Process MeNtOR 3.o

Uni-SEP

Content adapted from the deliverables prepared by Dr. Kostas Kontogiannis

EECS 4413 Team Project
Professor: Alvine Boaye Belle
Winter 2025
Deliverable 3

TEAM D
Vashavi Shah, Nafis Ahmed Awsaf, Nargis Rafie, Adrian Kaminski

Electric Vehicle E-commerce System **Design Document**

Version:	24
Print Date:	April 5, 2025
Release Date:	March 20, 2025
Release State:	
Approval State:	
Approved by:	
Prepared by:	Vashavi, Nafis, Nargis, Adrian
Reviewed by:	
Path Name:	
File Name:	
Document No:	

Document Change Control

Version	Date	Authors	Summary of Changes
DELIVERABLE 1			
1	*02/10/25*	*Nafis Ahmed Awsaf*	*Completed Main Page, Introduction and Shared document access to the team, Added in References.*
2	*02/11/25*	*Nafis Ahmed Awsaf*	*Completed 6. Architecture after completing part c) UML component and package diagram*
3	*02/12/25*	*Vashavi Shah*	*Added sequence diagrams and description for Use case: Registered Customers.*
4	*02/12/25*	*Vashavi Shah*	*Added Test Driven Development.*
5	*02/13/25*	*Nafis Ahmed Awsaf*	*Group Meeting Logs Updated, Updated Component Diagram and UML package Diagram with proper syntax*
6	*2/13/25*	*Adrian Kaminski*	*Completed Cloud Design Decisions as well as completed/reviewed product backlog.*
7	*2/14/25*	*Nargis Rafie*	*Added sequence diagrams and description for Use Case: Administrators (Store Owners.)*
8	*2/14/25*	*Nafis Ahmed Awsaf*	*Created the following: Main Page and Table of contents. Finalized the following sections: 2.Introductions 3. Major Design Decisions 4. Use Case Diagram 6. Architecture*
9	*2/14/25*	*Adrian Kaminski*	*Added Sequence diagrams 15 and 16 for Secondary Actor Use Cases.*
10	*2/14/25*	*Nargis Rafie*	*GANTT Diagram link with a schedule of the planned activities added.*
11	*2/14/25*	*Vashavi Shah*	*Transferred everything from the rough document to main document.*
DELIVERABLE 2			
12	*3/2/2025*	*Nafis Ahmed Awsaf*	• *Updated Introduction* • *Refined Components Diagram* • *Added Design Patterns*
13	*3/15/2025*	*Vashavi Shah*	• *Added Deployment Diagram* • *Documented Server-side Test Cases*
14	*3/16/2025*	*Nargis Rafie*	• *Added chatbot implementation*

			• *Design Patterns*
15	*3/17/20*	*Nargis Rafie*	• *GANTT Chart updated.* • *Updated Use Case Diagram and the description.* • *Updated Sequence Diagram descriptions and added links.*
16	*3/21/2025*	*Nafis Ahmed Awsaf*	• *Updated UML Package Diagram with proper relationships and entities with updated backend.* • *Description of Database Schema (evstore_DB)* • *Entity Relationship Diagram*
17	*3/21/2025*	*Nargis Rafie*	• *Updated References*
18	*3/21/2025*	*Adrian Kaminski*	• *Updated 6.2 Group Meeting Logs.* • *Cloud, Database, Troubleshooting and Security.*
19	*3/21/2025*	*Vashavi Shah*	• *Updated Architecture Section and Activities Plan* • *Transferred everything from the rough document to main document*
DELIVERABLE 3			
20	*4/2/2025*	*Vashavi Shah*	• *Updated Index* • *Added the Demo table* • *Updated the implementation of Chatbot* • *Updated Server-side implementation* • *Added Client-side implementation* • *Added all the Activity plans and group meeting logs.*
21	*4/2/2025*	*Nargis Rafie*	• *Added Chatbot flowchart* • *Updated GitHub links* • *Added the web design section* • *Updated GANTT chart*
22	*4/4/2025*	*Nafis Ahmed Awsaf*	• *Added explanation and implementation of the Security side of the system section 7.4:*

			• *Authentication & Authorization* • *Data Encryption* • *RBAC (Role Based Access Control)* • *Performance (expanded section with key metrics)* • *Section 3.4 Strengths, Weaknesses, and Insights of the System*
23	*4/5/2025*	*Adrian Kaminski*	• *Updated security and Performance.* • *Helped with 3.7 and 3.4*
24	*4/5/2025*	*Vashavi Shah*	• *Made the final edits* • *Updated the index*

Document Sign-Off

Name (Position)	**Signature**	**Date**
Vashavi Shah	Vashavi Shah	April 5, 2025
Nafis Ahmed Awsaf	Nafis Ahmed Awsaf	April 5, 2025
Adrian Kaminski	Adrian Kaminski	April 5, 2025
Nargis Rafie	Nargis Rafie	April 5, 2025

Contents

1 Demo Grading Table

Table 1 Grading criteria used for the demo

Feature to test/check during the demo (35% of Deliverables 3)	Mark	Comment made by the TA/prof
1) The system is cloud-native (4%)		
2) It is possible to register (2%)		
3) It is possible to sign in (2%)		
4) It is possible to sign out (2%)		
5) The system is secure (e.g., the system is using https in its url) (3%)		
6) It is possible to filter items (2%)		
7) It is possible to sort items (2%)		
8) It is possible to add items in the shopping cart (2%)		
9) It is possible to remove items from the shopping cart (2%)		
10) It is possible to check out items (2%)		
11) It is possible to write reviews on items and rate items using five stars (2%)		
12) It is possible to use the distinguished feature (2%)		
13) It is possible to use the chatbot (3%)		
14) It is possible to use the loan calculator (3%)		
15) It is possible to view hot deals (2%)		

2 Introduction

2.1 Purpose

The EvStore Project 4413 is an e-commerce platform dedicated to transforming how people shop for electric vehicles online. Our goal is to create a platform that simplifies the shopping process and delivers a secure, scalable, and high-performing experience. The system emphasizes core functionalities such as user authentication, product catalogue management, order processing, and payment integration to build a solution that users can trust and enjoy. The primary goal of the system is to offer a secure, scalable, and high-performing online shopping experience while ensuring ease of maintenance and extensibility.

2.2 Overview

The EvStore Project 4413 is built using modern cloud-native principles, meaning it leverages cloud-based infrastructure to deliver auto-scaling, high availability, and fault tolerance. The application is developed with Java EE (Jakarta EE) technologies and provides enterprise-grade robustness. Key technologies include:

- ***Spring Boot & Jakarta EE:** For a robust and reliable backend.*
- ***ReactJS:** For a user-friendly frontend.*
- ***Spring Security**: Ensuring secure authentication and authorization.*
- ***RESTful APIs:** Facilitating seamless client-server communication.*
- ***Microservices & Containerization (Docker & Kubernetes):** Delivering scalability and flexibility.*
- ***AWS:** Providing robust cloud hosting and efficient database management.*

This architecture is optimized for cloud deployment, ensuring high availability, security, and performance while maintaining the flexibility needed for future enhancements.

Key Features:

- ***User Authentication:** Secure registration, login, and profile management using Spring Security and JWT.*
- ***Product Catalog:** Interactive listings with advanced filtering, sorting, and 360° views.*
- ***Order & Payment:** Smooth shopping cart and secure checkout with integrated payment gateways.*
- ***Cloud-Native Architecture:** Auto-scaling, high availability, and fault tolerance via containerization on AWS.*
- ***Microservices Design:** Modular backend for independent scaling and easy maintenance.*
- ***Chatbot Support:** A lightweight, React-based chatbot that interacts with users in real-time, fetching user details, car information, and order status through RESTful API calls.*
- ***Admin Dashboard:** Lets administrators generate sales reports, view pending orders, update order statuses, manage vehicle listings and see user lists – all with secure, real-time backend communication via Axios*

2.3 Resources – References

- *https://www.lucidchart.com/pages/uml-component-diagram*
- *https://www.sencha.com/blog/web-application-development-top-frameworks/*
- *https://eclass.yorku.ca/pluginfile.php/6912126/mod_resource/content/1/EECS%20EECS4413_Week2_Slideset%201_AlvineBelle.pdf*
- *https://eclass.yorku.ca/pluginfile.php/6955259/mod_resource/content/1/EECS%20EECS4413_Week5_Slideset%201_AlvineBelle.pdf*
- *https://spring.io/guides/tutorials/rest*
- *https://www.w3schools.com/js/js_json_intro.asp*
- *https://www.w3schools.com/REACT/DEFAULT.ASP*
- *https://creately.com/blog/diagrams/sequence-diagram-tutorial/*
- *https://www.plutora.com/blog/deployment-diagrams-explained-in-detail-with-examples*
- *https://www.digitalocean.com/community/tutorials/java-session-management-servlet-httpsession-url-rewriting*
- *https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction*
- *https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#what-is-rest%2030*

3 Major Design Decisions

This section outlines the significant design choices and modularization criteria that shape our Electric Vehicle(EV) store E-commerce System. Our decisions focus on creating a robust, secure, and scalable system while adhering to REST principles and leveraging industry-standard technologies and design patterns.

3.1 Server-Side Implementation and REST Principles

Our backend is built on Java Spring Boot and follows a layered, modular architecture designed for robustness, scalability, and security. We strictly adhere to REST principles, ensuring that every endpoint is stateless and resource-based. This means that each URL uniquely identifies a resource (such as a user, product, or order), and our clients interact with these resources using standard HTTP methods (GET, POST, PUT, DELETE).

Key features:

- ***Stateless Communication****: Each request to the API carries all the information needed to process it. We avoid maintaining a session state on the server side, which not only simplifies scaling but also enhances security.*
- ***Uniform Interface:*** *Our endpoints are designed with a consistent structure and naming convention, (e.g. /users, /products, /orders), and standard HTTP status codes are used to communicate the results of each operation..*
- ***Resource-Based Design:*** *We structure our API around key business entities. Each resource, whether it's a user, product, or order, is represented as a unique URI, making the API intuitive and aligned with REST best practices.*
- ***JSON as the Data Exchange Format:*** *To facilitate smooth communication between the front end and the back end, we use JSON as our data exchange format. This lightweight and human-readable format is widely supported across different platforms.*

Backend Structure Overview:

- ***Configuration****: config/SecurityConfig.java configures Spring Security and JWT to secure our endpoints elegantly.*
- ***Controller Layer:*** *The controller package holds all our REST controllers (e.g., AdminReportController.java, AuthController.java, CartController.java, OrderController.java, ReviewController.java, VehicleController.java, etc.). These controllers handle incoming HTTP requests and delegate processing to the appropriate services.*
- ***DTO Layer:*** *In the* DTO *package, we have our Data Transfer Objects (e.g.,* CartDTO.java, OrderDTO.java, RegisterRequest.java*, etc.) that ensure efficient and secure data exchange between the client and server.*
- ***Model Layer:*** *The* model *package defines our core business entities such as* User.java, Vehicle.java, Orders.java*, and* Review.java*. These classes directly map to our database tables.*
- ***Repository Layer:*** *Our* repository *package includes Spring Data JPA repositories (e.g.,* UserRepository.java, VehicleRepository.java, OrderRepository.java*) that abstract the complexity of database interactions and provide robust CRUD operations.*
- ***Service Layer:*** *The* service *package encapsulates our business logic. Services such as* AuthenticationService.java, OrderService.java, *and* ReportService.java *implement specific business rules and processes.*

- ***Exception Handling:*** *The* exception *package defines custom exceptions (e.g.,* InvalidCredentialsException.java, ResourceNotFoundException.java*) to manage error conditions gracefully.*
- ***Security & Utilities:*** *Under* security*, classes like* JwtTokenFilter.java *ensure secure token validation, while utilities like* JwtSecretGenerator.java *support cryptographic operations.*
- ***Database Integration:*** *The* database *package manages the connection to our underlying relational database, ensuring smooth data persistence.*
- ***Entry Point & Initialization:*** Evstoreproject4413Application.java *and* ServletInitializer.java *serve as the main entry points to bootstrap our Spring Boot application.*

3.2 Client-Side Implementation and UI Principles

Our frontend is developed using ReactJS combined with CSS for styling, creating a dynamic, responsive, and modular single-page application (SPA). This design emphasizes reusability and maintainability while mirroring the RESTful architecture of our backend to provide a seamless end-to-end experience.

