REST API JPA CRUD Spring for Sensor Data Management in Autonomous Vehicles

Frida Cano Falcón

Java Backend Academy MTY
August 2024

Week 4

IndexSeptember 7, 2024

Index

1. Introduction

Self-driving cars rely heavily on a multitude of sensors like cameras, LiDAR, radar, and ultrasonic sensors to perceive their surroundings and make real-time decisions. This project presents a REST API that enables the management of sensor data for a self-driving vehicle. The system allows for creating, retrieving, updating, and deleting (CRUD) sensor information. The backend is developed using Java 17, Spring Boot, JPA, and MySQL.

2. Objectives

The primary objectives of this project are:

- **Build a REST API** for managing sensor data used in autonomous vehicle systems.
- **Implement CRUD operations** for sensor entities, allowing users to create, read, update, and delete sensors.
- **Use JPA for database management** to persist sensor data in a MySQL database.
- **Ensure scalability and modularity**, allowing easy integration of more sensors and future features.

3. Technology Stack

Backend Technologies

- **Java 17**: The programming language used for building the API.
- **Spring Boot 3.3.3**: A framework for rapid development of Java-based REST APIs.
- **JPA (Java Persistence API)**: A specification for handling database operations through objects.
- **MySQL**: The relational database used to store sensor data.

Tools

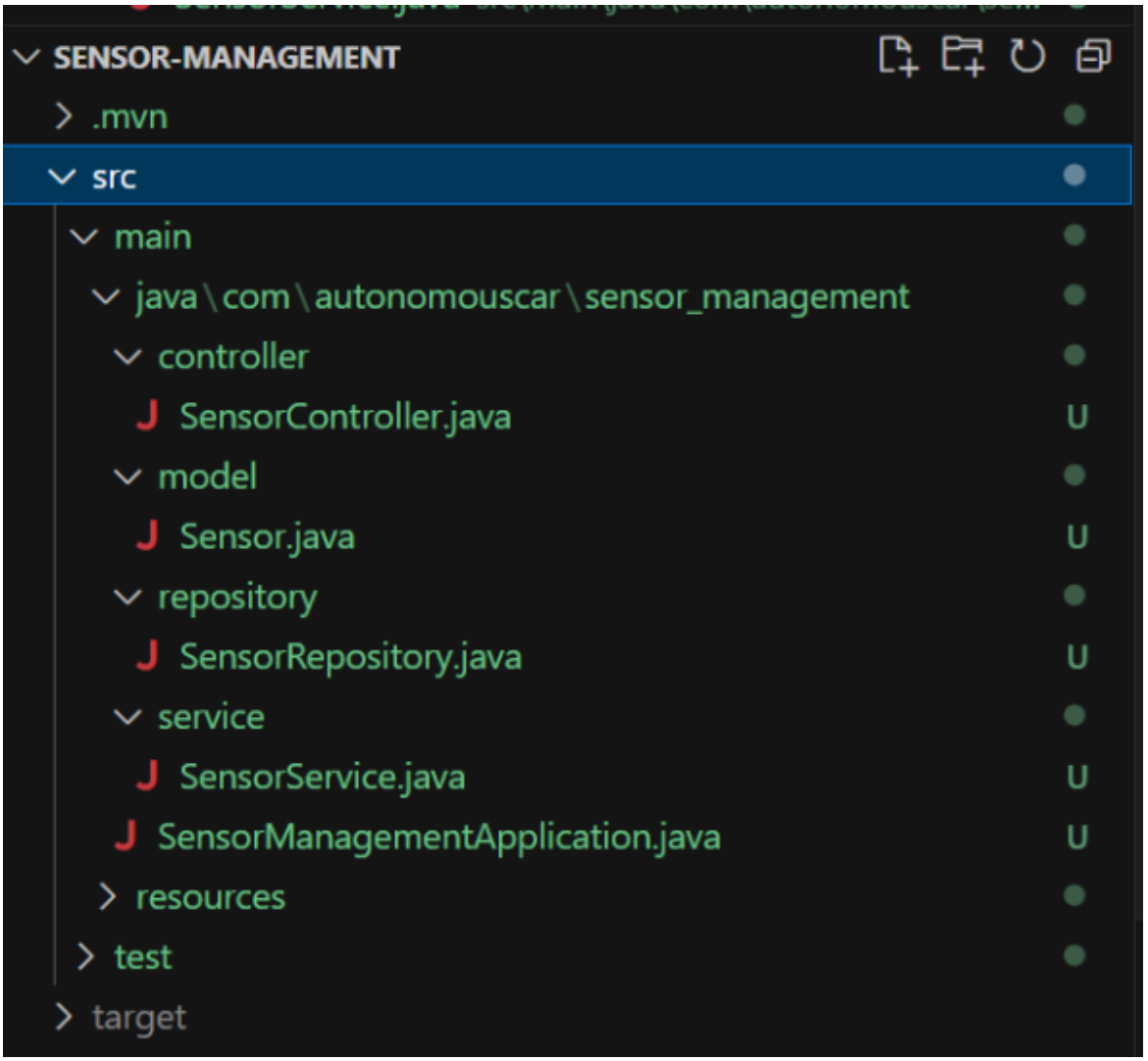
- **Visual Studio Code**: IDE for development.
- **Spring Initializr**: Used to initialize the Spring Boot project.
- **Maven**: Build and dependency management tool.
- **Postman**: For testing API endpoints.

