Flask Framework Complete Report

Overview of Flask Framework

Definition: Flask is a complete web framework created using the Python programming language. It is primarily used for developing end-to-end web applications.

Purpose: Flask is particularly useful for data scientists and machine learning engineers who need to showcase their models through web applications.

Key Components of Flask

WSGI (Web Server Gateway Interface)

Definition: WSGI is a protocol that facilitates communication between the web server and the web application.

Functionality:

1. When a user sends a request (e.g., accessing a homepage), the request is received by the web server.

2. The web server uses WSGI to redirect the request to the Flask web application.

3. The web application processes the request and sends a response back to the web server, which then delivers it to the user.

Diagram: WSGI Communication Flow

```
1. WSGI->
User       Request
    |
    v
+-----------+
| Web Server|
+-----------+
    |
    v
+-----------+
|   WSGI   |
+-----------+
    |
    v
+-----------+
| Flask App |
+-----------+
    |
    v
User       Response
```

Jinja2 Template Engine

Definition: Jinja2 is a web template engine that combines web templates with data sources to create dynamic web pages.

Functionality:

1. Jinja2 allows developers to create templates that can be populated with data from various sources (e.g., SQL databases, CSV files, machine learning models).

2. This enables the creation of dynamic web pages that can change based on user input or data retrieved from a data source.

Diagram: Jinja2 Template Engine Flow

```
2. JINJA2->
+-----------+
| Web Page  |
| (Template)|
+-----------+
     |
     v
+-----------+
| Data Source|
| (e.g., DB) |
+-----------+
     |
     v
+-----------+
| Jinja2    |
| Template   |
+-----------+
     |
     v
Dynamic Web Page
```

Practical Applications

Flask is essential for creating web applications that interact with machine learning models. For example:

- Image Classification: A web page with an upload button allows users to upload images, which are then processed by a machine learning model to classify the image (e.g., dog or cat).

- User Authentication: A login form that authenticates users against a database.

Importing Flask and Creating an Application Instance

To use Flask, you first need to import it and create an instance of the Flask class:

```
from flask import Flask, render_template

app = Flask(__name__)
```

This code imports the Flask class and the render_template function, which is used to render HTML templates. An instance of the Flask class is created, which will be your WSGI application.

Defining Routes

You can define routes using decorators. Here are examples of defining routes for the home page and an index page:

```
@app.route('/')
def welcome():
    return '<html><H1>Welcome to the best flask course</H1></html>'

@app.route('/index')
def idx():
    return render_template('index.html')
```

The @app.route('/') decorator defines the home page route, which returns a simple HTML welcome message. The @app.route('/index') decorator defines a route for the index page, which renders an HTML template named 'index.html'.

About Page Route

You can also define additional routes for other pages. Here is an example of an about page route:

```
@app.route('/about')
def about():
    return render_template('about.html')
```

This code defines a route for the about page, which renders an HTML template named 'about.html'.

Running the Application

Finally, you can run the application with the following code:

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

This condition checks if the script is being run directly. If true, the app.run() method starts the Flask application. The debug=True argument enables debug mode, which provides detailed error messages and automatically reloads the server when code changes are made.

Summary of Key Functions

Flask: The main class used to create a Flask application instance.

render_template: A function used to render HTML templates.

@app.route(): A decorator used to bind a URL to a function, defining the routes of the application.

def function_name(): Defines a function that will be executed when the associated route is accessed.

return: Sends a response back to the client (web browser) when a route is accessed.

app.run(): Starts the Flask application server.

Conclusion

Flask is a powerful framework for building web applications, especially for those in data science and machine learning. Understanding WSGI and Jinja2 is crucial for effectively using Flask to create dynamic and interactive web applications. This report covers the basics of Flask web development, including how to create an application instance, define routes, and render HTML templates. Each function and decorator plays a crucial role in handling web requests and generating responses.

Flask Web Development

Setting Up Flask

To set up a Flask application, you need to install Flask first. You can do this using pip:

```
pip install Flask
```

This command installs Flask and its dependencies.