Key features

- ***Single-Page Application (SPA):*** *The entire user interface is built as a SPA, reducing full-page reloads and delivering a fluid, app-like experience.*
- ***Component-Based Architecture:*** *Each UI section (e.g., Product Catalog, Shopping Cart, Admin Dashboard) is encapsulated within its own dedicated React component. This modular approach promotes high cohesion and simplifies testing and maintenance.*
- ***State Management with Hooks:*** *We use React Hooks (e.g., useState, useEffect) to manage component-level state and side effects. This functional approach keeps our logic organized and improves code readability.*
- ***Axios for HTTP Requests:*** *Communication with our Spring Boot backend is handled via Axios, which simplifies sending and receiving JSON data. Each API request includes the necessary headers (such as JWT tokens) to ensure secure, authenticated access.*
- ***JWT-Based Authentication:*** *User credentials are validated by the server, and the resulting authentication token (JWT) is stored client-side (typically in localStorage). Subsequent API calls include this token in the headers, ensuring stateless and secure interactions with protected endpoints.*
- ***Styling and Responsiveness:*** *We use plain CSS (and additional style libraries if needed) to provide a consistent look and feel across the site. Our responsive layouts ensure an optimal user experience on both desktop and mobile devices.*
- ***Routing (Optional):*** *When using React Router (or a similar library), we enable clean, client-side routing without full-page refreshes, effectively mirroring our REST endpoints on the front end.*

Frontend Structure Overview:

- ***Core Files & Entry Point:*** index.js *bootstrap our React application, while* App.js *acts as the main container component for the entire UI.*
- ***Component-Based Architecture:*** *The* components *directory contains reusable components such as:*

   - **Chatbot.js***: Implements a lightweight, React-based chatbot that simulates conversation flows using React Hooks for state management and Axios for backend API calls.*
   - **AdminDashboard.js***: Provides administrators with features like generating sales reports, viewing pending orders, updating order statuses, managing vehicle listings, and accessing a user list.*
   - ***Additional components*** *like* AddToCartComponent.js, Cart.js, Checkout.js, Login.js, Register.js*, etc., each encapsulate specific functionalities.*
- ***Page-Specific Components:*** *The* pages *directory holds view-specific components (e.g.,* Model3Page.js, ModelSPage.js, ModelXPage.js*) that represent different car models of our application.*
- ***Asset Management:*** *The* assets *folder contains images and other media (such as car images and logos) that enhance the UI.*
- ***Styling:*** *Global styles are maintained in* index.css*, with additional style sheets in the* styles *folder (including* AdminDashboard.css, Chatbot.css, CheckoutPage.css*, etc.), ensuring a consistent and responsive design across devices.*
- ***API Integration:*** *The* API.js *file centralizes our HTTP communication logic (using Axios), ensuring secure, authenticated interactions with the backends' RESTful endpoints.*

3.3 Technologies and Integration

- ***Backend Framework****: We chose Java Spring Boot due to its rapid development capabilities and built-in support for security and straightforward data management via Spring Data JPA.*
- ***Frontend Framework:*** *ReactJS is chosen to create a dynamic and intuitive user interface that engages users effectively.*
- ***Database:*** *Our system uses MySQL/PostgreSQL, accessed via Spring Data JPA, to ensure reliable and scalable relational data storage.*
- ***Authentication:*** *Security is handled by Spring Security with JWT tokens, safeguarding our endpoints and managing user sessions.*
- ***Cloud Integration:*** *By deploying our application on AWS, we achieve automatic scalability, high availability, and robust performance in a production-grade environment.*
- ***Chatbot Integration:*** *While we initially planned to integrate Amazon Lex, we now use a custom React-based chatbot that leverages React Hooks for state management and Axios for API calls. This approach provides a streamlined conversational flow, allowing users to inquire about car details, order statuses, and more.*
- ***Development Environment:*** *Visual Studio Code is our primary tool for coding, ensuring consistency throughout the development process*

3.4 Chatbot Implementation

Purpose:

- ***Customer Support:*** *Quickly answer questions about car specifications, prices, and availability.*
- ***Product Recommendations:*** *Assist users in choosing a car based on budget and features.*
- ***Order Assistance:*** *Provide support for order tracking and delivery updates.*

Technology Stack:

- ***Backend:*** *Java Spring Boot for API development.*
- ***Frontend:*** *React.js is used to build a custom chatbot widget. This implementation leverages React Hooks for state management and Axios for HTTP communication.*
- ***NLP:*** *A simple, rule-based logic is implemented to process user input and drive the conversation flow.*
- ***Database:*** *MySQL stores the core application data, while conversation logs can be managed as needed.*

Development and Integration:

- ***API Development:*** *RESTful endpoints in Spring Boot handle requests such as fetching user data based on email and retrieving car details.*
- ***Frontend Integration:*** *The chatbot widget is fully embedded within the React app. It uses a state-driven approach to simulate a natural conversation flow.*
- ***Rule-Based Processing:*** *Based on the conversation state (e.g., asking for email, selecting a car model, inquiring about features), the chatbot processes user inputs and responds accordingly using hard-coded data for quick responses.*
- ***Axios Communication:*** *Axios is used to call backend endpoints, ensuring secure and authenticated data exchange.*

Conversation Flow:

- ***Greeting & Email Prompt:*** *The chatbot starts by welcoming the user and asking for their email.*
- ***User Verification:*** *Upon receiving the email, it calls the backend API to fetch user details and personalizes the greeting.*
- ***Main Options:*** *The chatbot presents options such as "Available Cars," "Order Status," and "Something else."*
- ***Car Details:*** *If the user selects "Available Cars," the chatbot prompts for a car model (Model 3, Model S, or Model X), fetches the corresponding car details from the backend, and displays them.*
- ***Feature Exploration:*** *The user can then request specific features about the selected car. The chatbot uses a pre-defined set of hard-coded feature descriptions to provide immediate responses.*
- ***Additional Queries:*** *For queries such as order status or other issues, the chatbot processes the input and responds with appropriate messages, guiding the user through the conversation.*

Chatbot Flowchart Diagram (Conceptual): ***view the flowchart here.***

3.5 Significant Design Choices

*We leveraged several **design patterns** to enhance the modularity and maintainability of our system:*

- ***MVC (Model-View-Controller):** This pattern separates our data model, business logic, and presentation layer. The REST controllers and React-based front end work together seamlessly while the back-end services manage the business logic.*
- ***Repository Pattern:** Employed through Spring Data JPA, this pattern abstracts the complexity of database operations.*
- ***Singleton Pattern:** Critical services, such as database connection pools, are maintained as single instances throughout the application.*
- ***Factory Pattern:** This allows us to dynamically create service objects, making the system more flexible when handling various business scenarios.*

*We have implemented a clear **three-tier (layered) architecture**:*

- ***Presentation Layer:** Managed by React components that provide responsive user interactions for browsing, shopping, and checkout.*
- ***Business Logic Layer:** Composed of Spring Boot services, this layer implements key functions like user registration, order creation, and payment validation.*
- ***Data Layer:** Data persistence is handled by MySQL/PostgreSQL via JPA repositories, which keep the data layer separate from business logic.*

*Additionally, our architecture benefits from **several Java EE patterns:***

- ***Front Controller:** The Spring Boot REST API serves as the entry point for all requests.*
- ***DAO (Data Access Object):** This pattern isolates our persistence logic from the rest of the application.*
- ***Service Layer:** Coordinates the business rules and interactions between the DAO layer and REST controllers.*

3.6 Modularization Strategy

To ensure our system is maintainable and adaptable, we organize it with the:

- ***High Cohesion:** Every module is dedicated to a single responsibility. For example, the shopping cart module solely manages adding/removing items and computing totals, while the order management and payment processing modules work independently.*
- ***Low Coupling:** Modules interact only through clearly defined interfaces, which minimizes interdependencies. This means changes in one module (such as updating order processing logic) do not ripple through to other parts of the system.*

We've broken down the system into modules based on their functional responsibilities:

- ***Separation of Concerns:** Different areas like authentication, catalogue management, ordering, and user management are implemented in isolated modules.*
- ***Frontend and Backend Communication:** Our React-based UI interacts with the Spring Boot API using RESTful calls, preserving a clean separation between the presentation and business logic.*

- ***Integration with External Services (Deliverable 3):*** *By incorporating AWS for cloud deployment, Amazon Lex for the chatbot, a Payment Gateway for transaction processing, and an Email Service for notifications, our system remains highly modular and extensible.*

3.7 Strengths, Weaknesses and Insights of the System

- ***Security***
  Our system is architected with a strong emphasis on end-to-end security to protect both users and data integrity. Authentication and authorization are enforced using Spring Security integrated with JWT-based authentication. Every user who registers is automatically assigned the ROLE_USER, while ROLE_ADMIN is granted only through controlled backend seeding or admin registration workflows.

  Passwords are securely hashed using BCrypt before being stored in the database, ensuring no plaintext credentials are exposed. The tokens are signed using HMAC and stored securely in localStorage on the client-side, and any blacklisted token (e.g., post-logout) is invalidated by a dedicated blacklist service.

