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

Frida Cano Falcón

Java Backend Academy MTY
August 2024

Week 4

September 7, 2024

Index

1. Introduction

Self-driving cars rely heavily on various sensors like cameras, LiDAR, radar, and ultrasonic sensors to perceive their surroundings and make real-time decisions. This project presents a REST API designed to manage sensor data for autonomous vehicles. The system facilitates the creation, retrieval, updating, and deletion (CRUD) of sensor information. The backend is developed using Java 17, Spring Boot, and JPA (Java Persistence API), with MySQL as the database.

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 additional 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

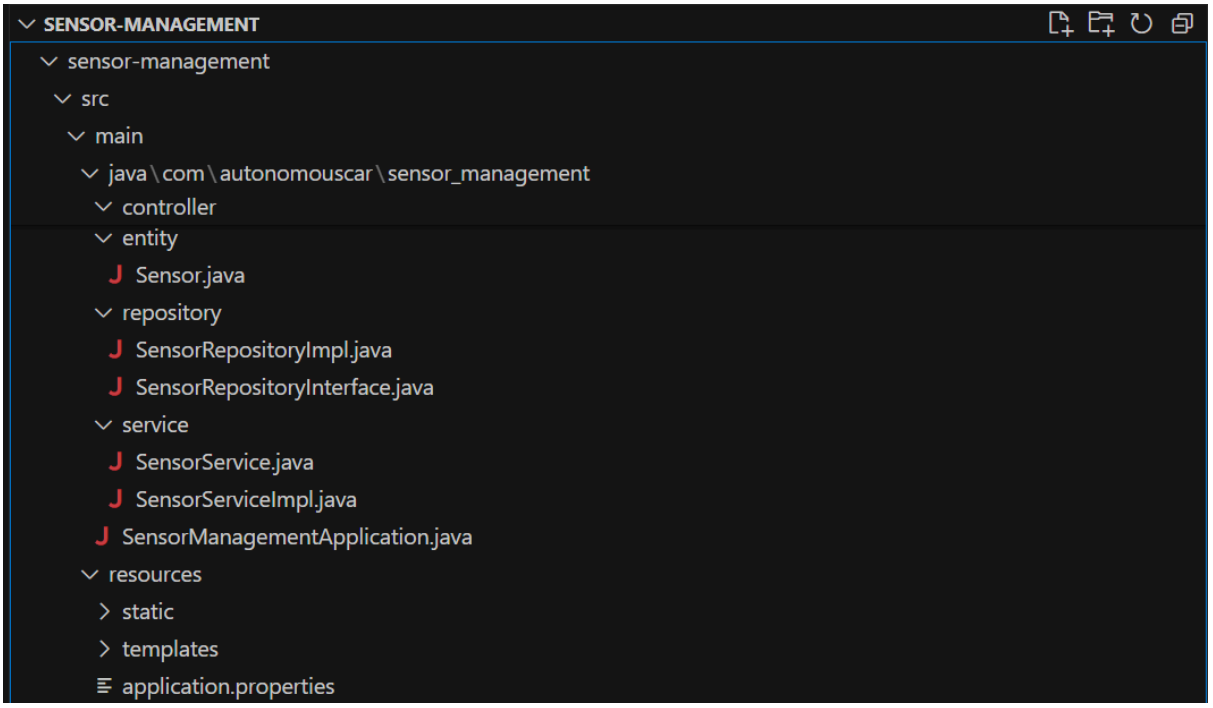
- **Visual Studio Code:** IDE for development.
- **Spring Initializr:** Used to initialize the Spring Boot project.
- **Maven:** Build and dependency management tool.
- **Postman:** For testing API endpoints.

