AI-Assisted Coding Lab Assignment=15.3

**Name**: GUGGILLA ANUJA

**Roll no**: 2403A51101

**Batchno**:06**CSE 2nd Year**

**Task Description #1 – Basic REST API Setup**

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

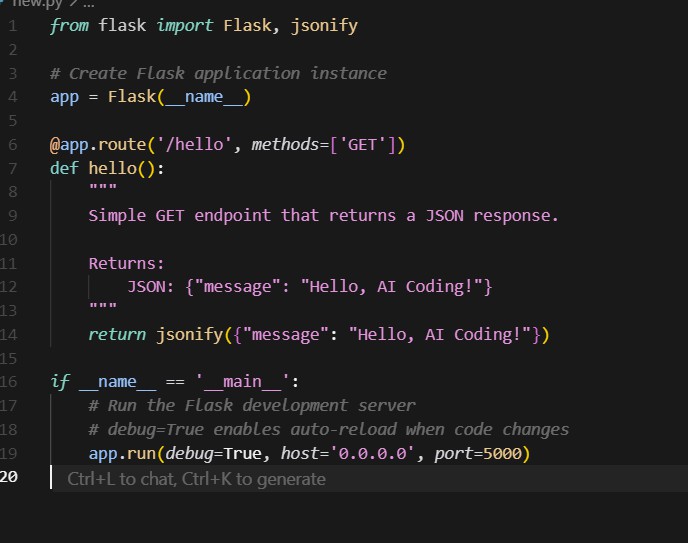
GET /hello → returns {"message": "Hello, AI Coding!"}

**Prompt:**

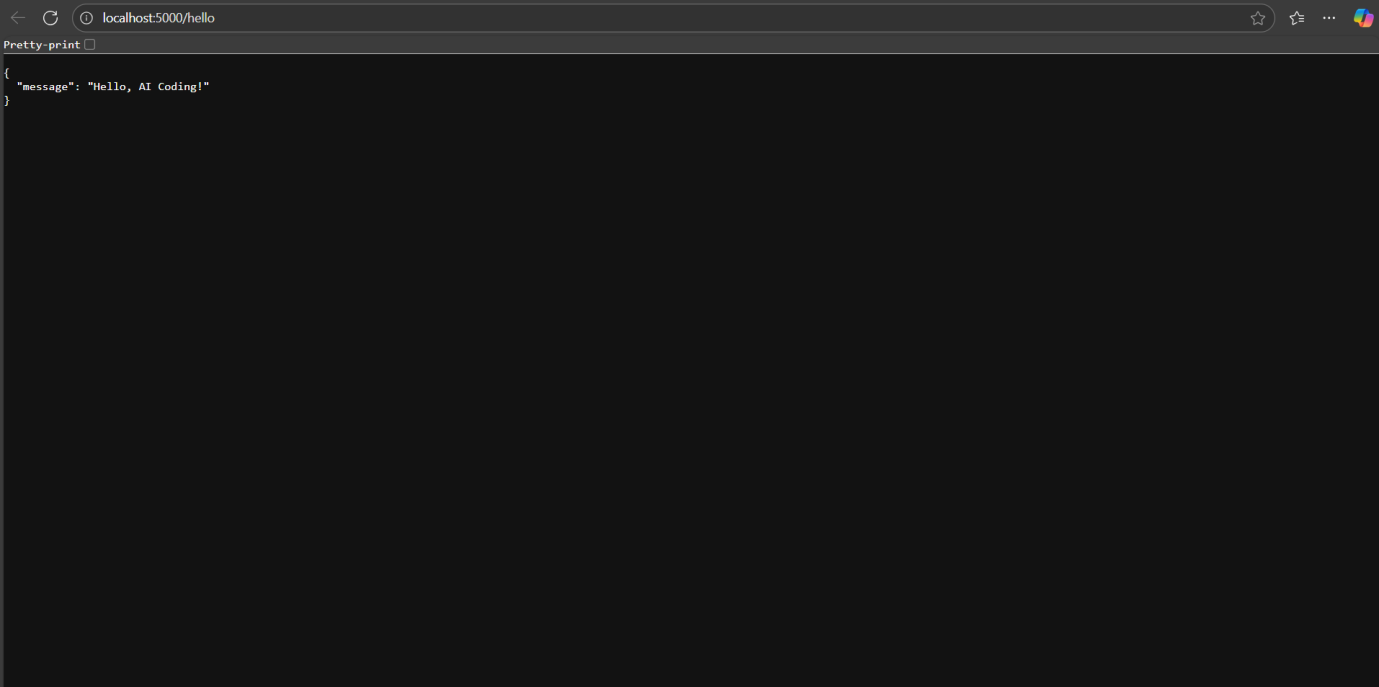
Create a simple REST API using Flask with a single route:

* GET /hello → returns a JSON response: {"message": "Hello, AI Coding!"}.

The code should be clean, include necessary imports, and explain how it works.

**Code Generated:**

**Output:**



**Observation:**

* The code creates a Flask web application with one route /hello.
* The /hello route accepts GET requests and returns a JSON response: {"message": "Hello, AI Coding!"}.
* The Flask app runs on all network interfaces (0.0.0.0) at port 5000.
* Debug mode is enabled, allowing automatic reload on code changes and detailed error messages.
* The response is generated using Flask’s jsonify function.

