Vehicle Performance Dashboard Report

by

Shivangi Srivastava

Anjalika Goswami

Under the Guidance of

Ruchi Gupta

(Master Trainer, CEA, GLA)

Declaration

I hereby declare that the work which is being presented in the B.Tech. Project **"Vehicle Performance Dashboard "**, in partial fulfillment of the requirements for the award of the ***Bachelor of Technology*** in Computer Science and Engineering and submitted to the Department of Computer Engineering and Applications of GLA University, Mathura, is an authentic record of my own work carried under the supervision of **Ruchi Gupta .**

(Master Trainer)

The contents of this project report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree.

ACKNOWLEDGEMENT

I would like to express my gratitude towards our mentor **Ruchi Gupta** for teaching and assisting us with the new technology and guiding us at every step, and it wouldn’t have been possible to complete the project in such short period of time if it were not for her motivation.

I would also like to thank extend my gratitude towards my group members and all those people who extended their wholehearted co-operation and have helped us in completing this project successfully.

ABSTRACT

The project **"Vehicle Performance Dashboard"** aims to develop a real-time monitoring system for displaying key vehicle performance metrics based on sensor data stored in a PostgreSQL database. The system backend is built using **Java 8**, leveraging Modern Java features and the Collections API for efficient data handling, while the frontend is developed using **HTML**, **CSS**, and **JavaScript** for an interactive user experience.

The dashboard retrieves, processes, and visualizes real-time sensor readings such as speed, engine temperature, fuel efficiency, and other critical parameters. This solution is especially useful for in-field vehicle testing, providing engineers with immediate insights into vehicle performance.

The project highlights the integration of backend and frontend technologies to create a reliable, scalable, and practical application for automotive data analysis.

List of Figures

CONTENTS

Chapter 1
Introduction

1.1 Background

The automotive industry has witnessed significant advancements in vehicle design, manufacturing, and performance optimization over the past few decades. With the evolution of technology, modern vehicles are equipped with a wide range of sensors that continuously monitor parameters such as speed, fuel efficiency, engine temperature, brake pressure, and overall health of various subsystems.

However, the raw data generated by these sensors often remains underutilized without an effective mechanism to visualize and interpret the data in real-time. In response to this need, a **Vehicle Performance Dashboard** becomes a crucial tool, offering engineers and testers the ability to monitor key vehicle parameters during in-field testing.

The project **Vehicle Performance Dashboard** aims to build a real-time data visualization platform that can display important vehicle performance metrics. It leverages backend technologies like **Java 8**, **Collections API**, and **PostgreSQL**, along with frontend technologies such as **HTML**, **CSS**, and **JavaScript**, to provide a seamless and interactive experience to users.

1.2 Problem Statement

Modern vehicles generate a vast amount of sensor data, but in many cases, real-time interpretation and visualization tools are either missing or too complex for regular testing teams. Manual data logging and post-test analysis not only slow down the testing process but also increase the chances of errors or missed critical insights during in-field tests.

Thus, there is a strong need for an efficient, real-time dashboard that can:

- Fetch live data from the database.
- Display essential vehicle performance metrics.
- Provide insights into the health and performance of the vehicle during in-field testing.
- Help engineers make quick decisions based on live sensor readings.

The lack of such systems in field operations can lead to delays, inaccurate performance analysis, and potential system failures going undetected during tests.

1.3 Objectives

The main objectives of this project are:

- To develop a dashboard that fetches real-time sensor data from a **PostgreSQL** database.
- To implement backend logic using **Java 8** and **Collections API** for efficient data processing.
- To design a simple, intuitive, and interactive frontend using **HTML**, **CSS**, and **JavaScript**.
- To ensure real-time visualization of key vehicle performance parameters such as speed, temperature, and fuel efficiency.
- To create a system that can be easily extended or modified based on new sensor inputs or additional vehicle parameters.

1.4 Scope of the Project

This project focuses on the following areas:

- Real-time integration between the PostgreSQL database and the dashboard application.
- Clean and readable frontend visualizations such as charts, tables, and meters.
- Reliable backend processing using Java for speed, security, and stability.
- In-field usability for vehicle performance testing during development or diagnostics.
- Simple deployment and maintenance using widely accepted tools and frameworks.

1.5 Significance

This project focuses on the following areas:

- Real-time integration between the PostgreSQL database and the dashboard application.
- Clean and readable frontend visualizations such as charts, tables, and meters.
- Reliable backend processing using Java for speed, security, and stability.
- In-field usability for vehicle performance testing during development or diagnostics.
- Simple deployment and maintenance using widely accepted tools and frameworks.

Chapter 2

Literature Review

- **2.1 Existing Systems Overview**

In recent years, the monitoring and management of vehicle performance have become a significant area of interest, especially with the growth of fleet management and real-time data monitoring. Various systems have been developed to track and manage vehicle performance, integrating data from numerous sensors installed in vehicles.

- **Vehicle Performance Dashboards:** Several commercial vehicle performance monitoring systems provide dashboards to monitor vehicle metrics in real-time. Systems like **Geotab**, **Fleet Complete**, and **Samsara** use telematics to track vehicle parameters such as fuel consumption, engine health, speed, GPS location, and more. These systems rely heavily on **On-Board Diagnostics (OBD-II)** sensors, which connect directly to the vehicle's electronic control unit (ECU). These sensors provide real-time data such as engine temperature, oil levels, and fuel efficiency.

- **Real-Time Data Processing:** Real-time data processing has become integral to these systems. Platforms like **Apache Kafka** and **WebSockets** are often used to handle streaming data from sensors. These technologies ensure that vehicle performance data is transmitted and processed in real time, enabling the system to update the dashboard with live data as it is collected.

- **PostgreSQL and Data Storage:** Relational databases like **PostgreSQL** are often employed for storing large amounts of structured data, including historical vehicle performance data. PostgreSQL offers powerful features such as ACID compliance, advanced indexing, and high performance, which make it a suitable choice for storing and querying real-time sensor data.

- **Frontend Technologies:** The dashboards typically use **HTML**, **CSS**, and **JavaScript** to display vehicle data on web interfaces. Technologies such as **React**, **Angular**, and **Vue** are commonly used to create interactive user interfaces that allow users to visualize and analyze vehicle performance data effectively.

- **Existing Java Applications for Vehicle Monitoring:** Many vehicle monitoring systems are developed using Java, leveraging Java 8 features such as **Streams**, **Lambda expressions**, and the **Collections API** for managing and processing data efficiently. Java's multi-threading capabilities allow for real-time data handling, while its wide array of libraries and frameworks supports building scalable and maintainable solutions.

2.2 Limitations of Current Systems

Despite the growing number of vehicle performance monitoring systems, there are several limitations that hinder their efficiency and usability in real-world applications.

- **Limited Real-Time Capabilities:** While many systems offer vehicle data monitoring, they often lack the ability to process and display data in real time. Delays in sensor data transmission can lead to outdated information on the dashboard, which is critical when real-time decisions need to be made, such as identifying a malfunctioning vehicle during infield testing.

- **Scalability and Data Handling:** As vehicle fleets grow, existing systems can struggle to manage the large volume of data generated by sensors. Commercial systems may face performance bottlenecks when handling millions of records, especially when multiple vehicles are being monitored simultaneously. Moreover, some systems store data in a way that makes querying for insights inefficient, leading to slow data retrieval.

- **Complexity and Usability:** Many existing systems are complex and require specialized knowledge to interpret the data presented on dashboards. Fleet managers or field engineers might find these systems difficult to navigate, especially in a fast-paced testing environment where quick decision-making is essential. The lack of user-friendly, intuitive interfaces can impede the effectiveness of these systems.

- **Lack of Customization:** Commercial solutions are often generic and lack the flexibility needed for specific use cases. Infield testing, for instance, may require customized dashboards that display only the most relevant metrics for a specific vehicle model or test condition. Many systems do not offer this level of customization, limiting their applicability to specific testing scenarios.

- **Integration Challenges:** The integration of different sensor types into a single system can be challenging. Some systems are designed to work with specific sensor types or manufacturers, which makes it difficult to integrate with third-party sensors. This limitation reduces the flexibility of existing systems and hinders their ability to handle diverse vehicle fleets with varying sensor setups.

2.3 Need for the Proposed System

Given the limitations of existing systems, there is a clear need for a vehicle performance dashboard that overcomes these challenges. The proposed system aims to address the gaps in real-time data monitoring, scalability, usability, and customization.

- **Real-Time Data Monitoring:** One of the primary goals of the proposed system is to provide real-time updates of vehicle performance metrics, ensuring that fleet managers and field engineers receive up-to-date information. This capability is particularly crucial in the context of **infield testing**, where immediate responses are necessary to identify potential issues with a vehicle during performance trials

.

- **Customization for Infield Testing:** The proposed system is designed to be highly customizable. It will allow users to define which vehicle metrics are most important for a given test scenario. This flexibility is essential for field engineers who need to monitor specific vehicle performance data in real time and make quick decisions based on that data.

- **Scalability and Efficiency:** Using **PostgreSQL** as the backend database will ensure that the system can efficiently handle large volumes of sensor data. The use of indexed tables, normalized databases, and optimized queries will allow for fast data retrieval and high system performance, even as the volume of data increases. This will ensure that the system remains scalable as the fleet grows or as additional sensors are added.

