Development of microservices-based application

Technical Report

Table of Contents

Introduction

This introduction provides a clear and brief of the project developed which is a microservice to Monitor environmental parameters responsible for collecting data from different IoT sensors. The main goal of the project (microservices) on monitoring the environment is to focus on things like Temperature, Rainfall, Humidity, Air Pollution, and CO2 level on a real-time basis. Firstly, collecting the data required from the IoT sensors and using it in our logic to understand the weather, what's happening, and the changes around us in the environment, using some calculations and logic. It helps us making better decisions, protecting the environment and to make it more eco-friendly.

Furthermore, discussing how this project is built on its idea, what it does, how it has been tested, and the flow of the project. It also shows pictures of the project in action and discusses what went well and what could have been even better.

Project Description

The main goal of the project is to focus on monitoring the environment with the help of IoT sensors that help in collecting real-time data on different parameters such as temperature, rainfall, humidity, air pollution, and CO2 emissions. This helps people make smart choices and take action to solve environmental problems before they become serious. or something big that affects the environment. This microservice has been built using the C# programming language and the ASP.NET framework within the Visual Studio Code platform and SSMS (SQL Server Management Studio) is the database sued to store the data.

The First Component which is the Sensors project, comprises of classes that are responsible for generating data for the parameters such as temperature, rainfall, humidity, air pollution, and CO2 emissions within the specified ranges. In the second one, the Sensors API is the intermediary step or plays the role of middleman that helps as a communication layer with the sensor classes to get all the real-time data. This API project ensures that the data-driven from all the sensors is moved or transmitted into the Monitoring Station API. The final project of the microservice is the Monitoring station API project. This API project serves as the main central hub which is responsible for collecting data from the sensor APIs. Moreover, the monitoring Station API project contains features that help display warnings if the collected data falls under the predefined and mentioned thresholds.

In summary, this microservice project does collection of data, transmission, stores the collected data, and uses it for analysis purposes. This powers the environmental authorities to make decisions and to take protective measures in response to environmental conditions.

Database Schema

i) Setup: ASP.NET API Project to SQL Server

In this Project, we have used SSMS (SQL Server Management Studio) as the database to be connected to our project. SSMS is a user-friendly platform that makes it easy for upcoming developers and administrators to interact with databases. Once the required platforms are downloaded, the setup connection between our project and database is done. For this, we are passing the database connection strings and details in the appsettings.json file in our project, and configuring Entity Framework Core to the project to use this connection string that we set. Finally, we inject the DbContext into our API controllers, which helps us to interact with the database.

ii) Database Overview

The database name has been set to ‘monitoringstation’ and this database has been used in our project to store all the tables and data in it. The database schema for the microservice includes the table named 'EnvironmentalData' that stores the sensor's readings of the environmental parameters. This table contains fields such as 'Id', 'SensorNumber', 'Timestamp', 'Temperature', 'Rainfall', 'Humidity', 'AirQualityIndex', 'CO2Emissions', 'Parameter', and 'Value'. The 'Id' column set is the primary key that helps in uniquely identifying sensors and their readings.

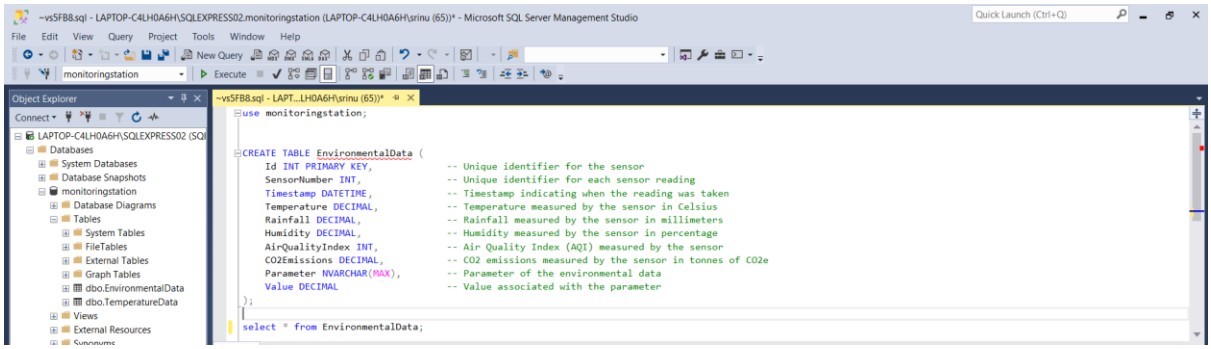

Fig. 1. DB overview in SSMS

Each row represents a specific sensor reading captured at a particular timestamp. This schema allows for the efficient storage and retrieval of environmental data collected by the microservice.

iii) Database Schema

The below mentioned schema allows for the efficient storage and retrieval of environmental data collected by the microservice.

Column Name	Data Type	Description
Id	INT	Unique identifier for the sensor data entry (Primary Key)
SensorNumber	INT	Unique identifier for each sensor reading
Timestamp	DATETIME	Timestamp indicating when the reading was taken
Temperature	DECIMAL	Temperature measured by the sensor in Celsius
Rainfall	DECIMAL	Rainfall measured by the sensor in millimeters
Humidity	DECIMAL	Humidity measured by the sensor in percentage
AirQualityIndex	INT	Air Quality Index (AQI) measured by the sensor
CO2Emissions	DECIMAL	CO2 emissions measured by the sensor in tonnes of CO2e
Parameter	NVARCHAR(MAX)	Parameter of the environmental data
Value	DECIMAL	Value associated with the parameter

API Development

i) API Endpoints and HTTP Methods

When I build and run the project locally, it will be assigned an HTTP port. For instance, if my port number is 5134, it will be set when the project is active. The link to access the API endpoints would then be:

http://localhost:5134/api/MonitoringStationAPI

GET Endpoints:

1. GET http://localhost:5134/api/MonitoringStationAPI

This GET endpoint is to retrieve all monitoring station data records.

