SMART INTERNZ – APSCHE,

AI / ML Training

Name: Gunti Vasanthi

Email: 208x1a0521@khitguntur.ac.in

College: Kallam Haranadhareddy Institute of Technology

Assessment 3

1.What is Flask, and how does it differ from other web frameworks?

ANS) Flask is a lightweight and versatile web framework for Python it is designed to make it easy to build web applications quickly and with minimal boilerplate code. It's often referred to as a micro-framework because it focuses on simplicity and extensibility, providing just the essentials for web development without imposing any particular way of structuring your application.

Here are some key aspects that differentiate Flask from other web frameworks:

1)Minimalistic: Flask is minimalistic by design, providing only the essential features needed for web development. This minimalism allows developers to have more flexibility and control over their applications.

2)Extensibility: Flask is highly extensible, meaning you can easily add additional functionality to your application using Flask extensions. These extensions cover a wide range of functionalities such as authentication, database integration, form validation, and more.

3)Flexibility: Flask gives developers the freedom to choose their own tools and libraries for various tasks. It doesn't enforce any specific way of doing things, allowing developers to use the tools they are most comfortable with.

4)No built-in ORM or database layer: Unlike some other web frameworks like Django, Flask does not come with a built-in ORM (Object-Relational Mapping) or database layer. This means developers have the freedom to choose their preferred database and ORM library, or even work without one if they prefer.

5)Built-in development server: Flask comes with a built-in development server, making it easy to run and test your application during development without the need for additional setup.

6)URL routing: Flask provides a simple yet powerful URL routing mechanism, allowing developers to map URLs to view functions easily.

7)Template engine: Flask includes a lightweight templating engine called Jinja2, which makes it easy to generate dynamic HTML content in your web applications.

Summary

Overall, Flask's simplicity, flexibility, and extensibility make it a popular choice for building web applications, particularly for projects that require a lightweight framework with minimal dependencies. However, it's worth noting that Flask's minimalistic approach means developers may need to make more decisions and do more configuration compared to more opinionated frameworks like Django.

2. Describe the basic structure of a Flask application.

A Flask application typically follows a basic structure that includes several key components:

1)Application Package: Flask applications are usually organized as packages, which are directories containing a collection of Python modules. This package contains all the files needed to run the Flask application.

2)Main Python Script: The main entry point of the Flask application is a Python script where the Flask application object is created. This script often has a name like app.py, main.py, or similar.

3)Virtual Environment: It's common practice to create a virtual environment to isolate the dependencies of the Flask application from other Python projects. This helps in managing dependencies effectively.

4)Requirements File: A requirements.txt file is used to list all the dependencies (Python packages) required by the Flask application. This file can be used to install all dependencies at once using pip.

5)Static Files: The static directory contains static files such as CSS, JavaScript, images, etc., that are served directly to the client without any processing.

6)Templates: Flask uses Jinja2 templating engine by default. Templates are HTML files with placeholders for dynamic content. These files are stored in the templates directory.

7)Configuration: Configuration parameters for the Flask application, such as database connection strings, secret keys, etc., are often stored in separate configuration files or as environment variables.

8)Routes: Routes define the URL endpoints of the application and the functions that are called when these endpoints are accessed. These are defined using decorators such as @app.route().

9)Views: Views are the Python functions associated with the routes. They handle requests, process data, and return responses. Views typically render templates or return JSON responses.

10)Models (Optional): If the application interacts with a database, models represent the structure of the data and provide an interface for querying and manipulating the data.

11)Forms (Optional): If the application accepts user input through forms, Flask-WTF or similar extensions can be used to define and validate forms.

Middleware (Optional): Middleware can be used to intercept and process requests and responses before they reach the views or after they are generated.

This basic structure provides a foundation for developing Flask applications, but it can vary based on the complexity and requirements of the specific project.

3. How do you install Flask and set up a Flask project?

To install Flask and set up a Flask project, you can follow these steps:

1)Install Python: First, ensure that Python is installed on your system. You can download and install Python from the official website: https://www.python.org/

2)Create a Virtual Environment (Optional): It's a good practice to create a virtual environment to isolate the dependencies of your Flask project. Navigate to your project directory in the terminal and run:

```
python -m venv myenv
```

This will create a virtual environment named myenv. You can replace myenv with any name you prefer.