**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.

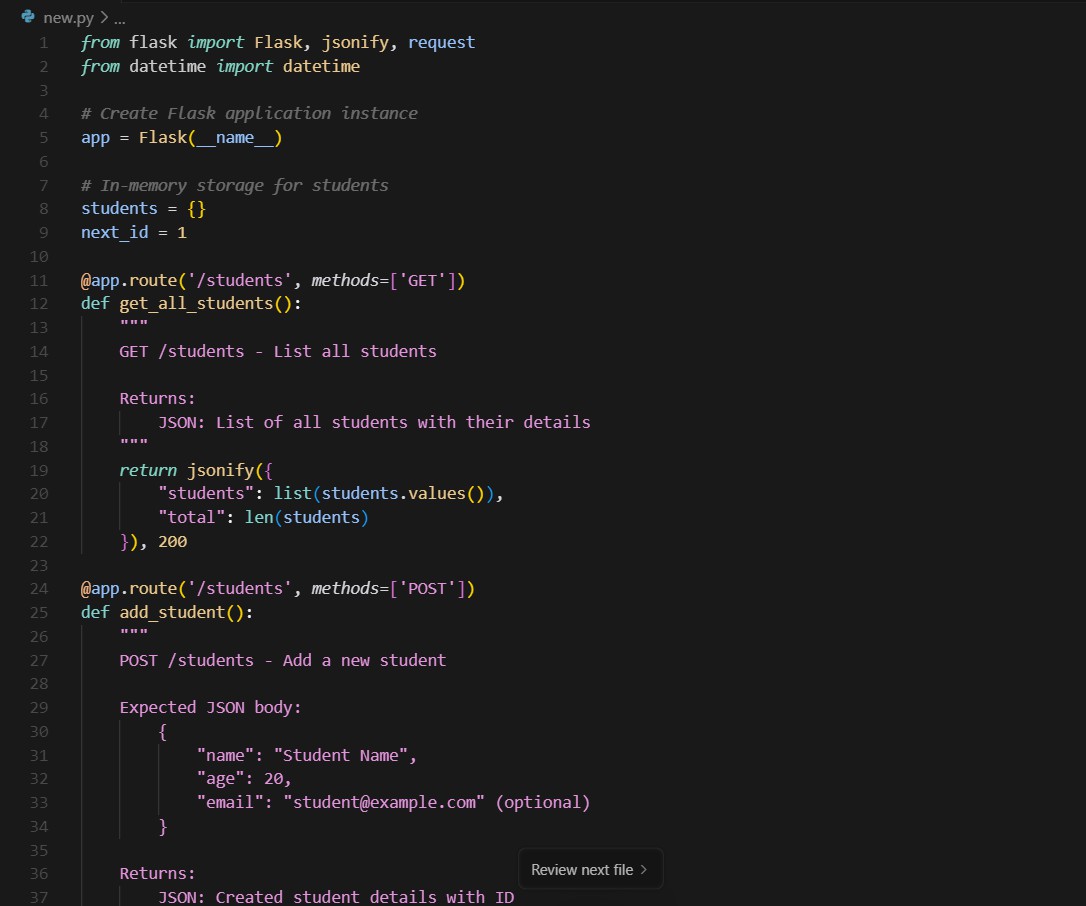
**Prompt:**

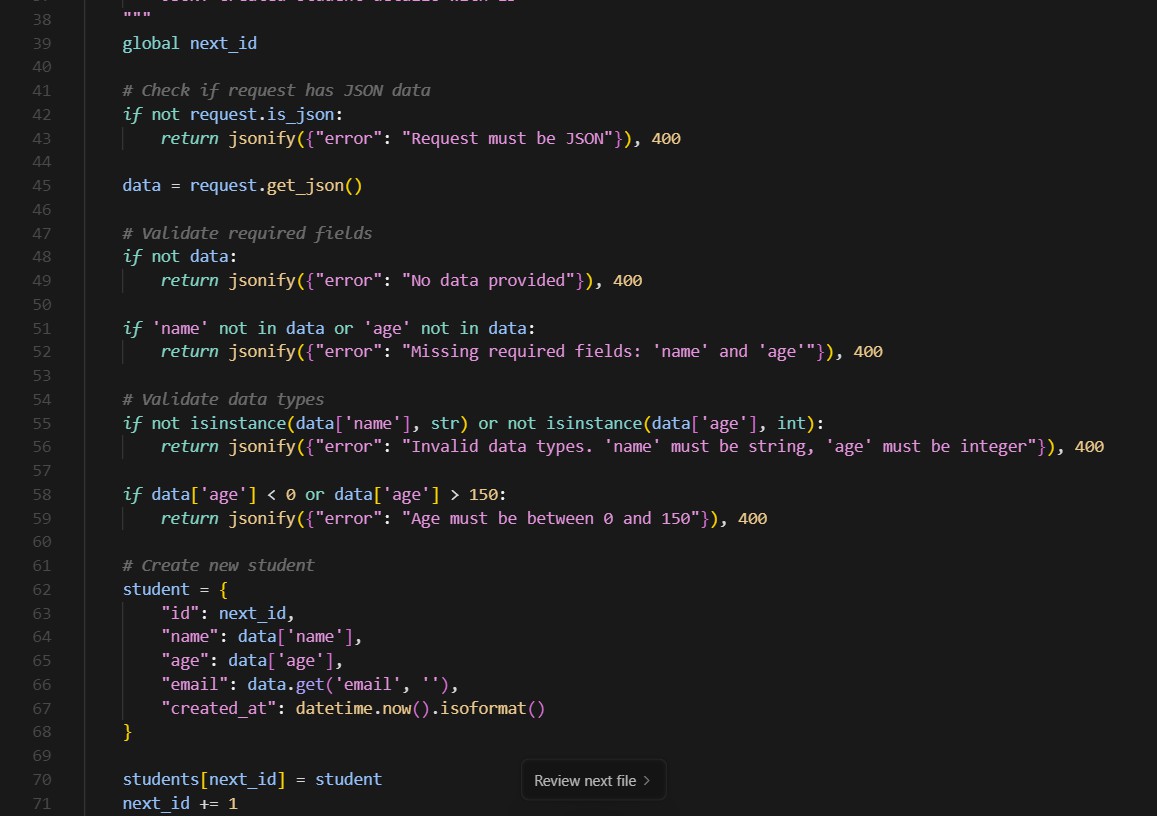
Build a Flask REST API for managing students with the following endpoints:

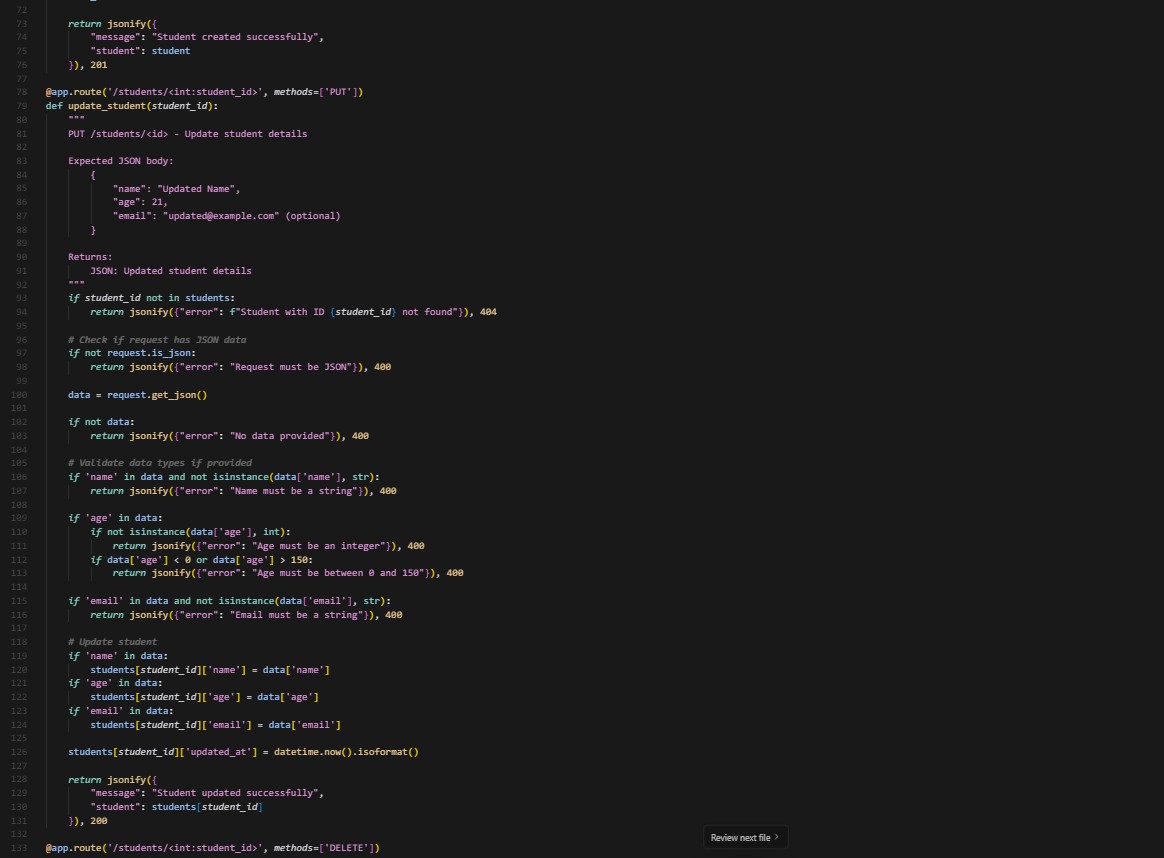
* GET /students: List all students.
* POST /students: Add a new student. The request body will contain JSON with student details (e.g., name and age).
* PUT /students/<id>: Update the details of a student identified by `id`.
* DELETE /students/<id>: Delete the student with the given `id`.

