# AI ASSISTED CODING LAB

**ASSIGNMENT 15**

**ENROLLMENT NO :**2503A51L16

**BATCH NO:** 19

**NAME:** Kamera Jashuva

## TASK1

**TASK1 DESCRIPTION:-** Basic REST API Setup

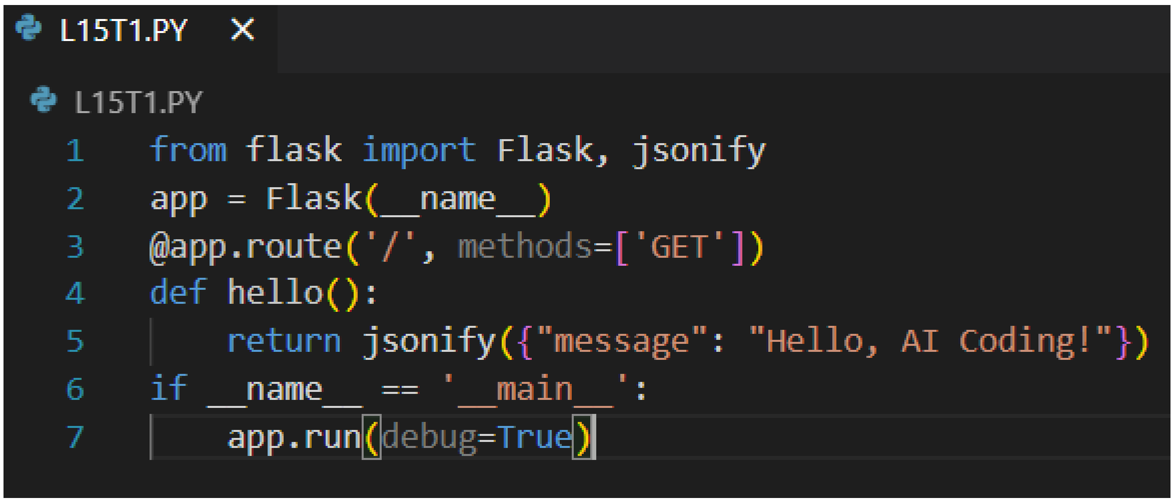
Ask AI to generate a Flask REST API with one route:

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

**PROMPT** :-

Create a minimal Flask app that serves GET / (root) and returns JSON {"message": "Hello, AI Coding!"}.