  All client-server communications are encrypted using HTTPS, which leverages TLS to protect data in transit. Additionally, sensitive elements in JWT (e.g., user roles and email) are cryptographically verified on every request. Where applicable, AES encryption is applied for server-side cryptographic operations, offering further protection of sensitive data.
  Access control is strictly enforced using Role-Based Access Control (RBAC). Endpoints are guarded based on the user's authority level using .hasAuthority("ROLE_USER") and .hasAuthority("ROLE_ADMIN"). This granular level of authorization ensures unauthorized access attempts result in 403 Forbidden, maintaining strict boundaries between user roles.
  Furthermore, the system benefits from input validation and Spring Security's built-in CSRF protection to mitigate injection and cross-site attacks. As demonstrated in the attached database screenshot, even the admin cannot access plaintext passwords, and roles are managed securely.
- ***Performance***
  Our application demonstrates commendable performance metrics validated through both Apache JMeter load testing and WebPageTest.org results.
  From the WebPageTest, the following key performance indicators were observed for our production deployment at https://evstoreproject4413.com:
  - ***Time to First Byte (TTFB): 0.781 seconds*** *— This reflects the responsiveness of the backend server. A TTFB under one second indicates fast backend processing and minimal server latency.*
  - ***Start Render: 2.500 seconds*** *— Users begin seeing page elements within 2.5 seconds, improving the perception of speed.*
  - ***First Contentful Paint (FCP): 2.533 seconds*** *— The point at which key content becomes visible. This is a strong metric for perceived interactivity.*
  - ***Speed Index: 4.622 seconds*** *— While slightly above the ideal benchmark, this shows relatively efficient page rendering.*
  - ***Largest Contentful Paint (LCP): 2.533 seconds*** *— This is well within the optimal range, meaning major content like images are quickly visible.*
  - ***Cumulative Layout Shift (CLS): 0.0*** *— Perfect layout stability with no unexpected shifts.*

- ***Total Blocking Time (TBT): 0.0 seconds** — The main thread is never blocked, ensuring uninterrupted interactivity.*
- ***Page Weight: 5,098KB** — The high weight is mainly due to image assets; optimization is advised using compressed or next-gen formats.*

In addition to these front-end metrics, we implemented backend performance boosters:

- ***Caching:** Frequently accessed data such as vehicle listings and user session tokens are cached using Redis, reducing database round-trips and improving API latency.*
- ***Load Balancing:** Traffic is distributed using an AWS Application Load Balancer (ALB) or Nginx, enabling horizontal scalability and fault tolerance.*
- ***Database Optimization:** Indexing (on userId, email, and orderStatus), query tuning, and connection pooling via HikariCP significantly reduce latency and improve concurrency handling.*

Apache JMeter tests were conducted for varying levels of concurrency (1–10, 1–100, 1–1000 users), targeting endpoints like product listing, review submission, and review retrieval. These stress tests confirm the backends' ability to handle high loads without degradation in performance or response times.

***Integration with External Services (Deliverable 3):** By incorporating AWS for cloud deployment, Amazon Lex for the chatbot, a Payment Gateway for transaction processing, and an Email Service for notifications, our system remains highly modular and extensible.*

4 Comprehensive Database Schema & ER Diagram

evstore_db: *https://github.com/awsaf11/evstoreproject4413/blob/main/Dump20250318.sql*

4.1 Core Data Structures

The EV Store application utilizes a MySQL relational database to efficiently manage electric vehicle sales. The schema is carefully designed to ensure data normalization, integrity, and robust validations through primary/foreign keys and constraints, while relationships are elegantly managed using JPA annotations in a Spring Boot backend. The database schema consists of the following key tables:

1. vehicles: Stores all vehicle configuration options and metadata.

- ***vehicle_id*** *(Primary Key, BIGINT) – Unique identifier for each vehicle.*
- ***type*** *(VARCHAR) – Can be New or Used.*
- ***model*** *(VARCHAR) – Limited to Model S, Model 3, Model X.*
- ***payment*** *(VARCHAR) – Options: Cash, Finance.*
- ***trim**, **paint**, **wheels**, **interior** – Vehicle customization options with limited values via CHECK constraints.*
- ***seat_layout*** *(VARCHAR) – Options: Five Seat Interior, Seven Seat Interior.*
- ***performance_upgrade**, **tow_hitch**, **hot_deal** (BOOLEAN/bit(1)) – Optional upgrades and promotions.*
- ***price*** *(DECIMAL(10,2)) – Final configured price of the vehicle.*
- ***year**, **mileage**, **cover_image**, **exterior_colour**, **tires**, **range**, **top_speed**, **kmh** – Additional descriptive fields.*

2. orders: Tracks purchases made by users.

- ***order_id*** *(Primary Key)*
- ***user_id*** *(Foreign Key to users)*
- ***total_amount*** *(DECIMAL)*
- ***order_status*** *(e.g., Pending, Completed)*
- ***created_at**, **payment_method***

3. order_items: Handles the many-to-many relationship between orders and vehicles.

- ***order_item_id*** *(Primary Key)*
- ***order_id*** *(Foreign Key)*
- ***vehicle_id*** *(Foreign Key)*
- *Each row represents one vehicle in one order.*

4. users: Store information for each registered customer or admin.

- ***user_id*** *(Primary Key)*
- ***email**, **first_name**, **last_name**, **password**, **role**, etc.*

5. reviews: Users can leave reviews on vehicles.

- ***review_id*** *(Primary Key)*
- ***vehicle_id*** *(Foreign Key)*
- ***user_id**, **rating**, **comment***

*This schema ensures **normalization**, and clear **entity relationships**, and allows for robust **data validation** through constraints. Entity relationships are managed via JPA annotations in the Spring Boot backend (e.g., @OneToMany, @ManyToOne.)*

6. cart: Each user has a shopping cart for holding unconfirmed vehicle selections.

- *cart_id (PK)*
- *user_id (FK → users) – Owner of the cart.*
- *created_at – When the cart was initialized.*

7. cart_items: Items currently added to a user's cart.

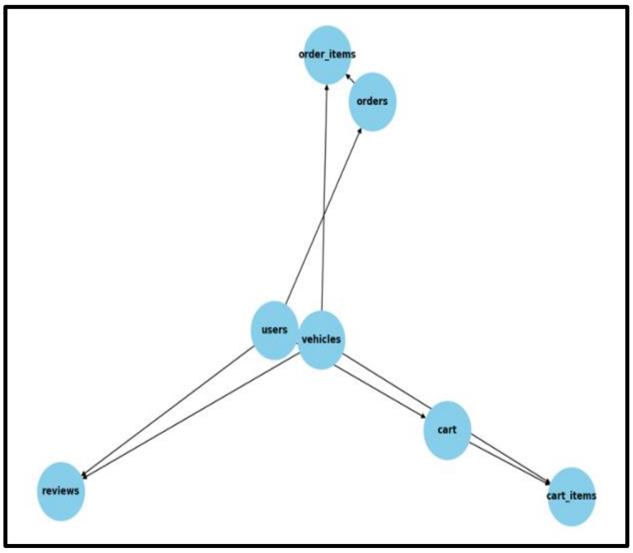
- *cart_item_id (PK)*
- *cart_id (FK → cart) – Which cart does this item belong to.*
- *vehicle_id (FK → vehicles) – Selected vehicle.*
- *Supports multiple configurable vehicles per cart.*

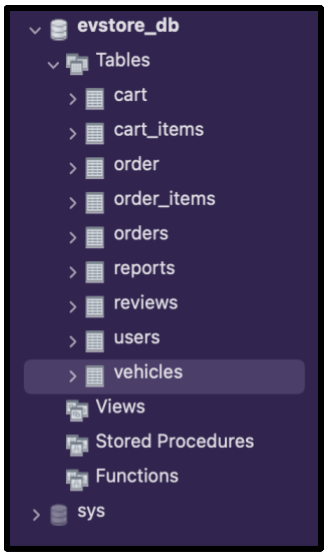
8. reports: Stores generated admin reports, such as sales summaries.

- *report_id (PK)*
- *report_type – Type of report (e.g., Sales, Inventory).*
- *report_data – JSON/text-based report content.*
- *created_at – Timestamp of generation.*
- *Used for admin dashboard and data analysis.*

4.2 Entity Relationship (ER) Diagram

This ER diagram represents the relationships between major entities in the EV Store database.





5 Use case Diagram

*This use case diagram represents the interactions between three main actors — **Guest**, **Registered Customer**, and **Administrator (Store Owner)** — in an electric vehicle e-commerce platform.*

- ***Guests** can browse available electric vehicles, view vehicle details, check current hot deals, ask the chatbot for help, and register for an account if they wish to make a purchase.*
- ***Registered Customers** have access to advanced functionalities, including signing in/out, filtering and sorting vehicles, comparing options, customizing vehicles, using a loan calculator, adding items to the cart, proceeding to checkout, writing reviews, and interacting with the chatbot.*
- ***Administrators** manage vehicle listings, run sales and usage reports, and edit user accounts to keep the platform running smoothly.*

The diagram highlights a seamless journey from browsing as a guest to becoming a registered customer and provides powerful tools for administrators to maintain system efficiency.

View the Use Case Diagram Here.

6 Sequence Diagrams (SD)

6.1 Registered Customers

SD1: Register: *View Register (UC1) SD Here.*
The customer submits their registration details (username, password, and email) via the front end. The API Gateway validates the input and checks for duplicate accounts by querying the Authentication Service and Vehicle Database. If valid, the password is hashed and stored. The customer receives a success message or error feedback if there are issues with the input or account duplication.

SD2: Sign In and Sign Out (UC2 & UC3): *View Sign In and Sign Out (UC2 & UC3) SD Here.*
The customer logs in by entering their username and password, which the API Gateway sends for validation. If correct, a JWT session token is issued. For sign-out, the customer requests logout through the front-end, which invalidates the session, confirming successful log-out or an error.

SD3: Listing Electric Vehicles and Viewing Hot Deals (UC4 & UC8): *Viewing Electric Vehicles and Viewing Hot Deals (UC4 & UC8) SD Here.*
The customer requests vehicle listings or hot deals, and the API Gateway interacts with the Catalog Service and Vehicle Database to fetch relevant data. If vehicles or deals are available, they are shown, otherwise, the customer is notified.

SD4: Sorting and Filtering Vehicles (UC5 & UC6: *View Sorting and Filtering Vehicles (UC5 & UC6) SD Here.*
The customer selects sorting options or applies filters. The API Gateway queries the

Catalog Service, retrieves sorted or filtered vehicles, and displays the results. If no matches are found, an appropriate message is shown.

SD5: Viewing Vehicle Details (UC7): *View Viewing Vehicle Details (UC7) SD Here.*
The customer selects a vehicle, prompting the system to query the Vehicle Database for detailed information. The details, such as specifications and pricing, are displayed, or an error message is shown if the vehicle isn't found.

SD6: Use Loan Calculator (UC9): *View Use Loan Calculator (UC9) SD Here.*
The customer enters loan parameters into the Loan Calculator. The system validates the inputs and calculates the monthly payments, querying the Vehicle Database for prices when necessary. Results are shown, or error messages are displayed for invalid inputs.

SD7: Compare Vehicles (UC10): *View Compare Vehicles (UC10) SD Here.*
The customer selects multiple vehicles to compare, and the system fetches data from the Vehicle Database. The comparison is displayed side-by-side. If no data is available, an error message is shown.

SD8: Customize Vehicle (UC11): *View Customize Vehicle (UC11) SD Here.*
The customer requests customization options for a vehicle. The system queries the Vehicle Database, and available options are displayed. Once selected, the customizations are saved. Errors are shown if no options are available or if the customizations can't be saved.

SD9: Add to Cart and Edit/Remove from Cart (UC12 & UC13): *View Add to Cart and Edit/Remove from Cart (UC12 & UC13) SD Here.*
The customer adds a vehicle to their cart. The system checks availability and confirms success. For editing or removal, the cart is updated, and a success message is shown or an error if there's an issue.

SD10: Checkout (UC14): *View Checkout (UC14) SD Here.*
The customer proceeds to checkout by providing payment and shipping details. The system validates the session, processes payment via the Payment Service, and stores the order. A success message is shown after payment confirmation or an error if payment fails.

SD11: Write Reviews & Rate Vehicles (UC15): *View Write Reviews & Rate Vehicles (UC15) SD Here.*
After vehicle delivery, the customer is prompted to write a review. The review and rating are saved in the Vehicle Database, and the customer receives confirmation. If an image is uploaded, it is linked to the review.