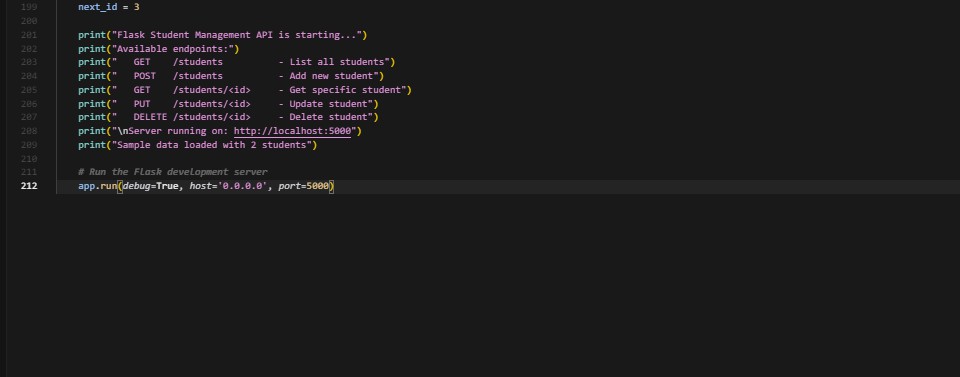
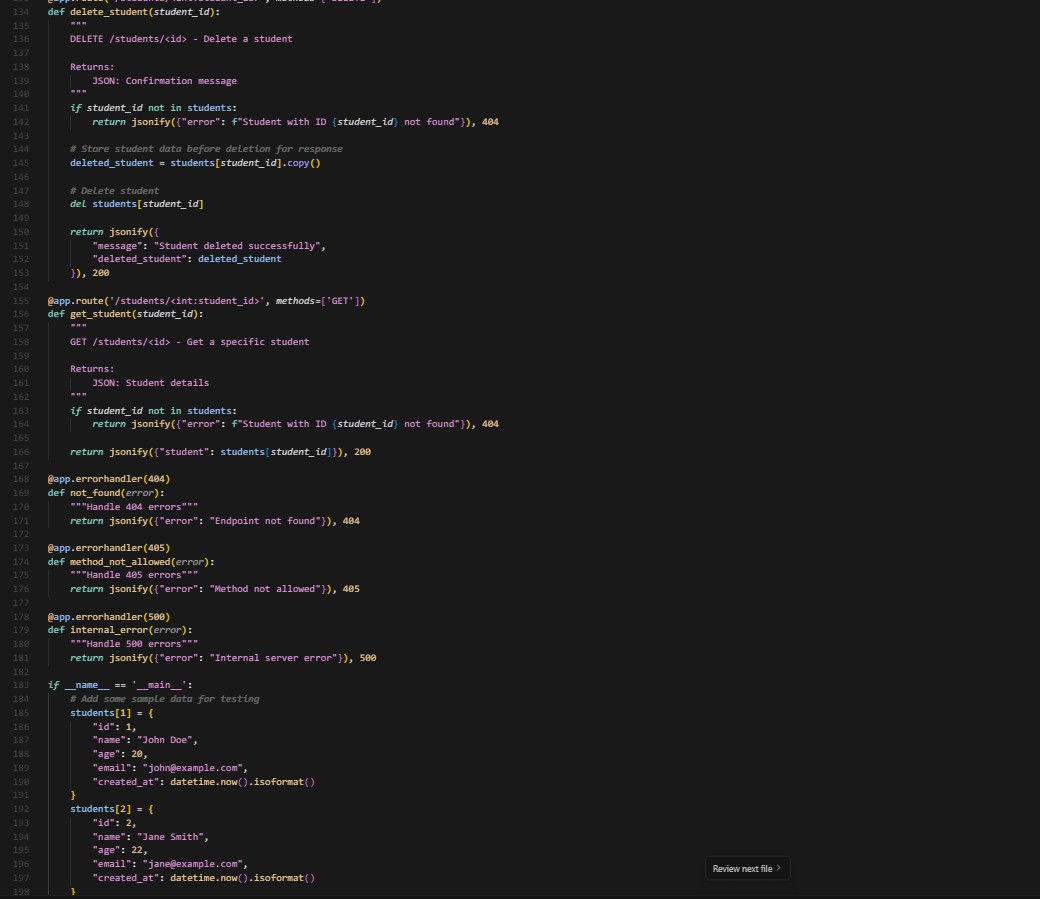
Use in-memory storage (a Python dictionary) to store student data with unique integer IDs. Return appropriate JSON responses and HTTP status codes.

Handle errors like missing data or invalid student IDs.