Creating a Simple Flask App

Here is a simple Flask application that returns 'Hello, World!':

```
from flask import Flask

app = Flask(__name__)

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

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

In this code, we create a Flask application and define a route that returns 'Hello, World!' when accessed.

Running the Application

To run the application, save the code in a file named 'app.py' and execute it:

```
python app.py
```

This command starts the Flask development server, and you can access the app at http://999.9.9.9:9999.

Understanding Routes

In Flask, routes are defined using the @app.route decorator. This allows you to map URLs to Python functions. For example:

```
@app.route('/about')
def about():
    return render_template('about.html')
```

This code defines a route for the '/about' URL, which renders an 'about.html' template.

GET and POST Methods

In web development, GET and POST are two common HTTP methods used to send and receive data between a client and a server.

GET Method:

1. The GET method is used to request data from a specified resource.

2. Data sent using the GET method is appended to the URL as query parameters.

3. It is generally used for retrieving data and should not have side effects (i.e., it should not change the state of the server).

4. Example: Accessing a URL like 'http://example.com/page?name=John' uses the GET method.

POST Method:

1. The POST method is used to send data to a server to create or update a resource.

2. Data sent using the POST method is included in the body of the request, not in the URL.

3. It is generally used for submitting forms and can change the state of the server.

4. Example: Submitting a form with user data uses the POST method.

Differences between GET and POST:

1. Data Transmission: GET appends data to the URL, while POST sends data in the request body.

2. Data Length: GET has limitations on the amount of data that can be sent (URL length), while POST can send larger amounts of data.

3. Security: GET is less secure as data is visible in the URL, while POST is more secure as data is not exposed in the URL.

4. Use Cases: GET is used for retrieving data, while POST is used for submitting data.

Complete Flask Application Example

Here is the complete code for a simple Flask application:

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

```
app = Flask(__name__)

@app.route("/")
def welcome():
    return "<html><h1>Welcome to the Flask Course</h1></html>"

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

@app.route('/about')
def about():
    return render_template('about.html')

@app.route('/form', methods=['GET', 'POST'])
def form():
    if request.method == 'POST':
        name = request.form['name']
        return f"Hello {name}!"
    return render_template('form.html')

@app.route('/submit', methods=['GET', 'POST'])
def submit():
    if request.method == 'POST':
        name = request.form['name']
        return f"Hello {name}!"
    return render_template('form.html')

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

This code creates a Flask application with multiple routes, including a welcome page, an index page, an about page, and a form page that accepts user input.

HTML Templates

The following HTML templates are used in the Flask application:

index.html

```
<html lang="en"><head>
    <meta charset="UTF-8">
    <title>Flask App</title>
</head>
<body>
    <h1>Welcome to My Flask App!</h1>
    <p>This is a simple web application built with Flask.</p>
</body></html>
```

about.html

```
<html lang="en"><head>
    <meta charset="UTF-8">
    <title>About</title>
</head>
<body>
    <h1>About</h1>
    <p>This is the about page of my Flask app.</p>


</body></html>
```

form.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Form</title>
</head>
<body>
    <h1>Submit a Form</h1>
    <form action="/submit" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Flask Advanced Concepts Report

Introduction

This report covers advanced concepts in Flask, including dynamic URL building, variable rules, and the Jinja2 template engine. We will explore how to create a Flask application that utilizes these features effectively.

Building URL Dynamically

In Flask, you can build URLs dynamically using the `url_for` function. This function generates a URL to the specified function based on the function name and any parameters you provide.

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

app = Flask(__name__)

@app.route('/submit', methods=['POST', 'GET'])
def submit():
    total_score = 0
    if request.method == 'POST':
        science = float(request.form['science'])
        maths = float(request.form['maths'])
        c = float(request.form['c'])
        data_science = float(request.form['datascience'])

        total_score = (science + maths + c + data_science) / 4
    else:
        return render_template('getresult.html')

    return redirect(url_for('successres', score=total_score))
```

Explanation:

1. The `submit` function handles both GET and POST requests. When the form is submitted (POST), it calculates the average score from the input fields.

2. If the request method is GET, it renders a template named 'getresult.html'.

3. After calculating the total score, it redirects to the 'successres' route using `url_for`, passing the total score as a parameter.

Variable Rule

Flask allows you to define dynamic routes using variable rules. You can capture values from the URL and pass them to your view functions.