2. GET http://localhost:5134/api/MonitoringStationAPI/{ID}

It retrieves a specific monitoring station data record by its ID.

POST Endpoint:

3. POST http://localhost:5134/api/MonitoringStationAPI

It creates a new monitoring station data record.

PUT Endpoint:

4. PUT http://localhost:5134/api/MonitoringStationAPI/{ID}

It updates an existing monitoring station data record.

DELETE Endpoint:

5. DELETE http://localhost:5134/api/MonitoringStationAPI/{ID}

It deletes a specific monitoring station data record.

ii) Request and Response Format

A request format is the context of data sent from a client to a server when performing a call request to the API. Some of the request formats used in our project are HTTP Method, Endpoint, Header, and body. As mentioned in the above context some of the HTTP methods used in our project are GET, POST, PUT, and DELETE. The endpoint is like the address where the API request is sent. Headers may contain some additional information about the request sent and the Body contains the actual data sent to the server, like POST and PUT.

iii) Error Handling with example

While working on this project we might face some errors while trying to test like '404 Not Found', '500 Internal Server Error', ‘400 Bad Request’, and so on; these are the common errors. To get the expected outcome, these errors should be fixed. You can resolve errors by carefully tracing the steps to pinpoint where they occur, understanding the reasons behind the errors, identifying the specific parts of the code that need to be fixed or corrected, and then retesting the application to ensure the issue has been resolved.

For instance, below is the explained example of the error, description and resolution of the error I faced.

Error: 404 Not Found.

Error Description: It indicates that the requested resource could not be found on the server.

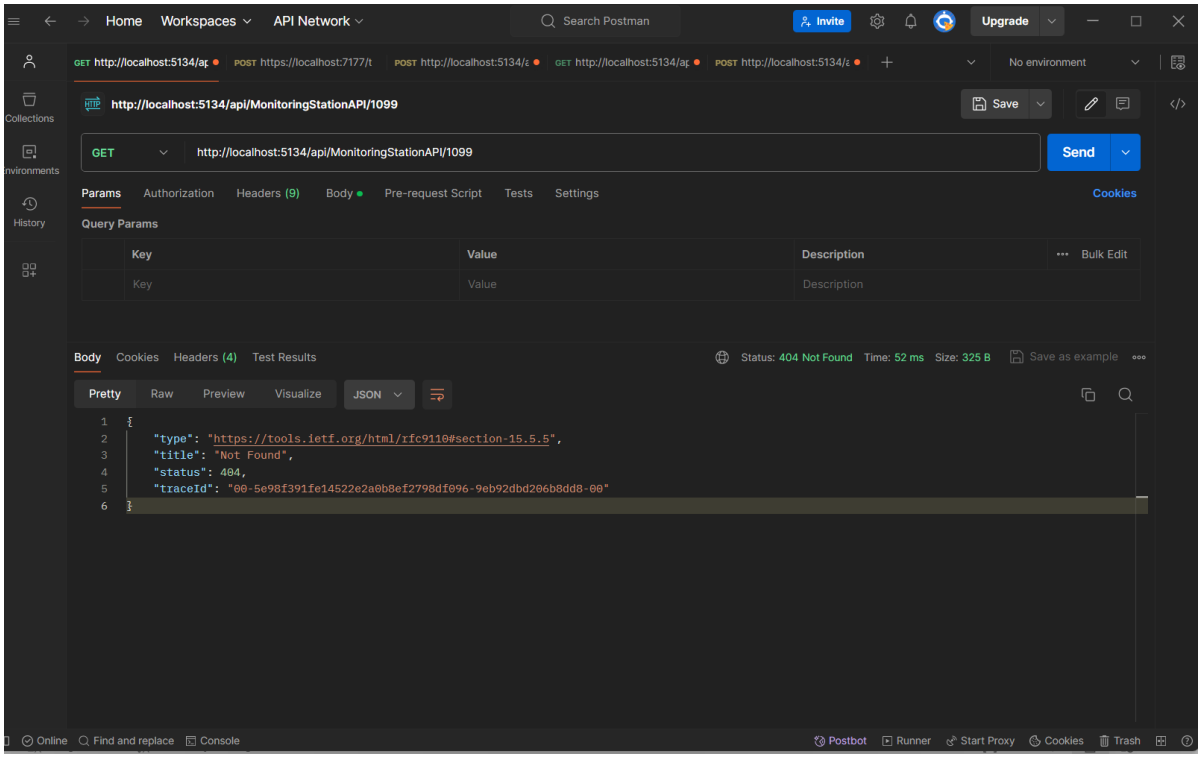

Fig. 2. 404 Not Found Error

Resolution: Upon investigating the error, I discovered that the ID I provided in the GET request did not exist in the database. Consequently, the API returned a "404 Not Found" error because it couldn't locate the requested resource. To rectify this issue, I ensured that I passed a valid primary key that corresponded to an existing record in the database. In this instance, the ID I initially passed was 1099, which was not present in the database. After correcting the ID to a valid value, I successfully received the desired output from the endpoint /api/MonitoringStationAPI/2.