4. Project Design and Structure

The project follows the **Model-View-Controller (MVC)** pattern:

- **Model**: Defines the sensor entity and its fields.
- **Repository**: Provides the interface to interact with the database.
- **Service**: Contains business logic and data processing.
- **Controller**: Manages HTTP requests and provides appropriate responses.

Project Structure



5. System Components

5.1. Sensor Entity

The *Sensor.java* class represents the sensor data that is stored in the MySQL database. It has the following fields:

- **ID**: Unique identifier for the sensor.
- **Sensor Name**: Name of the sensor.
- **Sensor Type**: Type of sensor (Camera, LiDA etc.).
- **Status**: The operational status of the sensor (calibrated, not calibrated).

src > main > java > com > autonomouscar > sensor_management > model > J S

```java
package com.autonomouscar.sensor_management.model;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

@Entity
public class Sensor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name="sensor_name")
    @Getter
    @Setter
    private String sensorName;

    @Column(name="sensor_type")
    @Getter
    @Setter
    private String sensorType;

    @Column(name="status")
    @Getter
    @Setter
    private String status;
```

```
29      public Sensor() {
30      }
31
32      public Sensor(String sensorName, String sensorType, String status) {
33          this.sensorName = sensorName;
34          this.sensorType = sensorType;
35          this.status = status;
36      }
37  }
```

Implementation of Sensor Entity

5.2. Repository Layer

The SensorRepository.java is an interface that extends JpaRepository. This is responsible for interacting with the database. It uses methods provided by Spring Data JPA such as save(), findAll(), findById(), deleteById(), etc.