```
@app.route('/success/<int:score>')
def success(score):
    res = ''
    if score >= 50:
        res = "PASSED"
    else:
        res = "FAILED"

    return render_template('result.html', results=res)
```

Explanation:

1. The `success` function is defined with a variable rule that captures an integer score from the URL.

2. It checks if the score is greater than or equal to 50 and sets the result to 'PASSED' or 'FAILED'.

3. Finally, it renders a template named 'result.html', passing the result as a variable.

Jinja2 Template Engine

Jinja2 is a powerful template engine for Python that allows you to create dynamic HTML pages. It uses the following syntax:

Jinja2 Syntax:

1. {{ ... }}: Used to print output in HTML.

2. {% ... %}: Used for control structures like conditions and loops.

3. {# ... #}: Used for comments.

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/about')
def about():
    return render_template('about.html')
```

Explanation:

1. The `about` function renders a template named 'about.html'.

2. Jinja2 allows you to include dynamic content in your HTML templates, making it easy to create interactive web pages.

Complete Flask Application Example

Here is the complete code for a Flask application that demonstrates dynamic URL building and Jinja2 templates:

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

app = Flask(__name__)

@app.route("/")
def welcome():
    return "<html><H1>Welcome to the flask course</H1></html>"

@app.route("/index", methods=['GET'])
def index():
    return render_template('index.html')

@app.route('/about')
def about():
    return render_template('about.html')

@app.route('/success/<int:score>')
def success(score):
    res = ''
    if score >= 50:
        res = "PASSED"
    else:
        res = "FAILED"

    return render_template('result.html', results=res)

@app.route('/successres/<int:score>')
def successres(score):
    res = ''
    if score >= 50:
        res = "PASSED"
    else:
        res = "FAILED"

    exp = {'score': score, 'res': res}

    return render_template('result1.html', results=exp)

@app.route('/submit', methods=['POST', 'GET'])
def submit():
    total_score = 0
    if request.method == 'POST':
        science = float(request.form['science'])
        maths = float(request.form['maths'])
        c = float(request.form['c'])
        data_science = float(request.form['datascience'])

        total_score = (science + maths + c + data_science) / 4
    else:
        return render_template('getresult.html')

    return redirect(url_for('successres', score=total_score))

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

Explanation:

1. The application starts by importing necessary modules from Flask.

2. The `welcome` function returns a simple HTML welcome message.

3. The `index` function renders the 'index.html' template.

4. The `about` function renders the 'about.html' template.

5. The `success` and `successres` functions handle dynamic routes and render results based on the score.

6. The `submit` function processes form data, calculates the average score, and redirects to the appropriate result page.

HTML Templates

The following HTML templates are used in the Flask application:

index.html

```
<html lang="en"><head>
    <meta charset="UTF-8">
    <title>Flask App</title>
```

```
</head>
<body>
    <h1>Welcome to My Flask App!</h1>
    <p>This is a simple web application built with Flask.</p>
</body></html>
```

about.html

```
<html lang="en"><head>
    <meta charset="UTF-8">
    <title>About</title>
</head>
<body>
    <h1>About</h1>
    <p>This is the about page of my Flask app.</p>


</body></html>
```

result.html

```
<h1>
    Based on the marks, You have {{ results }}

    {% if results>=50 %}
    <h1>You have passed with marks {{ results }}</h1>
    {% else %}
    <h2>You have failed with marks {{ results }}</h2>
    {% endif %}
</h1>
```

result1.html

```
<html>
    <h2>Final Results</h2>
    <body>

        {% for key,value in results.items() %}
        {#This is the comment section #}
        <h1>{{ key }}</h1>
        <h2>{{ value }}</h2>

        {% endfor %}
    </body>
</html>
```

getresult.html

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

```
<body>

    <h2>HTML FORMS</h2>

    <form action="/submit" method="post">
        <label for="Science">Science:</label><br>
        <input type="text" name="science" id="science" value="0"><br><br>
        <label for="Maths">Maths:</label><br>
        <input type="text" name="maths" id="maths" value="0"><br><br>
        <label for="C ">C:</label><br>
        <input type="text" id="c" name="c" value="0"><br><br>
        <label for="datascience">Data Science:</label><br>
        <input type="text" id="datascience" name="datascience" value="0"><br><br>
        <input type="submit" value="Submit">
    </form>

    <p>If you click the "Submit" button, the form-data will be sent to a page called "/su
</body>
</html>
```

Flask To-Do List API Report

1. Overview of Flask Framework

Flask is a micro web framework written in Python. It is designed to make it easy to build web applications quickly and with minimal code.