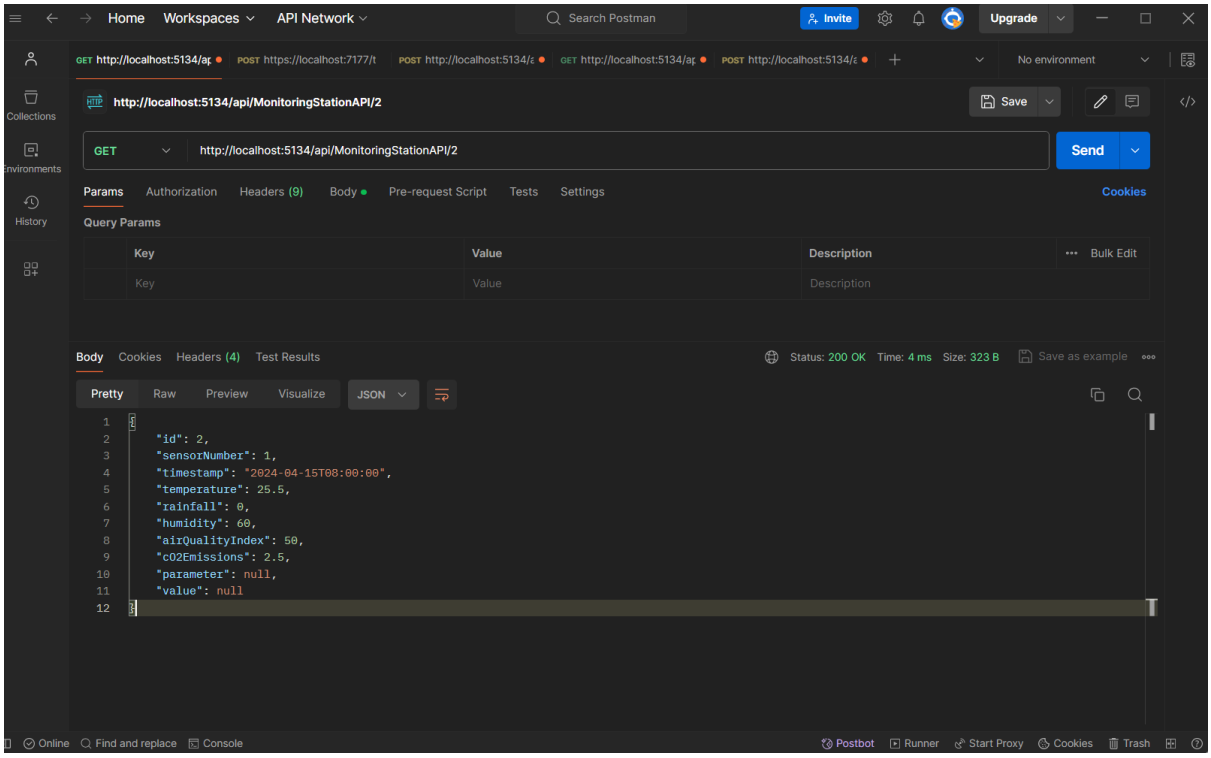

Fig. 3. Fix of '404 Not Found Error'

Testing

i) Testing Tools

I used POSTMAN and Swagger tools to test and ensure the API's functionality and reliability.

1. POSTMAN: Used for API testing, allowing comprehensive testing of various endpoints, request methods, and payloads. POSTMAN facilitated both manual testing and the creation of automated test suites.

2. Swagger: Utilized for interactive API documentation and testing. Swagger provided a user-friendly interface to explore and test API endpoints directly within the browser.

ii) Types of Tests

1. Unit Tests: Developed and executed unit tests to validate individual components and functionalities of the microservice, ensuring that each unit gives the output as expected.

2. Integration Tests: Conducted integration tests to verify the interaction and collaboration between different modules or services within the microservice architecture.

3. End-to-end Tests: Performed end-to-end tests to validate the entire flow of the application, from client requests to server responses, ensuring seamless operation across all components.

iii) Test results

Screenshots of the test results from POSTMAN and Swagger are provided below:

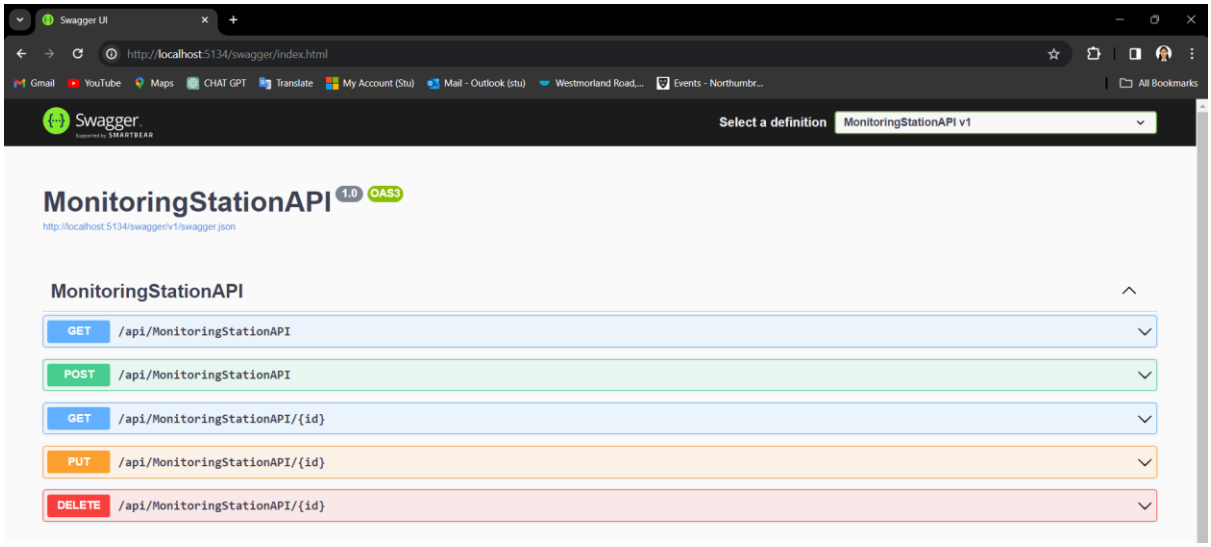

Fig. 4. Overall result (Tool: Swagger)