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 stored in the MySQL database. It includes the following fields:

- **ID:** Unique identifier for the sensor.
- **Sensor Name:** Name of the sensor (e.g., Camera, LiDAR).
- **Sensor Type:** Type of sensor (Optical, Radar, etc.).
- **Status:** The operational status of the sensor (Active, Inactive).

sensor-management > src > main > java > com > autonomouscar > sensor_management > entity > Sensor.java > Sensor

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

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

@Entity
public class Sensor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    @Getter
    @Setter
    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;

    public Sensor() {
    }

    public Sensor(String sensorName, String sensorType, String status) {
        this.sensorName = sensorName;
        this.sensorType = sensorType;
        this.status = status;
    }
}
```

Implementation of Sensor Entity

5.2. Repository Layer

The *SensorRepositoryInterface.java* is an interface that defines the CRUD operations. The *SensorRepositoryImpl.java* provides the implementation using *EntityManager* for database interactions.

Interface: *SensorRepositoryInterface.java*

sensor-management > src > main > java > com > autonomouscar > sensor_management > repository > J SensorRepositoryInterface.java > ...

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

import com.autonomouscar.sensor_management.entity.Sensor;

import java.util.List;

public interface SensorRepositoryInterface {

    List<Sensor> findAll();

    Sensor findById(int id);

    Sensor save(Sensor sensor);

    void deleteById(int id);

}
```

Implementation of Sensor Repository Interface

Implementation: *SensorRepositoryImpl.java*

sensor-management > src > main > java > com > autonomouscar > sensor_management > repository > J SensorRepositoryImpl.java > SensorRepositoryImpl >

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

import com.autonomouscar.sensor_management.entity.Sensor;

import jakarta.persistence.EntityManager;
import jakarta.persistence.TypedQuery;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class SensorRepositoryImpl implements SensorRepositoryInterface {

    // define field for entitymanager
    private EntityManager entityManager;

    // set up constructor injection
    @Autowired
    public SensorRepositoryImpl(EntityManager TheEntityManager) {
        entityManager = TheEntityManager;
    }

```

```java
    @Override
    public List<Sensor> findAll() {
        // create a query
        TypedQuery<Sensor> theQuery = entityManager.createQuery(qlString:"FROM Sensor", resultClass:Sensor.class);

        // execute query and get result list
        List<Sensor> sensors = theQuery.getResultList();

        // return the results
        return sensors;
    }

    @Override
    public Sensor findById(int id) {
        return entityManager.find(entityClass:Sensor.class, id);
    }

    @Override
    public Sensor save(Sensor sensor) {
        return entityManager.merge(sensor);
    }

    @Override
    public void deleteById(int id) {
        entityManager.remove(findById(id));
    }

}
```

Implementation of Sensor Repository Implementation

5.3. Service Layer

The *SensorService.java* contains the business logic for managing sensor data, and *SensorServiceImpl.java* provides the implementation.

Interface: *SensorService.java*

sensor-management > src > main > java > com > autonomouscar > sensor_management > service > SensorService.java > SensorService

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

import java.util.List;

import com.autonomouscar.sensor_management.entity.Sensor;

public interface SensorService {

    List<Sensor> findAll();

    Sensor findById(int id);

    Sensor save(Sensor sensor);

    void deleteById(int id);


}
```

Implementation of Sensor Service Interface

Implementation: *SensorServiceImpl.java*

sensor-management > src > main > java > com > autonomouscar > sensor_management > service > SensorServiceImpl.java > SensorServiceImpl

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

import java.util.List;

import com.autonomouscar.sensor_management.entity.Sensor;
import com.autonomouscar.sensor_management.repository.SensorRepositoryInterface;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class SensorServiceImpl implements SensorService {

    private SensorRepositoryInterface sensorRepository;

    @Autowired
    public SensorServiceImpl(SensorRepositoryInterface TheSensorRepository) {
        sensorRepository = TheSensorRepository;
    }

    @Override
    public List<Sensor> findAll() {
        return sensorRepository.findAll();
    }

    @Override
    public Sensor findById(int id) {
        return sensorRepository.findById(id);
    }

    @Transactional
    @Override
    public Sensor save(Sensor sensor) {
        return sensorRepository.save(sensor);
    }

    @Transactional
    @Override
    public void deleteById(int id) {
        sensorRepository.deleteById(id);
    }