SD12: Ask Chatbot for Help (UC16): *View Ask Chatbot for Help (UC16) SD Here.*
The customer submits a query via the front end, which the Chatbot Service processes. The chatbot responds, querying relevant databases for information when needed. The response is returned to the customer for assistance.

SD13: 360-Degree View and Zoom Feature (UC17): *View 360-Degree View and Zoom Feature (UC17) SD Here.*
The customer accesses the 360-degree view or zoom feature for a vehicle. The system

provides a 3D model or high-resolution images from the database. If unavailable, a message informs the customer.

6.2 Administrators:

SD14: Run Reports and Manage Listings and User Accounts (UC18): *View Register (UC1) SD Here.*
The customer submits their registration details (username, password, and email) via the front end. The API Gateway validates the input and checks for duplicate accounts by querying the Authentication Service and Vehicle Database. If valid, the password is hashed and stored. The customer receives a success message or error feedback if there are issues with the input or account duplication.

SD15: Process Payments (UC22): *View Process Payments (UC22) SD Here.*
The customer submits payment details, which are validated by the backend API and processed by the Payment Gateway. Upon successful payment, the order is updated, and a confirmation is sent. If declined, an error message is displayed.

SD16: Send Notifications (Notification Service): *View Send Notifications (Notification Service) SD Here.*
The Notification Service processes notifications, including purchase confirmations, hot deals, and promotional emails. Notifications are delivered via email/SMS, and failure handling or opt-out preferences are managed.

7 Architecture

This section details our system's architecture through refined component and package diagrams, module descriptions, and our cloud deployment strategy. Our design not only meets functional requirements but also supports key quality attributes.

7.1 System Decomposition (Package and Component Diagram)

Package Diagram: *View the Package Diagram Here.*

Our package diagram organizes the system into several distinct tiers:

- ***Frontend Package:***
  - ***Technologies:*** *React.js and CSS*
  - ***Role:*** *Manages user interfaces such as the Login, Registration, Admin, Vehicle Catalogue, Chatbot, Order Now and Checkout pages*
- ***Backend Package:***
  - ***Technologies:*** *Java Spring Boot REST API with Spring Security (JWT)*
  - ***Role:*** *Processes business logic, handles authentication, and manages data processing.*
- ***Database Package:***
  - ***Technologies:*** *MySQL/PostgreSQL*
  - ***Role:*** *Stores persistent data including user profiles, product listings, and order details.*
- ***Cloud Integration Package***
  - ***Technologies:*** *AWS services (such as EC2/Elastic Beanstalk for the backend, S3/CloudFront for the frontend, and RDS for the database)*
  - ***Role:*** *Ensures our system is deployed in a scalable, reliable, and cloud-native environment.*

Component Diagram: *View the Component Diagram Here.*

Our component diagram further breaks down the backend into interacting modules:

- ***Front-End Components***
  - ***Catalog View:*** *Displays product listings with advanced filtering and 360° view and zoom features.*
  - ***Shopping Cart View:*** *Manages item additions, removals, and price calculations.*
  - ***Checkout View:*** *Collects payment information and confirms orders.*
  - ***Registration & Login Views:*** *Allows users to create accounts, log in, and maintain sessions.*
  - ***Chabot:*** *Allows users to answer basic queries regarding car models and order history.*
  - ***Admin/Analytics View:*** *Shows sales data, usage metrics, and system reports for administrators.*
- ***Back-End Components***
  - ***Catalog Service:*** *Handles listing, filtering, and sorting of products.*
  - ***Ordering Service:*** *Processes shopping cart items, checkout flows, and payment integration.*
  - ***Identity Management Service:*** *Manages user registration, authentication, and session tokens.*
  - ***Analytics Service:*** *Aggregates data for sales and usage reports.*
  - ***Data Access Component:*** *Central point for database operations (via JPA repositories).*

- ***Web API Gateway:***
  - ***Unified Entry Point:*** *Routes requests from the frontend to the correct service.*
  - ***Cross-Cutting Logic:*** *Can manage caching, logging, or rate limiting.*

7.2 Description of Modules

Modules			
Module Name	**Description**	**Exposed Interface Names**	**Interface Description**
M1 Ordering Service	*"Manages shopping cart, checkout, and transaction processing.*	*M1:I1* *M1:I2*	*M1:I1 "Place vehicle order"* *M1:I2 "Obtain order details"*
M2 Catalogue Service	*"Handles vehicle listings, sorting, filtering, and comparison."*	*M2:I3*	*M2:I3 "Retrieve product details"*
M3: Identity Management Service	*"Manages user authentication (sign in, sign-out, sign up)*	*M3:I4*	*M3:I4 "Verify User"*
M4: Analytics Service	*"Generates reports on sales trends and user activity."*	*M4:I5*	*M4:I5 "Retrieve sales and usage reports*
M5: Users Service	*"Manages user profiles"*	*M5:I6*	*M5:I6 "Manages user profiles, retrieve order history, update account settings"*

Interfaces		
Interface Name	**Operations**	**Operation Descriptions**
M1:I1	*<return type> I1:Op1() used by M2, M3* *<return type> I1:Op2(int x) used by M3*	*M1:I1:Op1(): "places an order for a selected vehicle"* *M1:I1:Op2(int x): "Retrieves order details based on order ID"*
M1:I2	*<return type> M1:I2:Op3() used by M3* *M1:I2:Op3() used by M1*	*M1:I2:Op3() "Fetches past order history"* *"Internal order validation"*
M2:I3	*<return type> I3:Op1() used by M1, M3*	*"Obtain vehicle listings based on filters"*
M3:I4	*<return type> I4:Op1() used by M1, M2*	*"Validates user credentials using login"*
	<return type> I4:Op2() used by all front-end components	*"Manages session authentication and token generation"*
M4:I5	*<return type>* *I5:Op1(),* *I5:Op2()*	*"Generate Sales reports and provide application usage analytics"*
M5:I6	*<return type>* *I6:Op1(), I6:Op2(),* *I6:Op3()*	*"I6:Op1() - Retrieves user profile details, I6:Op2() - Updates user account settings, I6:Op3() - Retrieves order history for the user"*

7.3 Deployment Diagram

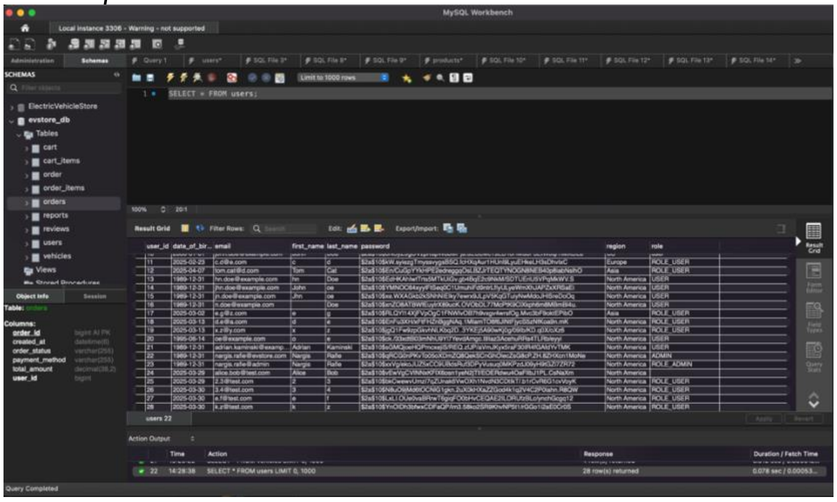
View the deployment diagram here.

Our deployment architecture is designed to harness the full benefits of a cloud-native environment:

- ***Backend Deployment:*** *The backend runs on AWS using EC2 instances or Elastic Beanstalk. This allows us to dynamically scale our services and perform easy updates.*
- ***Frontend Deployment:*** *The frontend is hosted on AWS S3, with CloudFront serving as the Content Delivery Network (CDN) to ensure fast and reliable access.*
- ***Database Deployment:*** *We host our MySQL/PostgreSQL database on AWS RDS, which provides replication and automated backups for high availability.*
- ***CI/CD Pipeline:*** *Using GitHub Actions or Jenkins, our CI/CD pipeline automates builds, tests, and deployments, ensuring smooth integration and rapid iteration.*
- ***Monitoring and Security:*** *AWS CloudWatch monitors performance and logs, while security is enforced through HTTPS, JWT-based authentication, and robust role-based access controls.*

7.4 Quality Attributes Supported by the Architecture

Our architecture is carefully designed to support several critical quality attributes:

- ***Security***
  - *Authentication & Authorization: Implemented via Spring Security with JWT or OAuth 2.0 to ensure secure user authentication.*
  - *Users can register and login securely using JWT tokens. Roles: "ROLE_USER" (default), "ROLE_ADMIN" (for admin privileges).*
  - *Tokens are stored in localStorage on the frontend.*
  - *JWT-based authentication is implemented with Spring Security.*
  - *Authorization controls access to protected routes via role-based checks.*
  - *Blacklisted tokens are rejected (logout supported).*
  - *Admin users redirected to /admindashboard, regular users to /vehiclefilter.*
  - 

```
SELECT * FROM users;
```

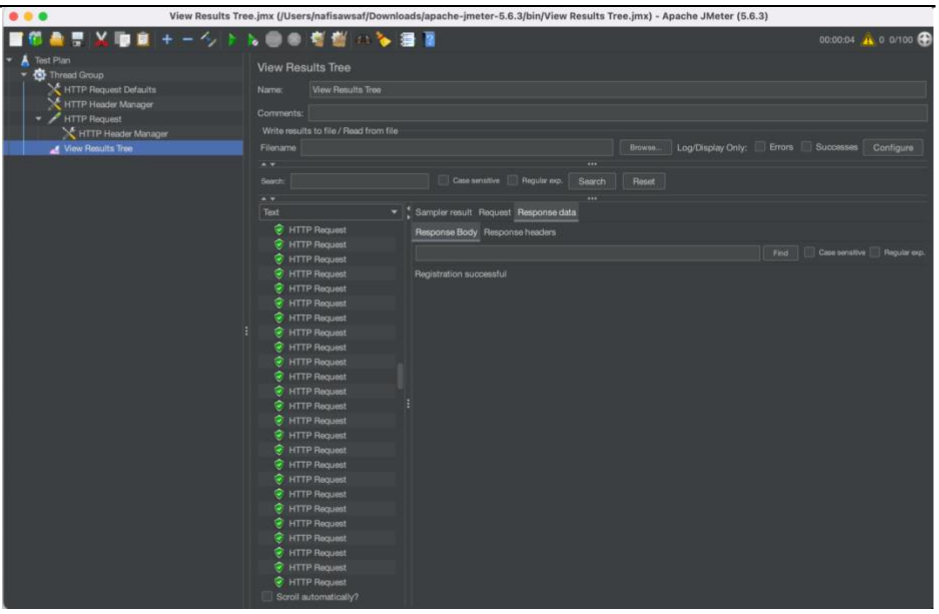
user_id	date_of_bir...	email	first_name	last_name	password	region	role
11	2025-02-23	c.d@e.com	c	d	[illegible]	Europe	ROLE_USER
12	2025-04-07	tom.cat@d.com	Tom	Cat	[illegible]	Asia	ROLE_USER
13	1989-12-31	hn.doe@example.com	hn	Doe	[illegible]	North America	USER
14	1989-12-31	jhn.doe@example.com	John	oe	[illegible]	North America	USER
15	1989-12-31	jn.doe@example.com	Jhn	oe	[illegible]	North America	USER
16	1989-12-31	n.doe@example.com		Doe	[illegible]	North America	USER
17	2025-03-02	e.g@z.com	e	g	[illegible]	Asia	ROLE_USER
18	2025-03-13	d.e@a.com	d	e	[illegible]	North America	ROLE_USER
19	2025-03-13	x.z@y.com	x	z	[illegible]	North America	ROLE_USER
20	1995-06-14	oe@example.com	o	e	[illegible]	North America	USER
21	1989-12-31	adrian.kaminski@examp...	Adrian	Kaminski	[illegible]	North America	USER
22	1989-12-31	nargis.rafie@evstore.com	Nargis	Rafie	[illegible]	North America	ADMIN
23	1989-12-31	nargis.rafie@admin	Nargis	Rafie	[illegible]	North America	ROLE_ADMIN
24	2025-03-29	alice.bob@test.com	Alice	Bob	[illegible]	North America	
25	2025-03-29	2.3@test.com	2	3	[illegible]	North America	ROLE_USER
26	2025-03-30	3.4@test.com	3	4	[illegible]	North America	ROLE_USER
27	2025-03-30	e.f@test.com	e	f	[illegible]	North America	ROLE_USER
28	2025-03-30	k.z@test.com	k	z	[illegible]	North America	ROLE_USER

```
     Time      Action                              Response            Duration / Fetch Time
22   14:28:38  SELECT * FROM users LIMIT 0, 1000   28 row(s) returned  0.078 sec / 0.00053...
```

When a new user is created, even the admin accesses the evstore_db database, the passwords are hashed as from the screenshot above and the user by default gets ROLE_USER access. No one can get ROLE_ADMIN unless our existing administrator decides.

