**Assignment # 2**

**Muharrem Kaya**

**041176702**

**Algonquin College**

**Remote Data and RT Applications**

**Sep 24, 2024**

IMPLEMENTING RESTFUL API FOR IOT DEVICES

**Introduction**

In this project, I developed a RESTful API using Flask to handle sensor data entries. While I have some experience working with Flask, most of my professional background is in Java and the Spring Boot framework, where I’ve implemented several API-driven solutions in enterprise environments. This assignment allowed me to expand my knowledge of Python and Flask, comparing its API routing and request-handling capabilities with the Spring Boot framework.

I used GitHub Codespaces and VSCode for the development environment, which provided a fully cloud-based, pre-configured setup, streamlining the development process. Once the API was built, I created a shell script to automate the testing of the Python code, verifying the functionality of all endpoints. This made the process more efficient and ensured that the API performed as expected across different scenarios.

**Solution Overview**

The solution involves creating a Python Flask application that exposes RESTful endpoints to handle sensor data. The core tasks implemented were as follows:

1. GET /average-temperature: This endpoint retrieves the average temperature from the sensor data and returns it as a JSON response. The logic checks if there is data available and calculates the average using a simple sum and division.
2. POST /sensor-data: This endpoint allows adding new sensor data entries. The required fields are timestamp, temperature, humidity, and pressure. If any of these fields are missing, the API returns an error message indicating the missing information.
3. PUT /sensor-data/<timestamp>: This endpoint updates an existing sensor data entry based on the provided timestamp. The client sends a JSON object containing the new values for the sensor entry, and if a matching timestamp is found, the entry is updated.
4. PATCH /sensor-data/<timestamp>: The PATCH method, which I added to extend the functionality of the API, allows for partial updates of the sensor data. It only updates the fields provided in the request body, leaving other fields unchanged.
5. DELETE /sensor-data/<timestamp>: The DELETE method, another addition I implemented, removes a sensor data entry based on the provided timestamp. If the timestamp is found, the entry is deleted from the dataset, and a success message is returned.

**Expected Outputs:**

|  |  |  |  |
| --- | --- | --- | --- |
| **METHOD** | **COMMAND (**BASE\_URL=<http://127.0.0.1:5000>) | **EXPECTED OUTPUT** | **REAL OUTPUT** |
| GET | # Test GET /average-temperature curl -X GET "${BASE\_URL}/average-temperature" | { "average\_temperature": 25.8 } | { "average\_temperature": 25.8 } |
| POST | # Test POST /sensor-data curl -X POST "${BASE\_URL}/sensor-data" -H "Content-Type: application/json" -d '{"timestamp": "2023-05-28T10:50:00", "temperature": 28.0, "humidity": 65.0, "pressure": 1015.0}' | { "message": "Data added successfully"  } | { "message": "Data added successfully"  } |
| PUT | # Test PUT /sensor-data/<timestamp> curl -X PUT "${BASE\_URL}/sensor-data/2023-05-28T10:50:00" -H "Content-Type: application/json" -d '{"temperature": 29.0, "humidity": 63.0, "pressure": 1014.0}' | { "message": "Data updated successfully",  "updated\_data": {  "timestamp": "2023-05-28T10:50:00",  "temperature": 29.0,  "humidity": 63.0,  "pressure": 1014.0  }  } | { "message": "Data updated successfully",  "updated\_data": {  "timestamp": "2023-05-28T10:50:00",  "temperature": 29.0,  "humidity": 63.0,  "pressure": 1014.0  }  } |
| PATCH | # Test PATCH /sensor-data/<timestamp> curl -X PATCH "${BASE\_URL}/sensor-data/2023-05-28T10:50:00" -H "Content-Type: application/json" -d '{"temperature": 30.0}' | { "message": "Data partially updated",  "updated\_data": {  "timestamp": "2023-05-28T10:50:00",  "temperature": 30.0,  "humidity": 63.0,  "pressure": 1014.0  }  } | { "message": "Data partially updated",  "updated\_data": {  "timestamp": "2023-05-28T10:50:00",  "temperature": 30.0,  "humidity": 63.0,  "pressure": 1014.0  }  } |
| DELETE | # Test DELETE /sensor-data/<timestamp> curl -X DELETE "${BASE\_URL}/sensor-data/2023-05-28T10:50:00" | {  "message": "Data deleted successfully"  } | {  "message": "Data deleted successfully"  } |

**Output Screenshots**

**Flusk Server:**

A screenshot of a computer code

Description automatically generated

**./test.sh**   
A screenshot of a computer

Description automatically generated

**Code Snippets:**

**GET Method**

@app.route('/average-temperature', methods=['GET'])

def get\_average\_temperature():