3)Activate the Virtual Environment: Activate the virtual environment. In Windows, run:

```
myenv\Scripts\activate
```

In macOS/Linux, run:

```
source myenv/bin/activate
```

4)Install Flask: With the virtual environment activated, use pip to install Flask:

```
pip install Flask
```

5)Set Up Your Flask Project Structure: Create a directory for your Flask project. Inside this directory, you can organize your project as follows:

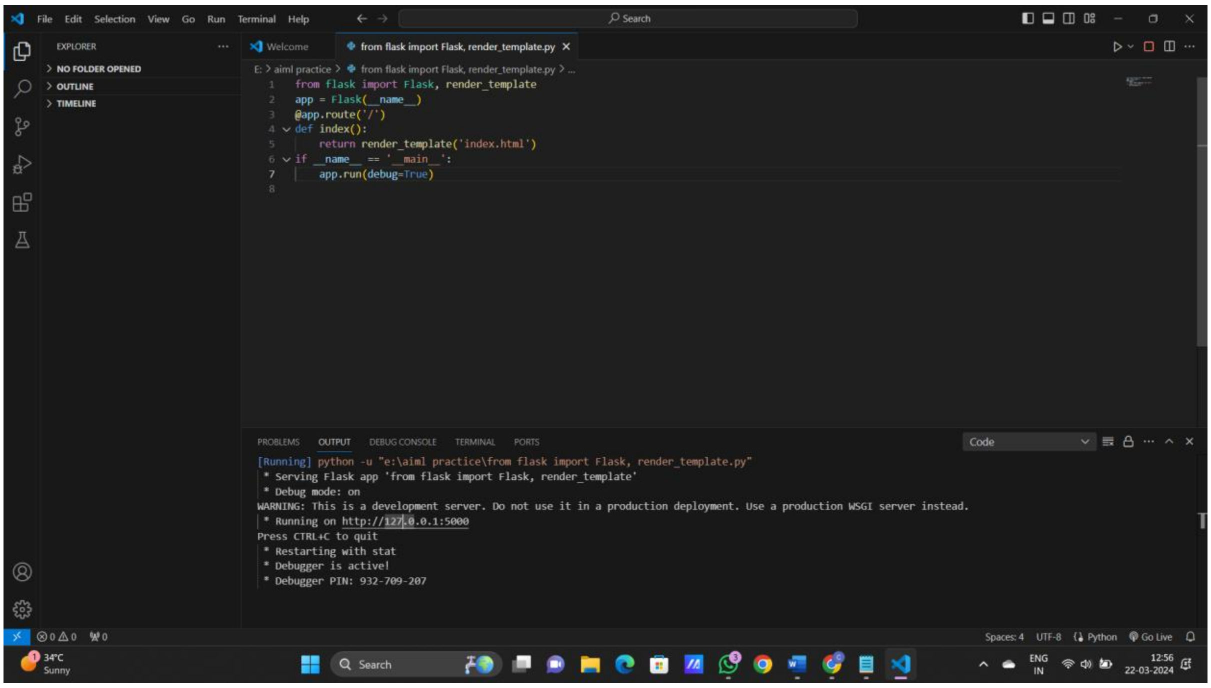
```
my_flask_project/
├── app.py
├── static/
│   └── css/
│       └── style.css
├── templates/
│   └── index.html
└── venv/ (virtual environment directory)
```

app.py: This is where you'll define your Flask application and its routes.

static/: This directory holds static files like CSS, JavaScript, and images.

templates/: This directory contains HTML templates used by your Flask views.

Create Your Flask Application: In your app.py file, you'll define your Flask application. Here's a minimal example:



```python
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)
```

```
[Running] python -u "e:\aiml practice\from flask import Flask, render_template.py"
 * Serving Flask app 'from flask import Flask, render_template'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 932-709-207
```

7)Run Your Flask Application: With your virtual environment activated and inside your project directory, run your Flask application:

python app.py

Your Flask application should now be running locally. You can access it by navigating to http://localhost:5000 in your web browser.

