Flask

Flask is a web application framework written in Python. A Web Application Framework or a simply a Web Framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on.

Installation

`pip install flask`

Getting Started

`my_app.py:`

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello() -> str:
    return "<h1>Hello World</h1>"

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

1. Import Flask: The first line `from flask import Flask` imports the Flask class from the flask module, which is necessary to use Flask in your application.
2. Create Flask App: The line `app = Flask(__name__)` creates a Flask application object. The `__name__` argument is a special Python variable that represents the name of the current module. This is necessary for Flask to determine the root path of your application.
3. Define Route: The `@app.route("/")` decorator is a Flask decorator that binds the URL route "/" to the `hello()` function defined below it. In other words, when a user accesses the root URL of your application, Flask will call the `hello()` function.
4. Define View Function: The `hello()` function is a view function that returns the string `<h1>Hello World</h1>` as the response body. The `<h1>` tags denote a header HTML

element, so the returned string will be displayed as a heading with the text "Hello World".

5. Run the App: The `if __name__ == "__main__":` block checks if the script is being run directly (as opposed to being imported as a module). If it is being run directly, Flask's built-in development server is started with the `app.run(debug=True)` method, which enables debug mode to show detailed error messages during development.

Adding Routes

On adding the code below to our app (created in the previous section) will create a new ‘about’ page in the route http://127.0.0.1:5000/about

```
@app.route("/about")
def about() -> str:
    return "<h1>About</h1>"
```

We can bind two routes to a same function as follows:

```
@app.route("/about")
@app.route("/introduction")
def about() -> str:
    return "<h1>About</h1>"
```

Templates

Templates increase Modularity and make the code look cleaner. To add templates, create a directory named `templates` in the same directory where your main Python file is located. In the templates directory, create the desired HTML file, for example, `hello.html`.

`hello.html:`

```
<!DOCTYPE html>
<html>

<head>
```

```
    <title>Hello World</title>
</head>

<body>
    <h1>Hello World</h1>
</body>

</html>
```

Now modify the main python file as follows:

`my_app.py:`

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/")
def hello() -> str:
    return render_template("hello.html")

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

In Flask, `render_template` is a function that allows you to render and return an HTML template as the response from a view function. The `render_template` function searches for the template file in the `templates` folder in the same directory as your Flask application.

Template Variables

We can add template variables as an argument to the render_template function, which can then be accessed in the HTML template file.

`my_app.py:`

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route("/")
def hello() -> str:
    return render_template("hello.html", name="Bob")

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

To access the template variable in the HTML template file, we enclose the variable name in double curly braces, like `{{name}}` in the example above. When the template is rendered, Flask replaces the `{{name}}` variable with the value of the `name` argument passed to the `render_template` function.

`hello.html:`

```
<!DOCTYPE html>
<html>

<head>
    <title>Hello</title>
</head>

<body>
    <h1>Hello {{name}}</h1>
</body>

</html>
```

Code Blocks

You can use code blocks to include Python code within your HTML templates. This allows you to dynamically generate content and apply logic using if statements and for loops.

`my_app.py:`

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def index():
    fruits = ['apple', 'banana', 'cherry']
    return render_template('index.html', fruits=fruits)

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

`index.html:`

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

```
    <title>Flask Template Example</title>
</head>
<body>
    <h1>Fruits:</h1>
    <ul>
        {% for fruit in fruits %}
        <li>{{ fruit }}</li>
        {% endfor %}
    </ul>

    <h2>Is banana in the list?</h2>
    {% if 'banana' in fruits %}
    <p>Yes, banana is in the list of fruits.</p>
    {% else %}
    <p>No, banana is not in the list of fruits.</p>
    {% endif %}
</body>
</html>
```

In this template, we're using Flask's built-in templating engine called Jinja2. We're using curly braces `{}` to wrap expressions that are evaluated by Flask. Here are some key points:

- We're using a for loop with `{% for fruit in fruits %}` and `{% endfor %}` to iterate over the list of fruits and generate an HTML list with `<li>` tags.
- We're using an if statement with `{% if 'banana' in fruits %}` and `{% else %}` to check if the word 'banana' is in the list of fruits and display a message accordingly.

Static Files

A web application often requires a static file such as a **javascript** file or a **CSS** file supporting the display of a web page.

Flask provides a way to serve static files such as CSS and JavaScript from a directory called `static` in the root of your application.

`hello.html:`

```
<!DOCTYPE html>
<html>

<head>
    <title>Hello</title>
</head>

<body>
```

```
    <h1 id="hello">Hello {{name}}</h1>

    <script src="../static/script.js"></script>
    <link rel="stylesheet" href="../static/style.css">
</body>

</html>
```

HTML Forms: Review

Text Input

```
<form>
    <label for="fname">First name:</label><br>
    <input type="text" id="fname" name="fname"><br>
    <label for="lname">Last name:</label><br>
    <input type="text" id="lname" name="lname">
</form>
```

1. `<form>`: This is the opening tag for the form element, which is used to create a container for the form contents. The `form` element is used to define a form on a webpage, which can contain various input elements for users to enter data.
2. `<label>`: This is a label element used to provide a text label for an input field. It is associated with the input field using the `for` attribute, which is linked to the `id` attribute of the corresponding input field. The `for` attribute is used to improve accessibility and allows users to click on the label to focus on the associated input field.
3. `<input>`: This is an input element used to create an input field where users can enter data. It has various attributes such as `type`, `id`, and `name` that define the type of input field, an identifier for the input field, and the name of the input field respectively.
4. `type="text"`: This is the `type` attribute of the input field, which specifies that the input field is a text field where users can enter text data. `id` is necessary to correctly identify the input element with the respective label. `name` attribute is the name with which the input response is sent from the form. This is necessary without which the response of that input element will not be sent.