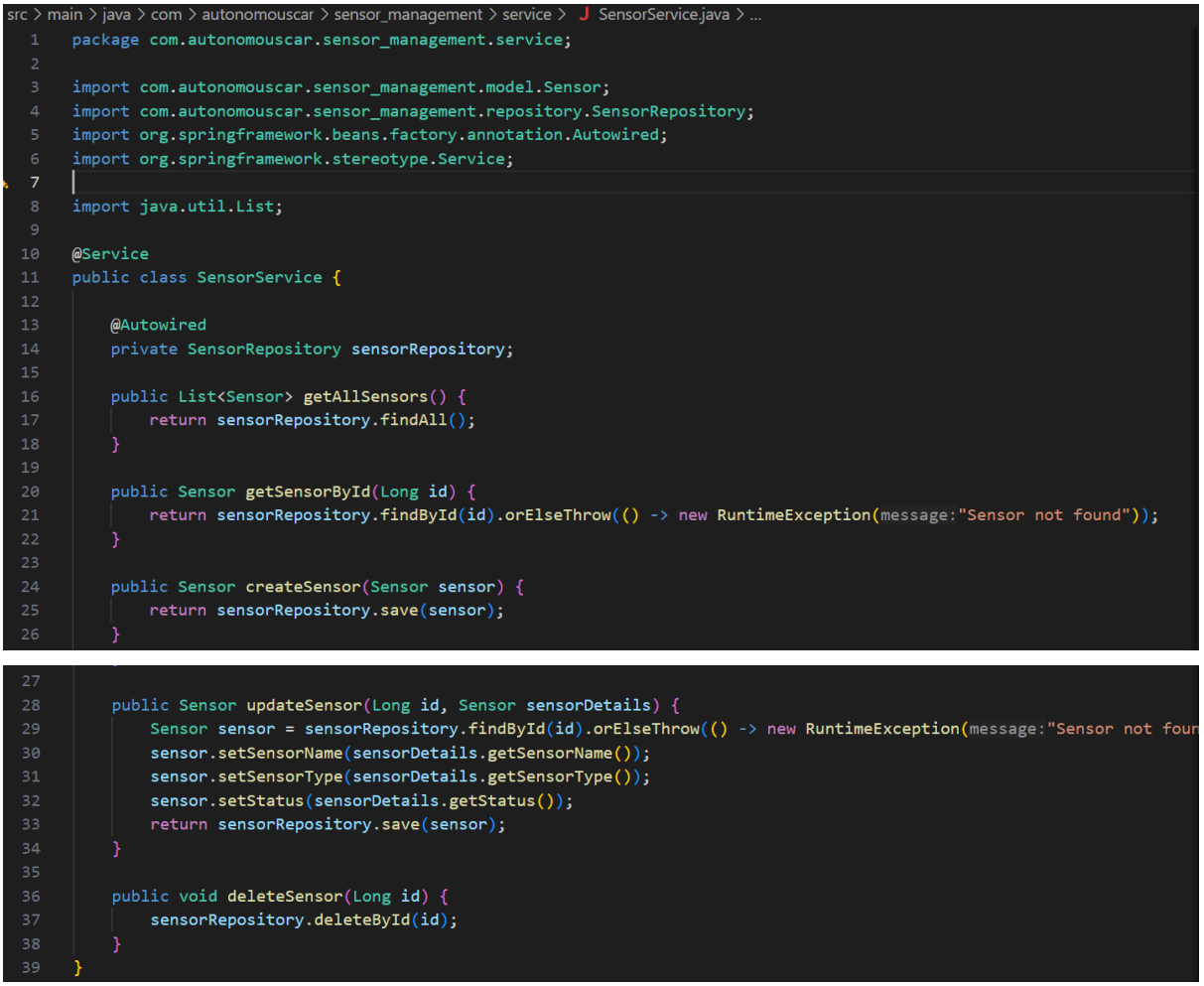
src > main > java > com > autonomouscar > sensor_management > repository > SensorRepository.java

```
1   package com.autonomouscar.sensor_management.repository;
2
3   import com.autonomouscar.sensor_management.model.Sensor;
4   import org.springframework.data.jpa.repository.JpaRepository;
5   import org.springframework.stereotype.Repository;
6
7   @Repository
8   public interface SensorRepository extends JpaRepository<Sensor, Long> {
9   }
```