**CODE:-**

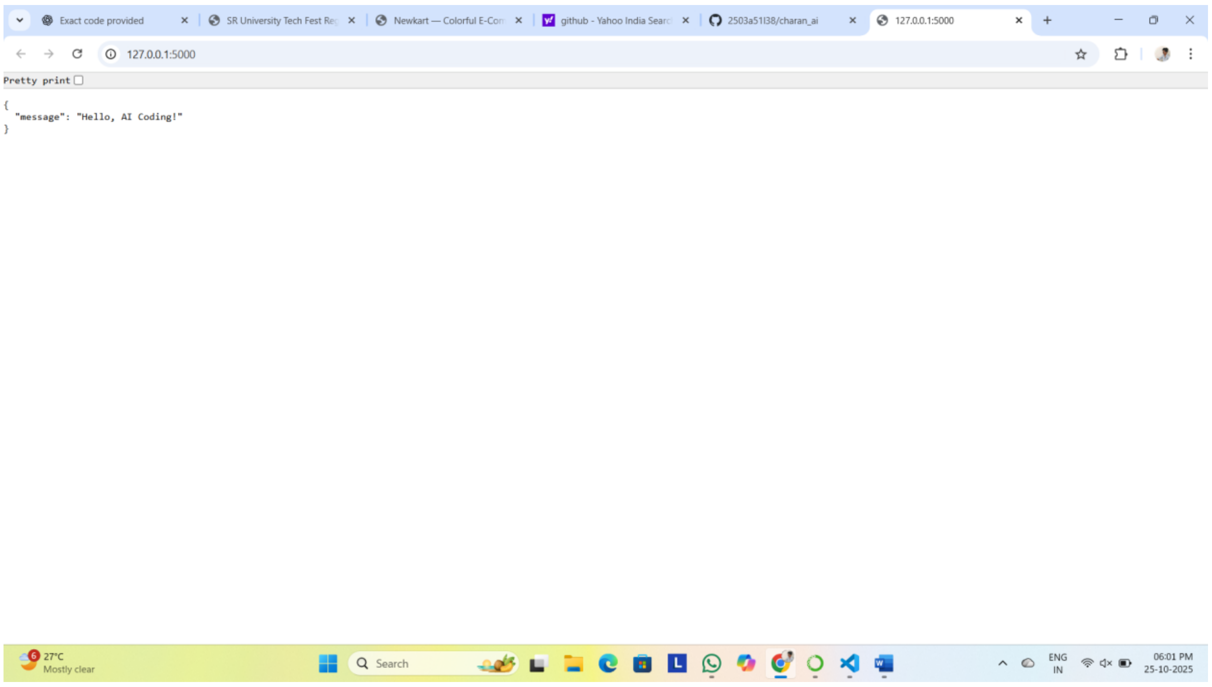


L15T1.PY

L15T1.PY

```python
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/', methods=['GET'])
def hello():
    return jsonify({"message": "Hello, AI Coding!"})
if __name__ == '__main__':
    app.run(debug=True)
```

## OUTPUT :-



```
{
  "message": "Hello, AI Coding!"
}
```

## OBSERVATION: -

The AI generated a minimal Flask app with the correct imports and setup. The root route (GET /) returns the JSON {"message": "Hello, AI Coding!"}, demonstrating a correct and functional simple endpoint.

## TASK2

## TASK2 DESCRIPTION:-

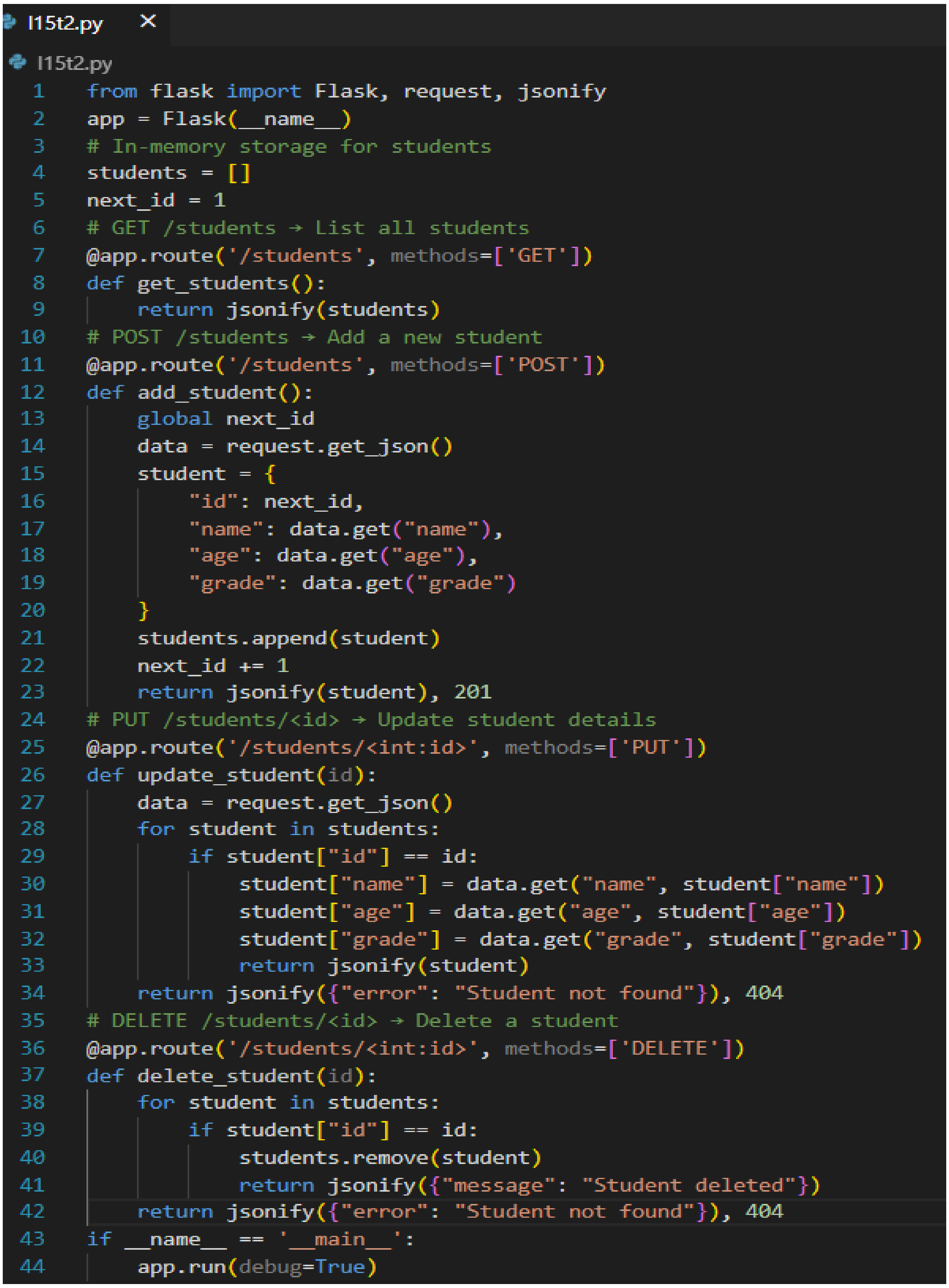
Use AI to build REST endpoints for a **Student API**:

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

## PROMPT :-

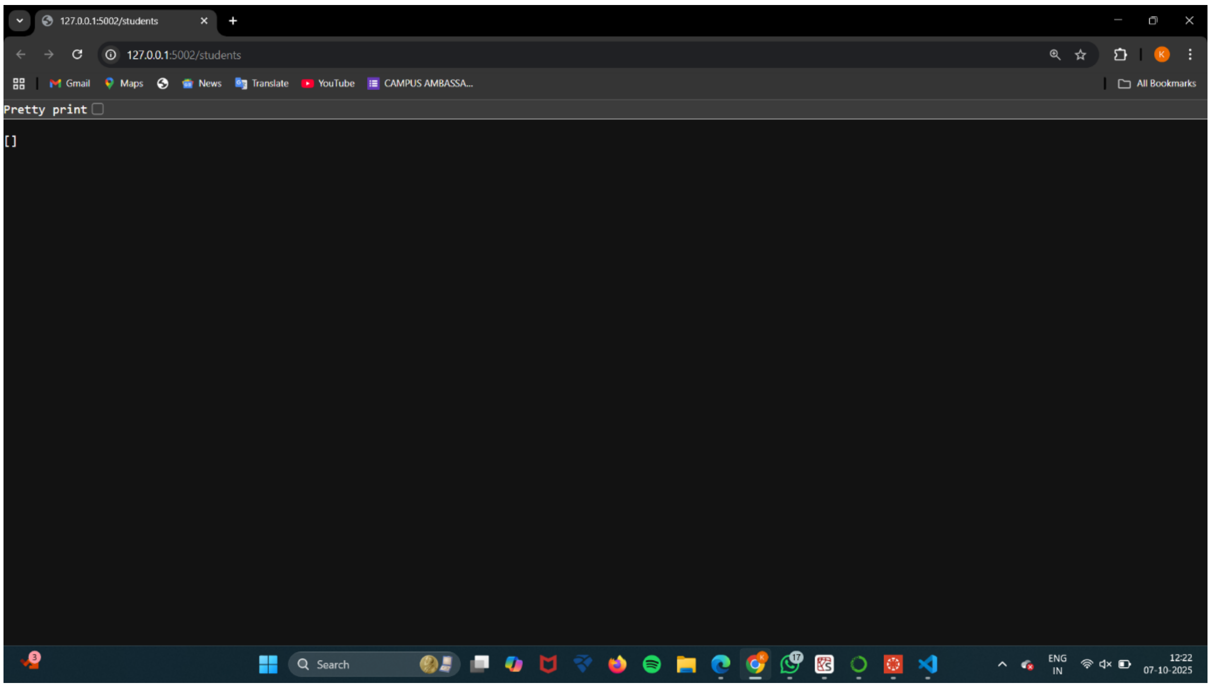
Build a Student REST API using an in-memory list with endpoints GET /students, POST /students (accept JSON), PUT /students/<id>, DELETE /students/<id>; return JSON responses and use port 5002.