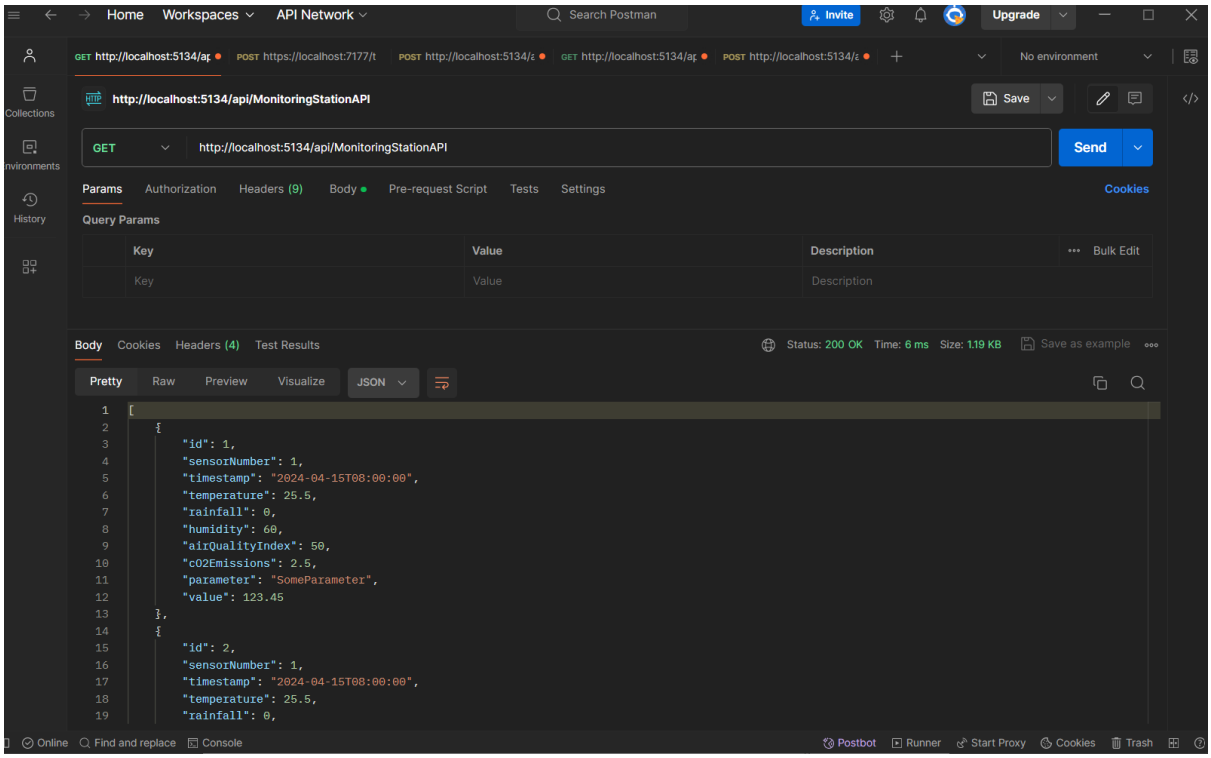

Fig. 5. Overall result (Tool: POSTMAN)

Testing Summary with Screenshots

1. Below attached is the screenshot of successfully executed part and the output for action ‘GET’ tested in both POSTAM and Swagger.