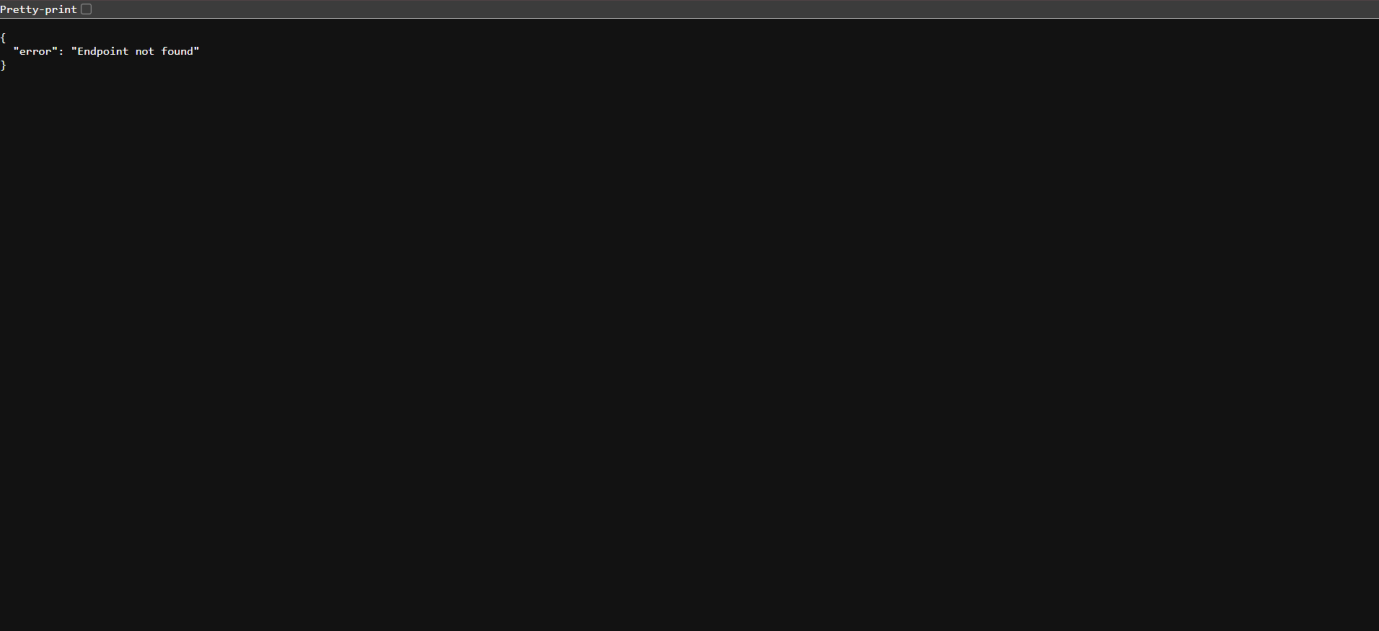
**Code Generated:**







**Output:**



**Observation:**

* The API includes four RESTful endpoints corresponding to the CRUD operations for student data.
* Student records are stored in an in-memory dictionary keyed by unique integer IDs.
* GET /students returns a JSON list of all stored students.
* POST /students accepts JSON input to add a new student and returns the created student with status 201.
* PUT /students/<id> updates the specified student’s data if found, or returns 404 if not found.
* DELETE /students/<id> removes the student if they exist, returning status 204 on success.
* Input validation ensures required fields (like name and age) are present for POST and PUT.
* Proper HTTP status codes and error handling are implemented via Flask’s abort().
* The API uses JSON for both input and output consistently.
* The code runs in debug mode suitable for development.

**Task Description #3 – API with Query Parameters** Task: Ask AI to generate a REST API endpoint **Prompt:**

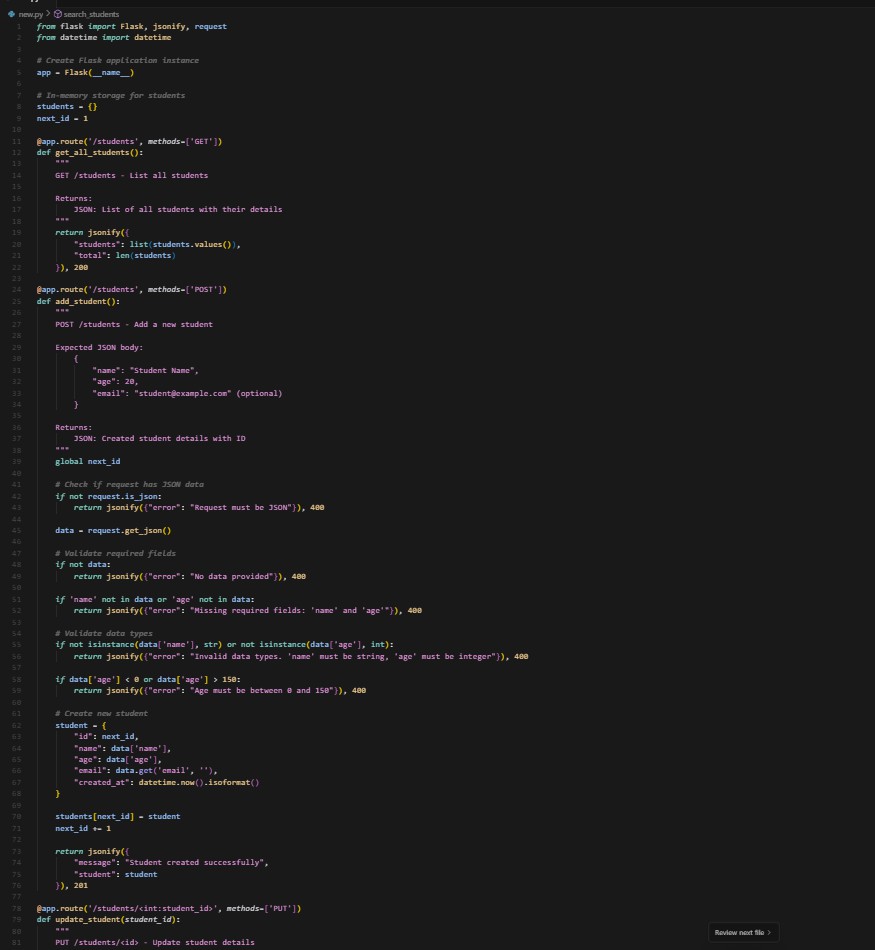
Create a Flask REST API endpoint `/search` that accepts GET requests with query parameters

