Self-Driving Vehicle Sensor and Battery Charge Management

Frida Cano Falcón

Java Backend Academy MTY
August 2024

Final Project

September 7, 2024

Index

Index **2**
1. Introduction **3**
2. Project Structure **3**
3. Class Explanations **4**
3.1. Config Layer: 4
3.2. Controller Layer: 4
3.3. Entity Layer: 4
3.4. Repository Layer: 5
3.5. Service Layer: 5
3.6. Application Class: 5
3.7 Integration of SQL Database 5
4. Technologies Used **7**
4.1. Java 17: 7
4.2. Spring Data JPA: 7
4.3. Spring Batch: 7
4.4. Mockito for Unit Testing: 7
4.4.1 Benefits of High Test Coverage 8
4.4.2. Benefits of High Test Coverage 9
4.4.3. What Could Be Improved 9
4.5. Object-Oriented Programming (OOP): 10
4.5.1. Encapsulation 10
4.5.2. Abstraction 10
4.5.3. Inheritance 11
4.5.4. Polymorphism 11
4.5.5. Example of OOP in Action 12
4.5.6. Why OOP is Important 12
5. Conclusion **12**

1. Introduction

The increasing complexity of autonomous vehicles has highlighted the need for efficient sensor management and battery charge tracking. As part of my work in a Self-Driving Vehicle project, I observed the necessity to monitor the history of battery charges, track who charged them, and when the process occurred. This system ensures battery efficiency, identifies charged batteries ready for use, and detects any faults or issues during the charging process. With this in mind, I developed a Self-Driving Vehicle Sensor and Battery Charge Management System using technologies such as Java 17, Spring Data JPA, Mockito for unit testing, and OOP principles.

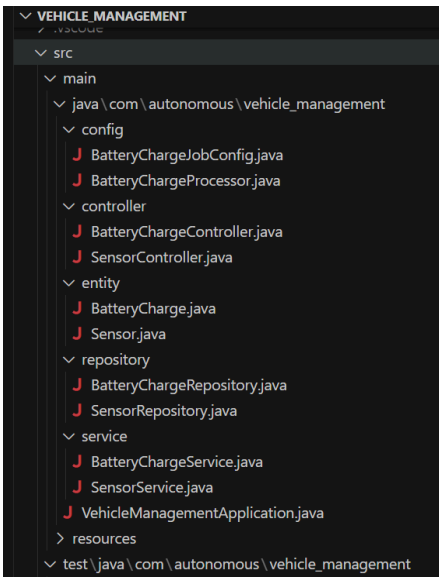
For this project, I utilized several key technologies:

- **Java 17**: The latest stable version of Java, offering new features and performance improvements, was chosen for this project.
- **Visual Studio Code**: As my Integrated Development Environment (IDE), VS Code provides a powerful and lightweight toolset for developing Java applications.
- **Spring Data JPA**: For managing database interactions, Spring Data JPA simplifies data access with built-in support for CRUD operations.
- **Mockito**: This framework was used to write unit tests for the service and controller layers, ensuring reliable behavior without a full database setup.
- **Object-Oriented Programming (OOP)**: OOP principles were applied to design modular, reusable, and maintainable code.
- **Spring Batch**: For managing large-scale data processing, such as importing and exporting battery charge data.

These technologies work together to build a robust, scalable, and maintainable solution for managing battery and sensor data in self-driving vehicles.

2. Project Structure

The project is structured into several key layers: **Config**, **Controller**, **Entity**, **Repository**, and **Service**, following the separation of concerns principle. Below is the project structure:



Project Structure from VS Code

3. Class Explanations

3.1. Config Layer:

- **BatteryChargeJobConfig.java**: This class defines the Spring Batch job responsible for importing and exporting battery charge history to/from the database. It configures the steps and readers/writers for processing CSV file.

src > main > resources > battery_charge_data.csv > data

```
1  id,battery_id,charge_start_time,charge_end_time,charging_area,charger_id,charge_status,issue_detected,issue_description
2  1,A,2024-09-09 10:00:00,2024-09-09 11:00:00,Software,A,successful,FALSE,NULL
3  2,B,2024-09-09 12:00:00,2024-09-09 13:00:00,Electronics,B,failed,TRUE,Charger malfunction during operation.
```

CSV file data

- **BatteryChargeProcessor.java**: This processor defines the business logic for transforming raw data into `BatteryCharge` objects, performing validation and filtering.