Implementation of Repository Layer

5.3. Service Layer

The SensorService.java contains the business logic of the application. It manages operations related to sensor data like adding, retrieving, updating, and deleting sensors.



```
src > main > java > com > autonomouscar > sensor_management > service > SensorService.java > ...
1   package com.autonomouscar.sensor_management.service;
2
3   import com.autonomouscar.sensor_management.model.Sensor;
4   import com.autonomouscar.sensor_management.repository.SensorRepository;
5   import org.springframework.beans.factory.annotation.Autowired;
6   import org.springframework.stereotype.Service;
7
8   import java.util.List;
9
10  @Service
11  public class SensorService {
12
13      @Autowired
14      private SensorRepository sensorRepository;
15
16      public List<Sensor> getAllSensors() {
17          return sensorRepository.findAll();
18      }
19
20      public Sensor getSensorById(Long id) {
21          return sensorRepository.findById(id).orElseThrow(() -> new RuntimeException(message:"Sensor not found"));
22      }
23
24      public Sensor createSensor(Sensor sensor) {
25          return sensorRepository.save(sensor);
26      }
27
28      public Sensor updateSensor(Long id, Sensor sensorDetails) {
29          Sensor sensor = sensorRepository.findById(id).orElseThrow(() -> new RuntimeException(message:"Sensor not four
30          sensor.setSensorName(sensorDetails.getSensorName());
31          sensor.setSensorType(sensorDetails.getSensorType());
32          sensor.setStatus(sensorDetails.getStatus());
33          return sensorRepository.save(sensor);
34      }
35
36      public void deleteSensor(Long id) {
37          sensorRepository.deleteById(id);
38      }
39  }
```

Implementation of Service Layer

5.4. Controller Layer

The SensorController.java is a REST controller that handles incoming HTTP requests and returns appropriate responses.

- **GET /all**: Fetches all sensors.
- **POST /create**: Adds a new sensor.
- **PUT /update/{id}**: Updates an existing sensor by ID.
- **DELETE /delete/{id}**: Deletes a sensor by ID.

src > main > java > com > autonomouscar > sensor_management > controller > SensorController.java > SensorController > deleteSensor(Long)

```java
package com.autonomouscar.sensor_management.controller;

import com.autonomouscar.sensor_management.model.Sensor;
import com.autonomouscar.sensor_management.service.SensorService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/sensors")
public class SensorController {

    @Autowired
    private SensorService sensorService;

    @GetMapping()
    public String index() {
        return "Welcome to the Sensor Management System!";
    }

    @GetMapping("/all")
    public List<Sensor> getAllSensors() {
        return sensorService.getAllSensors();
    }

    @GetMapping("/oneSensor/{id}")
    public ResponseEntity<Sensor> getSensorById(@PathVariable Long id) {
        return ResponseEntity.ok(sensorService.getSensorById(id));
    }

    @PostMapping("/create")
    public String createSensor(@RequestBody Sensor sensor) {
        sensorService.createSensor(sensor);
        return "Sensor created successfully!";
    }

    @PutMapping("/update/{id}")
    public String updateSensor(@PathVariable Long id, @RequestBody Sensor sensorDetails) {
        sensorService.updateSensor(id, sensorDetails);
        return "Sensor updated successfully!";
    }

    @DeleteMapping("/delete/{id}")
    public String deleteSensor(@PathVariable Long id) {
        sensorService.deleteSensor(id);
        ResponseEntity.noContent().build();
        return "Sensor deleted successfully!";
    }
}
```

Implementation of Controller Layer

6. Database Configuration and Initialization

6.1. Database Setup

```sql
create database if not exists sensor_management;

USE sensor_management;

drop table if exists sensor;

CREATE TABLE `sensor` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `sensor_name` VARCHAR(255) DEFAULT NULL,
    `sensor_type` VARCHAR(255) DEFAULT NULL,
    `status` VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
```

Database setup in MySQL Workbench 8.0

6.2. Database Configuration in Spring Boot

The database connection information is stored in the application.properties file.