- **Simplified User Experience:** The user interface will be designed to be intuitive, with a focus on simplicity and usability. The dashboard will present data in an easily understandable format, reducing the complexity faced by users. Interactive features such as **real-time graphs**, **alerts**, and **interactive data tables** will allow users to quickly assess vehicle performance without needing advanced technical knowledge.

- **Cost-Effectiveness:** The proposed system will be built using open-source technologies such as **Java**, **PostgreSQL**, **HTML**, **CSS**, and **JavaScript**, which makes it a cost-effective solution compared to proprietary commercial systems. The use of open-source tools not only reduces the cost but also ensures that the system remains flexible and adaptable to future updates and modifications.

Chapter 3
Technology Used

3.1 Java 8 Features

Java 8 introduced several important features that make it a powerful language for building modern applications. For the Vehicle Performance Dashboard, Java 8 features play a critical role in handling real-time data, processing sensor readings, and ensuring a responsive and scalable backend.

- **Lambda Expressions:**

Lambda expressions allow you to write more concise and readable code. They are used to provide clear and efficient representations of one-method interfaces (functional interfaces). In your project, lambdas can be used to process collections of vehicle performance data, such as filtering or sorting sensor data efficiently.

- **Streams API:**

  The **Streams API** is another key feature introduced in Java 8 that enables functional-style operations on sequences of elements, such as collections. It is especially useful in scenarios where large datasets need to be processed, such as filtering sensor data or transforming data from the database before displaying it in the dashboard.

- **Default Methods:**

  Java 8 allows the use of **default methods** in interfaces, which means you can add methods to an interface without affecting its implementing classes. This feature is particularly useful for maintaining backward compatibility in large applications, enabling easier integration of new features without breaking existing code.

- **Time API (java.time):**

  Java 8 introduced a new Date-Time API to handle dates, times, and durations more accurately and efficiently. In your dashboard, it can be used for logging and calculating the time of sensor data readings or when specific performance metrics are logged.

  These features, along with the powerful multithreading capabilities of Java, make it an ideal language for building a scalable, efficient, and responsive backend for the Vehicle Performance Dashboard.

3.2 Collections API

The **Collections API** in Java is a powerful framework for storing and manipulating data. In your Vehicle Performance Dashboard project, the Collections API is used to manage the sensor data and perform various operations like sorting, filtering, and searching for vehicle metrics.

- **List, Set, and Map:**

1. **List** is an ordered collection that allows duplicates. It is useful for storing data such as vehicle sensor readings over time, where the order of insertion is important.

2. **Set** is a collection that does not allow duplicates. This is useful for ensuring that there are no repeated vehicle IDs in the dataset.

3. **Map** is a collection that stores key-value pairs. It is particularly useful when associating vehicle IDs with their performance data or when querying vehicle metrics based on their ID.

- **Iterators:**

  The **Iterator** interface is part of the Collections API, allowing you to traverse collections. This is especially useful when processing large amounts of sensor data and performing operations like filtering or aggregating data.

- **Sorting and Searching:**

  The **Collections.sort()** method allows you to sort lists of vehicle data based on certain metrics like speed or engine temperature. Sorting helps in displaying the most relevant data on the dashboard, such as vehicles with the highest fuel consumption or speed.

- **Concurrency:**

  The Collections API also supports concurrent collections like **CopyOnWriteArrayList** and **ConcurrentHashMap**, which are useful when handling data in multi-threaded environments. This can be beneficial when the system is handling large volumes of sensor data from multiple vehicles simultaneously.

3.3 PostgreSQL Database

PostgreSQL is an open-source relational database management system (RDBMS) known for its robustness, scalability, and support for complex queries. It plays a vital role in your Vehicle Performance Dashboard by storing and managing sensor data from vehicles.

- **Why PostgreSQL?**

- **ACID Compliance**: PostgreSQL ensures that your data transactions are reliable and consistent. For vehicle performance data, this means that even if a system crashes, your data remains intact, and no data is lost.
- **Extensive SQL Features**: PostgreSQL supports complex queries, joins, and indexing, which is useful for retrieving historical sensor data and performing complex analytics, such as identifying trends in vehicle performance or comparing vehicles.

- **Data Integrity**: PostgreSQL allows for data validation and integrity checks, which ensures that only valid sensor data is stored, preventing any corrupt data from being entered into the system.

- **Database Schema:**

  In your system, PostgreSQL will store data about vehicles, sensor readings, and performance metrics. For instance, you could have tables like:

- **vehicles**: stores vehicle information (ID, make, model, etc.).
- **sensor_readings**: stores individual sensor readings (timestamp, speed, engine temperature, etc.).

- **Indexing and Query Optimization:** PostgreSQL supports indexing, which is used to speed up data retrieval operations. In your project, indexes will be crucial for quickly retrieving vehicle performance data based on specific metrics or time ranges.

3.4 HTML, CSS, JavaScript

HTML, CSS, and JavaScript form the backbone of the frontend for your Vehicle Performance Dashboard, providing an interactive and responsive user interface.

- **HTML**: HTML (Hypertext Markup Language)
  HTML is the standard markup language used to create the structure and content of web pages. It acts as the skeleton for any web application, providing the fundamental building blocks such as headings, paragraphs, tables, forms, and links. In the context of your Vehicle Performance Dashboard, HTML will be

used to create the various components of the user interface (UI) to display vehicle performance metrics and other related information.

HTML Elements for Vehicle Dashboard: In your Vehicle Performance Dashboard, the structure of the page will be built using various HTML elements that represent different types of data and functionality. Here’s how HTML will be utilized:

- **Header Section**: The header of the dashboard will contain the title and possibly navigation links or user options.
- **Vehicle Data Display**: Data such as vehicle speed, engine health, fuel efficiency, and temperature readings will be shown in tables or as individual metrics.
- **Forms for User Input**: If users need to input data (e.g., adding a new vehicle or configuring sensor thresholds), you would use forms.

- **CSS (Cascading Style Sheets)**

CSS is used to style and layout web pages. While HTML provides the structure, CSS controls the look and feel of the web page. It dictates things like colors, fonts, spacing, positioning, and responsiveness, which ultimately enhance the user experience.

In the Vehicle Performance Dashboard project, CSS is essential for creating a clean, intuitive, and visually engaging interface for users to interact with vehicle data.

- **Key Features of CSS for Vehicle Performance Dashboard:**

1. **Layout and Positioning**:

- CSS provides tools like **Flexbox** and **Grid** for creating responsive layouts that adapt to different screen sizes (e.g., desktop, tablet, mobile). These features are critical for ensuring that the dashboard looks good on all devices.

2. **Styling Data Tables**:

- The vehicle data tables can be styled to make them more readable and organized. For instance, alternating row colors, hover effects, and clear borders can enhance the user interface.

3. **Typography and Visual Hierarchy**:

- CSS is used to adjust fonts, text sizes, and font weights to make the most important information stand out. For example, the vehicle speed might be highlighted in a larger font to grab the user’s attention.

4. **Responsive Design**:

- CSS enables **media queries**, allowing the dashboard layout to adjust based on the device’s screen size. For example, the vehicle data table might display as a stacked list on smaller screens rather than a full table.

5. **Animations and Transitions**:

- CSS can be used to add subtle animations and transitions that make the dashboard more interactive. For example, you can animate the changes in vehicle data as it is updated in real-time.

- JavaScript

  JavaScript is a programming language used to create dynamic, interactive features on web pages. Unlike HTML and CSS, which are used to define the structure and style, JavaScript allows you to manipulate the page content, respond to user actions, and update the page dynamically.

  In your Vehicle Performance Dashboard, JavaScript will be used for tasks like:

  1.Real-Time Data Fetching:

    - JavaScript will allow you to fetch vehicle data from the backend server (possibly using APIs). By using the **Fetch API** or **AJAX**, the dashboard can retrieve new sensor readings in real time without requiring a page reload.

2. **Dynamic Content Updates**:

- Using JavaScript, the vehicle performance data can be dynamically updated in the dashboard. For example, when new sensor data is fetched, JavaScript can update the values displayed in the HTML table without reloading the page.

3. **Event Handling**:

- JavaScript enables you to add interactivity to your dashboard by responding to user actions. For instance, clicking on a vehicle could display additional details or switching between different vehicle metrics.

4. **Data Visualization**:

- JavaScript libraries like **Chart.js** or **D3.js** can be used to visualize vehicle performance metrics in the form of charts, graphs, or gauges. For example, you could display a real-time graph of speed or engine temperature.

5. **Asynchronous Behaviour**:

- JavaScript's ability to handle asynchronous behavior (using **Promises**, **async/await**, or **setTimeout**) ensures that real-time data fetching, processing, and display updates happen smoothly without blocking other operations on the page.

3.5 Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a lightweight but powerful code editor that provides an excellent development environment for building your Vehicle Performance Dashboard.

- **Why VS Code?**
- **Extensions**: VS Code supports a wide range of extensions for Java, PostgreSQL, HTML, CSS, JavaScript, and Git, allowing you to work efficiently across multiple technologies.
- **Integrated Terminal**: VS Code has an integrated terminal, making it easier to run commands, scripts, and interact with the backend server or database directly within the editor.
- **Git Integration**: VS Code integrates seamlessly with **Git**, allowing you to manage version control and push your code to GitHub.
- **Intelligent Code Completion**: With features like **IntelliSense**, VS Code helps you write code faster by providing suggestions and auto-completion for Java methods, SQL queries, HTML elements, and JavaScript functions.