That's it! You've now installed Flask and set up a basic Flask project. You can expand upon this structure by adding more routes, views, templates, and static files as needed for your application**.**

4. Explain the concept of routing in Flask and how it maps URLs to Python functions.

In Flask, routing refers to the process of mapping URLs (Uniform Resource Locators) to specific Python functions within your application. When a user makes a request to a URL, Flask's routing system determines which Python function should handle that request based on the URL pattern defined in your application.

Here's how routing works in Flask:

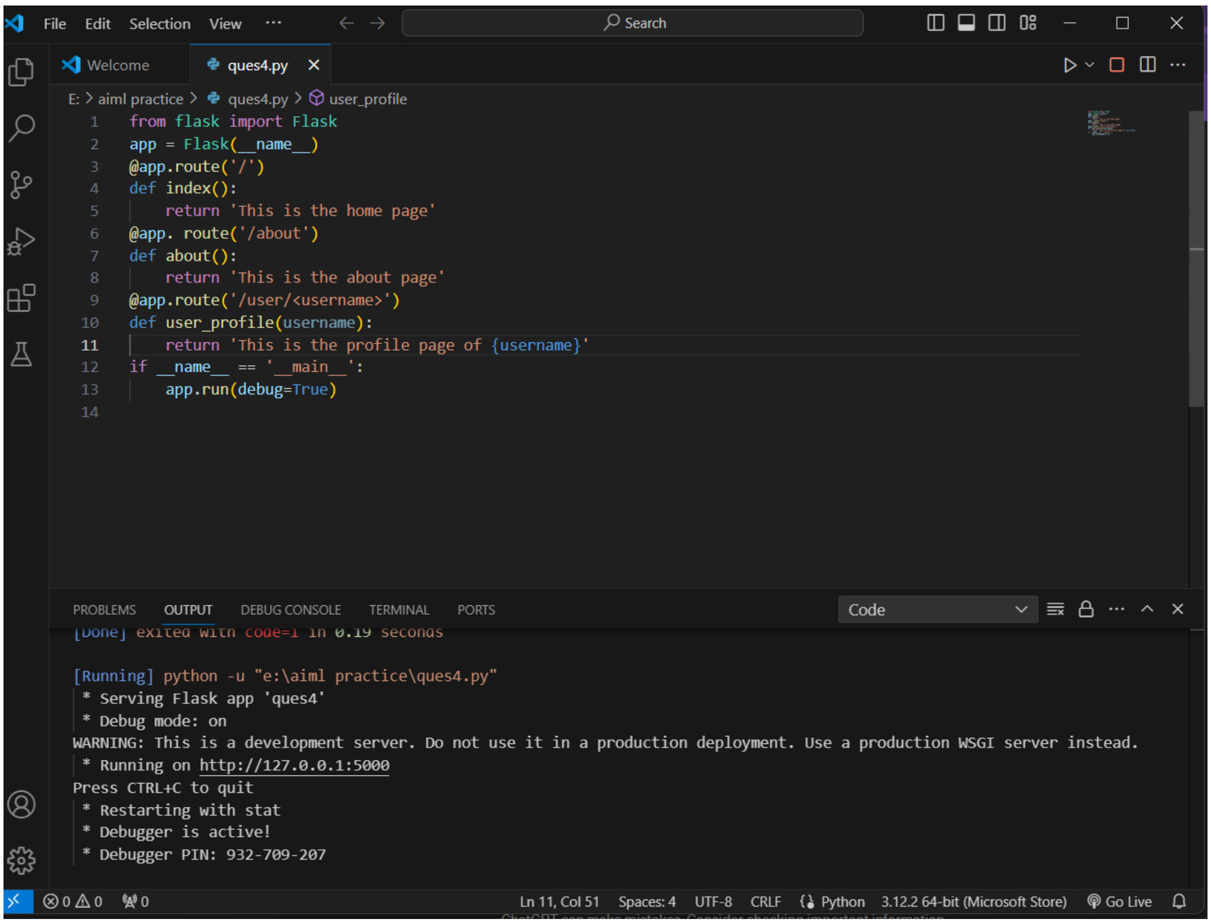
1)Route Decorators: Flask uses route decorators to associate URL patterns with Python functions. These decorators are provided by the @app.route() decorator, where app is an instance of the Flask class.