Radio Buttons

Run the code below to see how the radio buttons are implemented in a HTML form!

```
<form>
    <input type="radio" id="html" name="fav_language" value="HTML">
    <label for="html">HTML</label><br>
    <input type="radio" id="css" name="fav_language" value="CSS">
    <label for="css">CSS</label><br>
    <input type="radio" id="javascript" name="fav_language" value="JavaScript">
    <label for="javascript">JavaScript</label>
</form>
```

All radio buttons have the same `name` attribute value. The `name` attribute is used to group radio buttons together so that only one option can be selected from the group.

Checkbox

Run the code below to see how the Checkboxes are implemented in a HTML form!

```
<form>
    <input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
    <label for="vehicle1"> I have a bike</label><br>
    <input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
    <label for="vehicle2"> I have a car</label><br>
    <input type="checkbox" id="vehicle3" name="vehicle3" value="Boat">
    <label for="vehicle3"> I have a boat</label>
</form>
```

Form Attributes

The `action` attribute specifies the URL or endpoint where the form data should be sent after the form is submitted. In this case, the form data will be sent to "**http://localhost:5000/results**", which is a local URL that the form is configured to submit to.

The `method` attribute specifies the HTTP method to be used when submitting the form data. The most commonly used methods are "GET" and "POST". The details about these two methods are given in the following sections.

```
<form action = "http://localhost:5000/data" method = "POST">
        <!-- form elements -->
```

```
</form>
```

Handling GET Requests

What is a GET Request?

A GET request is a type of HTTP (Hypertext Transfer Protocol) request made by a client (such as a web browser) to a server in order to retrieve information or data from the server. The server responds to a GET request by providing the requested data or resource as the response body, typically in the form of HTML, XML, JSON, or other formats that can be interpreted by the client.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

@app.route('/data', methods=['GET'])
def get_data():
    data = {'name': 'John', 'age': 30, 'city': 'New York'}
    return data

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

The `methods` argument in the `@app.route()` decorator allows you to specify the allowed HTTP methods for a particular route. In this example, we specified `methods=['GET']` to only allow GET requests for the "/data" route. You can specify multiple methods by providing them as a list, such as `methods=['GET', 'POST']`.

Handling POST Requests

What is a POST Request?

A POST request is a type of HTTP (Hypertext Transfer Protocol) request made by a client (such as a web browser) to a server in order to submit data or information to the server for further processing. In other words, it is a request made by a client to a server to accept and store data sent in the request body.

```
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

@app.route('/data', methods=['POST'])
def handle_post_request():
    data = request.json
    return jsonify(data)

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

The route, `'/data'`, is mapped to the `handle_post_request()` function, which is configured to only allow 'POST' requests using the `methods=['POST']` argument in the `@app.route()` decorator. This function takes the JSON data from the request body using `request.json` and returns the same data as a JSON response using `jsonify(data)`. This is an example of a route that handles 'POST' requests and returns a JSON response.

Testing API Calls

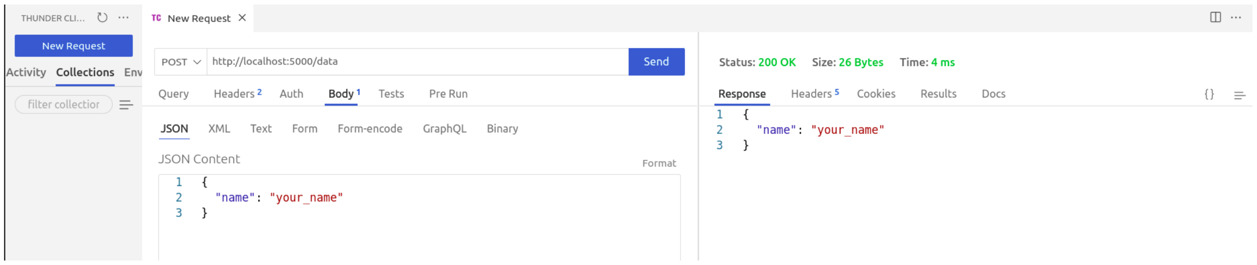
What is API?

API stands for application programming interface, which is a set of definitions and protocols for building and integrating application software. API calls typically involve sending a request from one application (the client) to another application (the server) over the internet using HTTP (Hypertext Transfer Protocol) or another communication protocol.

From the previous sections, the POST and the GET requests made to the `/data` route are the examples of API calls that we make to the Flask server.

Tools like `Postman` or simple VS code extension such as `Thunder Client` can be used to test API calls to the Flask server. Postman is a popular tool used for testing APIs and it allows users to send requests to a server and view the responses in a user-friendly interface.

After installing the VS code extension Thunder Client, we can test our `handle_post_request()` method by making an API call:



POST http://localhost:5000/data Send

JSON

JSON Content

```
{
  "name": "your_name"
}
```

Status: 200 OK Size: 26 Bytes Time: 4 ms

```
{
  "name": "your_name"
}
```

Resources

- https://www.youtube.com/watch?v=MwZwr5Tvyxo&list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH&index=1
- https://www.tutorialspoint.com/flask/index.htm