GET: http://localhost:5134/api/MonitoringStationAPI/31

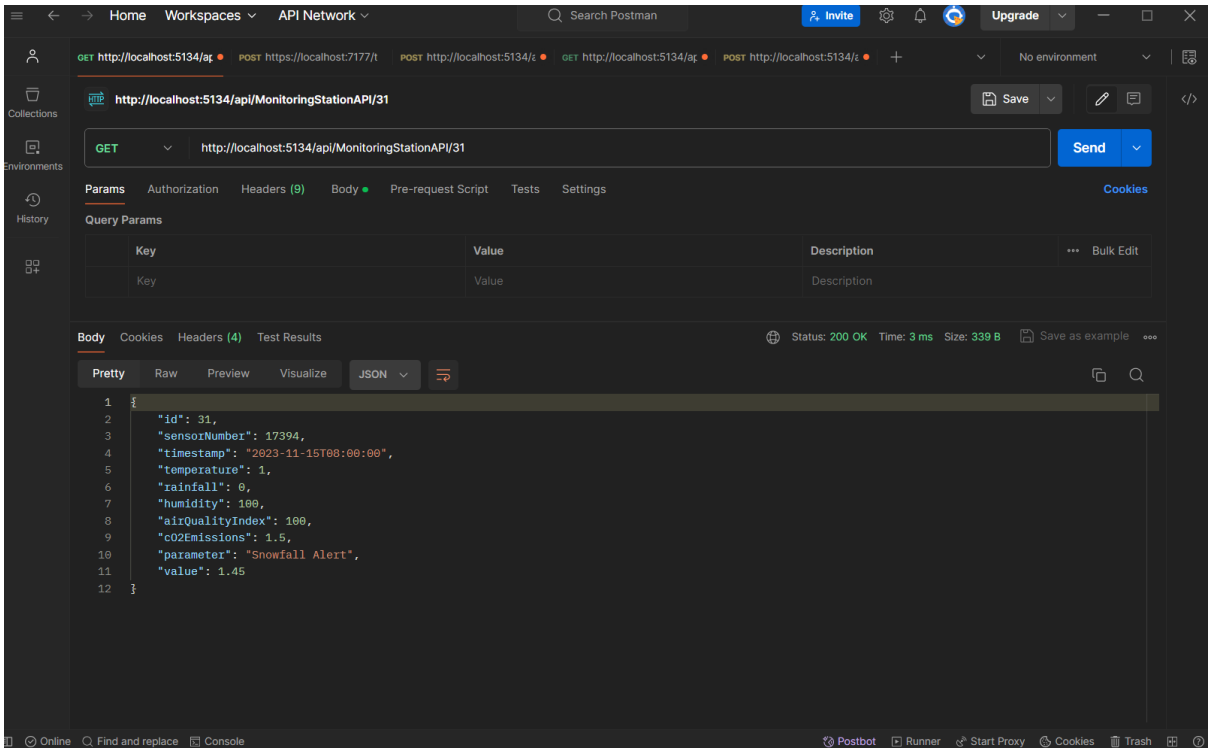

Fig. 6. GET with specific ID

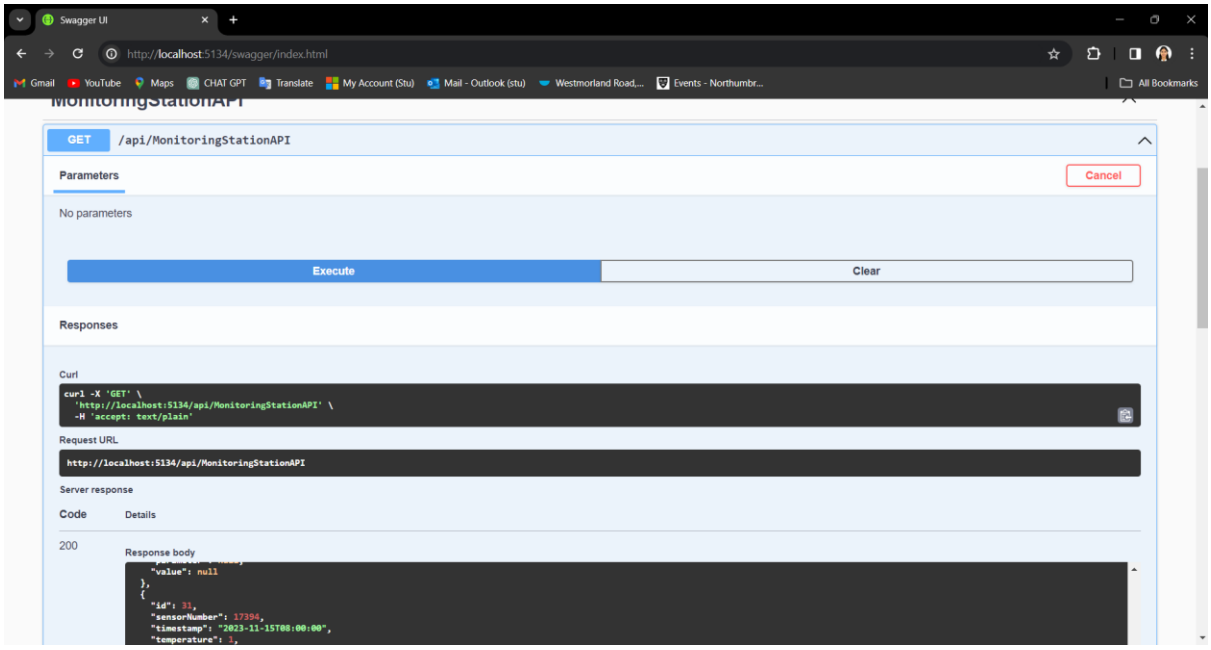

Fig. 7. Action GET tested in Swagger

2. Below attached is the screenshot of successfully executed part and the expected output for action ‘POST’ tested in both POSTAM and Swagger.

POST: http://localhost:5134/api/MonitoringStationAPI

BODY:
Specified in: raw; JSON


```
{
    "id": 493058,
    "sensorNumber": 1834907394,
    "timestamp": "2024-01-15T08:00:00",
    "temperature": 1,
    "rainfall": 0,
    "humidity": 100,
    "airQualityIndex": 100,
    "cO2Emissions": 1.5,
    "parameter": "Rainfall Alert!",
    "value": 1.45
}
```