}
```

Implementation of Sensor Service Implementation

5.4. Controller Layer

The *SensorController.java* manages HTTP requests and provides RESTful endpoints.

REST Endpoints

- **GET /api/sensors**: Fetches all sensors.
- **GET /api/sensors/{id}**: Fetches a single sensor by ID.
- **POST /api/sensors/add**: Adds a new sensor.
- **PUT /api/sensors/update/{id}:** Update a single sensor by ID.
- **DELETE /api/sensors/delete/{id}**: Deletes a sensor by ID.

6. Database Configuration and Initialization

6.1. Database Setup

Create a database named *sensor_management* in MySQL.

```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 details are specified in *application.properties*.

```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
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

server.port=8082

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

Sample sensor data can be added using the *data.sql* file.

```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');
```

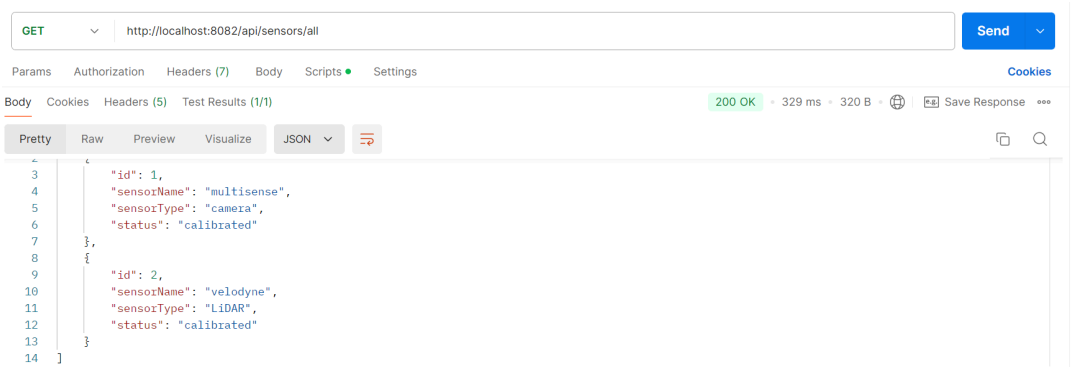
Example of a initialization of data

7. Testing the API

You can test the REST API using Postman or cURL.

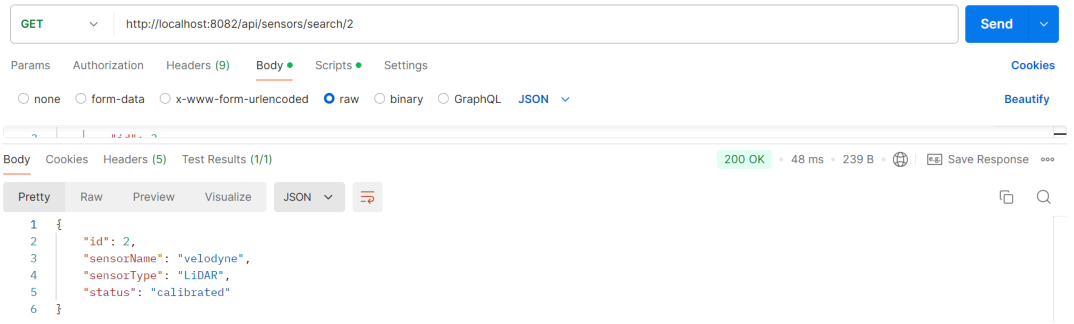