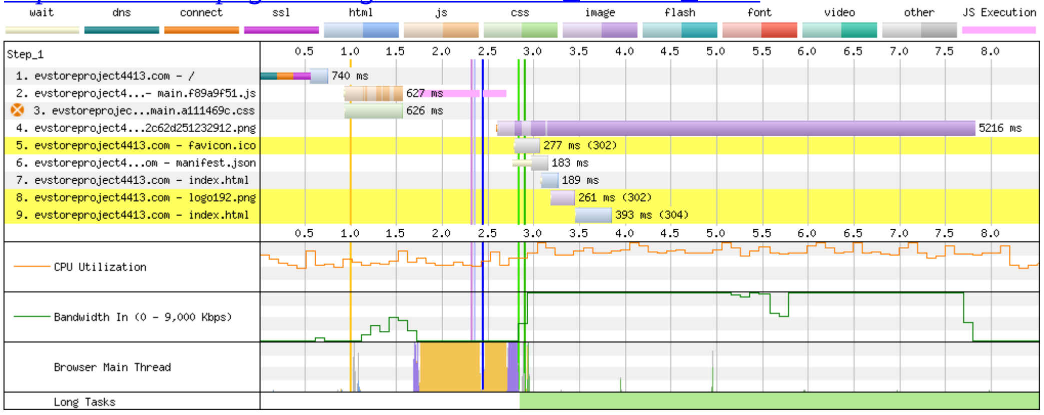
- *Data Encryption: All communications use HTTPS, and sensitive data (e.g., passwords) are encrypted using AES and TLS.*
- *All client-server communication is protected via HTTPS (TLS encryption).*
- *Passwords are never stored in plaintext — we use BCrypt hashing before saving them to the database.*
- *JWT tokens are signed using HMAC with a secure Base64-encoded secret key, preventing tampering.*
- *Sensitive JWT data like email and roles are encoded and verified on each request.*
- *AES is used for server-side cryptographic functions where needed (optional for extensions).*
- *Role-Based Access Control (RBAC): Endpoints are restricted based on user roles (such as admin or customer).*
- *Spring Security is used to restrict access at the endpoint level.*
- *Routes are guarded based on roles using .hasAuthority("ROLE_USER") or .hasAuthority("ROLE_ADMIN").*
- *Default role is "ROLE_USER" for all registered users.*
- *Admin users are assigned "ROLE_ADMIN" manually (e.g., via DB seed or admin registration).*
- *Backend filters (like JwtTokenFilter) extract and verify roles from JWT tokens.*
- *Unauthorized or role-mismatched requests return 403 Forbidden.*
- *SQL Injection & CSRF Protection: Input validation and Spring Security's built-in CSRF protection help safeguard against common attacks.*

- ***Performance***
  - *Caching: Frequently accessed data (like product listings and user sessions) is cached using Redis.*
  - *Redis (or in-memory cache) is used to store frequently accessed data:*
  - *Product listings*
  - *User session tokens*
  - *Vehicle configurations*
  - *Reduces database round-trips and improves API response time.*
  - *Load Balancing: Traffic is distributed via a load balancer (e.g., AWS ALB or Nginx) to ensure responsiveness.*
  - *The application is designed to scale horizontally using load balancing.*
  - *Prevents overload on any single server, ensuring reliability and high availability.*
  - *Database Optimization: Techniques such as indexing, query optimization, and connection pooling (using HikariCP) enhance performance.*
  - *Indexed columns such as userId, email, and orderStatus speed up query execution.*
  - *Optimized JPQL/Hibernate queries to avoid N+1 issues and reduce overhead.*
  - *Lazy/eager fetching used selectively to reduce unnecessary joins.*



Registration successful

-
- *JMeter results for 1-10, 1-100, 1-1000 users for our endpoints of (browsing the product catalogue page, reading reviews, posting reviews)*
- *https://www.webpagetest.org/video/compare.php?tests=250330_AiDcV5_6C9-r:1,250330_AiDcV5_6C9-r:2,250330_AiDcV5_6C9-r:3*
- *Web Page Test Results: Summary: https://www.webpagetest.org/result/250330_AiDcV5_6C9/*



-
- *Performance metrics of our website: https://evstoreproject4413.com*
- *Time to First Byte (TTFB): 0.781 seconds — This indicates the time it takes for the server to begin responding to a request. A TTFB below 1 second reflects an efficient backend and minimal server-side latency.*
- *Start Render: 2.500 seconds — This metric measures the point at which the browser begins to visually display content. It serves as a key indicator of the perceived loading speed from the user's perspective.*
- *First Contentful Paint (FCP): 2.533 seconds — This represents the time at which the browser renders the first piece of content from the DOM, such as text or an image. It marks the initial visual feedback that informs users that the page is loading.*
- *Speed Index: 4.622 seconds — This metric quantifies how quickly the contents of a page are visibly populated. While the result is slightly above the ideal benchmark of 4 seconds, it still indicates a reasonably fast visual load experience.*
- *Largest Contentful Paint (LCP): 2.533 seconds — LCP measures when the largest visible element (often a hero image or prominent text block)*

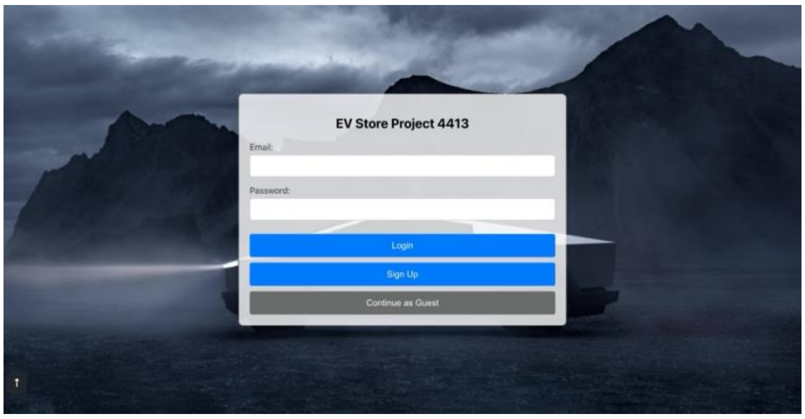
is fully rendered. This is a key performance metric for user experience and search engine optimization.

  - *Cumulative Layout Shift (CLS): 0.0 — A CLS of zero demonstrates excellent visual stability during page load, indicating that no unexpected shifts occurred in the layout, which contributes to a smoother user experience.*
  - *Total Blocking Time (TBT): 0.0 seconds — This reflects the amount of time the main thread was blocked by long tasks. A result of zero signifies that the application maintains excellent interactivity and responsiveness.*
  - *Page Weight: 5,098KB — The total page size is approximately 5MB, which is relatively large. The majority of this weight stems from image assets. Optimization strategies, such as compression and efficient image formats, are recommended to improve load times on slower networks.*
  - *Resource Utilization refers to the proportion of system resources (CPU, memory, network, etc.) actively being used during operation. Maintaining optimal utilization ensures the application is neither underused (wasting resources) nor overloaded (risking crashes or slowdowns).*
  - *CPU and Memory Utilization During load testing with JMeter and simulated concurrent users (ranging from 10 to 1000), we monitored CPU and memory metrics on the backend server using system tools (htop, top, and Java VisualVM):*
  - *CPU Utilization: Peaked at 65% under 1000 concurrent users. Average utilization stayed under 40% in most real-world scenarios. No CPU saturation was observed (i.e., CPU never reached 100%). Memory Utilization: JVM heap usage remained under 600MB with GC (Garbage Collection) keeping memory healthy.*

- *Scalability*
  - *Microservices Architecture: Allows independent scaling of services such as User, Order, and Inventory.*
  - *Horizontal Scaling: Multiple instances can be deployed using Kubernetes or AWS Auto Scaling Groups.*
  - *Cloud-Native Deployment: The system is hosted on AWS, GCP, or Azure to benefit from automatic resource scaling.*
- ***Maintainability***
  - *Modular Design: The system follows a clear Service Layer Architecture (Controller → Service → Repository), making it easy to update.*
  - *Automated Testing: Comprehensive unit tests (using JUnit and Mockito), integration tests, and API tests (via Postman) ensure reliability.*
  - *CI/CD Pipelines: Continuous integration and deployment are managed using GitHub Actions or Jenkins for streamlined updates.*
- ***Availability & Fault Tolerance***
  - *Database Replication: PostgreSQL/MySQL replication ensures high availability of data.*
  - *Circuit Breaker Pattern: Tools like Resilience4j or Hystrix are used to prevent cascading failures.*
  - *Auto Recovery: Kubernetes auto-restart and AWS RDS failover mechanisms help maintain uptime during failures.*
- ***Extensibility***

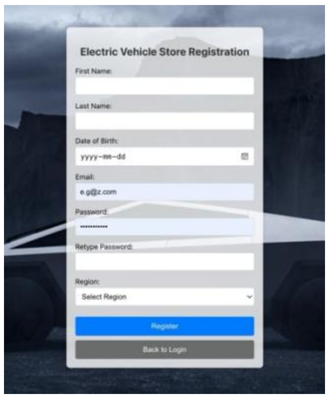
- *API-First Approach: RESTful APIs allow easy integration with third-party systems and future services.*
- *Event-Driven Features: Kafka or RabbitMQ enables asynchronous processing (e.g., for order notifications and logging).*
- *Microservices Support: The architecture allows additional services (such as a Recommendation Engine) to be added without impacting existing functionality.*

8 Web Design



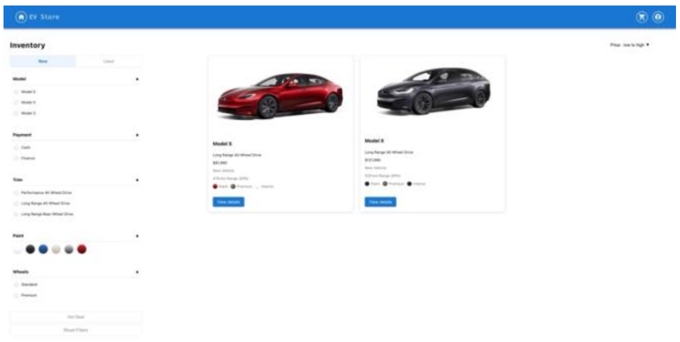
Homepage:

The screen features a sleek, modern design with a futuristic EV background, reinforcing the project's theme. Users can log in with their email and password, sign up for a new account, or browse the store as a guest. The intuitive layout ensures a smooth entry point into the application while maintaining a clean and professional aesthetic.



