



Assignment Code: DA-AG-004

Restful API & Flask | Assignment

Instructions: Carefully read each question. Use Google Docs, Microsoft Word, or a similar tool to create a document where you type out each question along with its answer. Save the document as a PDF, and then upload it to the LMS. Please do not zip or archive the files before uploading them. Each question carries 20 marks.

Total Marks: 200

Question 1 : What is a RESTful API?

Answer:

A RESTful API (Representational State Transfer Application Programming Interface) is a web service that allows communication between different systems over the internet using standard HTTP methods such as GET, POST, PUT, and DELETE.

It follows the principles of REST architecture, where each piece of data or functionality is treated as a resource and can be accessed through a specific URL (endpoint). The data is usually exchanged in JSON or XML format.

RESTful APIs are stateless, meaning each request from the client to the server must contain all the information needed to understand and process it, and the server does not store the client's session data.

Question 2: What is Flask, and why is it popular for building APIs?

Answer:

Flask is a micro web framework written in Python that is used to create web applications and APIs. It is called a “micro” framework because it provides the basic tools needed to build a web app but lets developers add extra features as required through extensions.

Why Flask is popular for building APIs:

Lightweight and Simple:

Flask is easy to understand and doesn't have unnecessary features, making it ideal for small to medium-sized projects.

Flexible:

Developers can choose their own tools and libraries according to their project needs.

Easy to Build RESTful APIs:

Flask allows quick creation of API routes and supports HTTP methods like GET, POST, PUT, and DELETE easily.

Extensible:

Many extensions are available for adding database support, authentication, and more.

Built-in Development Server and Debugger:

It helps in quick testing and debugging during development.

Question 3:What are HTTP methods used in RESTful APIs?

Answer:

In RESTful APIs, HTTP methods are used to perform different operations on resources (like data or objects).

Each method defines a specific type of action.

Here are the main HTTP methods used in RESTful APIs:

1. GET

- Purpose: To retrieve or read data from the server.
- Example:
GET /users → Fetch all users.
GET /users/5 → Fetch the user with ID 5.

2. POST

- Purpose: To create a new resource on the server.
- Example:
POST /users → Add a new user.

3. PUT

- Purpose: To update or replace an existing resource completely.
- Example:
PUT /users/5 → Replace the data of user with ID 5.

4. PATCH

- Purpose: To partially update an existing resource.
- Example:
PATCH /users/5 → Update only specific fields of user 5.

5. DELETE

- Purpose: To remove a resource from the server.
- Example:
DELETE /users/5 → Delete user with ID 5.

In summary:

HTTP Method	Action	Description
GET	Read	Fetch data from the server
POST	Create	Add new data
PUT	Update	Replace existing data
PATCH	Partial Update	Modify part of the data
DELETE	Delete	Remove data

Question 4: What is the purpose of the @app.route() decorator in Flask?

Answer:

In Flask, the @app.route() decorator is used to define the URL (route) that will trigger a specific function in the application.

It connects a web address (URL) to a Python function, so when a user visits that URL, the function runs and returns a response. This is how Flask knows which code to execute for each page or API endpoint.

Example:

```
@app.route('/home')
def home():
    return "Welcome to the Home Page"
```

Question 5: What is the role of Flask-SQLAlchemy?**Answer:**

Flask-SQLAlchemy is an extension of Flask that helps developers work with databases more easily using Python objects instead of writing complex SQL queries directly.

Role of Flask-SQLAlchemy:**1. Database Integration:**

It connects Flask applications to databases like MySQL, PostgreSQL, or SQLite.

2. Object Relational Mapping (ORM):

It allows developers to interact with the database using Python classes and objects instead of raw SQL commands.

3. Simplifies Database Operations:

Tasks like adding, updating, deleting, or retrieving data become much simpler.

4. Schema Definition:

You can define database tables as Python classes, where each class represents a table and each attribute represents a column.