Flask is particularly useful for data scientists and machine learning engineers who need to showcase their models through web applications.

Flask is lightweight and modular, allowing developers to choose the components they need for their applications.

2. Key Concepts

- **Flask**: A micro web framework for Python that allows for the creation of web applications.

- **HTTP Methods**: The API utilizes various HTTP methods to interact with resources:

- **GET**: Retrieve data from the server.

- **POST**: Send data to the server to create a new resource.

- **PUT**: Update an existing resource on the server.

- **DELETE**: Remove a resource from the server.

- **JSON**: The API communicates using JSON (JavaScript Object Notation), a lightweight data interchange format.

3. Code Explanation

3.1 Complete Code

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

# Initial Data in my to do list
items = [
    {"id": 1, "name": "Item 1", "description": "This is item 1"},
    {"id": 2, "name": "Item 2", "description": "This is item 2"}
]

@app.route('/')
def home():
    return 'Welcome To The Sample TO-DO List App'

# GET: Retrieve all the items
@app.route('/items', methods=['GET'])
def get_items():
    return jsonify(items)

# GET: Retrieve a specific item by id
@app.route('/items/<int:item_id>', methods=['GET'])
def get_item(item_id):
    item = next((item for item in items if item['id'] == item_id), None)
    if item is None:
        return jsonify({"error": "item not found"})
    return jsonify(item)

# POST: Create a new task - API
@app.route('/items', methods=['POST'])
def create_item():
    if not request.json or not 'name' in request.json:
        return jsonify({"error": "item not found"})
    new_item = {
        "id": items[-1]['id'] + 1 if items else 1,
        "name": request.json['name'],
        'description': request.json['description']
    }
    items.append(new_item)
    return jsonify(new_item)

# PUT: Update an existing item
@app.route('/items/<int:item_id>', methods=['PUT'])
def update_item(item_id):
    item = next((item for item in items if item['id'] == item_id), None)
    if item is None:
        return jsonify({"error": "Item not found"})
    item['name'] = request.json.get('name', item['name'])
    item['description'] = request.json.get('description', item['description'])
    return jsonify(item)

# DELETE: Delete an item
@app.route('/items/<int:item_id>', methods=['DELETE'])
def delete_item(item_id):
    global items
    items = [item for item in items if item['id'] != item_id]
    return jsonify({"result": "Item deleted"})

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

3.2 Explanation of the Code

This code implements a simple To-Do List API using Flask. It allows users to manage tasks with the following functionalities:
- **Home Route**: Returns a welcome message.
- **GET /items**: Retrieves all items in the to-do list.
- **GET /items/**: Retrieves a specific item by its ID.
- **POST /items**: Creates a new task. The request must include a JSON body with a 'name' and optionally a 'description'.
- **PUT /items/**: Updates an existing item based on its ID.
- **DELETE /items/**: Deletes an item based on its ID.

The API uses a list of dictionaries to store the tasks, where each task has an ID, name, and description. The ID is automatically incremented when a new task is created.

4. Testing the API

You can test the API using tools like Postman or curl. Here are some example requests:
- **GET all items**: `GET http://999.9.9.9:9999/items`
- **GET item by ID**: `GET http://999.9.9.9:9999/items/1`
- **POST new item**: `POST http://999.9.9.9:9999/items` with JSON body `{ "name": "Item 3", "description": "This is item 3" }`
- **PUT update item**: `PUT http://999.9.9.9:9999/items/1` with JSON body `{ "name": "Updated Item 1", "description": "Updated description" }`
- **DELETE item**: `DELETE http://999.9.9.9:9999/items/1`

5. Conclusion

This Flask To-Do List API demonstrates how to create a simple RESTful API that allows users to manage tasks. The use of HTTP methods and JSON for data interchange makes it a practical example of web service development. Flask's simplicity and flexibility make it an excellent choice for building web applications.