src > main > resources > application.properties

```
spring.application.name=sensor-management
spring.datasource.url=jdbc:mysql://localhost:3306/sensor_management
spring.datasource.username=jpa_project_W4
spring.datasource.password=jpa_project_W4

server.port = 8081
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Application Properties File

6.3. Data Initialization

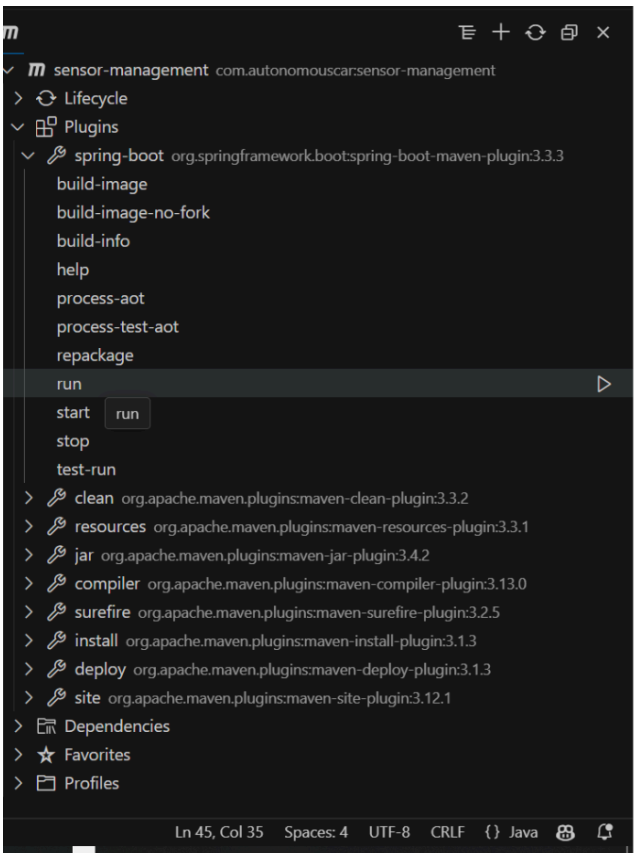
You can populate the database with sample sensor data:

```sql
INSERT INTO `sensor` (`id`,`sensor_name`,`sensor_type`,`status`) VALUES (1,'multisense','camera', 'calibrated');
INSERT INTO `sensor` (`id`,`sensor_name`,`sensor_type`,`status`) VALUES (2,'velodyne','LiDAR', 'calibrated');
```

Example of a initialization of data

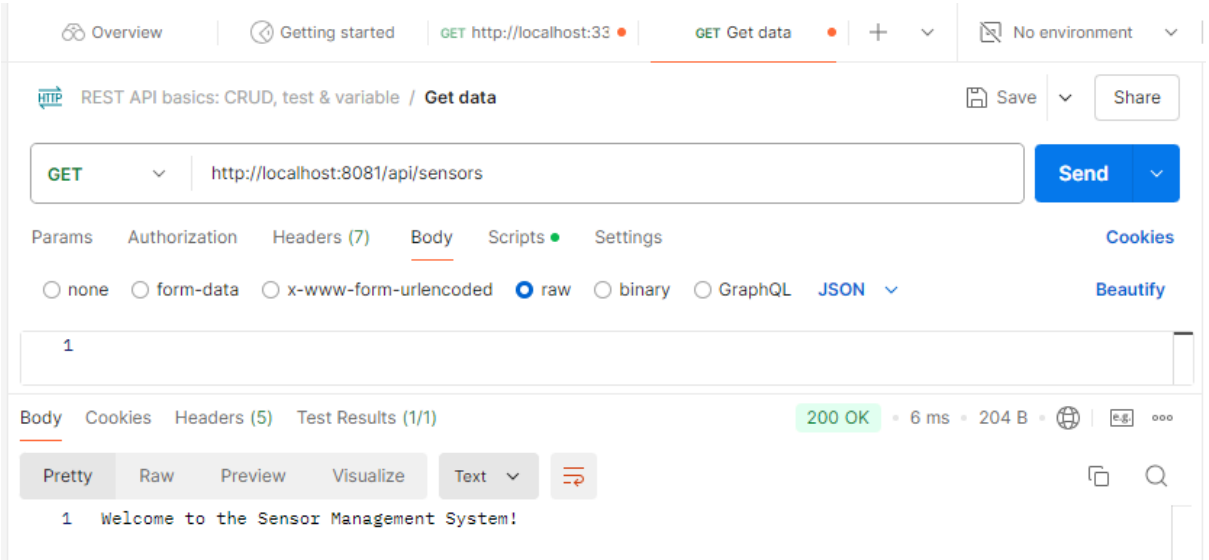
7. Testing the API

To run the Spring Boot, we use the Maven control, the *run* pluggin:



Maven Controls in VS Code IDE

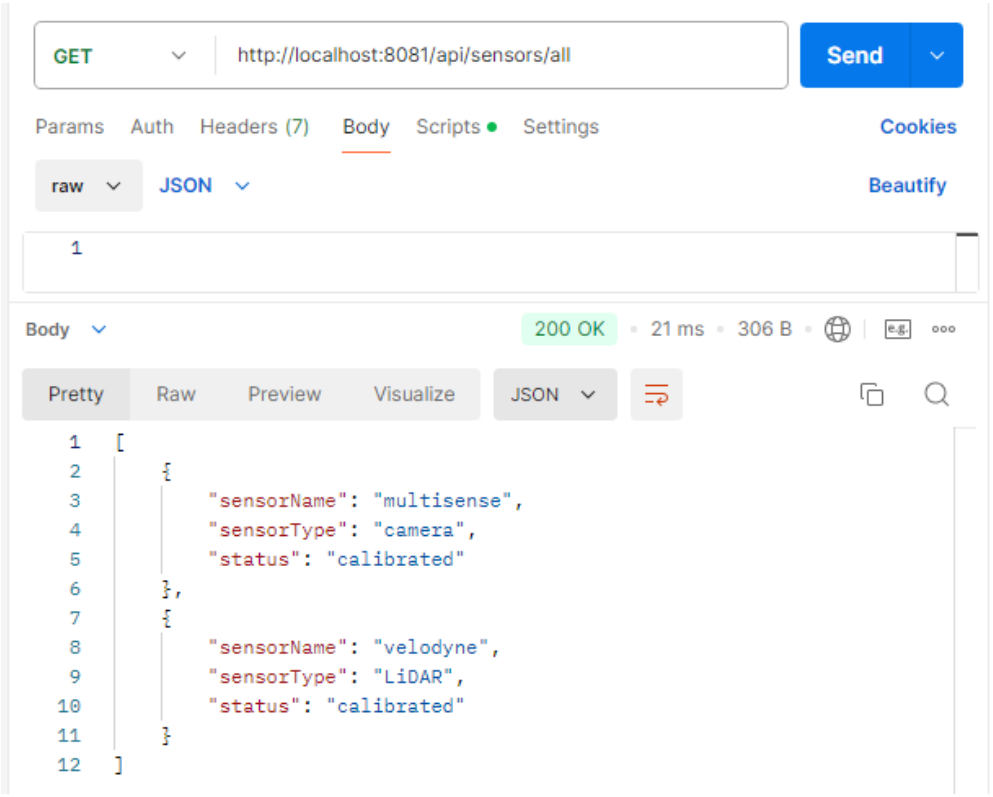
You can test the REST API using **Postman**



First Page

GET /all

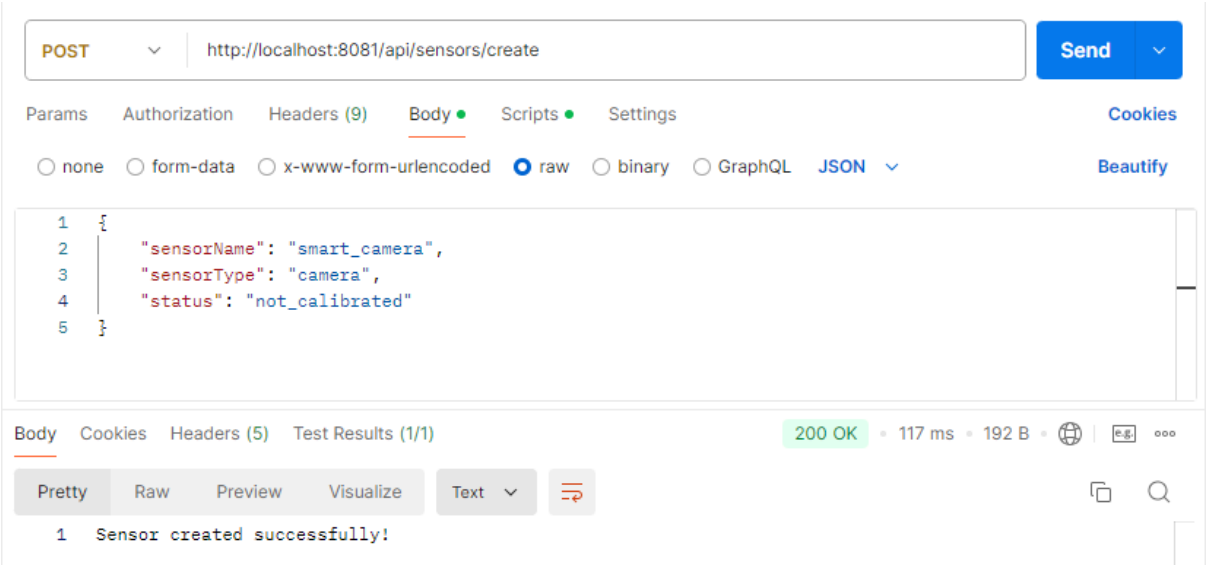
Retrieve all sensors:



See all the sensors in Postman

POST /create

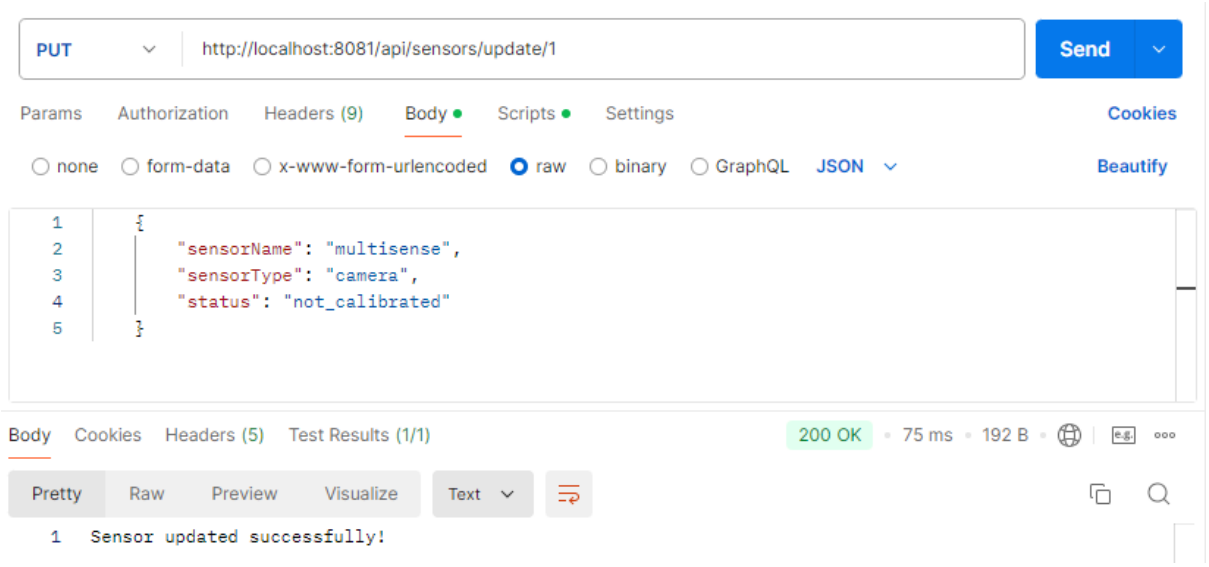
Create a new sensor:



Creating a sensor in Postman

PUT /update/{id}

Update an existing sensor:



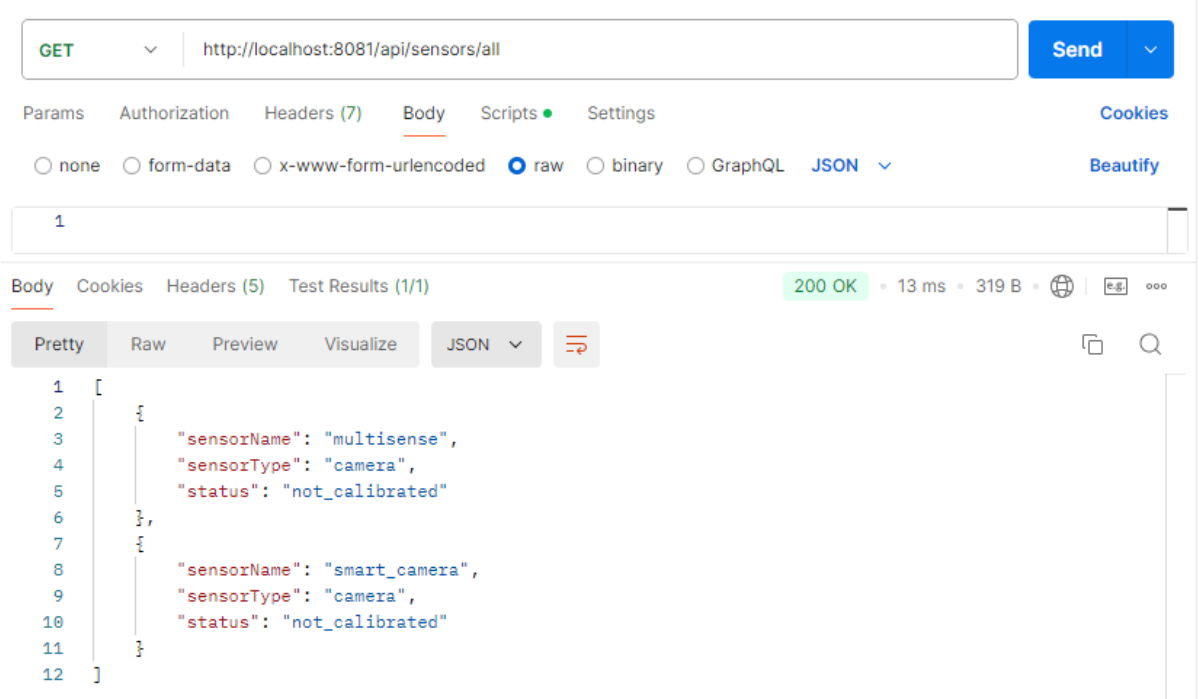
Updating the sensor 1 in Postman

DELETE /delete/{id}

Delete a sensor by ID:

Deleting the sensor 2 in Postman

Check the changes applied:



Deploy of all the sensors of the database updated in Postman

8. Future Enhancements

- **Authentication & Authorization**: Implementing security features using Spring Security to ensure only authorized users can modify sensor data.
- **Monitoring & Logging**: Adding proper logging for better diagnostics and performance monitoring.
- **Scaling the System**: Supporting multiple vehicles by associating sensors with specific vehicles.

9. Conclusion

This project successfully implements a REST API for managing sensor data in self-driving vehicles. It demonstrates how Spring Boot and JPA can be used to build scalable and maintainable backend services. By using a layered architecture (controller, service, repository), the system ensures separation of concerns and modularity. The project can serve as a foundation for building more complex autonomous vehicle data management systems.