5. Example:

```
6. from flask_sqlalchemy import SQLAlchemy
7. db = SQLAlchemy()
8.
9. class User(db.Model):
10.     id = db.Column(db.Integer, primary_key=True)
11.     name = db.Column(db.String(50))
```

Question 6: How do you create a basic Flask application?**Answer:**

To create a basic Flask application, follow these steps:

1. Install Flask using pip:

```
pip install Flask
```

2. Create a Python file (e.g., app.py) and write the following code:

```
from flask import Flask
```

```
# Create a Flask application instance
app = Flask(__name__)
```

```
# Define a route and its function
@app.route('/')
def home():
    return "Hello, World!"
```

```
# Run the application
if __name__ == '__main__':
    app.run(debug=True)
```

3. Run the application in the terminal:

```
python app.py
```

- This starts a local server (usually at <http://127.0.0.1:5000/>).
- Visiting this URL in a browser will display “Hello, World!”.
-

A basic Flask application involves installing Flask, creating an app instance, defining routes, and running the server.

Question 7: How do you return JSON responses in Flask?

Answer:

Flask, you can return JSON responses using the `jsonify()` function, which converts Python dictionaries or lists into JSON format and sets the correct Content-Type header.

Example:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/user')
def user():
    data = {
        "id": 1,
        "name": "Atish",
        "email": "atish@example.com"
    }
    return jsonify(data)

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

Visiting /user will return:

```
{
    "id": 1,
    "name": "Atish",
    "email": "atish@example.com"
}
```

Key Points:

1. `jsonify()` automatically converts Python objects to JSON.
2. It sets the Content-Type to `application/json`, which is required for APIs.
3. You can return lists, dictionaries, or nested structures.

Question 8: How do you handle POST requests in Flask?

Answer

In Flask, you can handle POST requests by specifying the methods parameter in the `@app.route()` decorator and then accessing the data sent by the client using `request.form` (for form data) or `request.get_json()` (for JSON data).

Example with JSON data:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add_user', methods=['POST'])
def add_user():
    # Get JSON data from the request
    data = request.get_json()

    name = data.get('name')
    email = data.get('email')

    # Here you can process or save the data
    return jsonify({"message": f"User {name} with email {email} added successfully!"})

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

Key Points:

1. Specify POST method:

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

2. Access request data:

- For JSON: `request.get_json()`
- For form data: `request.form`

3. Return a response: Usually as JSON using `jsonify()` for APIs.

To handle POST requests in Flask, set `methods=['POST']` in the route and use `request.get_json()` or `request.form` to access the data sent by the client.

Question 9: How do you handle errors in Flask (e.g., 404)?

Answer:

In Flask, errors like 404 Not Found or 500 Internal Server Error can be handled using the `@app.errorhandler()` decorator. This allows the application to return custom responses when an error occurs.

Example:

Handling 404 Error

```
from flask import Flask, jsonify
```

```
app = Flask(__name__)
```

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

```
def home():
```

```
    return "Welcome to the Home Page!"
```

```
# Handle 404 errors
```

```
@app.errorhandler(404)
```

```
def page_not_found(e):
```

```
    return jsonify({"error": "Page not found", "status": 404}), 404
```

```
if __name__ == '__main__':
```

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

- If a user visits a non-existent URL, the app will return:

```
{
```

```
    "error": "Page not found",
```

```
    "status": 404
```

```
}
```

The `@app.errorhandler()` decorator in Flask is used to handle HTTP errors and provide custom responses for them.

Question 10: How do you structure a Flask app using Blueprints?

In Flask, Blueprints allow you to organize your application into modular components instead of having all routes and logic in a single file. This is especially useful for large applications.

Steps to structure a Flask app using Blueprints:

1. Create the project structure

```
myapp/
|
+-- app.py
+-- run.py
+-- /auth
|   +-- __init__.py
|   +-- routes.py
+-- /blog
    +-- __init__.py
    +-- routes.py
```

2. Define a Blueprint

auth/routes.py

```
from flask import Blueprint, jsonify

auth = Blueprint('auth', __name__)

@auth.route('/login')
def login():
    return jsonify({"message": "Login Page"})
```

blog/routes.py

```
from flask import Blueprint, jsonify

blog = Blueprint('blog', __name__)

@blog.route('/posts')
def posts():
    return jsonify({"message": "Blog Posts"})
```

3. Register Blueprints in the main app

app.py

```
from flask import Flask
from auth.routes import auth
from blog.routes import blog
```

- The /auth/login route is handled by the auth Blueprint.
- The /blog/posts route is handled by the blog Blueprint.
- Blueprints let you group related routes and logic, making the app modular and maintainable.

Blueprints in Flask allow you to divide your app into reusable components, register them with the main app, and keep routes, views, and logic organized.