3.2. Controller Layer:

- **BatteryChargeController.java**: This controller exposes REST APIs for creating, reading, updating, and deleting battery charge records. It interacts with the `BatteryChargeService` to handle business logic.
- **SensorController.java**: Similarly, this controller handles the sensor-related operations, providing endpoints to manage sensor data and monitor vehicle performance in real-time.

3.3. Entity Layer:

- **BatteryCharge.java**: The entity representing a battery charge, including fields like `batteryId`, `startTime`, `endTime`, `status`, and the area of the project responsible for the charge. It maps directly to the `battery_charge` table in MySQL.
- **Sensor.java**: This entity represents a sensor, with fields for sensor type, status, and additional metadata. It maps to the `sensor` table in the database.

3.4. Repository Layer:

- **BatteryChargeRepository.java**: This interface extends `JpaRepository`, providing standard CRUD operations for the `BatteryCharge` entity. Custom queries can also be added to fetch specific charge data.
- **SensorRepository.java**: Similar to `BatteryChargeRepository`, this repository interface allows for CRUD operations on sensor data.

3.5. Service Layer:

- **BatteryChargeService.java**: The service layer contains business logic for managing battery charge data. It integrates with the repository to handle operations and communicate with the controllers.
- **SensorService.java**: This service handles the business logic for managing and monitoring sensors in the vehicle.

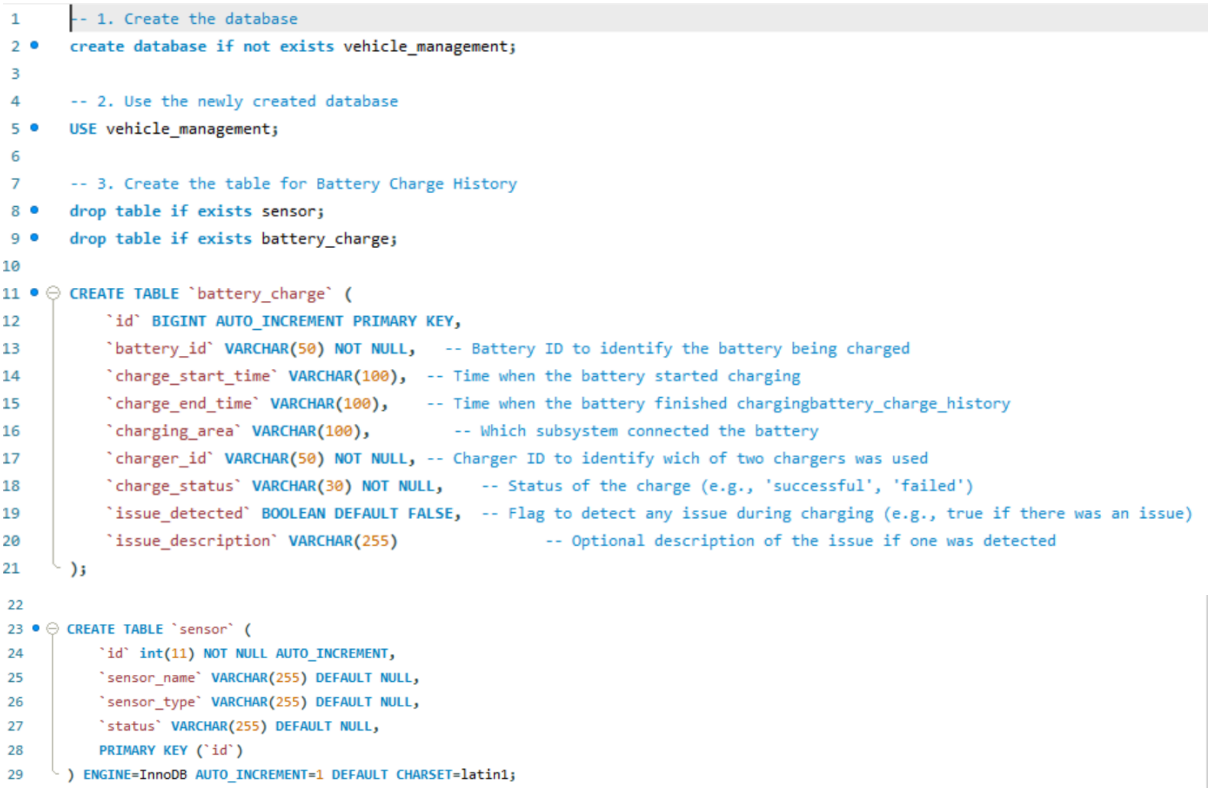
3.6. Application Class:

- **VehicleManagementApplication.java**: This is the main class that bootstraps the Spring Boot application, initializing all services, controllers, and configurations.

3.7 Integration of SQL Database

The database integration for the Self-Driving Vehicle Sensor and Battery Charge Management system is achieved using MySQL. This relational database was chosen for its reliability, performance, and ease of use when managing structured data. MySQL allows for efficient storage and retrieval of sensor and battery charge history data, which is essential for tracking the performance of autonomous vehicles.

The SQL scripts below demonstrate the creation of the database and the tables necessary for this project:



```sql
-- 1. Create the database
create database if not exists vehicle_management;

-- 2. Use the newly created database
USE vehicle_management;

-- 3. Create the table for Battery Charge History
drop table if exists sensor;
drop table if exists battery_charge;

CREATE TABLE `battery_charge` (
    `id` BIGINT AUTO_INCREMENT PRIMARY KEY,
    `battery_id` VARCHAR(50) NOT NULL,   -- Battery ID to identify the battery being charged
    `charge_start_time` VARCHAR(100),  -- Time when the battery started charging
    `charge_end_time` VARCHAR(100),    -- Time when the battery finished chargingbattery_charge_history
    `charging_area` VARCHAR(100),         -- Which subsystem connected the battery
    `charger_id` VARCHAR(50) NOT NULL, -- Charger ID to identify wich of two chargers was used
    `charge_status` VARCHAR(30) NOT NULL,    -- Status of the charge (e.g., 'successful', 'failed')
    `issue_detected` BOOLEAN DEFAULT FALSE,  -- Flag to detect any issue during charging (e.g., true if there was an issue)
    `issue_description` VARCHAR(255)               -- Optional description of the issue if one was detected
);

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 and tables creation using MySQL Workbench 8.0 CE

This script performs the following tasks:

1. **Create the Database**: The `vehicle_management` database is created if it does not already exist.

2. **Create the Battery Charge Table**:

    - The `battery_charge` table is designed to store the history of battery charges, including details such as the battery ID, start and end times, the subsystem (charging area) responsible for the charge, the charger ID, and the status of the charge (e.g., successful or failed).
    - There is also a provision for recording any issues detected during the charge process, with a flag (`issue_detected`) and an optional description field (`issue_description`) to detail the nature of the problem.

3. **Create the Sensor Table**:

    - The `sensor` table stores information about the sensors on the autonomous vehicles, including the sensor's name, type (e.g., LiDAR, radar, camera), and operational status.

- This table serves as the basis for tracking sensor performance and identifying any malfunctioning sensors that need attention.

These tables form the backbone of the system's data layer, with the `battery_charge` table being used to log battery charge activities and any issues encountered, and the `sensor` table used to monitor the status of various sensors in the self-driving vehicles.

The integration of this SQL schema into the project is handled through **Spring Data JPA**, which maps these tables to corresponding Java entities (`BatteryCharge` and `Sensor`). CRUD operations are easily performed using JPA repositories, ensuring a seamless connection between the Java application and the MySQL database. Additionally, this structure allows for batch processing using **Spring Batch** to handle large amounts of data, such as importing and exporting CSV files related to battery charge history.

4. Technologies Used

4.1. Java 17:

Java 17 was chosen due to its new features and improvements, such as sealed classes and pattern matching. These enhancements contribute to the maintainability and performance of the project.

4.2. Spring Data JPA:

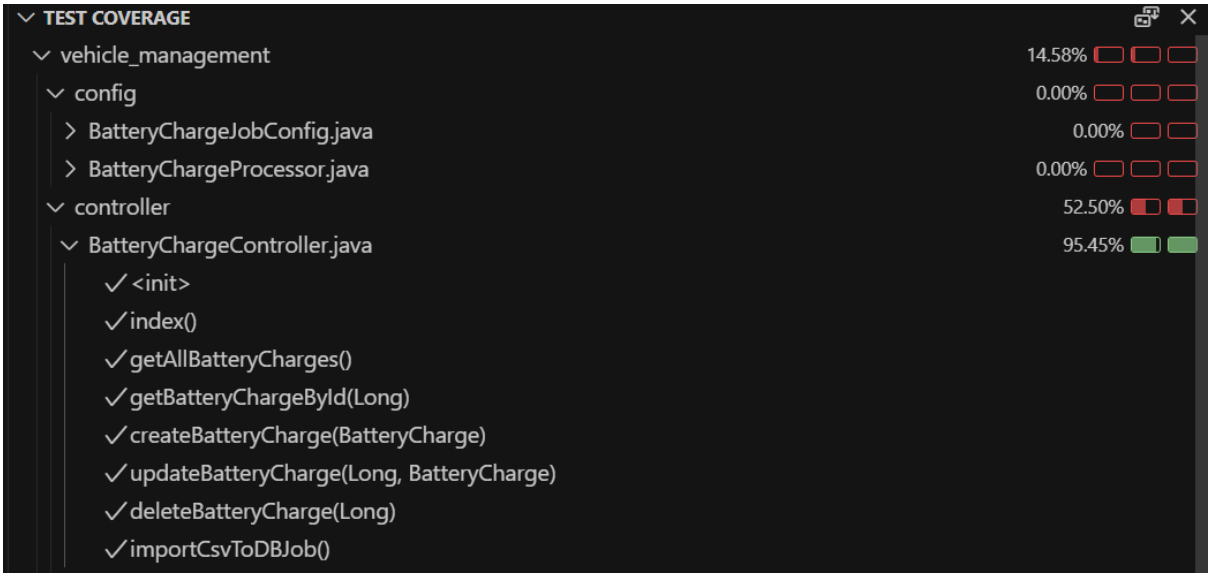
Spring Data JPA simplifies database access, reducing the amount of boilerplate code needed for repository management. It also supports efficient querying and transaction management.

4.3. Spring Batch:

Spring Batch is used to handle large-scale data operations, like importing battery charge history from CSV files and exporting data to external systems. This automation is key to managing data efficiently.

4.4. Mockito for Unit Testing:

Testing is a crucial part of software development, ensuring that the system works as expected and is reliable. In this project, Mockito was used to test the `BatteryChargeController` class, achieving a high level of test coverage, as shown in the image, which indicates 95.45% coverage for the `BatteryChargeController`.



Let's explore how Mockito was used, the tested functions, and the importance of achieving such a high test coverage.

Mockito is a popular testing framework for Java that allows developers to create mock objects for testing purposes. These mock objects mimic the behavior of real objects, allowing you to test components in isolation without relying on the actual implementation of their dependencies. This is particularly useful in unit testing where you want to focus on testing a single class or method.

In this project, the `BatteryChargeController` class depends on services that handle the business logic, such as `BatteryChargeService`. Mockito was used to mock these service layer components so that the controller could be tested in isolation.

- **`@InjectMocks`** is used to create an instance of `BatteryChargeController` and inject the mocked dependencies (in this case, `BatteryChargeService`) into it.
- **`@Mock`** is used to create a mock version of `BatteryChargeService`.
- The **`when(...).thenReturn(...)`** method is used to specify the behavior of the mocked `batteryChargeService.getAllBatteryCharges()` method. When this method is called, it returns a predefined list of battery charges.
- The **`verify(...)`** method ensures that the mocked service method is called the expected number of times.

4.4.1 Benefits of High Test Coverage

The image shows that the `BatteryChargeController` has a coverage of 95.45%, which means that almost all the methods were successfully tested. Here's a breakdown of the methods tested using Mockito:

1. `index()`: This method might return a simple message or view, and was tested to ensure it's working properly.
2. `getAllBatteryCharges()`: This method retrieves all the battery charges. The test ensures that the controller interacts with the service correctly, and that it returns the expected list of battery charges.
3. `getBatteryChargeById(Long id)`: This method retrieves a specific battery charge by its ID. The test ensures that the service is called with the correct ID and that the response is as expected.
4. `createBatteryCharge(BatteryCharge batteryCharge)`: This method creates a new battery charge. The test ensures that the controller passes the correct data to the service layer for creating a new entry.
5. `updateBatteryCharge(Long id, BatteryCharge batteryCharge)`: This method updates an existing battery charge. The test verifies that the controller correctly interacts with the service to update a record.
6. `deleteBatteryCharge(Long id)`: This method deletes a battery charge by ID. The test ensures that the deletion functionality works as expected and the correct service method is called.
7. `importCsvToDBJob()`: This method likely imports battery charge data from a CSV file to the database using Spring Batch. The test ensures that the import process is triggered correctly.