Test Endpoints

- **GET /api/sensors/all**: Fetches all sensors.



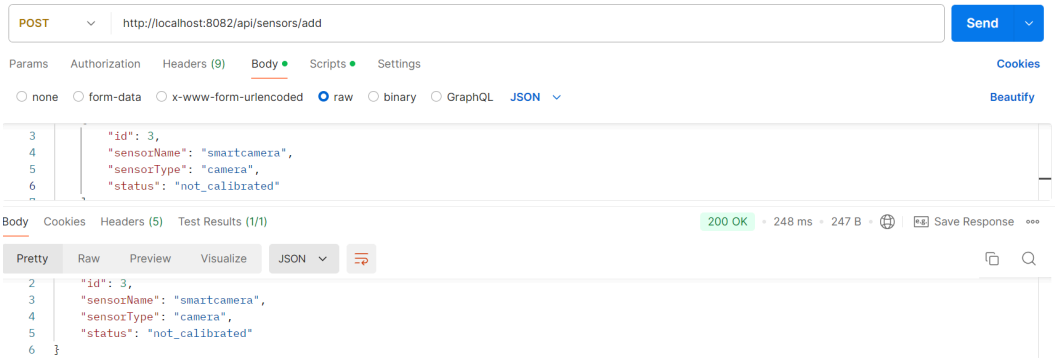
See all the sensors in Postman

- **GET /api/sensors/search/{id}**: Fetches a single sensor by ID.



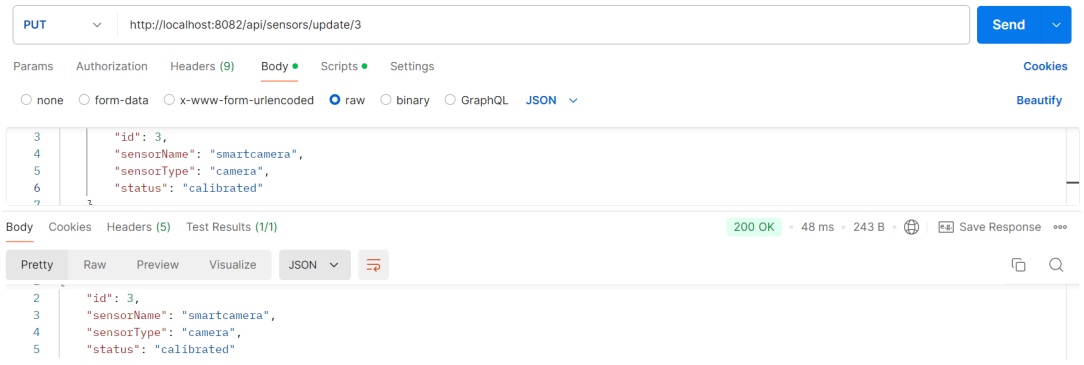
Searching a sensor in Postman

- **POST /api/sensors/add**: Adds a new sensor.



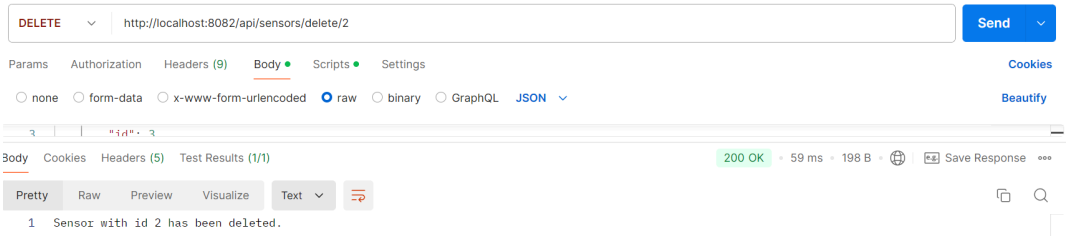
Creating a sensor in Postman

- **PUT /api/sensors/update/{id}:** Update a single sensor by ID.



Updating the sensor 1 in Postman

- **DELETE /api/sensors/delete/{id}**: Deletes a sensor by ID.



Deleting the sensor 2 in Postman

8. Future Enhancements

- **Authentication & Authorization:** Implement security features using Spring Security to ensure only authorized users can modify sensor data.
- **Monitoring & Logging:** Add logging for better diagnostics and performance monitoring.
- **Scaling the System:** Support 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 the use of Spring Boot and JPA to build scalable and maintainable backend services. By employing a layered architecture (controller, service, repository), the system ensures a clear separation of concerns