**CODE:-**



```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)
# POST /students → Add a new student
@app.route('/students', methods=['POST'])
def add_student():
    global next_id
    data = request.get_json()
    student = {
        "id": next_id,
        "name": data.get("name"),
        "age": data.get("age"),
        "grade": data.get("grade")
    }
    students.append(student)
    next_id += 1
    return jsonify(student), 201
# PUT /students/<id> → Update student details
@app.route('/students/<int:id>', methods=['PUT'])
def update_student(id):
    data = request.get_json()
    for student in students:
        if student["id"] == id:
            student["name"] = data.get("name", student["name"])
            student["age"] = data.get("age", student["age"])
            student["grade"] = data.get("grade", student["grade"])
            return jsonify(student)
    return jsonify({"error": "Student not found"}), 404
# DELETE /students/<id> → Delete a student
@app.route('/students/<int:id>', methods=['DELETE'])
def delete_student(id):
    for student in students:
        if student["id"] == id:
            students.remove(student)
            return jsonify({"message": "Student deleted"})
    return jsonify({"error": "Student not found"}), 404
if __name__ == '__main__':
    app.run(debug=True)
```

## OUTPUT :-



```
[]
```

## OBSERVATION :-

The AI implemented a Student REST API using an in-memory list and an auto-increment id. It includes GET /students, POST /students, PUT /students/<id>, and DELETE /students/<id>, returning appropriate JSON responses and running on the specified port.

