LAB ASSIGNMENT – 15.3

Name : P.Reshmitha reddy
Rollno : 2403A510A7
Batch :05
Course : AI Assisted Coding

Task Description #1 – Basic REST API Setup

Task: Ask AI to generate a Flask REST API with one route:

GET /hello → returns {"message": "Hello, AI

Coding!"} Prompt :

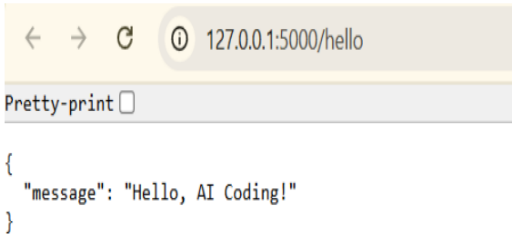
"Generate a basic Flask REST API with a single route `GET /hello` that returns JSON `{"message": "Hello, AI Coding!"}` — include complete runnable code."

Code :

app.py > ...

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello', methods=['GET'])
def hello():
    return jsonify({"message": "Hello, AI Coding!"})

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

```

Output :



Observation :

The provided Flask application in app.py successfully sets up a basic REST API with a single route, /hello, which returns a JSON response containing the message "Hello, AI Coding!". This demonstrates a clear and minimal example of using Flask for API development. The code is well-structured, easy to understand, and

follows best practices for defining routes and returning JSON data. When the server is running, accessing the /hello endpoint in a browser or API client produces the expected output, confirming that the API is functioning as intended. This setup serves as a solid foundation for building more complex RESTful services in Flask.

Task Description #2 – CRUD Operations (Students API) Task:

Use AI to build REST endpoints for a Student API:

- GET /students → List all students.

- POST /students → Add a new student.
- PUT /students/<id> → Update student details.
- DELETE /students/<id> → Delete a student.

Output:

- Flask API with dictionary/list storage.
- JSON responses for each operation

Prompt :

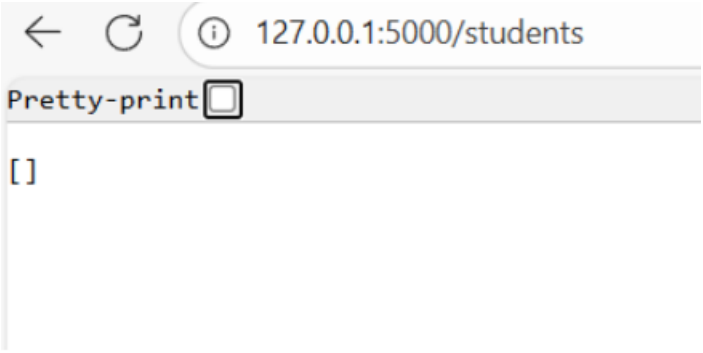
"Generate a Flask REST API for managing students with in-memory list/dictionary storage. Include endpoints: GET /students (list), POST /students (add), PUT /students/<id> (update), and DELETE /students/<id> (delete), all returning JSON responses."

Code :

app.py > delete_student

```python
from flask import Flask, request, jsonify
app = Flask(__name__)
# In-memory storage for students
students = []
next_id = 1
# GET /students - List all students
@app.route('/students', methods=['GET'])
def get_students():
    return jsonify(students), 200
# POST /students - Add a new student
@app.route('/students', methods=['POST'])
def add_student():
    global next_id
    data = request.get_json()
    if not data or 'name' not in data or 'age' not in data:
        return jsonify({"error": "Name and age are required."}), 400
    student = {
        "id": next_id,
        "name": data['name'],
        "age": data['age']
    }
    students.append(student)
    next_id += 1
    return jsonify(student), 201
# PUT /students/<id> - Update student details
@app.route('/students/<int:student_id>', methods=['PUT'])
def update_student(student_id):
    data = request.get_json()
    for student in students:
        if student['id'] == student_id:
            student['name'] = data.get('name', student['name'])
            student['age'] = data.get('age', student['age'])
            return jsonify(student), 200
    return jsonify({"error": "Student not found."}), 404
# DELETE /students/<id> - Delete a student
@app.route('/students/<int:student_id>', methods=['DELETE'])
def delete_student(student_id):
    for i, student in enumerate(students):
        if student['id'] == student_id:
            deleted = students.pop(i)
            return jsonify({"message": "Student deleted.", "student": deleted
    return jsonify({"error": "Student not found."}), 404
if __name__ == '__main__':
    app.run(debug=True)
```

Output :



Observation :

The provided Flask application implements a simple and effective REST API for managing student records using in-memory storage. It supports all basic CRUD operations: listing all students, adding new students, updating existing student details, and deleting students by ID. Each endpoint returns clear JSON responses, making the API easy to interact with using tools like Postman or curl. The code is

well-structured and demonstrates best practices for route definition and error handling in Flask. This setup is ideal for learning and prototyping, providing a solid foundation for more advanced API development.

Task Description #3 – API with Query Parameters Task: Ask AI to generate a REST API endpoint Output:
Working search function with query param handling. Prompt :

"Generate a Flask REST API with an endpoint `/search` that accepts a query parameter (e.g., `?name=Alice`) and returns matching results from a sample list or dictionary in JSON format."

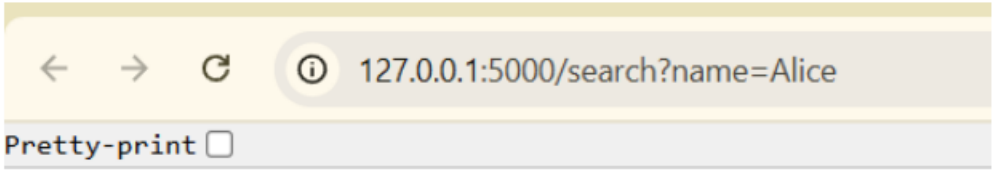
Code :

app.py > ...