2)URL Patterns: URL patterns are specified as arguments to the @app.route() decorator. These patterns define the paths that users can visit to access different parts of your application.

3)View Functions: Each route is associated with a view function, which is a Python function that handles requests to that route. When a user visits a URL that matches a route pattern, Flask calls the associated view function.

4)Dynamic URLs: Flask allows you to define dynamic parts within URL patterns using <variable_name>. These dynamic parts can be captured and passed as arguments to the associated view function.

Here's an example to illustrate routing in Flask:


File Edit Selection View
Search
Welcome
ques4.py
E: > aiml practice > ques4.py > user_profile
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Code
Ln 11, Col 51 Spaces: 4 UTF-8 CRLF Python 3.12.2 64-bit (Microsoft Store) Go Live

```python
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
    return 'This is the home page'
@app. route('/about')
def about():
    return 'This is the about page'
@app.route('/user/<username>')
def user_profile(username):
    return 'This is the profile page of {username}'
if __name__ == '__main__':
    app.run(debug=True)
```

```
[Done] exited with code=1 in 0.19 seconds

[Running] python -u "e:\aiml practice\ques4.py"
 * Serving Flask app 'ques4'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 932-709-207
```

In this example:

The @app. route ('/') decorator maps the root URL (http://localhost:5000/) to the index () function.

The @app.route('/about') decorator maps the /about URL (http://localhost:5000/about) to the about() function.

The @app.route('/user/<username>') decorator maps URLs like /user/john, /user/mary, etc., to the user_profile(username) function. The username variable in the URL pattern captures the username from the URL, which is passed as an argument to the user_profile() function.

When a user visits one of these URLs in the browser, Flask dispatches the request to the corresponding view function, which generates a response to be sent back to the client.

This routing mechanism in Flask provides a flexible way to define the structure of your web application and handle incoming requests efficiently.

5. What is a template in Flask, and how is it used to generate dynamic HTML content?

In Flask, a template is an HTML file that contains placeholders for dynamic content. These placeholders are typically filled with data from Python code within Flask views before being rendered and sent to the client's web browser. Flask uses the Jinja2 template engine by default, which provides powerful features for generating HTML dynamically.

Here's how templates are used in Flask to generate dynamic HTML content:

1)Creating Templates: Templates are typically stored in a directory named templates within your Flask project. Each template is an HTML file with the .html extension. You can use any text editor or IDE to create and edit template files.

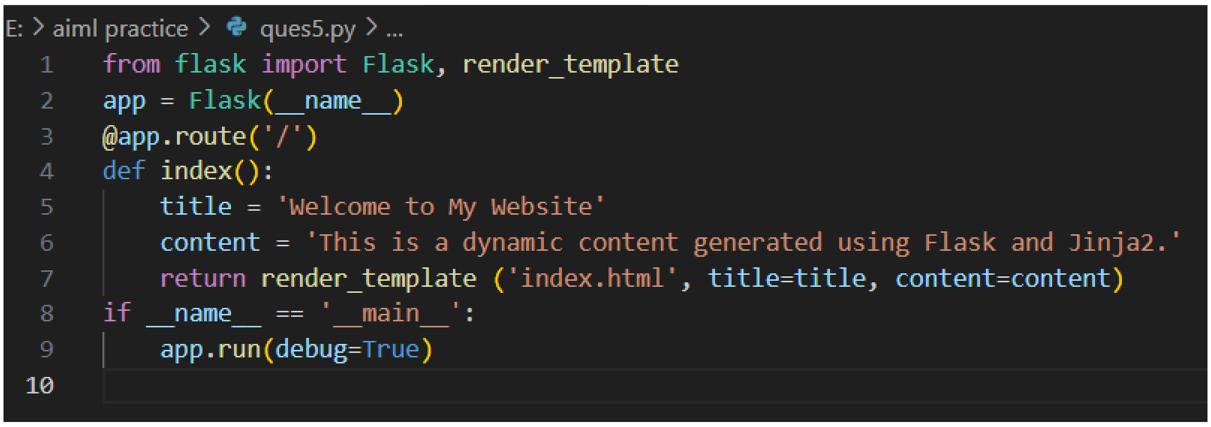
2)Using Jinja2 Syntax: Jinja2 syntax is used within template files to define placeholders and control structures. Placeholders are enclosed within double curly braces ({{ }}) and can contain variables, expressions, or function calls. For example, {{ title }} might be a placeholder for a dynamic title.