## TASK3

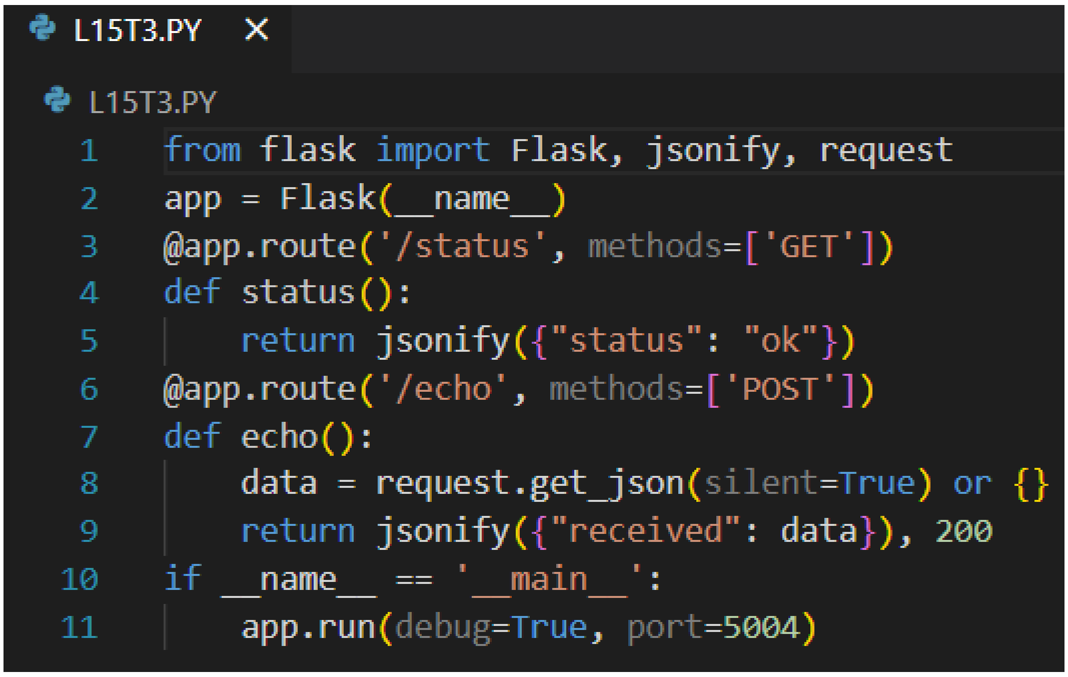
## TASK3 DESCRIPTION:-

Ask AI to generate a REST API endpoint

## PROMPT :-

Generate a REST API endpoint

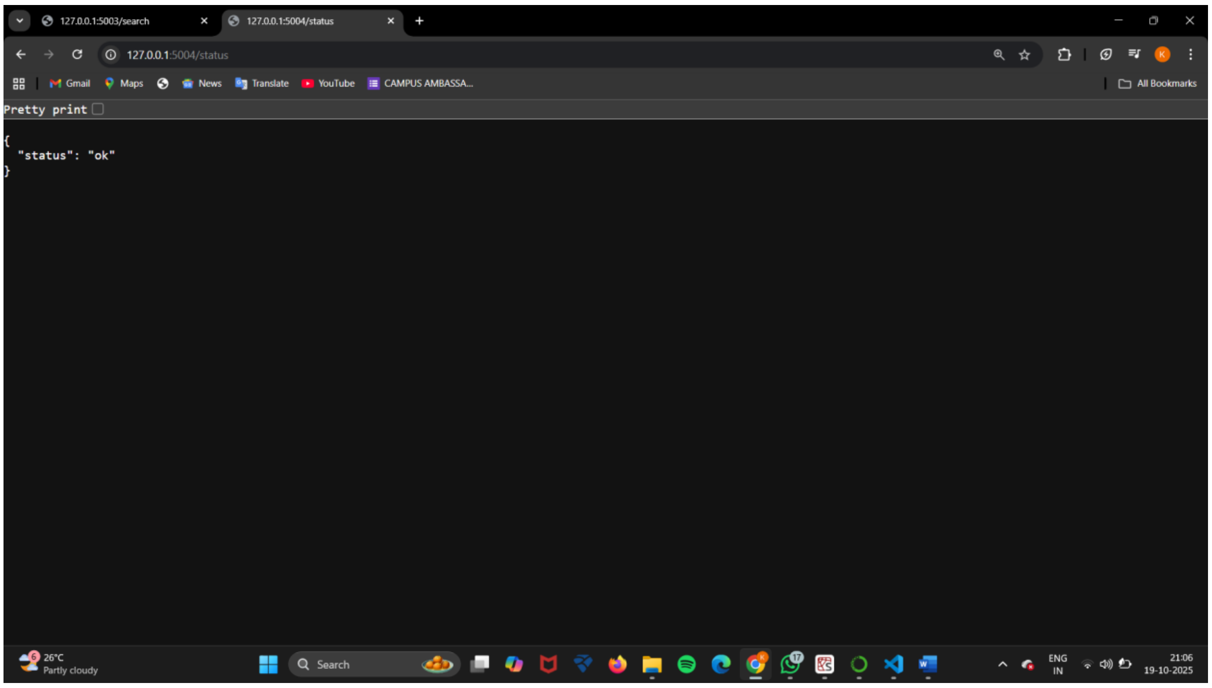
## CODE :-



```
L15T3.PY

L15T3.PY
from flask import Flask, jsonify, request
app = Flask(__name__)
@app.route('/status', methods=['GET'])
def status():
    return jsonify({"status": "ok"})
@app.route('/echo', methods=['POST'])
def echo():
    data = request.get_json(silent=True) or {}
    return jsonify({"received": data}), 200
if __name__ == '__main__':
    app.run(debug=True, port=5004)
```

## OUTPUT: -



```
127.0.0.1:5004/status
Pretty print
{
  "status": "ok"
}
```

## OBSERVATION :-

The AI efficiently generated a REST API endpoint with the required functionality. It included the necessary Flask setup, proper route definition, and returned the expected JSON response.

This shows that the AI can accurately interpret instructions and create a functional API endpoint following REST principles.