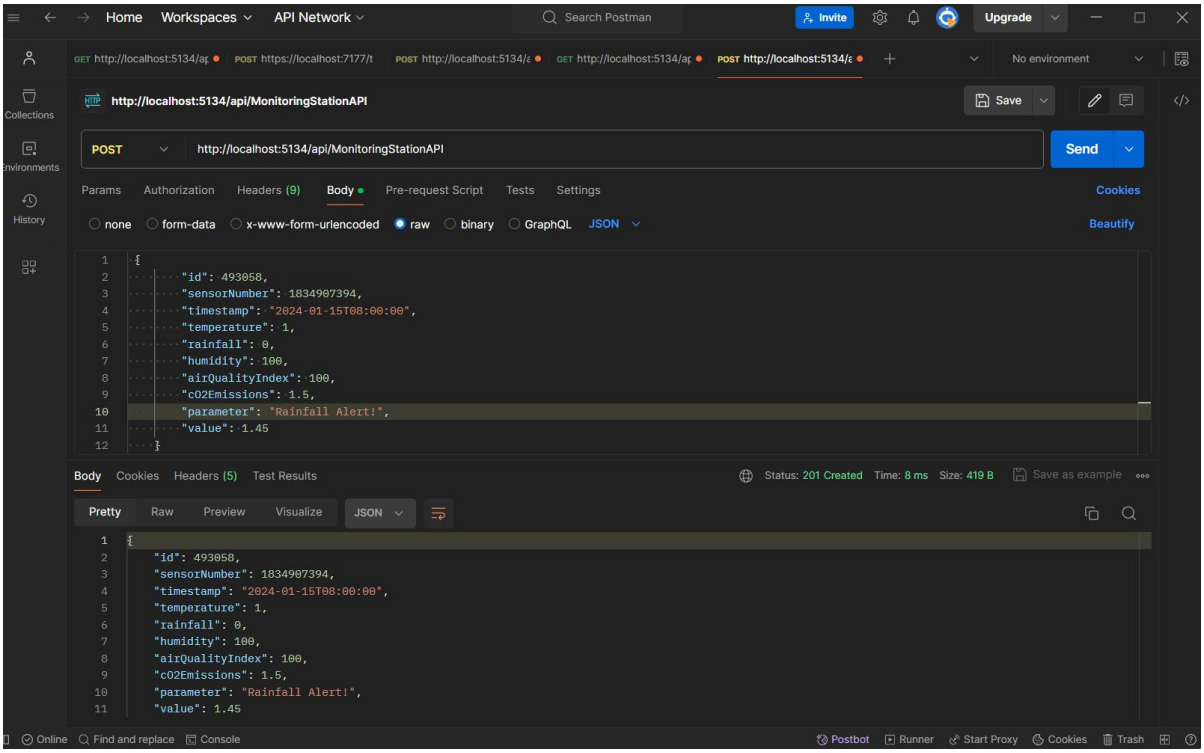

Fig. 8. Output of Action ‘POST’

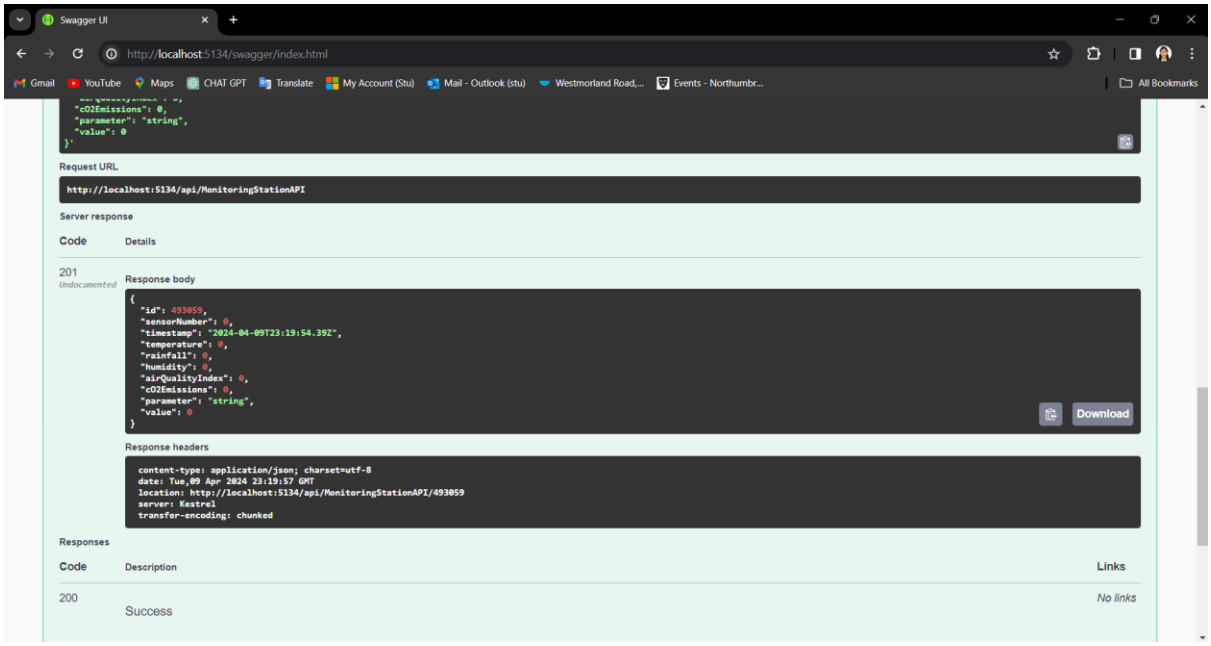

Fig. 9. Output of Action ‘POST’ in Swagger

3. Below attached is the screenshot of successfully executed part and the expected output for action ‘PUT’ tested in both POSTAM and Swagger.