Chapter 4
System Analysis

4.1 Requirement Gathering

Requirement gathering is the initial and critical phase of the project development life cycle. It involves collecting information about the project’s objectives, features, and limitations to ensure that the final system meets user needs.

For the Vehicle Performance Dashboard, the requirement gathering process included:

- Study of existing vehicle monitoring systems.
- Analysis of common vehicle performance parameters needed during testing (e.g., speed, fuel level, engine temperature, tire pressure).
- Identification of real-time data needs and database storage capabilities.
- Sources of Requirements:
- Research papers and documentation on infield vehicle testing systems.

- ## 4.2 Non-functional Requirements
- Non-functional requirements specify the system’s operational capabilities and constraints. They define **"how"** the system performs rather than **"what"** it does.
  1.Performance - System should process and display real-time data with minimal delay (<1 second).
  2.**Scalability**- System must support addition of new sensor types in future without
  3.**Reliability** – System must handle database downtime gracefully and ensure no data loss.

4.**Security** – Data must be securely transmitted using HTTPS; authentication is mandatory.

5.**Usability**- The dashboard interface should be simple , intuitive ,and user-friendly.

- ## 4.3 Use Case Diagrams

Use Case Diagrams graphically represent the interactions between users (actors) and the system to achieve specific goals.

- **Actors:**

1**.**Admin/User

2.Database

3. Java Backend

- **Use Cases**

- Enter or simulate sensor data
- Fetch data from database
- Display performance on dashboard
- Monitor performance history

The following diagram illustrates the key use cases of the Vehicle Performance Dashboard system, showing how users and system components interact:

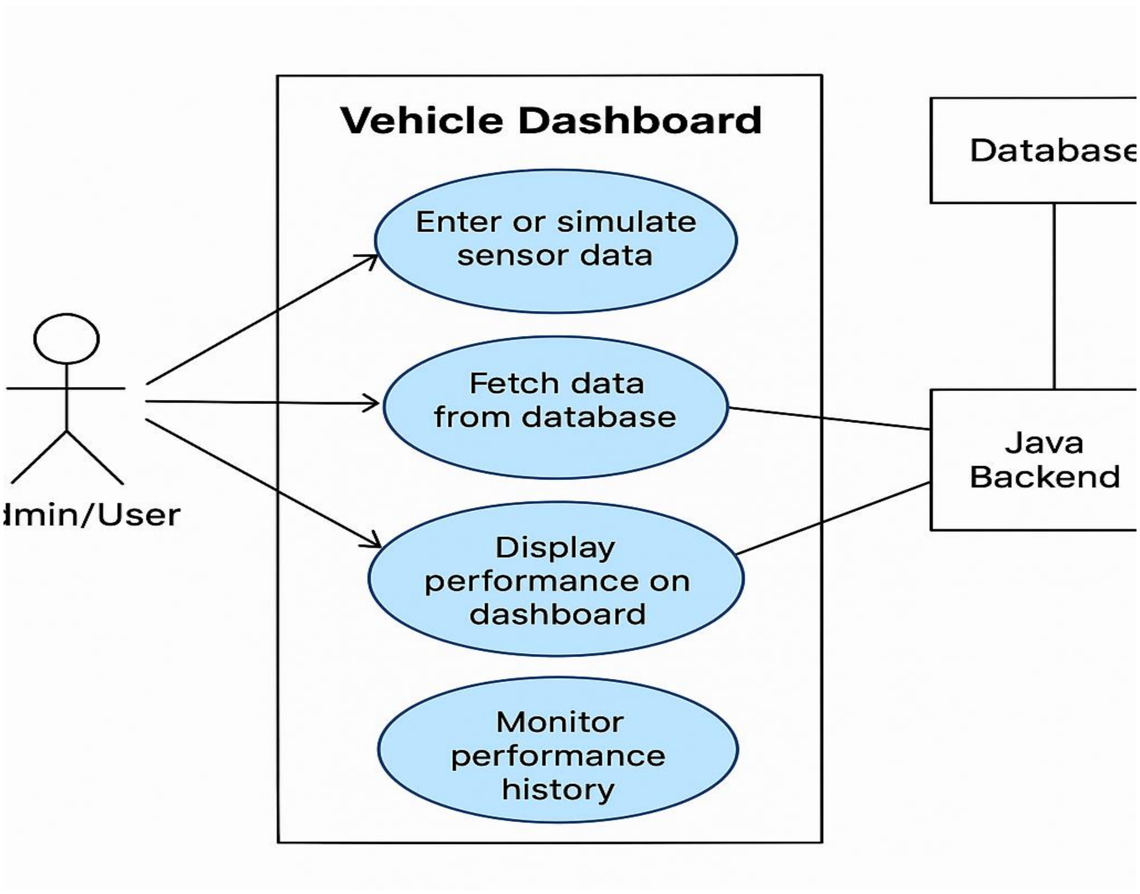

Vehicle Dashboard
Enter or simulate sensor data
Fetch data from database
Display performance on dashboard
Monitor performance history
Database
Java Backend
dmin/User

Fig 4.1

4.5 Data Flow Diagrams (DFD)

- A **Data Flow Diagram (DFD)** is a graphical representation that shows how data moves through a system — how it enters, is processed, and stored. It helps in understanding the logic of the system without going into the technical implementation details.

- Key Components of DFD

- **External Entity:** Outside sources or users that interact with the system (e.g., User, Sensor).
- **Process:** Operations or actions that transform data (e.g., Backend Processor).
- **Data Store:** Storage locations where data is kept (e.g., PostgreSQL Database).
- **Data Flow:** Arrows indicating the direction of data movement.

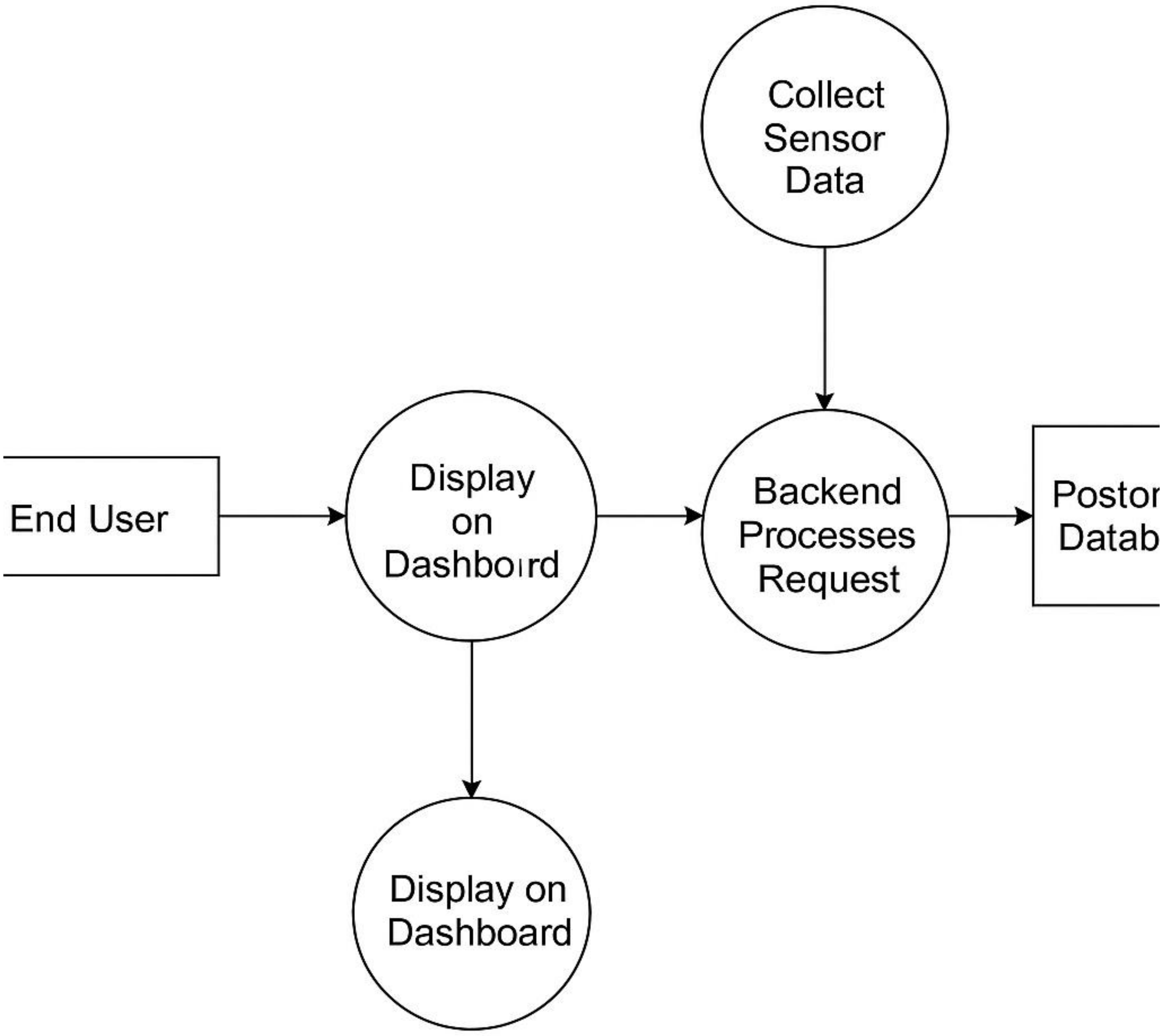


Fig 4.2

Chapter 5

System Design

The Vehicle Performance Dashboard system is built on a modular architecture that separates frontend, backend, and database layers. This helps ensure flexibility, scalability, and ease of maintenance. The primary components involved are:

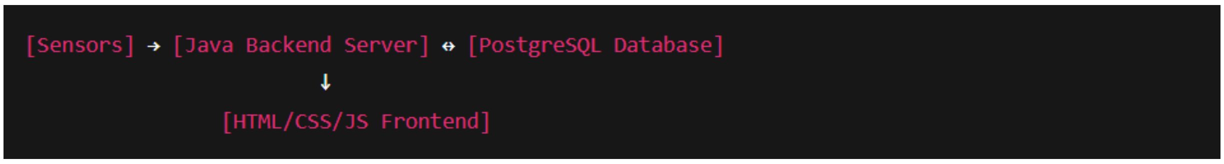
- **Frontend** – built using HTML, CSS, and JavaScript
- **Backend** – implemented using Java 8 with modern Java features and Collections API
- **Database** – PostgreSQL for structured sensor data storage
- **Development Environment** – VS Code used as the primary IDE

5.1 System Architecture

The Vehicle Performance Dashboard uses a three-tier architecture:

- **Presentation Layer (Frontend)**: Built with HTML, CSS, and JavaScript. It is responsible for displaying real-time sensor data and handling user interactions.
- **Application Layer (Backend)**: Developed using Java (Java 8 and Collections API). It processes requests, applies business logic, and communicates with the database.
- **Data Layer (Database)**: PostgreSQL is used to store sensor data, user details, and configuration settings.
- **Workflow**:

  • Vehicle sensors push data into the PostgreSQL database (assumed through a simulator or external source).
  • Java backend polls or retrieves latest data using JDBC.
  • Processed data is sent to frontend via REST API in JSON format.
  • Frontend renders data in visual format using HTML + JS.
  • User sees updated dashboard in real time.



```
[Sensors] → [Java Backend Server] ↔ [PostgreSQL Database]
                        ↓
               [HTML/CSS/JS Frontend]
```

Fig 5.1

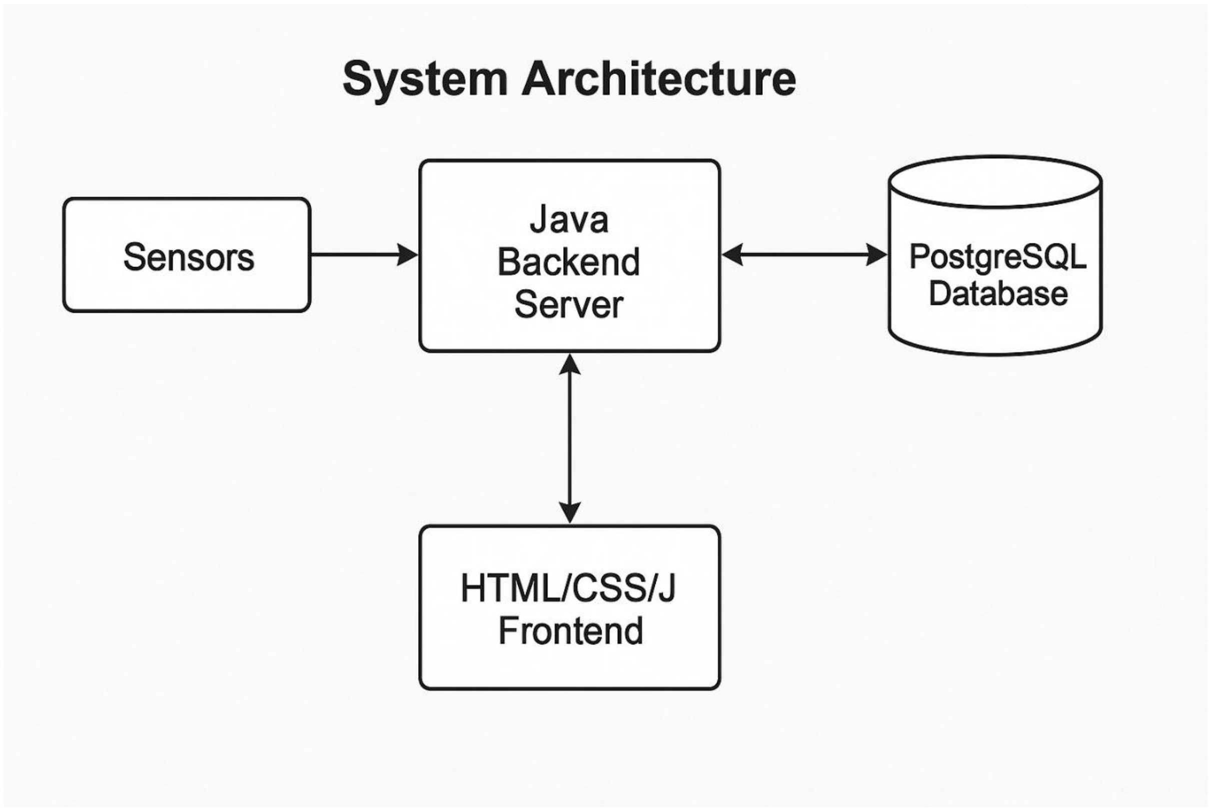


Fig 5.2

5.2 **Database Design (ER Diagram)**

The database plays a crucial role in storing and organizing sensor data and user management information.

Main Entities:

- **User**: Stores user credentials and roles (admin, tester).
- **Vehicle**: Information about the vehicle being tested.
- **Sensor Data**: Stores real-time sensor readings.
- **ThresholdConfig**: Stores limits for generating alerts.

5.3 Frontend Design

• Built using **HTML, CSS, and JavaScript**

• Displays live sensor data like speed, temperature, fuel level etc.

• Uses asynchronous JavaScript (AJAX) or fetch API to request backend data

• **Features include:**

- Live charts (Speed vs Time, Fuel Consumption)
- Real-time data updates
- Export data option
- Refresh button

5.4 Backend Logic Design

 Developed using **Java 8** with **Collections API** for efficient in-memory data handling

 Provides RESTful API endpoints to fetch data from PostgreSQL

 Handles logic for:

- Retrieving latest sensor readings
- Formatting data for charts and tables
- Calculating average or trends if needed

- **Database (PostgreSQL):**

- Stores time-stamped sensor data from vehicle
- Tables include:
  - sensor_readings (id, sensor_type, value, timestamp)
  - vehicles (current speed, temperature, fuel level, etc.)
- Optimized with indexes on timestamp for fast querying

- **Development Tools:**

- **VS Code** as IDE for frontend and Java backend coding
- **PostgreSQL pgAdmin** for DB management
- **Postman** or browser tools used for testing REST APIs

Important Concepts Used:

Java **Collections API** (ArrayList, HashMap) for temporary data handling.

- Java **Streams and Lambdas** for filtering and processing sensor data quickly.

- Java **JDBC** for database connectivity.