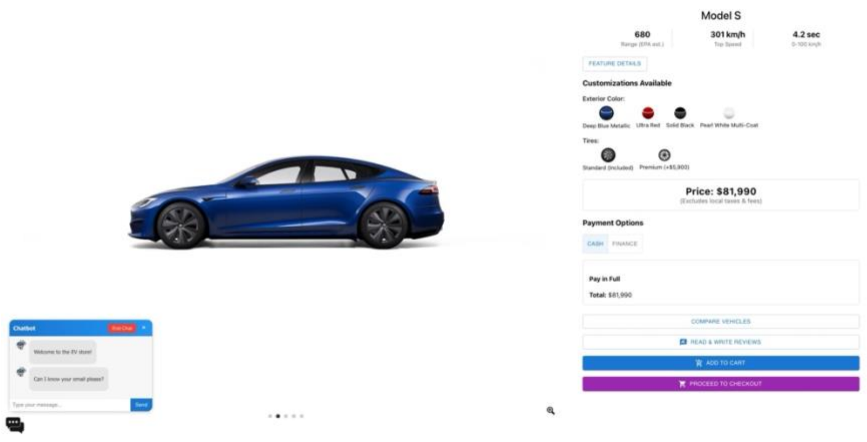
Registration Interface:

It allows new users to create an account by entering personal details including their first and last name, date of birth, email, password (with confirmation), and region. The form ensures user data is collected securely and includes a "Back to Login" button for easy navigation. With a modern, clean design and a sleek EV-themed background, this screen provides a user-friendly and professional onboarding experience for new customers.



Vehicle Catalog:

This inventory page offers users a seamless experience to explore and filter through electric vehicle listings. With powerful search filters and organized vehicle cards, users can easily compare models based on price, features, and style—all within a sleek, user-friendly interface.

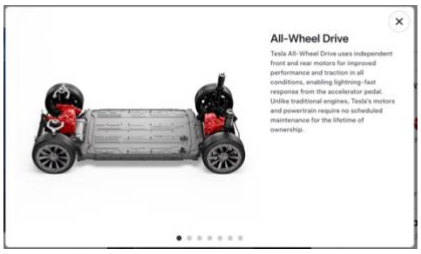


Vehicle Page:

This detailed product page showcases the vehicle with full specs, visual previews, customization options, and purchase tools. The user can tailor the car to their preferences, compare with other models, and easily proceed to checkout or seek help via the built-in chatbot—all within a visually clean and user-focused layout.

Zoom In Feature:

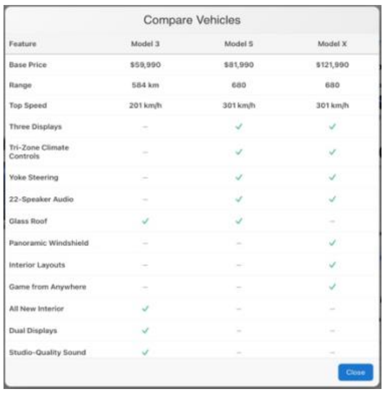
In the vehicles page, this feature helps to show the details in the car models. To access it, the users can use the plus icon at the bottom right side to zoom in and use the minus icon to zoom out.


All-Wheel Drive

Tesla All-Wheel Drive uses independent front and rear motors for improved performance and traction in all conditions, enabling lightning-fast response from the accelerator pedal. Unlike traditional engines, Tesla's motors and powertrain require no scheduled maintenance for the lifetime of ownership.

Features Details Button:

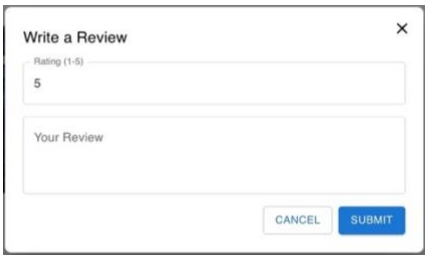
This button is located at right side of the vehicles page, it opens a pop-up showcasing different features of the selected car model – helping the customer to a detailed insight into the features offered by the product.


Compare Vehicles

Feature	Model 3	Model S	Model X
Base Price	$59,990	$81,990	$121,990
Range	584 km	680	680
Top Speed	201 km/h	301 km/h	301 km/h
Three Displays	–	✓	✓
Tri-Zone Climate Controls	–	✓	✓
Yoke Steering	–	✓	✓
22-Speaker Audio	–	✓	✓
Glass Roof	✓	✓	–
Panoramic Windshield	–	–	✓
Interior Layouts	–	–	✓
Game from Anywhere	–	–	✓
All New Interior	✓	–	–
Dual Displays	✓	–	–
Studio-Quality Sound	✓	–	–

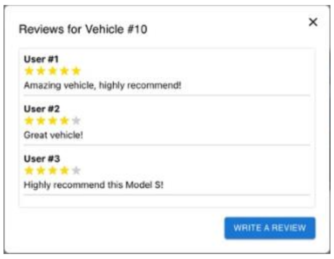
Compare Vehicles:

This **Vehicle Comparison Button** shows a pop-up that demonstrates the key differences between Model 3, Model S, and Model X. It compares price, range, top speed, and premium features like displays, climate control, yoke steering, and sound systems—helping users quickly spot what each model offers.


Write a Review

Rating (1-5): 5

Your Review


Reviews for Vehicle #10

User #1
★★★★★
Amazing vehicle, highly recommend!

User #2
★★★★☆
Great vehicle!

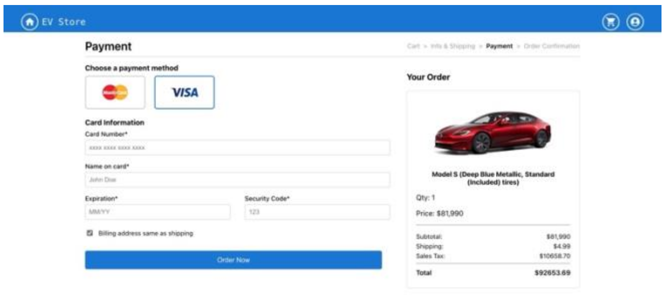
User #3
★★★★☆
Highly recommend this Model S!

Review Submission Popup:

This review submission popup enables users to rate EV products and share their experiences or to read the reviews. With a simple interface, it ensures feedback can be quickly added, helping future customers make informed decisions.

Payment Page:

This secure and cleanly designed checkout page allows customers to complete their purchase with a credit card. It presents all necessary order details and payment fields clearly, ensuring a smooth and transparent buying experience.



Payment

Choose a payment method

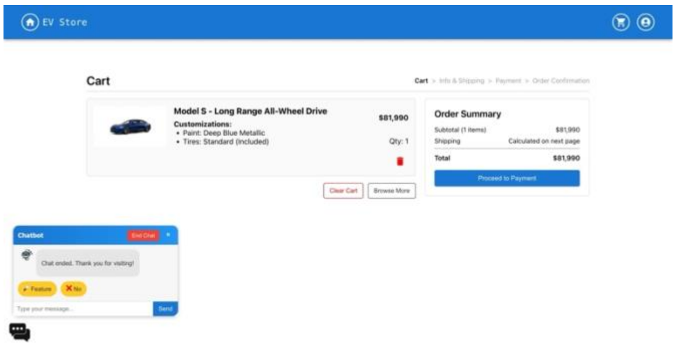
Card Information
Card Number*: xxxx xxxx xxxx xxxx
Name on card*: John Doe
Expiration*: MM/YY
Security Code*: 123
☑ Billing address same as shipping

Your Order
Model S (Deep Blue Metallic, Standard (Included) tires)
Qty: 1
Price: $81,990

Subtotal:	$81,990
Shipping:	$4.99
Sales Tax:	$10658.70
Total	$92653.69

Cart Page:

This populated cart view provides users with a clear summary of their selected EV and its configurations. It includes easy options to remove items, continue shopping, or move forward to payment, enhancing both flexibility and ease of navigation during checkout.



Cart

Model S - Long Range All-Wheel Drive — $81,990
Customizations:
- Paint: Deep Blue Metallic
- Tires: Standard (Included)

Qty: 1

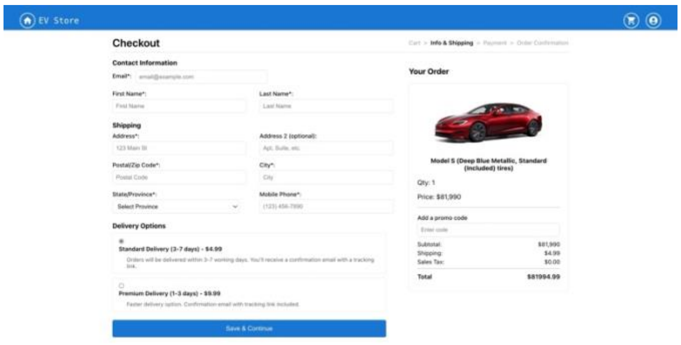
Order Summary

Subtotal (1 items)	$81,990
Shipping	Calculated on next page
Total	$81,990

Chat ended. Thank you for visiting!

Checkout Page:

This step of the checkout process collects essential shipping and contact information from the customer, along with their preferred delivery method. A clean summary of the order is shown on the right, helping users stay informed before moving on to the payment step.



Checkout

Contact Information
Email*: email@example.com
First Name*: First Name
Last Name*: Last Name

Shipping
Address*: 123 Main St
Address 2 (optional): Apt, Suite, etc.
Postal/Zip Code*: Postal Code
City*: City
State/Province*: Select Province
Mobile Phone*: (123) 456-7890

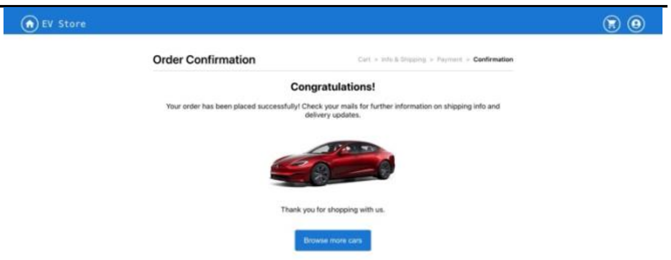
Delivery Options
● Standard Delivery (3-7 days) - $4.99
Orders will be delivered within 3-7 working days. You'll receive a confirmation email with a tracking link.
○ Premium Delivery (1-3 days) - $9.99
Faster delivery option. Confirmation email with tracking link included.