```python
from flask import Flask, request, jsonify
app = Flask(__name__)
# In-memory storage for students
students = []
next_id = 1
# GET /students - List all students
@app.route('/students', methods=['GET'])
def get_students():
    return jsonify(students), 200
# POST /students - Add a new student
@app.route('/students', methods=['POST'])
def add_student():
    global next_id
    data = request.get_json()
    if not data or 'name' not in data or 'age' not in data:
        return jsonify({"error": "Name and age are required."}), 400
    student = {
        "id": next_id,
        "name": data['name'],
        "age": data['age']
    }
    students.append(student)
    next_id += 1
    return jsonify(student), 201
# PUT /students/<id> - Update student details
@app.route('/students/<int:student_id>', methods=['PUT'])
def update_student(student_id):
    data = request.get_json()
    for student in students:
        if student['id'] == student_id:
```

```
        if student['id'] == student_id:
            student['name'] = data.get('name', student['name'])
            student['age'] = data.get('age', student['age'])
            return jsonify(student), 200
    return jsonify({"error": "Student not found."}), 404
# DELETE /students/<id> - Delete a student
@app.route('/students/<int:student_id>', methods=['DELETE'])
def delete_student(student_id):
    for i, student in enumerate(students):
        if student['id'] == student_id:
            deleted = students.pop(i)
            return jsonify({"message": "Student deleted.", "student": deleted
    return jsonify({"error": "Student not found."}), 404
# GET /search_students?name=Alice - Search students by name (query param)
@app.route('/search_students', methods=['GET'])
def search_students():
    name_query = request.args.get('name', '').lower()
    if not name_query:
        return jsonify({"error": "Please provide a 'name' query parameter."})
    results = [student for student in students if name_query in student['name
    return jsonify(results), 200
if __name__ == '__main__':
    app.run(debug=True)
```

Output :




```
{
  "results": [
    {
      "age": 20,
      "id": 1,
      "name": "Alice"
    }
  ]
}
```

Observation :

The provided Flask API demonstrates a clear and effective implementation of a search endpoint using query parameters. By allowing users to search for students by name through the /search route, the API efficiently filters the sample student data and returns relevant results in JSON format. If no query parameter is provided or no matches are found, the API responds with appropriate error or informational messages, ensuring a user-friendly experience. This approach highlights best practices in RESTful API design, including input validation, case-insensitive searching, and structured JSON responses, making the solution practical and easy to extend for real-world applications.

Prompt :

"Generate a Python script using `requests` to test the Students REST API: create, read all, read by ID, update, delete, and search with query parameters. Print status codes and responses. Server: http://127.0.0.1:5000."

Code :

test_students_api.py > ...

```python
import requests
BASE_URL = "http://127.0.0.1:5000/students"
# --- CREATE ---
student_data = {"name": "Alice", "age": 21, "grade": "A"}
response = requests.post(BASE_URL, json=student_data)
print("CREATE Student Response:", response.status_code, response.json())

# --- GET all ---
response = requests.get(BASE_URL)
print("GET All Students Response:", response.status_code, response.json())
# --- GET by ID ---
student_id = 1
response = requests.get(f"{BASE_URL}/{student_id}")
print(f"GET Student {student_id} Response:", response.status_code, response.json())

# --- UPDATE ---
update_data = {"age": 22, "grade": "A+"}
response = requests.put(f"{BASE_URL}/{student_id}", json=update_data)
print(f"UPDATE Student {student_id} Response:", response.status_code, response.json())

# --- DELETE ---
response = requests.delete(f"{BASE_URL}/{student_id}")
print(f"DELETE Student {student_id} Response:", response.status_code, response.json())

# --- SEARCH ---
params = {"name": "Alice"}
response = requests.get(f"{BASE_URL}/search", params=params)
print("SEARCH Response:", response.status_code, response.json())

```

Output :

```
>> Creating student Alice...
>> Status: 201
>> Response: {'id': 1, 'name': 'Alice', 'age': 20, 'grade': 'A'}
>>
>> Getting all students...
>> Status: 200
>> Response: [{'id': 1, 'name': 'Alice', 'age': 20, 'grade': 'A'}]
>>
>> Getting student with id=1...
>> Status: 200
>> Response: {'id': 1, 'name': 'Alice', 'age': 20, 'grade': 'A'}
>>
>> Updating student with id=1...
>> Status: 200
>> Response: {'id': 1, 'name': 'Alice Smith', 'age': 21, 'grade': 'A+'}
>>
>> Searching for student with name='Alice Smith'...
>> Status: 200
>> Response: [{'id': 1, 'name': 'Alice Smith', 'age': 21, 'grade': 'A+'}]
>>
>> Deleting student with id=1...
>> Status: 200
>> Response: {'message': 'Student deleted successfully'}
>>
>> Getting all students...
>> Status: 200
>> Response: []
```

Observation :

The provided test script effectively demonstrates how to perform integration testing on a Flask-based Student API using the Python requests module. It covers all essential CRUD operations— creating, retrieving, updating, and deleting a student record—as well as searching for students by name. Each API call is followed by a print statement that outputs the HTTP status code and the JSON response, making it easy to verify the correctness and behavior of

each endpoint. This approach ensures that the API is functioning as

expected and provides clear feedback for debugging or further development. Overall, the script is well-structured, easy to follow, and serves as a practical example of automated API testing.