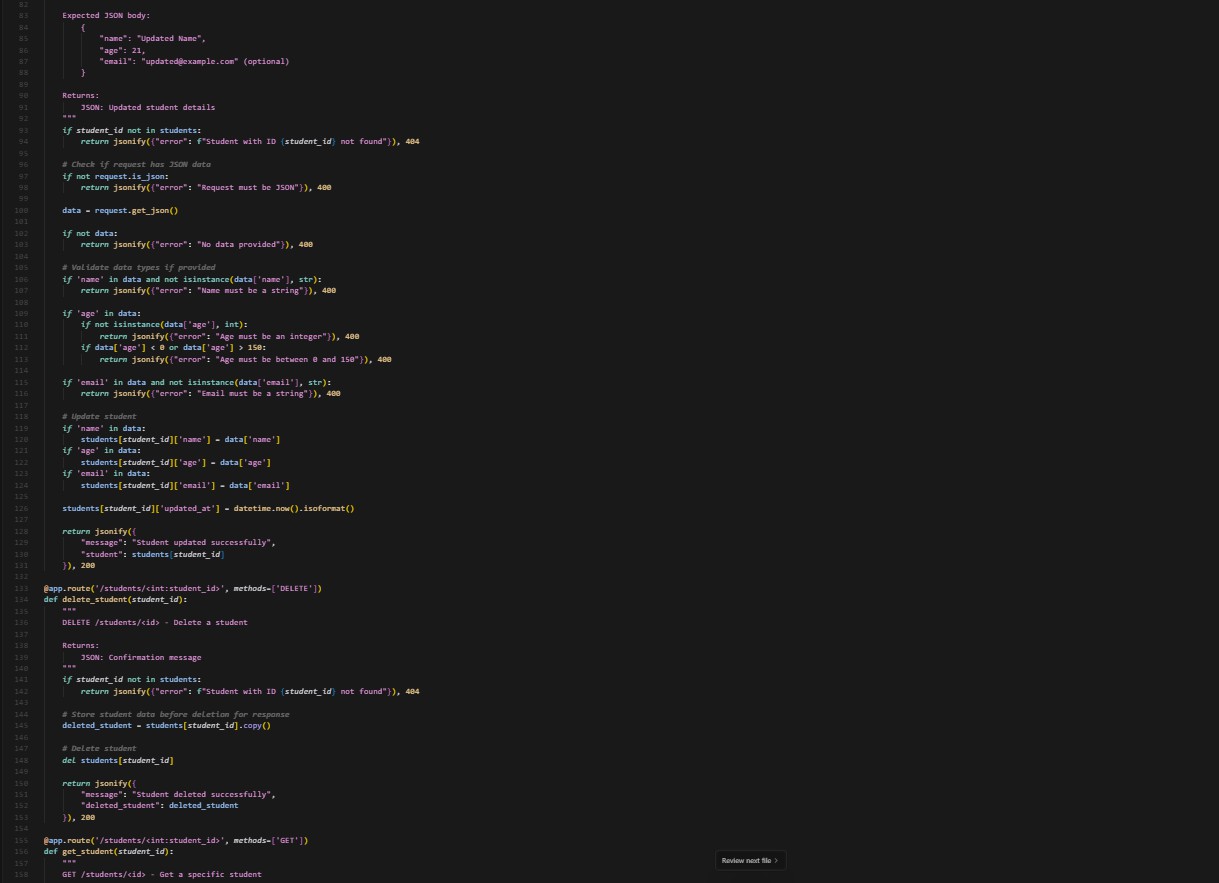
`name` and `age`.