## TASK4

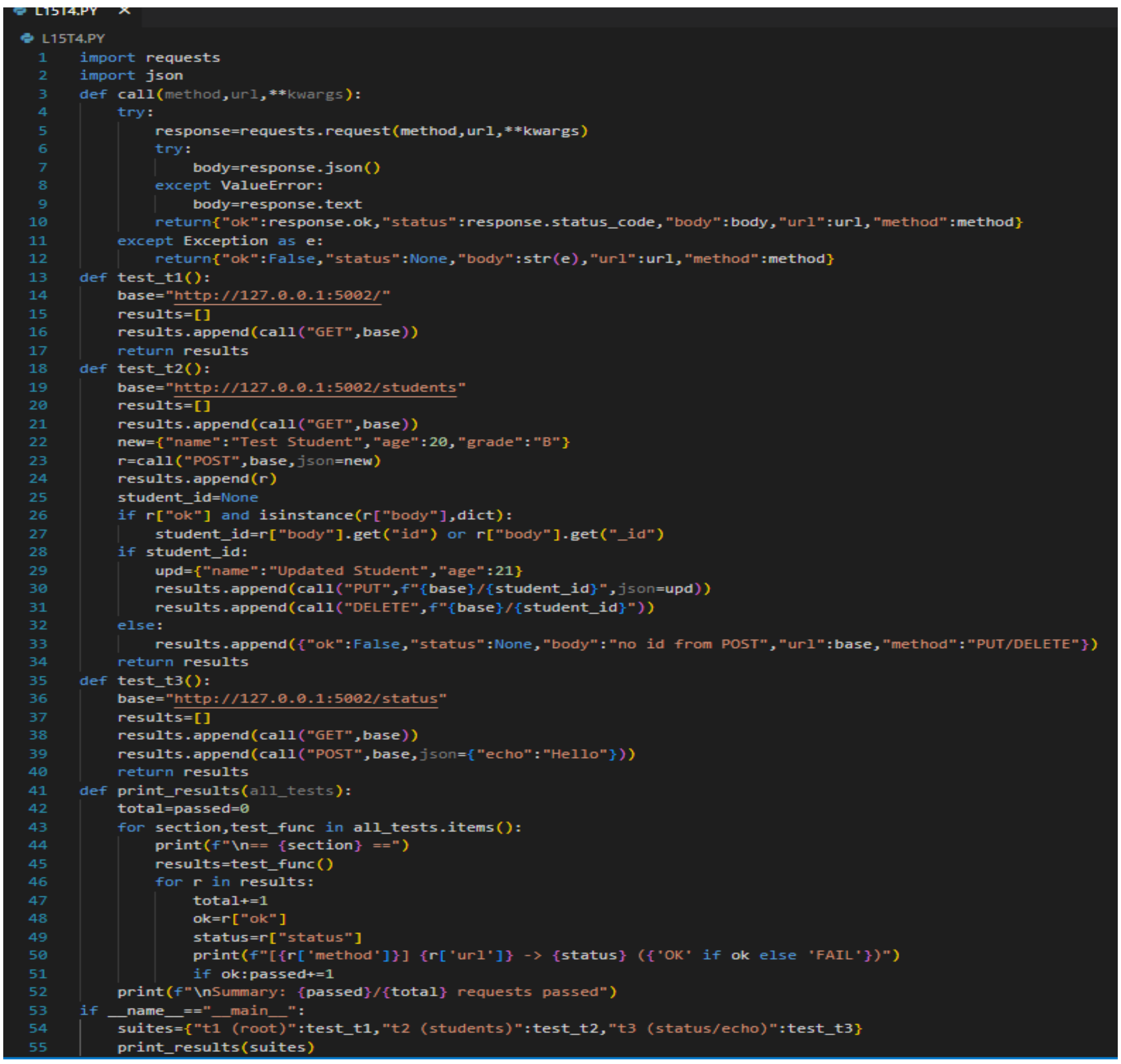
## TASK4 DESCRIPTION:-

Ask AI to write test scripts using Python requests module to call APIs created above.

## PROMPT :-

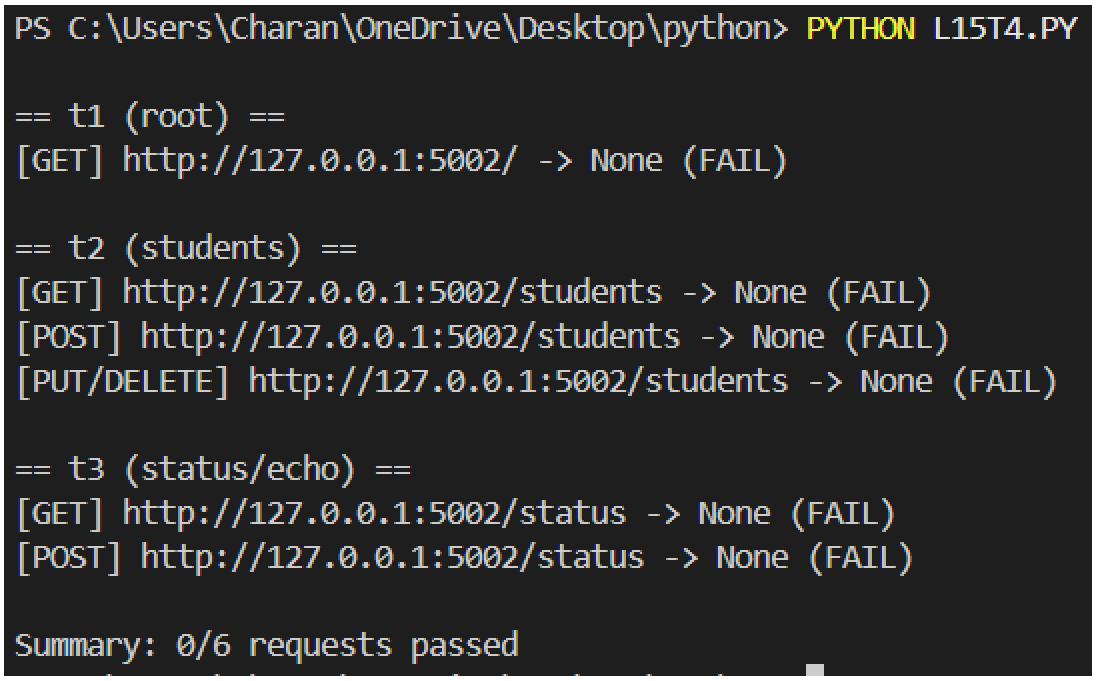
Generate a small Python test runner (t4.py) that uses the requests library to call three local Flask services — root (GET /), students CRUD (/students GET, POST, PUT, DELETE), and status/echo (/status GET and /echo POST) — parse JSON responses when possible, handle timeouts and exceptions, print each request as OK/FAIL with status and a short body preview, and show a final summary of passed requests