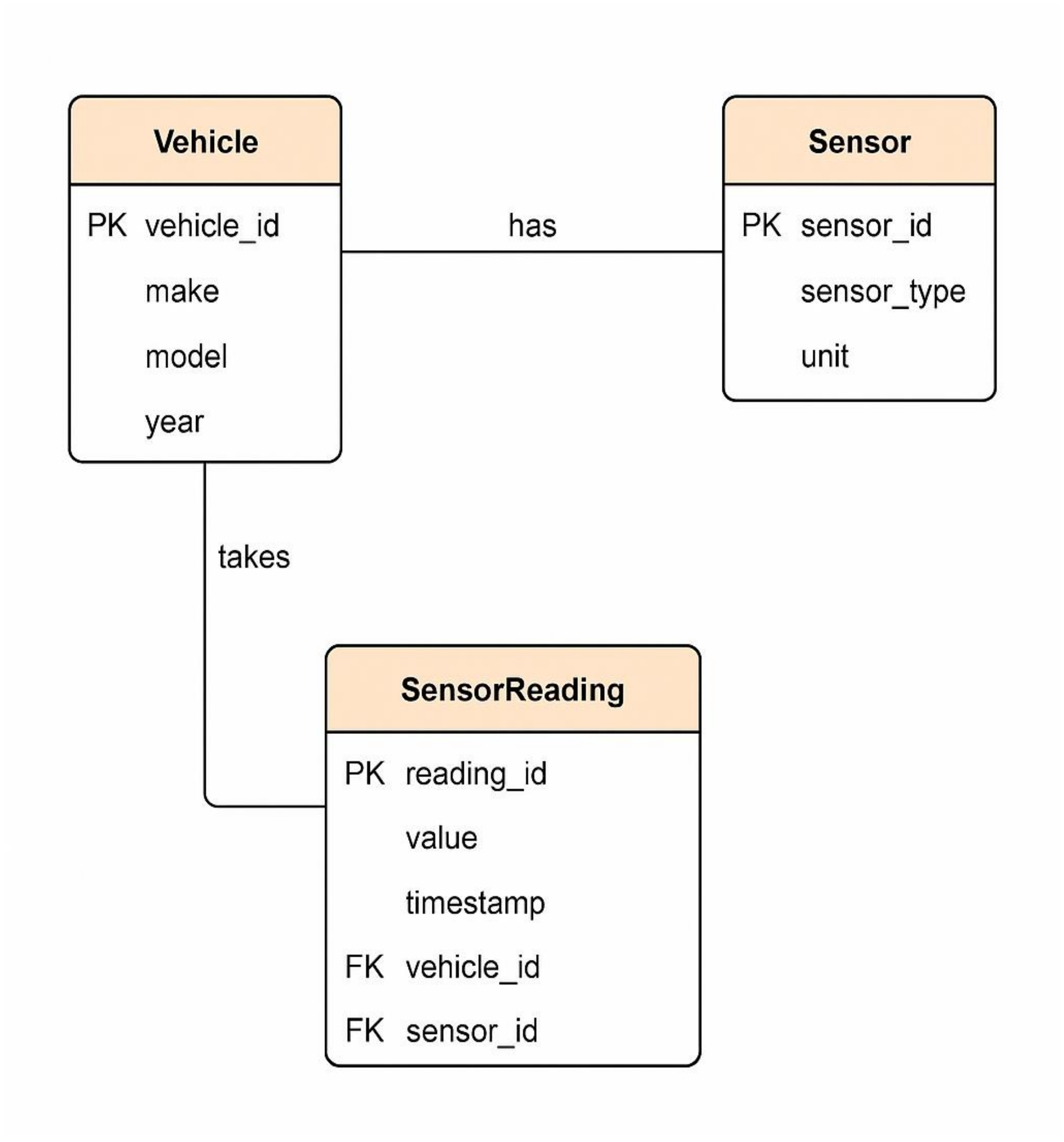


Fig 5.3

Chapter 6

Implementation

- **6.1 Introduction**
  The implementation phase focuses on converting the design and planned architecture into working code. The project was developed using **Java 8** for backend logic, **PostgreSQL** as the relational database, and **HTML, CSS, JavaScript** for the frontend interface. The development was done in **Visual Studio Code (VS Code)** environment.

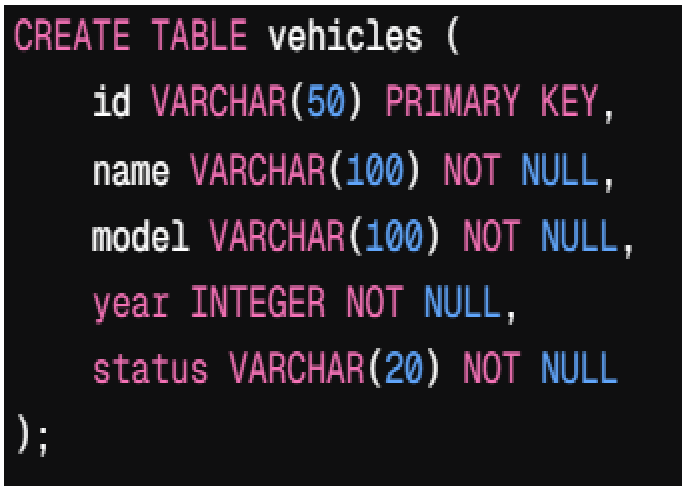
6.2 Technologies Used

Technology	
1.Java 8	- Backend logic and API development
2.Collection API	- Efficient in-memory data handling
3. JDBC	- Database connectivity with PostgreSQL
4. PostgreSQL	- Sensor data storage
5. HTML/CSS	- Web dashboard design
6.JavaScript	- Dynamic data updates and interactivity
7.VS Code	- Coding environment

6.3 Database Implementation

PostgreSQL Tables Created

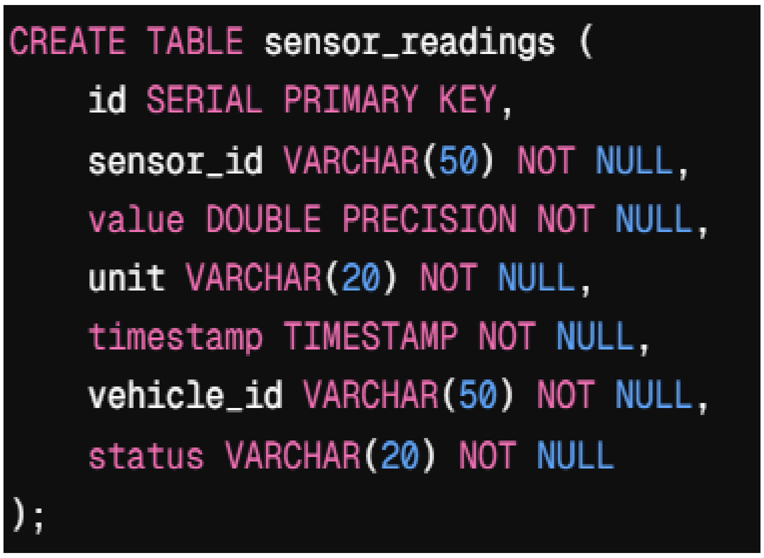
1. vehicle Table



```
CREATE TABLE vehicles (
    id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    model VARCHAR(100) NOT NULL,
    year INTEGER NOT NULL,
    status VARCHAR(20) NOT NULL
);
```

Fig 6.1

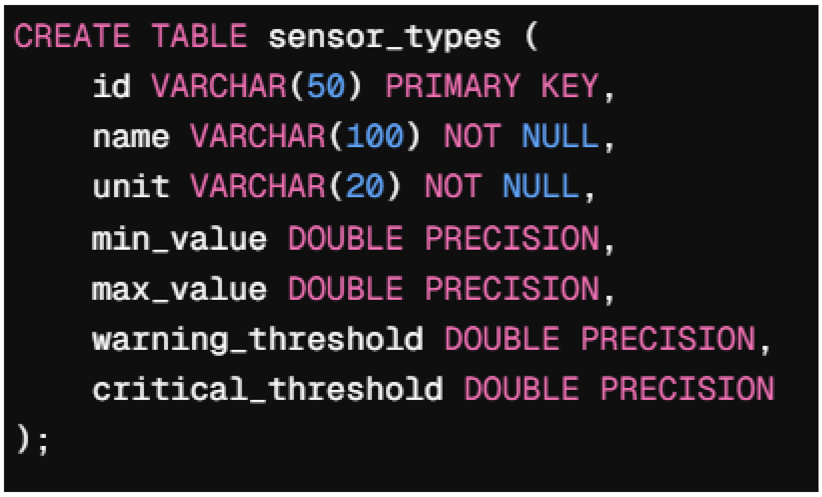
2. sensor_readings Table



```sql
CREATE TABLE sensor_readings (
    id SERIAL PRIMARY KEY,
    sensor_id VARCHAR(50) NOT NULL,
    value DOUBLE PRECISION NOT NULL,
    unit VARCHAR(20) NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    vehicle_id VARCHAR(50) NOT NULL,
    status VARCHAR(20) NOT NULL
);
```

Fig 6.2

3.sensor_types Table

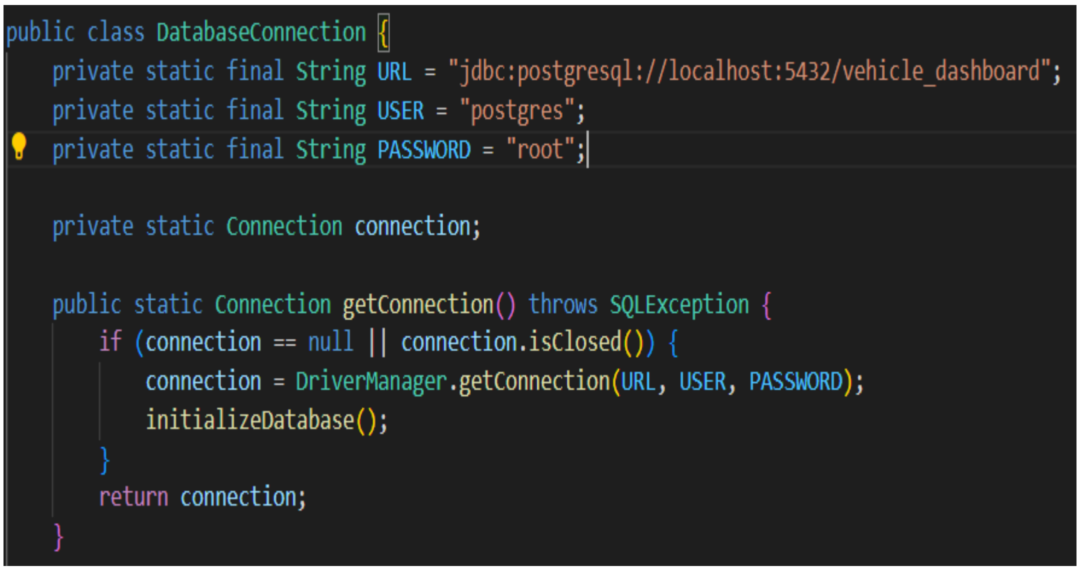


```sql
CREATE TABLE sensor_types (
    id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    unit VARCHAR(20) NOT NULL,
    min_value DOUBLE PRECISION,
    max_value DOUBLE PRECISION,
    warning_threshold DOUBLE PRECISION,
    critical_threshold DOUBLE PRECISION
);
```