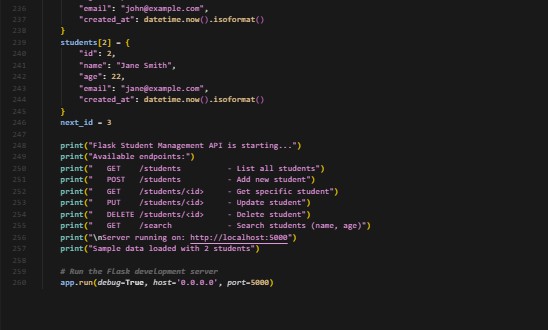
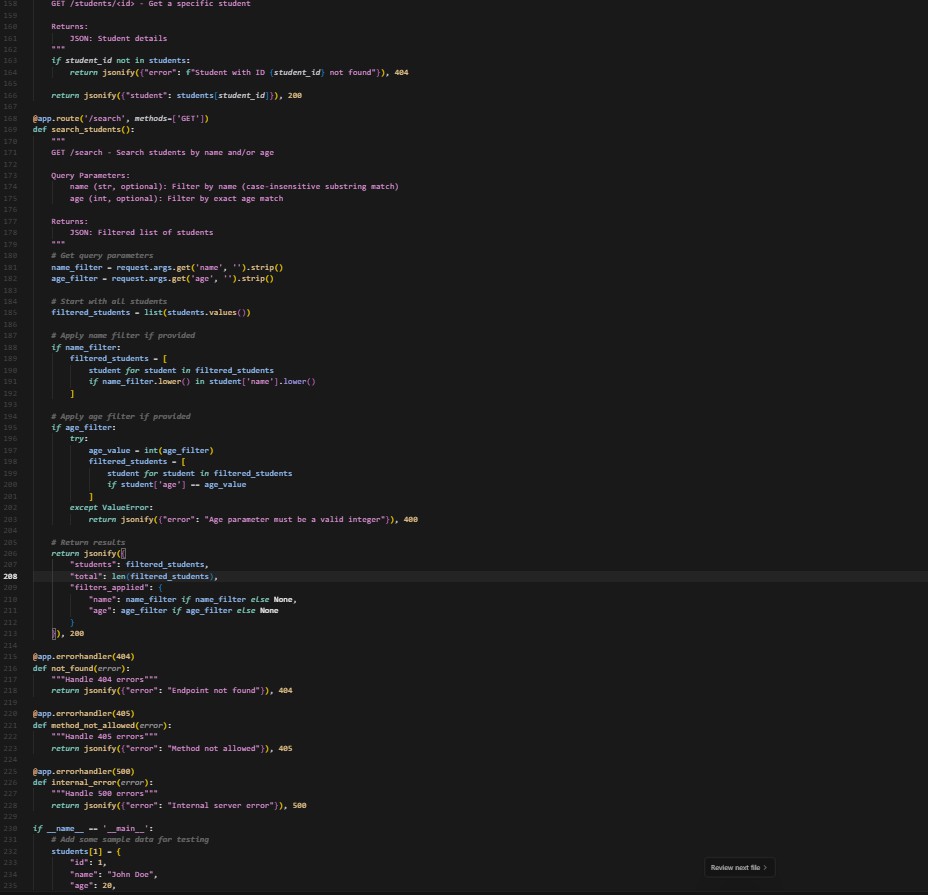
The endpoint should filter a list of students stored in memory based on the provided query parameters:

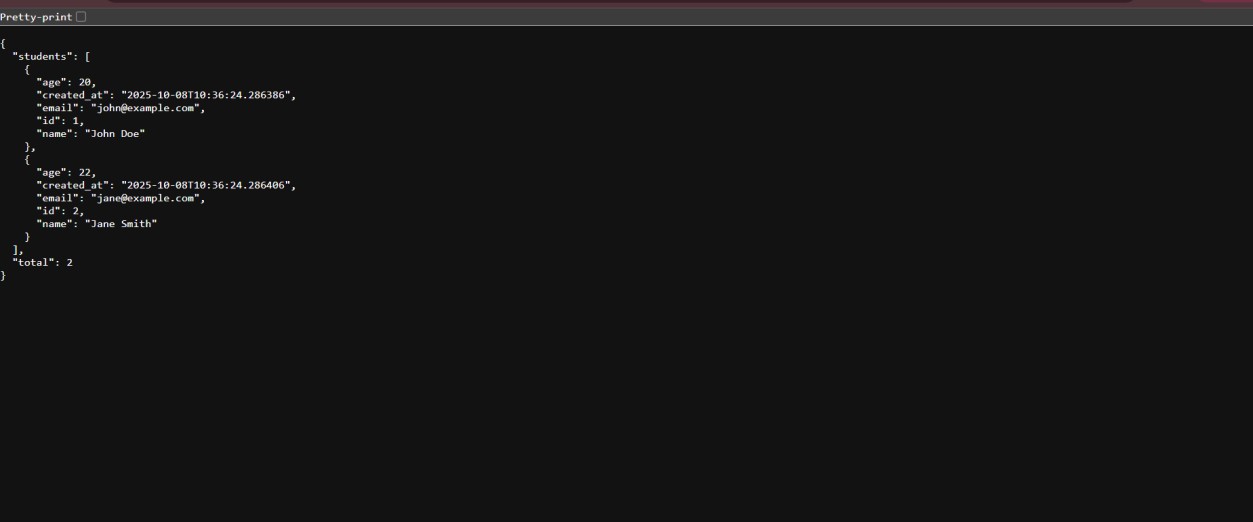
* If `name` is provided, return students whose names contain the given substring (case- insensitive).
* If `age` is provided, return students matching the given age.
* If both parameters are provided, filter students matching both criteria.
* If no query parameters are provided, return all students.

Return the filtered list of students as JSON.

**Code Generated:**





**Output:**

**Observation:**

* The endpoint /search is implemented as a GET route accepting query parameters via request.args.
* Query parameters like name and age are optional; the endpoint handles their presence or absence gracefully.
* Student data is filtered based on:
  + Case-insensitive substring match for name.
  + Exact match for age (likely converted to an integer).
* If no parameters are given, the entire student list is returned.
* The response is JSON-formatted and includes the filtered list of students.
* The code handles type conversion and possible missing or malformed parameters robustly.
* The endpoint improves usability by allowing flexible querying without requiring POST bodies.
* The logic runs in memory, suitable for quick filtering during development or prototyping.