PUT: http://localhost:5134/api/MonitoringStationAPI/1

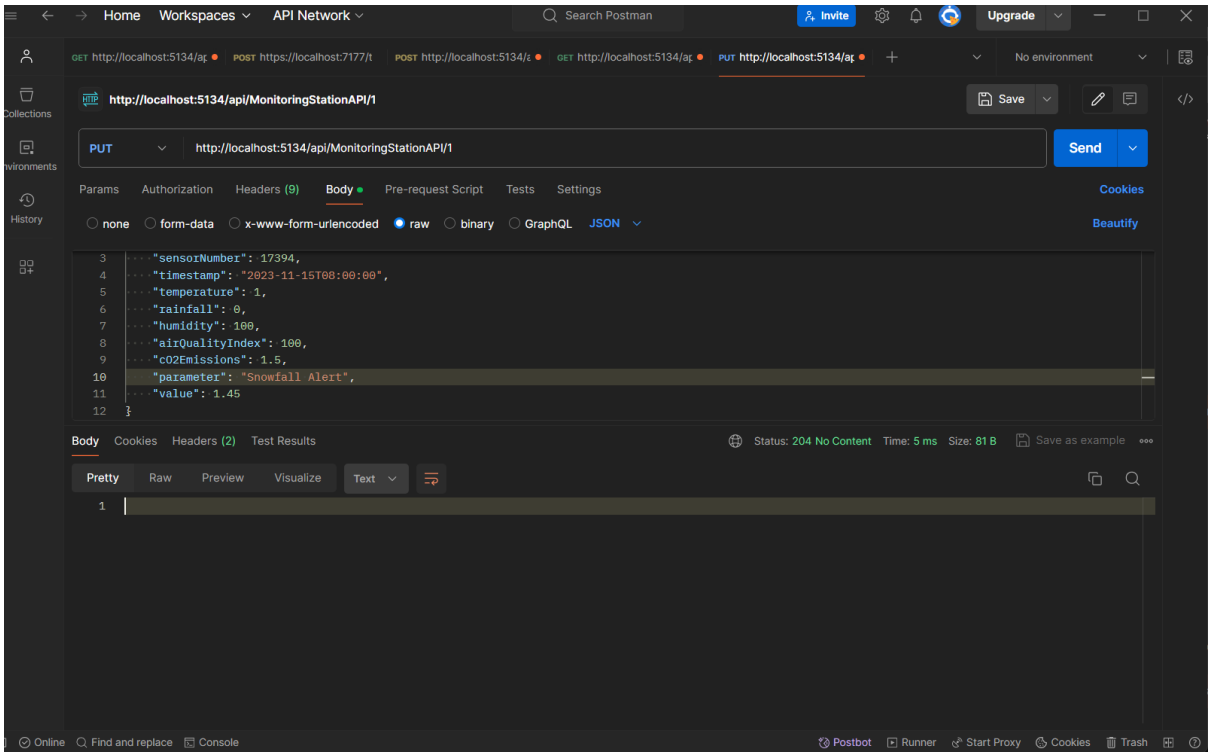

Fig. 10. Output of Action ‘PUT’ in POSTMAN

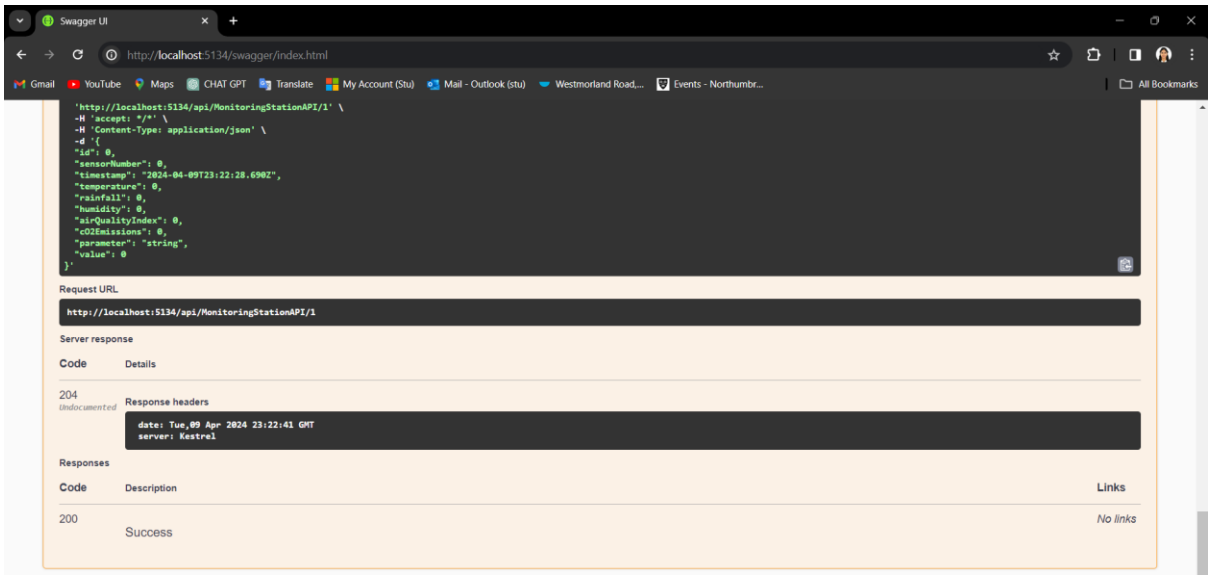

Fig. 11. Output of Action 'PUT' in Swagger

4. Below attached is the screenshot of successfully executed part and the expected output for action 'DELETE' tested in both POSTAM and Swagger.