Fig 6.3

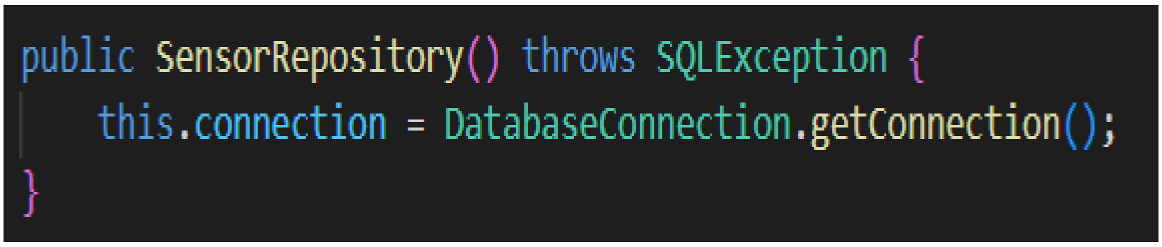
6.4 Backend Implementation (Java)

1. Database Connection (JDBC)



```java
public class DatabaseConnection {
    private static final String URL = "jdbc:postgresql://localhost:5432/vehicle_dashboard";
    private static final String USER = "postgres";
    private static final String PASSWORD = "root";

    private static Connection connection;

    public static Connection getConnection() throws SQLException {
        if (connection == null || connection.isClosed()) {
            connection = DriverManager.getConnection(URL, USER, PASSWORD);
            initializeDatabase();
        }
        return connection;
    }
```

(a)

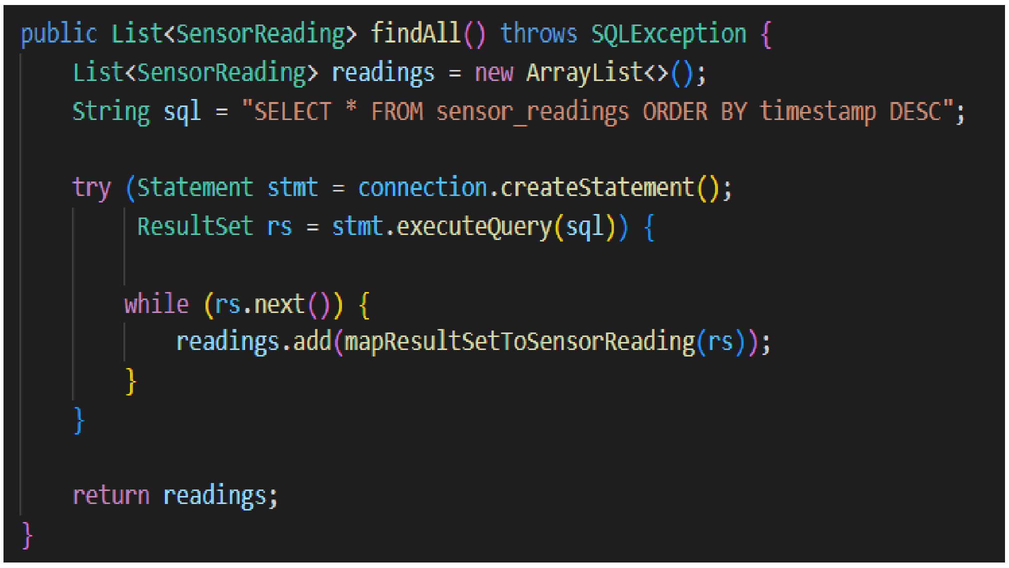


```java
public SensorRepository() throws SQLException {
    this.connection = DatabaseConnection.getConnection();
}
```

(b)

Fig 6.4

2. Fetching Latest Sensor Readings

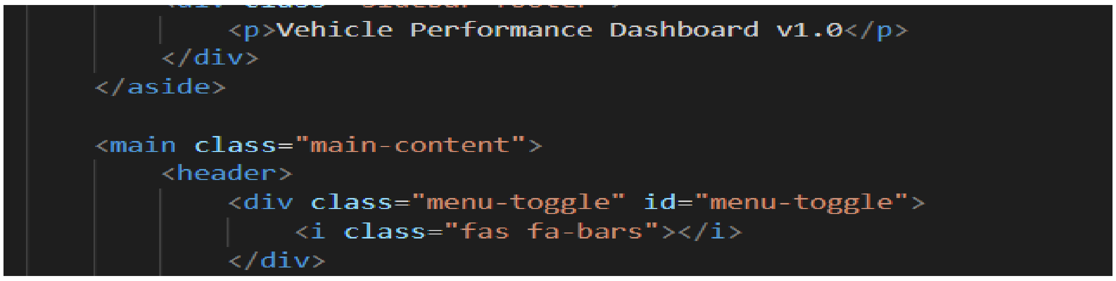


```java
public List<SensorReading> findAll() throws SQLException {
    List<SensorReading> readings = new ArrayList<>();
    String sql = "SELECT * FROM sensor_readings ORDER BY timestamp DESC";

    try (Statement stmt = connection.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            readings.add(mapResultSetToSensorReading(rs));
        }
    }

    return readings;
}
```

Fig 6.5

- **Frontend Implementation**

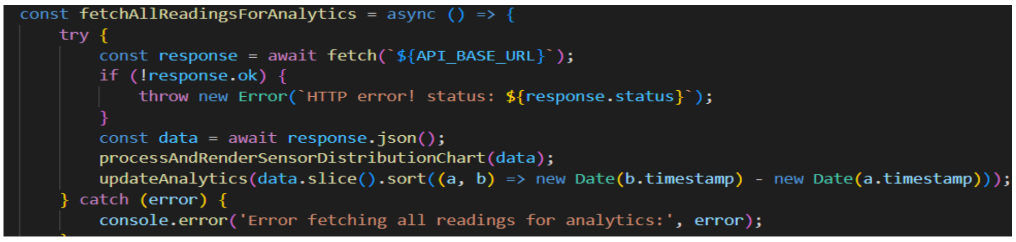
1. HTML Template



```html
                <p>Vehicle Performance Dashboard v1.0</p>
            </div>
        </aside>

        <main class="main-content">
            <header>
                <div class="menu-toggle" id="menu-toggle">
                    <i class="fas fa-bars"></i>
                </div>
```

Fig 6.6

2. JavaScript for Real-time Data

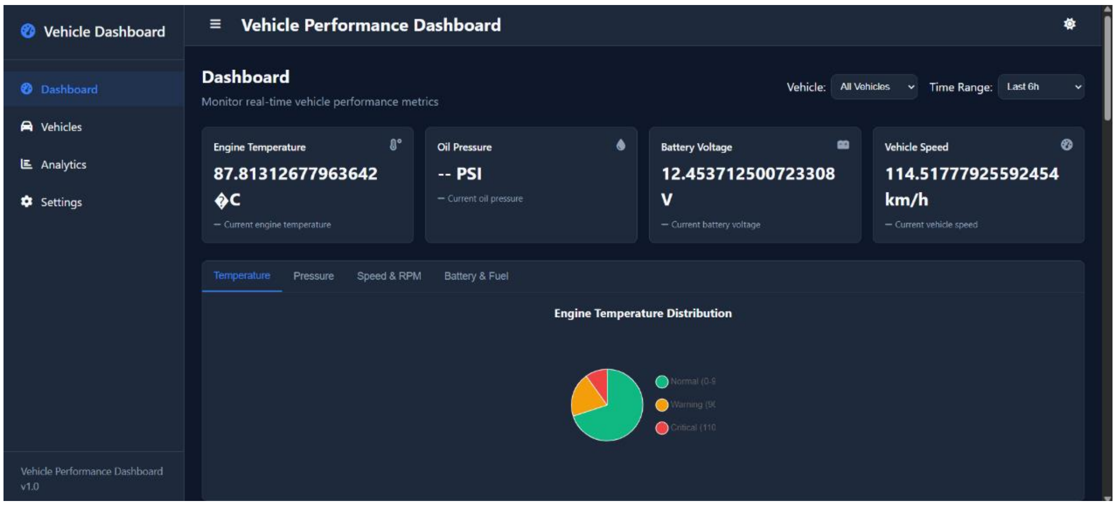


```javascript
const fetchAllReadingsForAnalytics = async () => {
    try {
        const response = await fetch(`${API_BASE_URL}`);
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const data = await response.json();
        processAndRenderSensorDistributionChart(data);
        updateAnalytics(data.slice().sort((a, b) => new Date(b.timestamp) - new Date(a.timestamp)));
    } catch (error) {
        console.error('Error fetching all readings for analytics:', error);
```