4.4.2. Benefits of High Test Coverage

Achieving 95.45% test coverage in the `BatteryChargeController` is a strong indicator of reliability. Here are the benefits:

- **Code Reliability**: High coverage ensures that most of the functionality has been tested, minimizing the chance of bugs.
- **Bug Detection**: By covering all critical paths, the tests help catch any edge cases or potential bugs early in the development process.
- **Confidence in Refactoring**: With high test coverage, future code changes can be made confidently, knowing that the tests will catch any unintended side effects.
- **Regression Testing**: The tests serve as regression tests, ensuring that any future changes do not break existing functionality.

4.4.3. What Could Be Improved

While 95.45% is excellent, there are still a few areas (like `BatteryChargeJobConfig.java` and `BatteryChargeProcessor.java`) that show 0% coverage in the image. These components could be tested to achieve even higher overall coverage:

- **`BatteryChargeJobConfig`**: Testing configuration classes ensures that the job configuration for Spring Batch is correctly set up, particularly if any custom configurations are used.
- **`BatteryChargeProcessor`**: This class likely contains business logic for processing battery charge data. Testing it ensures that the data is processed correctly before it is written to the database.

Adding tests for these components would provide complete coverage for the project and further increase its reliability.

4.5. Object-Oriented Programming (OOP):

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects," which can contain both data (attributes) and methods (functions or behaviors). In this project, OOP plays a crucial role in the design and implementation of various components.

4.5.1. Encapsulation

Encapsulation is about bundling data (fields) and methods (functions) that operate on that data into a single unit, or class, and controlling access to that data through access modifiers (`private`, `public`, `protected`).

In this project:

- **Entities like `BatteryCharge` and `Sensor`** encapsulate the relevant data related to battery charges and sensors. For example, the `BatteryCharge` class encapsulates attributes such as `battery_id`, `charge_start_time`, and `issue_detected`. These fields are marked as `private`, ensuring that the data can only be accessed through getter and setter methods, thereby controlling how the data is accessed and modified.

- Encapsulation ensures that sensitive data, such as the `issue_description`, can only be modified through the appropriate methods, ensuring the integrity of the system's state.

4.5.2. Abstraction

Abstraction involves hiding complex implementation details and exposing only the necessary functionalities through well-defined interfaces or abstract classes. In this project:

- **Service Layer (e.g., `BatteryChargeService` and `SensorService`)** abstracts the business logic from the controller, providing a clean and simple interface for performing operations like saving or retrieving data. The controller does not need

to know the details of how data is persisted in the database; it just calls methods exposed by the service layer, such as `getBatteryChargeHistory()` or `saveBatteryCharge()`.

- The repository interfaces (e.g., `BatteryChargeRepository`) are abstractions that allow interaction with the database without exposing the underlying implementation, which could change (for example, switching from MySQL to another database technology). This enhances maintainability and flexibility.

4.5.3. Inheritance

Inheritance allows a class to acquire properties and methods from a parent class, promoting code reuse and reducing redundancy. In this project:

- While the project doesn't explicitly have inheritance in the current structure, it could be extended to include it. For example, if you had multiple types of sensors (e.g., temperature sensors, motion sensors), they could inherit from a base class `Sensor`, which contains common fields like `sensor_name` and `sensor_type`. Specialized sensor types could extend this base class and introduce additional functionality specific to that sensor type.

- Similarly, if there are different types of battery charge statuses (e.g., fast charging, slow charging), they could be implemented as classes that inherit from a base `ChargeStatus` class, which could define common behavior for charging.

4.5.4. Polymorphism

Polymorphism allows one interface or method to be used in different forms, enabling flexibility and extensibility. In this project:

- **Service and Repository interfaces** provide polymorphism by allowing different implementations of the same interface. For example, the `BatteryChargeService` interface can be implemented in different ways to accommodate future changes in business logic, such as adding a caching layer or using an external service for managing battery charges. The controller would still interact with the service interface, and the underlying implementation can be swapped without affecting the rest of the application.

- **Spring Batch Processor** (`BatteryChargeProcessor`) could also utilize polymorphism if there are different ways to process different types of battery charges. Multiple processing strategies can be created that implement a common interface, and the processor can dynamically choose the appropriate one at runtime.

4.5.5. Example of OOP in Action

The class `BatteryCharge` is a concrete example of OOP in action. Here, the `BatteryCharge` class encapsulates the battery charge data and provides a clear interface for interacting with it. This class can later be extended, and polymorphism could be introduced if there are different ways to handle specific types of battery issues.

4.5.6. Why OOP is Important

- **Modularity and Maintainability**: By breaking down the system into reusable objects (entities, services, repositories), OOP ensures that the code is modular, meaning it's easier to maintain and extend over time.

- **Reusability**: Once you create objects like `BatteryCharge` or `Sensor`, they can be reused across different parts of the project or even in future projects involving self-driving vehicle management systems.

- **Scalability**: The system can be easily extended. For instance, if new functionalities need to be added (e.g., new types of sensors or different battery charging protocols), OOP allows us to extend the existing codebase with minimal modifications.

- **Testability**: The use of interfaces in the service and repository layers, combined with dependency injection provided by Spring, makes it easier to mock dependencies and write unit tests using frameworks like Mockito.

By following OOP principles, this project not only achieves the required functionality for managing sensors and battery charges in a self-driving vehicle system but also ensures that the code is modular, maintainable, scalable, and testable.

5. Conclusion

This Self-Driving Vehicle Sensor and Battery Charge Management project demonstrates how modern Java technologies like Spring Boot, Spring Data JPA, Spring Batch, and Mockito can be integrated to solve real-world problems. The application efficiently tracks sensor data and battery charge history, ensuring that batteries are ready for use and potential issues are detected early. The modular architecture allows for scalability, and the use of unit tests ensures reliability and stability in production environments.

This project offers a foundation for further expansion into other areas of autonomous vehicle management, such as real-time monitoring and predictive maintenance.