Your Order
Model S (Deep Blue Metallic, Standard (Included) tires)
Qty: 1
Price: $81,990
Add a promo code: Enter code

Subtotal:	$81,990
Shipping:	$4.99
Sales Tax:	$0.00
Total	$81994.99

Order Confirmation Page:

This clean and minimal confirmation page provides closure to the purchase journey, assuring the customer that their order has been received. It offers next steps through email and gives users the opportunity to continue browsing.



Order Confirmation

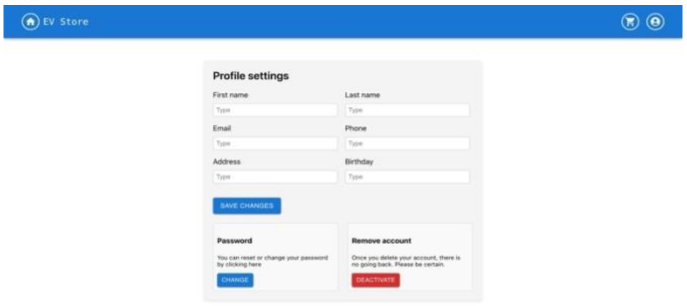
Congratulations!

Your order has been placed successfully! Check your mails for further information on shipping info and delivery updates.

Thank you for shopping with us.

Profile Settings Page:

This user profile page gives customers full control over their personal details, allowing them to update info, change their password, or permanently delete their account. It's designed to be simple, secure, and user-friendly.



Profile settings

First name: Type
Last name: Type
Email: Type
Phone: Type
Address: Type
Birthday: Type

Password

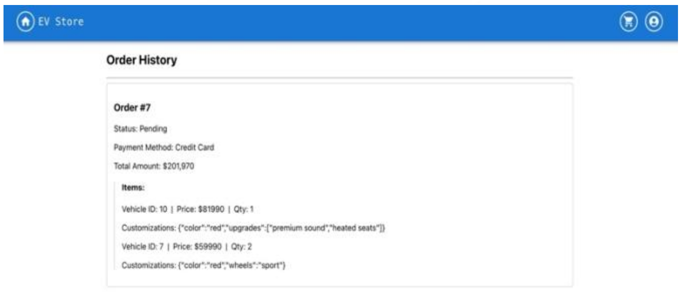
You can reset or change your password by clicking here

Remove account

Once you delete your account, there is no going back. Please be certain.

Order History Page:

This page presents a clear breakdown of the user's past orders, including item-level details like vehicle ID, price, quantity, and applied customizations. It helps users track their order status and review their purchase history, offering transparency and convenience.



Order History

Order #7

Status: Pending

Payment Method: Credit Card

Total Amount: $201,970

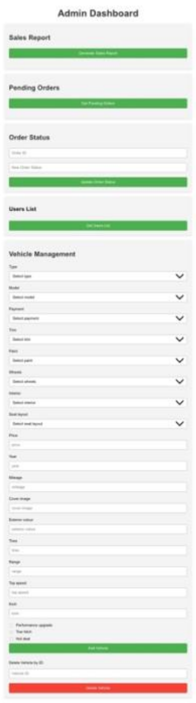
Items:

Vehicle ID: 10 | Price: $81990 | Qty: 1

Customizations: {"color":"red","upgrades":["premium sound","heated seats"]}

Vehicle ID: 7 | Price: $59990 | Qty: 2

Customizations: {"color":"red","wheels":"sport"}



Admin Dashboard:

This is an **Admin Dashboard** designed to manage and monitor our ecommerce site. It gives admins full control over key operations like:

- **Sales Reports:** Generate up-to-date sales data at the click of a button.
- **Pending Orders:** View orders that need attention.
- **Order Status Management:** Update the status of any order using its ID.
- **Users List:** Fetch and review all registered users.
- **Vehicle Management:** Add or delete vehicles with detailed specs like type, model, payment method, trim, paint, wheels, interior, seat layout, price, year, mileage, color, tires, range, speed, and extra features like performance upgrade, tow hitch, or hot deals.

It's basically our all-in-one command center for running things smoothly behind the scenes. Clean layout, straightforward inputs, and total control—just what every admin dreams of.

9 Activities Plan

9.1 Project Backlog and Sprint Backlog

Project Backlog:

- ***User Authentication & Account Management***: *1.1 User Registration (Sign-up) 1.2 User Login (Sign-in) 1.3 User Logout (Sign-out) 1.4 Password Encryption & Secure Storage 1.5 Implement JWT Token-based Authentication 1.6 Validate User Credentials Against Database 1.7 Implement Account Lock After Failed Attempts*
- ***Vehicle Listings & Browsing:*** *2.1 Display Electric Vehicles with Filters 2.2 Implement Sorting Options (Price, Brand, Year) 2.3 Implement Filtering (Self-Driving Feature, Price Range, etc.) 2.4 Paginated Listings for Efficient Browsing 2.5 Retrieve Vehicle Details from Database 2.6 Implement Vehicle Detail Page (Full Specifications, Price, Images)*
- ***Shopping Cart Management:*** *3.1 Add Vehicle to Cart 3.2 Remove Vehicle from Cart 3.3 Update Cart (Change Quantity, Remove Items) 3.4 Store Cart Data Persistently 3.5 Display Cart Summary with Total Price 3.6 Validate Vehicle Availability Before Adding to Cart*
- ***Secure Checkout & Payment Processing:*** *4.1 Implement Checkout Flow (Step-by-Step Navigation) 4.2 Collect and Validate Payment Information 4.3 Integrate Secure Payment Gateway 4.4 Implement Order Confirmation Process 4.5 Store Transaction History in Database 4.6 Send Order Confirmation Email 4.7 Implement Refund & Cancellation Process*
- ***Customer Reviews & Ratings:*** *5.1 Enable Users to Submit Reviews & Ratings 5.2 Moderate and Approve Reviews 5.3 Display Reviews on Vehicle Detail Pages 5.4 Allow Users to Upload Images with Reviews 5.5 Implement Review Filtering & Sorting.*
- ***Admin & Reporting Dashboard:****6.1 Implement Admin Login & Role-Based Access 6.2 Generate Sales Reports 6.3 Display User Activity Analytics 6.4 Manage Vehicle Listings (Add, Update, Delete) 6.5 Process Refunds & Order Cancellations 6.6 Manage User Accounts (Ban, Update, Delete).*
- ***Additional Features:*** *7.1 Implement 360-Degree View & Zoom Feature for Vehicles 7.2 Enable Vehicle Comparison Feature 7.3 Implement Chatbot for Customer Support 7.4 Integrate Loan Calculator for Financing Options 7.5 Implement Guest Checkout 7.6 Provide Personalized Vehicle Recommendations.*
- ***Cloud Infrastructure & Deployment:*** *8.1 Deploy Backend on AWS Elastic Beanstalk 8.2 Deploy Frontend on AWS S3 & CloudFront 8.3 Set Up RDS Database (PostgreSQL) 8.4 Implement API Gateway for Request Management 8.5 Configure Load Balancer & Auto-scaling 8.6 Enable HTTPS & Secure Communication.*
- ***Testing & Quality Assurance:*** *9.1 Unit Testing for Backend Services 9.2 Integration Testing for API & Database 9.3 Frontend UI Testing 9.4 Security Testing (Authentication, SQL Injection, etc.) 9.5 Performance & Load Testing 9.6 End-to-End Testing of Purchase Flow*

Sprint Backlog: GANTT Chart

Please visit the following link to view the GANTT Chart: ***E-commerce site - EECS 4413.xlsx***

9.2 Group Meeting Logs

Date & Time	*Attendance*	*Topics discussed and Tasks done*
	DELIVERABLE 1	
Saturday, January 18th – Zoom Sessions *Time: 1:00 pm - 1:45 pm* *Session 1 (25 minutes)* *Session 2 (15 minutes)*	*Session 1 - Nafis, Nargis, Vashavi* *Session 2 - Nafis, Adrian*	• *Reviewed the overall project requirements.* • *Discussed which frameworks and technologies the team is comfortable with for the backend, frontend, and database.* • *Continued the discussion on project requirements.* • *Further clarified technical preferences and confirmed initial roles for each team member.*
Saturday, January 25th – WhatsApp	*All team members present*	• *Tasks were assigned through the WhatsApp group.* • *Focused on developing the use case diagrams and sequence diagrams.* • *Coordinated responsibilities for Deliverable 1 using the project backlog as a reference.*
Saturday, February 1st – Zoom *Time: 1:04 PM (20 minutes)*	*All team members present*	• *Established the GitHub repository for the project.* • *Completed the sequence diagrams and committed initial work to the repository.* • *Discussed details of the component diagram and Gantt chart.* • *Began writing up the initial sections of the project report.*
Sunday, February 9th – Zoom *Time: 12:57 PM (22 minutes)*	*All team members present*	• *Reviewed and refined the project architecture.* • *Finalized the UML component diagram and package diagram.* • *Collaborated on various report sections to ensure consistency and clarity.*
Wednesday, February 12th – Zoom	*All team members present*	*Part 1 – Finalizing Deliverable 1 Report:* • *Main Page: Ensure all team members' names are added after completing their assigned sections.* • *Introduction: Completed by Nafis.* • *Major Design Decisions: Completed by Adrian.* • *Use Case Diagrams: Finalized by Nafis.*

Time: 10:00 AM (33 minutes)		• *Sequence Diagrams: Completed by Vashavi, Nargis, and Adrian.* • *Component & Package Diagrams + Architecture: Completed by Nafis.* • *Product Backlog: Completed by Adrian.* ○ *Gantt Chart: Finalized by Nargis.* • *TDD (Test Cases): Three test cases (sign-in, sign-out, registration) completed by Vashavi.* *Additional Instructions:* • *Part 2: All team members must review every section of the report to verify completeness.* • *Part 3: Format the report for final submission.* • *Part 4: After finalizing the frontend, backend, and database, create the login page to integrate these components. (Note: The login page is already completed.)* • *Discuss scheduling the next meeting before submitting Deliverable 1.*
Friday, February 14th – Zoom *Time: 7:00 PM (17 minutes)*	*All team members present*	• *Final Review: Finalize all sections of the report collaboratively.* • *Project Creation (Part 4): Confirm that the login page is fully operational with the server up and running.* • *Next Meeting: Tentatively scheduled for Monday, February 17th at 7:00 PM to finish Part 4 and ensure the website is fully integrated and operational.*
DELIVERABLE 2		
Session 1: Monday February 17th, *Time: 7:00 PM (21 minutes)* *Session 2: Tuesday February 18th, 12:00PM* *Time: 12:00 PM (12 minutes)*	*Session 1: Nafis, Nargis, Adrian* *Tasks assigned - Through chat Vashavi* *Session 2 - Nafis, Vashavi*	*Modules and tasks assigned to each member:* • *Module1 : Vehicle Catalogue* ○ *Vehicle Catalogue - UI Page (Front-End)* ○ *Need to decide what items to include?* ○ *Self-Driving cars filter implementation.* *(Nargis)* —--- • *Module 2: Ordering Service* ○ *Order Now - UI Page (Front-End)* ○ *Order class - (Back-End)* ○ *Zoom in Feature implementation.* *(Vashavi)* —--- • *Module 5: Users* ○ *Log in - UI Page - (Front-End)* ○ *Registration - UI Page - (Front-End)* ○ *Users class - (Back-End)* *(Nafis)* • *Database* ○ *Build model* ○ *Store model in cloud* ○ *Create website (evstore.com) (domain - change)* *(Adrian)*