Fig 6.7

- **Screenshots**

1. Dashboard


Vehicle Dashboard

- Dashboard
- Vehicles
- Analytics
- Settings

Vehicle Performance Dashboard v1.0

≡ Vehicle Performance Dashboard

Dashboard

Monitor real-time vehicle performance metrics

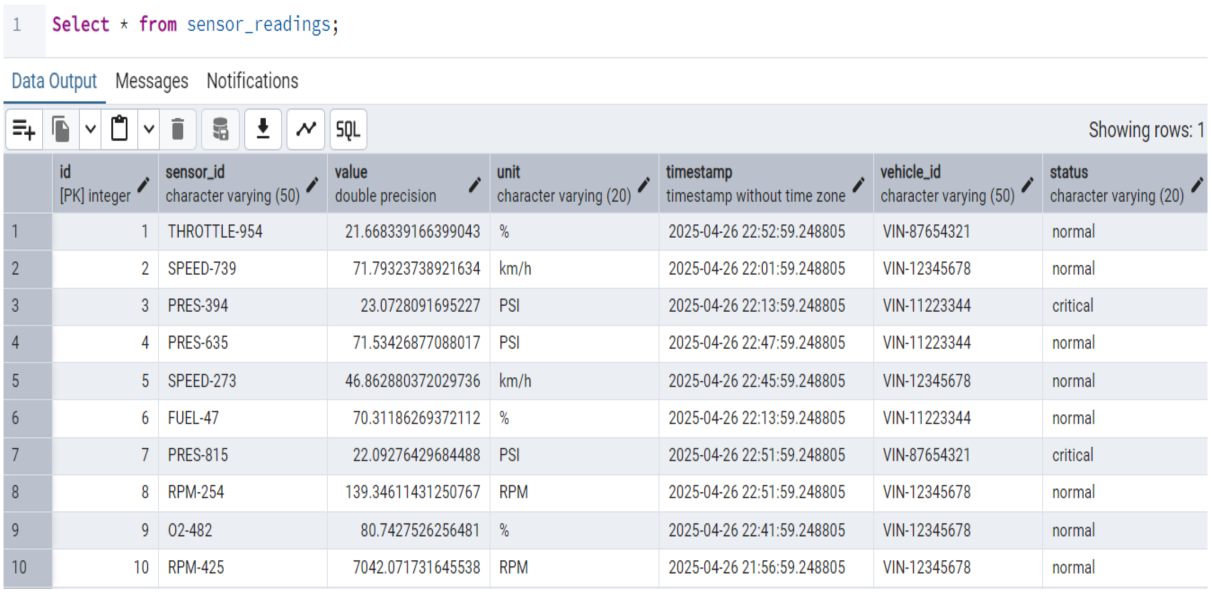
Vehicle: All Vehicles Time Range: Last 6h

Engine Temperature	Oil Pressure	Battery Voltage	Vehicle Speed
87.81312677963642 �C	-- PSI	12.453712500723308 V	114.51777925592454 km/h
— Current engine temperature	— Current oil pressure	— Current battery voltage	— Current vehicle speed

Fig 6.8

2. PostgreSQL DB table view

    a. Sensor_readings Table


```
Select * from sensor_readings;
```

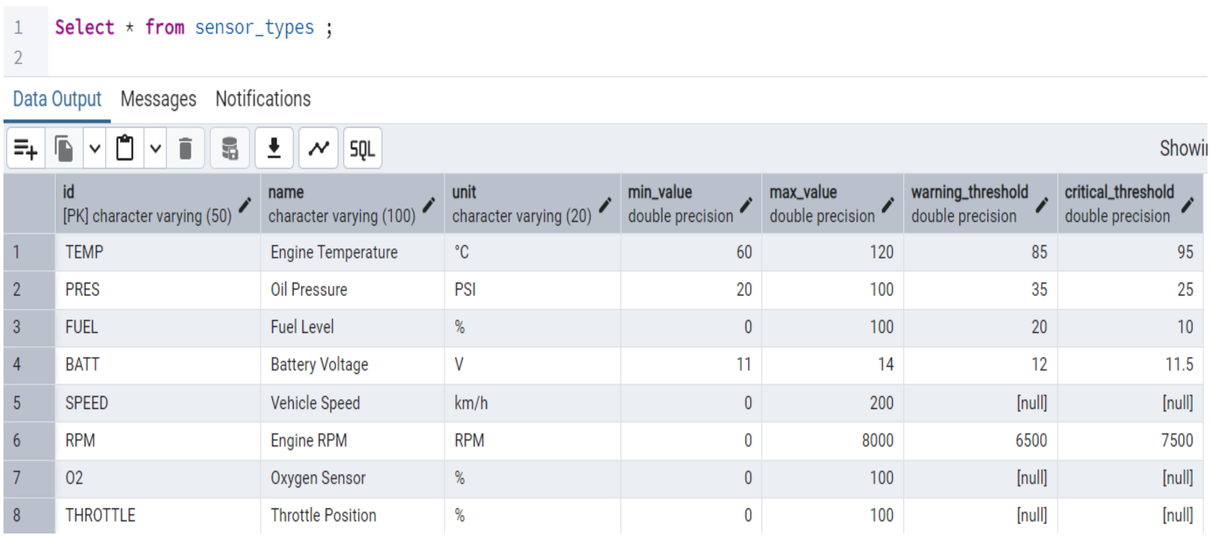
Data Output Messages Notifications

Showing rows: 1

	id [PK] integer	sensor_id character varying (50)	value double precision	unit character varying (20)	timestamp timestamp without time zone	vehicle_id character varying (50)	status character varying (20)
1	1	THROTTLE-954	21.668339166399043	%	2025-04-26 22:52:59.248805	VIN-87654321	normal
2	2	SPEED-739	71.79323738921634	km/h	2025-04-26 22:01:59.248805	VIN-12345678	normal
3	3	PRES-394	23.0728091695227	PSI	2025-04-26 22:13:59.248805	VIN-11223344	critical
4	4	PRES-635	71.53426877088017	PSI	2025-04-26 22:47:59.248805	VIN-11223344	normal
5	5	SPEED-273	46.862880372029736	km/h	2025-04-26 22:45:59.248805	VIN-12345678	normal
6	6	FUEL-47	70.31186269372112	%	2025-04-26 22:13:59.248805	VIN-11223344	normal
7	7	PRES-815	22.09276429684488	PSI	2025-04-26 22:51:59.248805	VIN-87654321	critical
8	8	RPM-254	139.34611431250767	RPM	2025-04-26 22:51:59.248805	VIN-12345678	normal
9	9	O2-482	80.7427526256481	%	2025-04-26 22:41:59.248805	VIN-12345678	normal
10	10	RPM-425	7042.071731645538	RPM	2025-04-26 21:56:59.248805	VIN-12345678	normal

Fig 6.9

    b. sensor_types Table

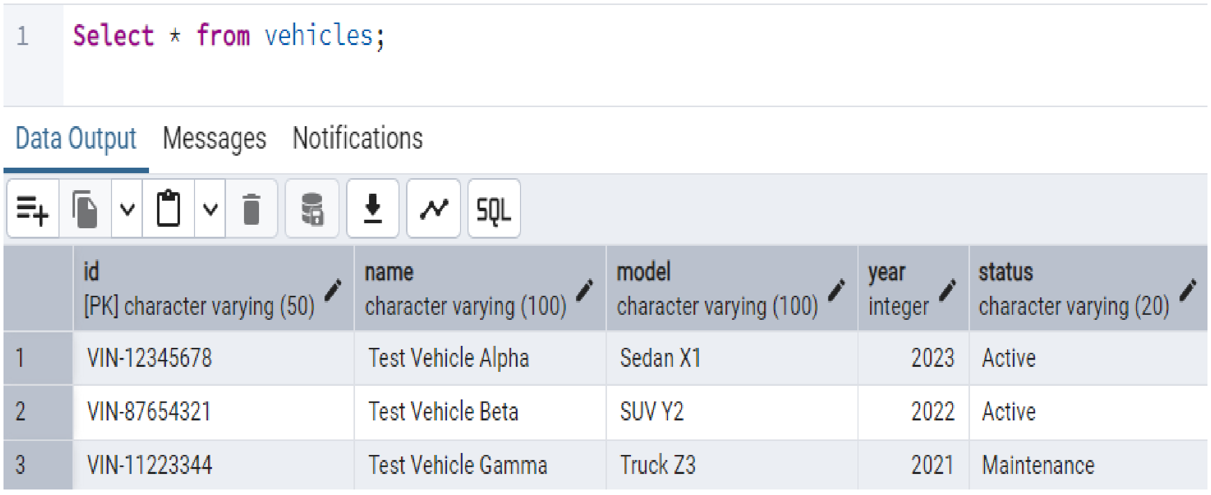


```
Select * from sensor_types ;
```

	id [PK] character varying (50)	name character varying (100)	unit character varying (20)	min_value double precision	max_value double precision	warning_threshold double precision	critical_threshold double precision
1	TEMP	Engine Temperature	°C	60	120	85	95
2	PRES	Oil Pressure	PSI	20	100	35	25
3	FUEL	Fuel Level	%	0	100	20	10
4	BATT	Battery Voltage	V	11	14	12	11.5
5	SPEED	Vehicle Speed	km/h	0	200	[null]	[null]
6	RPM	Engine RPM	RPM	0	8000	6500	7500
7	O2	Oxygen Sensor	%	0	100	[null]	[null]
8	THROTTLE	Throttle Position	%	0	100	[null]	[null]

Fig .6.10

c. vehicles Table



```
Select * from vehicles;
```

	id [PK] character varying (50)	name character varying (100)	model character varying (100)	year integer	status character varying (20)
1	VIN-12345678	Test Vehicle Alpha	Sedan X1	2023	Active
2	VIN-87654321	Test Vehicle Beta	SUV Y2	2022	Active
3	VIN-11223344	Test Vehicle Gamma	Truck Z3	2021	Maintenance

Fig 6.11

Chapter 7
Conclusion And Future Scope

7.1 Conclusion

The **Vehicle Performance Dashboard** project successfully demonstrates how real-time vehicle data can be collected, processed, and displayed using modern software technologies. This system integrates:

- **Java 8** for robust backend logic,
- **PostgreSQL** for reliable sensor data storage, and
- **HTML, CSS, JavaScript** for a dynamic, user-friendly dashboard.