# TODO: Implement code to calculate the average temperature

# and return it as a JSON response

if sensor\_data:

total\_temp = sum(entry['temperature'] for entry in sensor\_data)

average\_temp = total\_temp / len(sensor\_data)

return jsonify({"average\_temperature": average\_temp})

return jsonify({"message": "No data available"}), 404

**POST Method**

# Task 2: Add a new sensor data entry

@app.route('/sensor-data', methods=['POST'])

def add\_sensor\_data():

# TODO: Implement code to add a new sensor data entry

# based on the request body JSON

new\_entry = request.get\_json()

required\_fields = ['timestamp', 'temperature', 'humidity', 'pressure']

if all(field in new\_entry for field in required\_fields):

sensor\_data.append(new\_entry)

return jsonify({"message": "Data added successfully"}), 201

return jsonify({"message": "Missing fields"}), 400

**PUT Method**

# Task 3: Update an existing sensor data entry by timestamp

@app.route('/sensor-data/<timestamp>', methods=['PUT'])

def update\_sensor\_data(timestamp):

# TODO: Implement code to update an existing sensor data entry

# based on the provided timestamp and request body JSON

updated\_data = request.get\_json()

for entry in sensor\_data:

if entry['timestamp'] == timestamp:

entry.update(updated\_data)

return jsonify({"message": "Data updated successfully", "updated\_data": entry})

return jsonify({"message": "Data not found"}), 404

**PATHCH Method**

# Task 4: Partially update an existing sensor data entry by timestamp

@app.route('/sensor-data/<timestamp>', methods=['PATCH'])

def patch\_sensor\_data(timestamp):

# This method allows partial updates to an existing sensor data entry

patch\_data = request.get\_json()

for entry in sensor\_data:

if entry['timestamp'] == timestamp:

entry.update(patch\_data)

return jsonify({"message": "Data partially updated", "updated\_data": entry})

return jsonify({"message": "Data not found"}), 404

**DELETE Method**

# Task 5: Delete an existing sensor data entry by timestamp

@app.route('/sensor-data/<timestamp>', methods=['DELETE'])

def delete\_sensor\_data(timestamp):

# This method deletes a sensor data entry based on the provided timestamp

for i, entry in enumerate(sensor\_data):

if entry['timestamp'] == timestamp:

del sensor\_data[i]

return jsonify({"message": "Data deleted successfully"}), 200

return jsonify({"message": "Data not found"}), 404

**Challenges and Difficulties**

* In the implementation process, one of the challenges I encountered was ensuring the DELETE method worked correctly. Since Python lists can be tricky to modify during iteration, I had to ensure that the entry removal was handled safely by using the index to delete the entry after iterating through the list.
* Writing the shell script to automate testing was straightforward but required some thought on how to sequence the tests correctly. The script ensures that all the API endpoints are tested sequentially, verifying the expected behaviour of each method.

**Additional Observations and Improvements**

* While the current implementation covers the necessary functionality, a few improvements could enhance the API:
* Error handling: The API could be further improved with more robust error handling, particularly for invalid data types or malformed JSON input.
* Database integration: In a real-world scenario, using a database to store the sensor data would make the API more scalable and persistent, compared to the in-memory storage used in this project.

**Development Environment: GitHub Codespaces**

For this project, I used GitHub Codespaces as my development environment. GitHub Codespaces offers several key benefits:

* Pre-configured environment: It provides a fully set-up environment, saving time on configuration and installation of dependencies.
* Cloud-based: It’s cloud-hosted, allowing me to develop from any device without worrying about local setup.
* Integration with GitHub: It integrates seamlessly with GitHub, making it easier to commit code, track changes, and collaborate with team members.
* Efficiency: GitHub Codespaces allowed me to focus on coding without needing to spend time managing the development environment itself, which increased my overall efficiency.

**Conclusion**

In this project, I implemented a RESTful API with multiple endpoints to handle sensor data, including adding, updating, retrieving, and deleting entries. I extended the functionality by adding PATCH and DELETE methods to support partial updates and entry deletion. Despite some challenges in handling list modification and ensuring data integrity, the solution works as expected. The addition of a shell script automated the testing process, making the API easier to verify and validate. Lastly, using GitHub Codespaces streamlined the development experience by providing a fully configured, cloud-based environment that is deeply integrated with GitHub.

**REFERENCES**

(PDF) an empirical study on evolution of API documentation. (n.d.). https://www.researchgate.net/publication/225147411\_An\_Empirical\_Study\_on\_Evolution\_of\_API\_Documentation

*GitHub Codespaces Overview*. GitHub Docs. (n.d.). https://docs.github.com/en/codespaces/overview

Lane, K. (2024, September 9). *Intro to apis: History of apis*. Postman Blog. https://blog.postman.com/intro-to-apis-history-of-apis/

What is rest api? (n.d.). https://www.visual-paradigm.com/guide/development/what-is-rest-api/