3)Passing Data to Templates: In your Flask views, you can pass data to templates by rendering the template with the render_template() function.

This function takes the name of the template file as its first argument and additional keyword arguments representing data to be passed to the template.

4)Rendering Templates: When a view function renders a template using render_template(), Flask processes the template file, replaces placeholders with actual values, and generates complete HTML content. This HTML content is then sent as a response to the client's request.

5)Example Usage:



E: > aiml practice > ques5.py > ...

```python
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def index():
    title = 'Welcome to My Website'
    content = 'This is a dynamic content generated using Flask and Jinja2.'
    return render_template ('index.html', title=title, content=content)
if __name__ == '__main__':
    app.run(debug=True)

```

In this example, the index () view function renders the index.html template and passes two variables (title and content) to the template.

6)Template Inheritance: Jinja2 also supports template inheritance, allowing you to create a base template with common layout elements (like headers, footers, etc.) and extend it in other templates. This promotes code reusability and makes it easier to maintain consistent designs across multiple pages of your website.

By using templates in Flask, you can generate dynamic HTML content efficiently, separate presentation from logic, and build web applications with a clean and maintainable structure.

6.Describe how to pass variables from Flask routes to templates for rendering.

In Flask, passing variables from routes to templates for rendering involves using the render_template() function along with keyword arguments to pass data to the template. Here's how you can pass variables from Flask routes to templates:

Import render_template(): First, make sure to import the render_template() function from the flask module in your Flask application script.

```
from flask import Flask, render_template
```

Define Route: Create a route in your Flask application where you want to render the template. Within the view function associated with this route, define the variables that you want to pass to the template.

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

def index():

    title = 'Welcome to My Website'

    content = 'This is a dynamic content generated using Flask and Jinja2.'

    return render_template('index.html', title=title, content=content)
```

Render Template with Variables: Use the render_template() function to render the template file (e.g., index.html) and pass the variables as keyword arguments. The first argument of render_template() should be the name of the template file, and subsequent keyword arguments are used to pass variables.

```
return render_template('index.html', title=title, content=content)
```

Access Variables in Template: In the template file (e.g., index.html), you can access the variables passed from the route using Jinja2 syntax. Placeholders for variables are enclosed within double curly braces ({{ }}). For example, {{ title }} will display the value of the title variable passed from the route.

```
<!DOCTYPE html>

<html>

<head>

    <title>{{ title }}</title>

</head>

<body>
```

```
    <h1>{{ title }}</h1>
    <p>{{ content }}</p>
</body>
</html>
```

Run Your Flask Application: After defining the route and template, run your Flask application. When you visit the URL associated with the route, Flask will render the template with the variables passed from the route, and the dynamic content will be displayed in the browser.

By following these steps, you can effectively pass variables from Flask routes to templates for rendering dynamic HTML content in your web application.

7.How do you retrieve form data submitted by users in a Flask application?

In Flask, you can retrieve form data submitted by users using the request object, which is part of the flask module. The request object provides access to the data submitted in a form via the POST method. Here's how you can retrieve form data in a Flask application:

Import request: First, make sure to import the request object from the flask module in your Flask application script.

```
from flask import Flask, request
```

Access Form Data in Route: In the route associated with the form submission, use the request.form attribute to access the submitted form data. This attribute provides a dictionary-like object containing the form data.

```
@app.route('/submit', methods=['POST'])
def submit_form():
    username = request.form['username']
    password = request.form['password']
    # Process the form data
    return f'Username: {username}, Password: {password}'
```

In this example, it is assumed that the form submitted by the user contains fields named username and password. The values of these fields are accessed using request.form['fieldname'].

Handle Form Submission: Ensure that the route is configured to accept POST requests by specifying the methods=['POST'] argument in the route decorator.

HTML Form: In your HTML template file, make sure to create a form that submits data using the POST method. The form action should point to the URL associated with the route handling the form submission.