The project meets its primary objective — to show real-time vehicle performance metrics using data stored in a database. The dashboard is responsive, functional, and visually informative. The modular architecture allows easy scaling and modification for future improvements.

The implementation and testing show that:

- Sensor data is fetched correctly from PostgreSQL,
- Data is displayed in a readable and intuitive format,
- The frontend and backend communicate efficiently via REST APIs.

Overall, this project provides a **useful and practical solution** for monitoring vehicles during **infield testing scenarios**, especially in automotive research and diagnostics.

- **7.2 Limitation**
- **No Offline Support**: The dashboard relies on a live connection with the backend and database; there is no caching or offline mode.

- **Limited Security**: Basic login functionality is implemented, but advanced security features like encryption, session timeout, and two-factor authentication are not included.
- **Basic Data Analysis**: The system does not offer in-depth analytics or predictive modeling (e.g., predictive maintenance).
- **Lack of Reporting Features**: Users cannot yet generate downloadable reports in Excel or PDF format.
- **Basic UI/UX**: While functional, the frontend lacks advanced UI/UX features like charts with real-time animations, dark mode, or accessibility options.

7.2 Future Scope

To make the system more powerful, user-friendly, and industry-ready, the following enhancements are proposed:

- **Real-Time Sensor Hardware Integration**

  Connect the backend with physical sensors using devices like Arduino, Raspberry Pi, or OBD-II interfaces. This would make the system suitable for direct deployment in vehicles.

- **Mobile App Interface**

  Develop a cross-platform mobile application (Android/iOS) or a responsive Progressive Web App (PWA) for field engineers to access live data from mobile devices.

- **Predictive Data Analytics**

  Use machine learning models to analyze historical data and predict issues like engine overheating, low battery, poor fuel efficiency, or worn-out brake pads.

- **Multi-Vehicle Support**

Extend the system to simultaneously monitor and display data from multiple vehicles. This is useful for fleet owners, transport companies, or vehicle testing labs.

- **Security Upgrades**

  Implement role-based access, password encryption using hashing (e.g., bcrypt), token-based authentication (JWT), and multi-factor authentication (MFA).

- **Real-Time Alerts & Notifications**
  Add SMS or email notification services for critical events like:

  1.High engine temperature

  2. Brake system failure

  3. Low fuel warning

- **Offline Data Caching**

  Use service workers or IndexedDB in browsers to temporarily store data and update the dashboard even when disconnected from the network.

- **Cloud Hosting and Scalability**

  Deploy the project on cloud platforms such as AWS, Microsoft Azure, or Google Cloud Platform to allow real-time access, load balancing, and secure backup of sensor data.

- **Report Generation Tools**

  Enable users to export vehicle performance summaries as PDF, CSV, or Excel files with charts and summaries.

- **Third-Party API Integration**

  Integrate mapping APIs (e.g., Google Maps) to show vehicle location data or use weather APIs to consider external factors affecting vehicle performance.

- **Enhanced Visualization Tools**

  Implement libraries such as Chart.js, D3.js, or Highcharts for advanced and interactive graphs and charts.

- **Voice Assistant or Chatbot Integration**

  Add a smart assistant feature to help users query the system using natural language (e.g., "Show me yesterday's fuel efficiency report.")

Chapter 8

Summary

The *Vehicle Performance Dashboard* project is a full-fledged solution aimed at addressing the need for real-time monitoring of vehicle performance during infield testing. It integrates a variety of modern and relevant technologies—including **Java 8** for backend logic, **PostgreSQL** for reliable data storage, and **HTML, CSS, and JavaScript** for an interactive and responsive user interface. The project was developed using **Visual Studio Code**, which provided a robust development environment with tools for debugging, version control, and UI development.

The dashboard fetches sensor data stored in a PostgreSQL database, processes it through a Java backend, and displays it in a visually interactive web interface. **RESTful APIs** are used to establish communication between the backend and frontend, ensuring real-time updates and a smooth user experience. The application was tested for functionality, reliability, and performance under simulated data input to validate system behavior under various testing conditions.

Although the current version has some limitations such as the absence of live hardware integration and advanced analytics, the system lays a strong foundation for future enhancements. These could include:

- **Real-time IoT sensor integration** using protocols like MQTT or WebSockets
- **Machine learning-based predictive analytics** to detect anomalies or forecast failures
- **Mobile and tablet support** through responsive design or native apps
- **Cloud-based deployment** for large-scale testing across distributed teams

The dashboard has been architected using a **modular, layered design**, separating concerns between frontend, backend, and database components. This makes the solution easy to maintain, upgrade, and scale for new requirements. Each component—frontend UI, backend processing logic, and database design—has been carefully engineered to ensure smooth integration and reliable operation.

This project also served as a hands-on learning experience in full-stack development, database modeling, software design principles, and user experience design. Implementing Java's **Collections API**, stream processing, and JDBC connectivity provided a real-world understanding of backend development. On the frontend side, integrating **asynchronous data fetching using JavaScript (AJAX/fetch)** and rendering live data using **dynamic DOM updates** helped bridge the gap between static design and functional interactivity.

From a system design perspective, **ER diagrams**, **Use Case Diagrams**, and **Data Flow Diagrams (DFD)** were created to model system behavior, data flow, and functional boundaries. This systematic planning enabled a smooth transition from concept to implementation, ensuring clarity at each stage of development.

The project also included authentication features for secure login, basic alert generation when sensor values crossed predefined thresholds, and dashboard components like speedometers, charts, and live metric panels. These features make the dashboard suitable not only for academic submission but also for use in real-world test labs and vehicle development environments.

In terms of performance, the application responds well to varying data volumes under simulated conditions and maintains a responsive UI for users. Furthermore, its modular code structure makes it easy to introduce new vehicle parameters or upgrade to more advanced data visualization libraries like D3.js or Chart.js in the future.

In conclusion, this project demonstrates a working model suitable for infield vehicle testing, offering a base framework for future extensions such as live sensor integration, predictive analytics, and mobile access. With its **scalable design**, **modular codebase**, and **real-time data visualization**, this Vehicle Performance Dashboard serves as an efficient and meaningful tool for vehicle diagnostics and performance evaluation. It reflects how theoretical computer science and software engineering concepts can be applied to solve real-world automotive challenges.

The experience gained through this project has enriched the developer’s understanding of:

- System architecture and software engineering workflows
- Client-server communication and REST API design
- Database design and query optimization
- Real-time data processing and visualization techniques

This project stands as a solid academic achievement and a strong portfolio component for practical industry exposure. It lays the groundwork for more complex and intelligent vehicle testing systems in the future, aligning with current trends in connected vehicles and smart mobility.

References

1. Oracle Java Documentation –

   https://docs.oracle.com/javase/8/docs/

2. PostgreSQL Official Documentation –

   https://www.postgresql.org/docs/

3. Mozilla Developer Network (MDN Web Docs) –

   https://developer.mozilla.org/

4. W3Schools –

   https://www.w3schools.com/

5. GeeksforGeeks –

   https://www.geeksforgeeks.org/

6. Stack Overflow –

   https://stackoverflow.com/

7. **"Design and Implementation of a Real-Time Vehicle Monitoring System"**

   International Journal of Computer Applications, 2019

8. **"Developing a Vehicle Performance Dashboard Using Java and Web Technologies"**

   Journal of Software Engineering and Applications, 2020

9. **"Real-Time Data Visualization for Vehicle Telemetry"**

   IEEE Conference on Intelligent Transportation Systems, 2018

10. **"Building RESTful APIs with Java and PostgreSQL"**

    Baeldung, 2021

11. **"Implementing Real-Time Dashboards with JavaScript and WebSockets"**

    Smashing Magazine, 2020

12. **"Java: The Complete Reference" by Herbert Schildt**

    A comprehensive resource covering Java programming, including features relevant to your project like JDBC and RESTful services.

13. **"Design and Implementation of a Real-Time Vehicle Monitoring System"**

    International Journal of Computer Applications, 2019

    This paper discusses the architecture and implementation of a real-time vehicle monitoring system using Java and PostgreSQL.

14. **"Developing a Vehicle Performance Dashboard Using Java and Web Technologies"**

    Journal of Software Engineering and Applications, 2020

15. **"Real-Time Data Visualization for Vehicle Telemetry"**

    IEEE Conference on Intelligent Transportation Systems, 2018

16. **"Real-Time Data Visualization for Vehicle Telemetry"**

    IEEE Conference on Intelligent Transportation Systems, 2018

17. **"Building RESTful APIs with Java and PostgreSQL"**

    Baeldung, 2021

    A comprehensive guide on creating RESTful APIs using Java and PostgreSQL, which can be applied to the backend of your dashboard.

"Java: The Complete Reference" by Herbert Schildt
A comprehensive resource covering Java programming, including features relevant to your project like JDBC and RESTful services.