DELETE: http://localhost:5134/api/MonitoringStationAPI/2

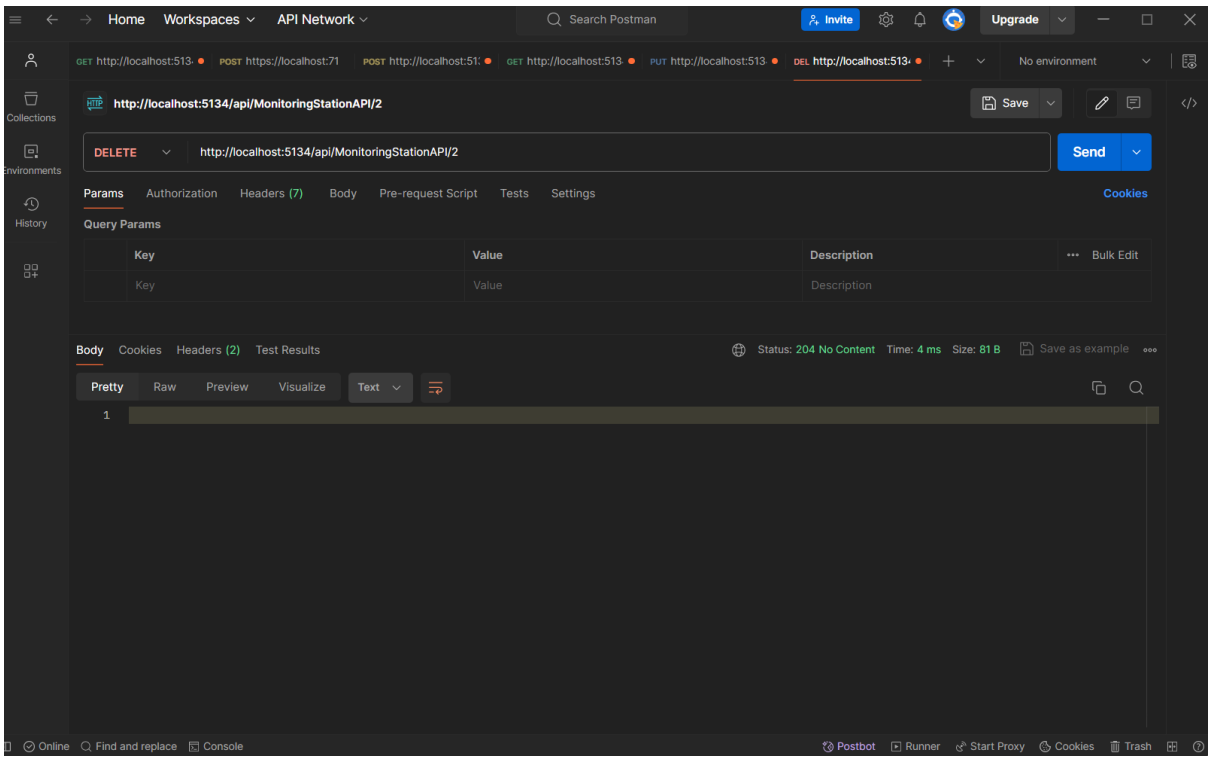

Fig. 12. Output of Action 'DELETE' in POSTMAN

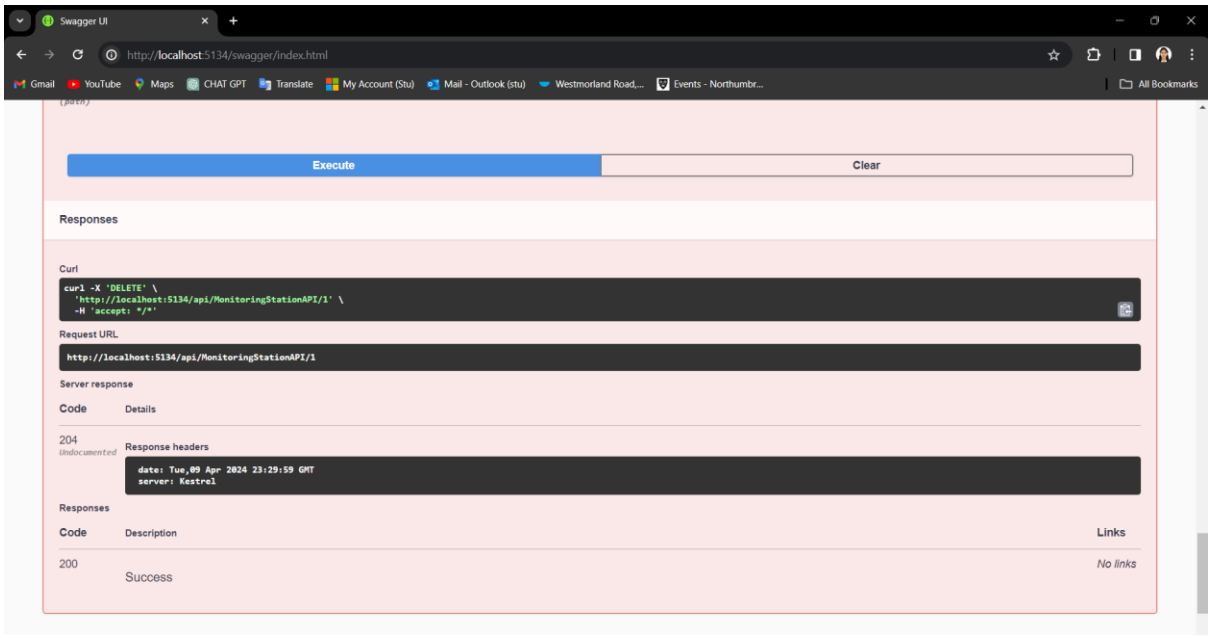

Fig. 13. Output of Action 'DELETE' in Swagger

Critical evaluation

i) Strengths

Throughout the development of the API project, many aspects were effectively managed, executed, and achieved. During the starting phase of the API project, research and planning to understand the project's requirements were set clear, because of this, it so was very helpful to major points like where to start, what's next on the flow, and things that must be fulfilled as per the requirements. Paying close attention to the project design like architecture and components was well-defined from the start. Choosing SQL Server for the database proved to be an excellent decision as it contributed to the stability and performance of the project. Furthermore, documenting each and every important note that has been followed in the project will make it easier for future maintenance and troubleshooting.

ii) Areas for Improvement in Project Development

Even testing the API project using tools like Postman and Swagger, still preferred to enhance the texting coverage by implementing more thorough unit and integration tests. Like incorporating automated testing frameworks in the development process which can be simplified and accelerate the testing process. This not only saves time but also increases the reliability of our testing, as it eliminates the possibility of human error.

References

https://learn.microsoft.com/en-us/ef/core/

https://learning.postman.com/docs/sending-requests/create-requests/test-data/

https://learn.microsoft.com/en-us/azure/azure-sql/database/connect-query-dotnet-core?view=azuresql

https://stackoverflow.com/questions/76949887/post-request-404-not-found-error-in-postman#:~:text=Check%20the%20URL%3A%20First%2C%20ensure,%3A8080%2Fadd%2Dcontact

https://community.intercom.com/api-webhooks-23/400-bad-request-error-but-works-fine-in-postman-4789