```
<form action="/submit" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username"><br>
    <label for="password">Password:</label>
    <input type="password" id="password" name="password"><br>
    <input type="submit" value="Submit">
</form>
```

Ensure that the name attribute of each input field corresponds to the keys used to access the form data in the Flask route.

Handle Form Data: Within the route function, you can then process the form data as needed, such as validating input, performing database operations, or generating a response.

By following these steps, you can retrieve form data submitted by users in a Flask application and handle it accordingly within your routes.

8.How do you retrieve form data submitted by users in a Flask application?

In Flask, you can retrieve form data submitted by users using the request object, which provides access to data submitted in a form. Here's how you can retrieve form data in a Flask application.

Import request: First, import the request object from the flask module in your Flask application script.

```
from flask import Flask, request
```

Access Form Data in Route: In the route associated with the form submission, use the request.form attribute to access the submitted form data. This attribute provides a dictionary-like object containing the form data.

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

def submit_form():

    username = request.form['username']

    password = request.form['password']

    # Process the form data

    return f'Username: {username}, Password: {password}'
```

In this example, it's assumed that the form submitted by the user contains fields named username and password. The values of these fields are accessed using request.form['fieldname'].

Handle Form Submission: Ensure that the route is configured to accept POST requests by specifying the methods=['POST'] argument in the route decorator.

HTML Form: In your HTML template file, create a form that submits data using the POST method. The form action should point to the URL associated with the route handling the form submission.

```
<form action="/submit" method="post">

    <label for="username">Username:</label>

    <input type="text" id="username" name="username"><br>

    <label for="password">Password:</label>

    <input type="password" id="password" name="password"><br>

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

</form>
```

Ensure that the name attribute of each input field corresponds to the keys used to access the form data in the Flask route.

Handle Form Data: Within the route function, process the form data as needed, such as validating input, performing database operations, or generating a response.

By following these steps, you can retrieve form data submitted by users in a Flask application and handle it accordingly within your routes.

9. Explain the process of fetching values from templates in Flask and performing arithmetic Calculations?

In Flask, you can fetch values from templates by passing them as variables from your routes to your templates using the render_template() function. Once the values are available in the template, you can perform arithmetic calculations using Jinja2 syntax directly within the HTML markup.

- Passing Variables to Templates:

```
@app.route('/')
def index():
        number1 = 10
        number2 = 5
return render_template('index.html',number1=number1,number2=number2)
```

- Accessing Variables in the Template:

```
   <body>

            <h1> Arithmetic Operations</h1>

                  <p>Number 1: {{number1}} </p>

                  <p>Number 1: {{number1}} </p>

                  <p>Sum: {{number1 + number2}} </p>

                  <p>Product: {{number1 * number2}} </p>

                  <p>Difference: {{number1 - number2}} </p>

      </body>
```

10.Discuss some best practices for organizing and structuring a Flask project to maintain scalability and readability?

- Use Blueprint

- Separate Concerns: principle of separation of concerns by separating different aspects of your application, such as routes, views, models, templates, and static files, into distinct directories or modules. For example, create separate directories for routes, templates, static files (CSS, JavaScript, images), and models.

• Use Application Factories: allows you to create multiple instances of your application with different configurations for development, testing, and production environments

• Organize Routes

• Separate Configuration: Store configuration variables such as database connection strings, secret keys, and API keys in separate configuration files (config.py, config_dev.py, config_test.py, config_prod.py) and load the appropriate configuration based on the environment.

• Use Flask Extensions: add additional functionality to your application, such as database integration, authentication, caching, and more

• Use Templates Inheritance: Use Jinja2 template inheritance to create reusable base templates with common layout and structure, and then extend or override specific sections of these templates in child templates. This promotes code reusability and consistency across your application's UI.

• Document Your Code: Good documentation helps other developers understand your codebase and contributes to maintainability.

• Follow PEP 8 Guidelines: Adhere to the PEP 8 style guide for Python code to maintain consistency and readability. Use meaningful variable and function names, follow naming conventions, and organize your code according to best practices.

• Use Version Control: (e.g., Git) to track changes to your Flask project and collaborate with other developers.