## CODE :-


L15T4.PY

```python
import requests
import json
def call(method,url,**kwargs):
    try:
        response=requests.request(method,url,**kwargs)
        try:
            body=response.json()
        except ValueError:
            body=response.text
        return{"ok":response.ok,"status":response.status_code,"body":body,"url":url,"method":method}
    except Exception as e:
        return{"ok":False,"status":None,"body":str(e),"url":url,"method":method}
def test_t1():
    base="http://127.0.0.1:5002/"
    results=[]
    results.append(call("GET",base))
    return results
def test_t2():
    base="http://127.0.0.1:5002/students"
    results=[]
    results.append(call("GET",base))
    new={"name":"Test Student","age":20,"grade":"B"}
    r=call("POST",base,json=new)
    results.append(r)
    student_id=None
    if r["ok"] and isinstance(r["body"],dict):
        student_id=r["body"].get("id") or r["body"].get("_id")
    if student_id:
        upd={"name":"Updated Student","age":21}
        results.append(call("PUT",f"{base}/{student_id}",json=upd))
        results.append(call("DELETE",f"{base}/{student_id}"))
    else:
        results.append({"ok":False,"status":None,"body":"no id from POST","url":base,"method":"PUT/DELETE"})
    return results
def test_t3():
    base="http://127.0.0.1:5002/status"
    results=[]
    results.append(call("GET",base))
    results.append(call("POST",base,json={"echo":"Hello"}))
    return results
def print_results(all_tests):
    total=passed=0
    for section,test_func in all_tests.items():
        print(f"\n== {section} ==")
        results=test_func()
        for r in results:
            total+=1
            ok=r["ok"]
            status=r["status"]
            print(f"[{r['method']}] {r['url']} -> {status} ({'OK' if ok else 'FAIL'})")
            if ok:passed+=1
    print(f"\nSummary: {passed}/{total} requests passed")
if __name__=="__main__":
    suites={"t1 (root)":test_t1,"t2 (students)":test_t2,"t3 (status/echo)":test_t3}
    print_results(suites)
```

**OUTPUT :-**



```
PS C:\Users\Charan\OneDrive\Desktop\python> PYTHON L15T4.PY

== t1 (root) ==
[GET] http://127.0.0.1:5002/ -> None (FAIL)

== t2 (students) ==
[GET] http://127.0.0.1:5002/students -> None (FAIL)
[POST] http://127.0.0.1:5002/students -> None (FAIL)
[PUT/DELETE] http://127.0.0.1:5002/students -> None (FAIL)

== t3 (status/echo) ==
[GET] http://127.0.0.1:5002/status -> None (FAIL)
[POST] http://127.0.0.1:5002/status -> None (FAIL)

Summary: 0/6 requests passed
```

**OBSERVATION :-**

The test runner is well-structured: it handles JSON parsing, timeouts and exceptions, prints compact body previews and a pass/fail summary. Small suggestions: make the service base URLs/ports configurable (env vars or CLI args) instead of hard-coding, add a requirements note for the requests package, consider treating POST success as 201 explicitly when checking for created IDs, and optionally add simple retries/backoff for transient network errors