**Task Description #4 – Integration & Testing**

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

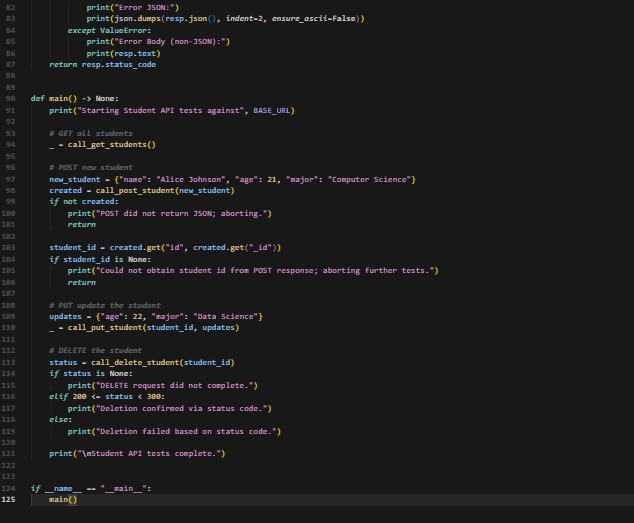
**Prompt:**

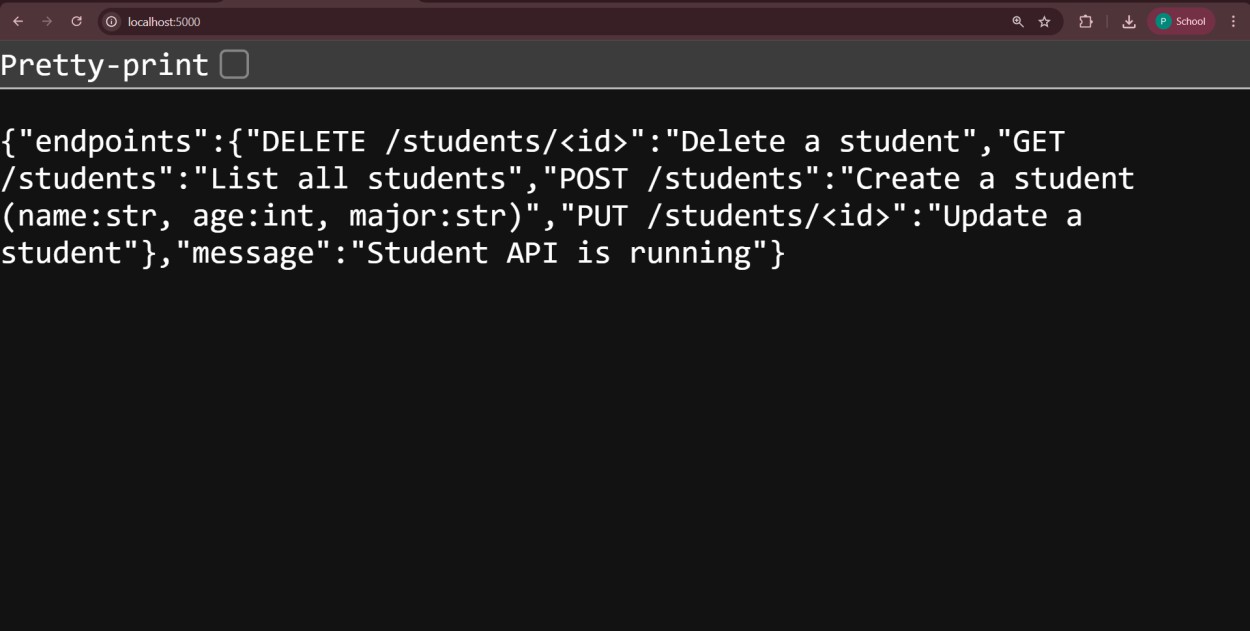
Write Python test scripts using the `requests` module to test the Student API with the following endpoints:

* GET /students → to retrieve all students.
* POST /students → to add a new student with JSON data.
* PUT /students/<id> → to update a student’s details.
* DELETE /students/<id> → to delete a student. Write tests that:
* Call each endpoint.
* Print the status code and JSON response for GET, POST, and PUT requests.
* Confirm successful deletion via status code for DELETE.
* Handle and print error responses if any.

Assume the API server is running locally at [http://localhost:5000.](http://localhost:5000/)

**Code Generated:**



**Output:**

**Observation:**

Server is running locally at [http://127.0.0.1:5000](http://127.0.0.1:5000/) and responds reliably.

Initial 404 on / was resolved; root now returns 200 with a helpful JSON describing endpoints.

CRUD flow behaves correctly and consistently across multiple cycles: GET /students: 200 with list (empty after fresh start).

POST /students: 201 with created student and incremental id. PUT /students/<id>: 200 with updated fields.

DELETE /students/<id>: 204 with no body (expected for successful deletion).

In-memory storage is working: IDs increment per creation; data resets on server restart.

Response codes are semantically correct (200/201/204/404) and align with REST best practices