" name="c_gain">btw:[0-99999]
<br>

<label for="c_loss">Capital Loss </label>

<input type="text" id="c_loss" name="c_loss">btw:[0-4356]
<br>

<label for="hours_per_week">Hours per Week </label>

<input type="text" id="hours_per_week" name="hours_per_week">btw:[1-99]
<br>

<label for="native-country">Native Country</label>

<select id="native-country" name="native-country">
    <option value="0">Cambodia</option>
    <option value="1">Canada</option>
    <option value="2">China</option>
    <option value="3">Columbia</option>
    <option value="4">Cuba</option>
    <option value="5">Dominican Republic</option>
```

```
<option value="6">Ecuador</option>
<option value="7">El Salvador</option>
<option value="8">England</option>
<option value="9">France</option>
<option value="10">Germany</option>
<option value="11">Greece</option>
<option value="12">Guatemala</option>
<option value="13">Haiti</option>
<option value="14">Netherlands</option>
<option value="15">Honduras</option>
<option value="16">HongKong</option>
<option value="17">Hungary</option>
<option value="18">India</option>
<option value="19">Iran</option>
<option value="20">Ireland</option>
<option value="21">Italy</option>
<option value="22">Jamaica</option>
<option value="23">Japan</option>
<option value="24">Laos</option>
<option value="25">Mexico</option>
<option value="26">Nicaragua</option>
<option value="27">Outlying-US(Guam-USVI-etc)</option>
<option value="28">Peru</option>
<option value="29">Philippines</option>
<option value="30">Poland</option>
```

```
<option value="11">Portugal</option>
<option value="32">Puerto-Rico</option>
<option value="33">Scotland</option>
<option value="34">South</option>
<option value="35">Taiwan</option>
<option value="36">Thailand</option>
<option value="37">Trinadad&Tobago</option>
<option value="38">United States</option>
<option value="39">Vietnam</option>
<option value="40">Yugoslavia</option>
</select>
<br>
<input type="submit" value="Submit">
</form>
</div>
</body>
</html>
```

Output :

Income Prediction Form

Age

Working Class Federal-gov ▾

Education 10th ▾

Marital Status divorced ▾

Occupation Adm-clerical ▾

Relationship Husband ▾

Race Amer Indian Eskimo ▾

Gender Female ▾

Capital Gain btw:[0-99999]

Capital Loss btw:[0-4356]

Hours per Week btw:[1-99]

Native Country Cambodia ▾

Note: In order to predict the data correctly, the corresponding values of each label should match the value of each input selected. For example — In the attribute Relationship, there are 6 categorical values. These are converted to numerical like this {'Husband': 0, 'Not-in-family': 1, 'Other-relative': 2, 'Own-child': 3, 'Unmarried': 4, 'Wife': 5}. Therefore we need to put the same values to the HTML form.

- Python3

```
# prediction function

def ValuePredictor(to_predict_list):

    to_predict = np.array(to_predict_list).reshape(1, 12)

    loaded_model = pickle.load(open("model.pkl", "rb"))

    result = loaded_model.predict(to_predict)

    return result[0]
```

```
@app.route('/result', methods = ['POST'])
```

```
def result():

    if request.method == 'POST':
```

```

to_predict_list = request.form.to_dict()

to_predict_list = list(to_predict_list.values())

to_predict_list = list(map(int, to_predict_list))

result = ValuePredictor(to_predict_list)

if int(result) == 1:

    prediction = 'Income more than 50K'

else:

    prediction = 'Income less than 50K'

return render_template("result.html", prediction = prediction)

```

Once the Data is posted from the form, the data should be fed to the model.

Flask script: Before starting with the coding part, we need to download flask and some other libraries. Here, we make use of a virtual environment, where all the libraries are managed which makes both the development and deployment job easier.

Here is the code to run the code using a virtual environment.

```

mkdir income-prediction
cd income-prediction
python3 -m venv venv
source venv/bin/activate

```

Now let's install Flask.

```
pip install flask
```

Let's create folder templates. In your application, you will use templates to render HTML which will display in the user's browser. This folder contains our HTML form file index.html.

```
mkdir templates
```

Create script.py file in the project folder and copy the following code. Here we import the libraries, then using `app=Flask(__name__)` we create an instance of flask. `@app.route('/')` is used to tell flask what URL should trigger the function `index()` and in the function `index`, we use

```
render_template('index.html') to display the script index.html in the browser.
```

Let's run the application.

```
export FLASK_APP=script.py #this line will work in linux  
set FLASK_APP=script.py # this it the code for windows.  
run flask
```

This should run the application and launch a simple server. Open <http://127.0.0.1:5000/> to see the html form.

Predicting the income value: When someone submits the form, the webpage should display the predicted value of income. For this, we require the model file (model.pkl) we created before in the same project folder.

Here, after the form is submitted, the form values are stored in the variable to_predict_list in the form of a dictionary. We convert it into a list of the dictionary's values and pass it as an argument to ValuePredictor() function. In this function, we load the model.pkl file and predict the new values and return the result.

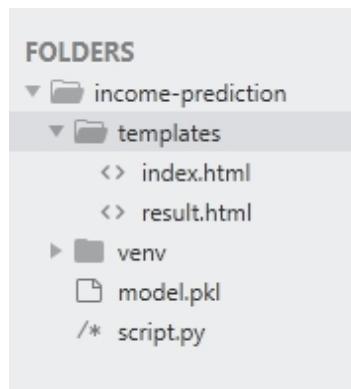
This result/prediction (Income more than or less than 50k) is then passed as an argument to the template engine with the HTML page to be displayed.

Create the following result.html file and add it to the templates folder.

- **HTML**

```
<!doctype html>  
  
<html>  
  
  <body>  
    <h1> {{ prediction }}</h1>  
  </body>  
  
</html>
```

Output:



Run the application again and it should predict the income after submitting the form and will display the output on result page.