		Deadline to complete modules: Sunday Feb 23rd. —--- *After completing the modules things to-do:* *(All team members)* • *Connect frontend pages - Log-in page, vehicle catalogue and order now page together* • *Connect Module1, Module2, Module 5 and Database together.* • *Discussion on when to tie Spring Boot, React JS, Java and cloud databases together?*
Sunday, February 23rd – Zoom *Time: 1:00 PM (30 minutes)*	*All team members present*	*Updates:* • *Order now Page (completed) - Vashavi* • *Log in, Registration Page (completed) - Nafis* • *Vehicle Catalogue page (in progress) - Nargis* • *Database (in progress) - Adrian* *TODO:* • *Review GitHub repository updates.* • *Domain configuration* • *Create as many front-end React pages as possible.* • *Finalizing the connection of backend and frontend.* • *Client Side/ Server Side with Design*

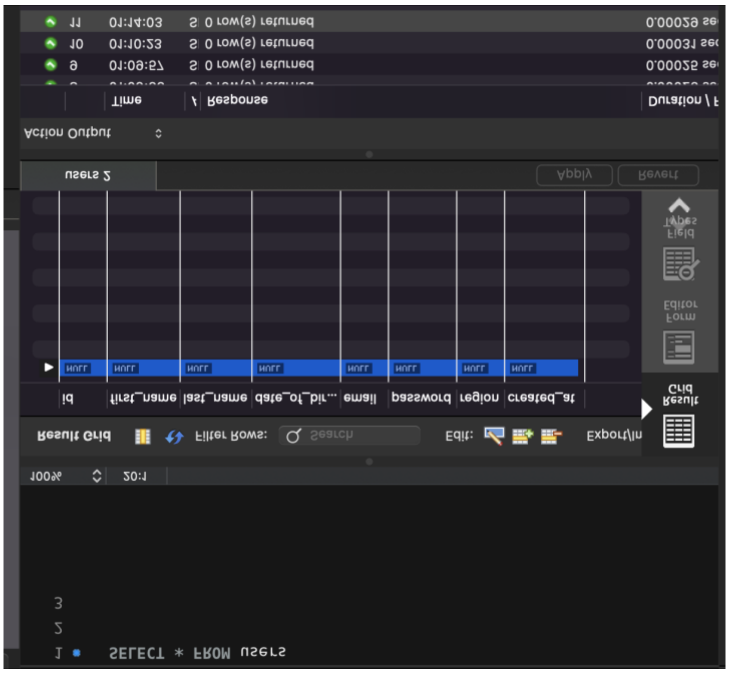
Monday February 23rd – Zoom *Time: 7:30 PM (120 minutes)*	*Nafis, Adrian*	*DATABASE and DOMAIN set-up*

Set up domain http://evstoreproject4413.com (completed) - Adrian
Password123

Adrian:
Deployed the starter project WAR file, connected MySQL, and created the database schema.

Nafis:

- *Fix issues with MySQL and spring boot application not running*
- *Database fixed (MySQL) and Spring Boot Application runs successfully*
- *User is not being added to the MySQL database. Shows null values in:*

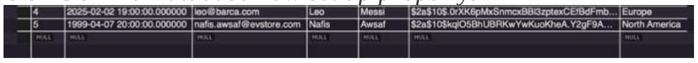


```
1 ●   SELECT * FROM users
2
3
```

id	first_name	last_name	date_of_bir...	email	password	region	created_at
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
      Time       Response                  Duration / F
 9    01:09:57   5  0 row(s) returned      0.00025 se
10    01:10:23   5  0 row(s) returned      0.00031 sec
11    01:14:03   5  0 row(s) returned      0.00029 se
```

OUTCOME: Database now set up properly.



4	2025-02-02 19:00:00.000000	leo@barca.com	Leo	Messi	$2a$10$.0rXK6pMxSnmcxBBl3zptexCEfBdFmb...	Europe
5	1999-04-07 20:00:00.000000	nafis.awsaf@evstore.com	Nafis	Awsaf	$2a$10$kqlO5BhUBRKwYwKuoKheA.Y2gF9A...	North America
NULL	NULL	NULL	NULL	NULL	NULL	NULL

- *Login and registration functionality are working as verified through MySQL Workbench.*
- *After logging in, a new dashboard appears (completed by Nafis)*

Saturday March 2nd *Time: 1:00 PM*	*Discussion Notes prepared by Nafis*	*Server Side Implementation TODO:* *Backend (Spring Boot - Java)* *1. Model Layer (Entity Classes - JPA)* *These represent database tables. Each entity is mapped to a table in MySQL.* *User.java – Represents users (customers, admins).* *Product.java – Represents products available for sale.* *Order.java – Represents customer orders.* *CartItem.java – Represents items in the shopping cart.* *Payment.java – Represents payment transactions.* *2. Repository Layer (DAO - Spring Data JPA)* *Handles database queries.* *UserRepository.java* *ProductRepository.java* *OrderRepository.java* *CartItemRepository.java* *PaymentRepository.java* *3. Service Layer (Business Logic)* *Implements business rules and interacts with repositories.* *UserService.java* *ProductService.java* *OrderService.java* *CartService.java* *PaymentService.java* *4. Controller Layer (REST API - Spring MVC)* *Handles HTTP requests.* *UserController.java* *ProductController.java* *OrderController.java* *CartController.java* *PaymentController.java* *5. Security Layer (Spring Security)* *Handles authentication & authorization.* • *SecurityConfig.java – Configures JWT authentication.* • *UserDetailsServiceImpl.java – Custom user authentication.* • *JwtUtil.java – Handles JWT token generation & validation.* *6. Exception Handling & Logging* *Handles errors properly.* • *GlobalExceptionHandler.java – Catches and formats exceptions.* • *LoggingAspect.java – Logs method calls and system events.* *Frontend (React.js)* *You need components, pages, and services to connect with the backend.* *1. Pages (React Components)* • *LoginPage.jsx / LoginComponent.ts* • *RegisterPage.jsx / RegisterComponent.ts* • *HomePage.jsx / HomeComponent.ts*

- ProductPage.jsx / ProductComponent.ts
- CartPage.jsx / CartComponent.ts
- CheckoutPage.jsx / CheckoutComponent.ts
- OrderHistory.jsx / OrderHistoryComponent.ts
- AdminDashboard.jsx / AdminDashboardComponent.ts

2. Services (API Communication)

These handle API calls to the backend.

- authService.js – Handles authentication.
- productService.js – Fetches product data.
- cartService.js – Manages cart actions.
- orderService.js – Handles order processing.

3. State Management (Redux / Context API)

- store.js – Manages application state.

4. UI Components

- Navbar.jsx
- Footer.jsx
- ProductCard.jsx
- CartItem.jsx

Test Coverage for Server-Side

For backend testing in Spring Boot, you should write unit tests & integration tests using JUnit & Mockito.

1. Unit Test Files (JUnit + Mockito)

- UserServiceTest.java
- ProductServiceTest.java
- OrderServiceTest.java
- CartServiceTest.java
- PaymentServiceTest.java

Each should contain at least 4 test cases per team member.

2. Integration Test Files (Spring Boot Test + TestContainers)

- UserControllerTest.java
- ProductControllerTest.java
- OrderControllerTest.java

3. Postman Test Collection

- A Postman Collection for testing API endpoints (GET /products, POST /login, POST /checkout, etc.)

Sunday, March 9th - Zoom *Time: 1:00 PM (42 minutes)*	*All team members present*	*Backend Functionality:* • *Finalized GitHub commits; discussed write access issues.* • *Reviewed new sections for Deliverable 2 and confirmed design patterns (now marked as DONE).* • *Discussed frontend-backend connectivity and cloud setups.* *Chatbot Implementation:* • *Reviewed technologies planned for the chatbot.* *Test Coverage:* • *Planned for 32 test cases (8 per team member) covering login, registration, and other functionalities.* *Assignments:* • *Vehicle Catalogue Page – Nargis.* • *Checkout page confirmation and complete additional TDD – Vashavi.* • *Cloud domain integration with login/registration – Adrian.* • *Backend core functionality – Nafis.* • *All team members connect endpoints.*
Friday, March 14th *Time: 10:00pm (30 minutes)*	*All team members present*	*Issues Reported:* • *Checkout failures.* • *Database not updating properly.* • *Data not being passed correctly among entities.* • *Frontend-backend connection issues with parameters.* • *Some endpoints not being hit in Postman.* *Assignments:* • *Vashavi & Nargis: Prepare Report, Chatbot documentation, and update the Deployment Diagram.* • *Nafis: Investigate and fix checkout implementation.* • *Adrian: Finalize cloud infrastructure set-up.* *Completion Sprints:* • *Troubleshooting endpoints and connecting services (Nafis and Adrian).* • *Vehicle Catalogue Page integration – Nargis.* • *Order Now and Checkout pages – Vashavi.* • *Documenting Server-Side Test Cases – Vashavi.* *Troubleshooting Meetings: Conducted additional breakout sessions on Zoom for frontend and backend integration issues.* • *Nafis, Nargis - Frontend connection (VehicleFilter.js)*

		• *Nafis, Vashavi - Frontend (OrderNowPage)* • *Nafis, Adrian - Backend*

Sunday March 16th *Time: 10:00 PM (25 minutes)*	*All team members present*	*Current Progress:* • *25 test cases have passed via Postman.* *Remaining Tasks:* • *Finalize test cases for Admin, Reviews, Chatbot, Zoomed In Feature, Payment method validation, Loan Calculator API, and Order Confirmation Notification.* • *Plan for complete TID33 User Flow and Admin Flow Testing.* *TODOs:* • *Pass all 38 test cases via Postman.* • *Expect necessary database changes.* • *Update and finalize the project report.* • *Grant TA access to the GitHub repository.* • *Create a basic admin page to showcase a sales report from the Orders database.* • *Document and implement the Chatbot service.* • *Connect the updated functional backend with the updated frontend.* *Final Overview of the Project Discussion* • *Complete Backend:* *Finalize and stabilize all backend functionalities to ensure smooth operations.* • *Connecting Backend, Database, and Frontend:* *Seamlessly integrate the backend, database, and frontend for real-time updates and data consistency.* • *Cloud Deployment:* *Configure and deploy the application using cloud-native services to leverage scalability and high availability.* • *Getting Ready for Demo:* *Prepare the complete system for a live demo, ensuring all integrations are functioning as expected.* *Potential Troubleshooting Issues (Integration of Final Frontend, Backend, and Live Database Updates)* • *Foreign Keys:* *Verify that associations (e.g., UserID linked to VehicleID, Orders, and Review Service) are correctly implemented.* • *Token Passing:* *Ensure that authentication tokens are properly passed through all frontend pages.* • *CORS Issues:* *Address any cross-origin resource sharing (CORS) problems that might arise during integration.*

- *Security Configuration:*
  Confirm that the security.config file is correctly set up to handle all authentication and authorization rules.
- *Dependencies:*
  Review and update dependencies in the POM.xml file to prevent version conflicts and ensure smooth builds.
- *Concurrency and Parameter Matching:*
  Avoid concurrency issues by ensuring parameter types match and that token generation (and placement in application.properties) is synchronized with any token expirations.
- *Role-Based Access Control:*
  Maintain a clear distinction between access rights for ROLE_USER and ROLE_ADMIN across all endpoints.
- *Endpoint Health:*
  Check that all API endpoints return a 200 status code during integration testing.

Questions to Ask the Professor

- *Clarification on the GitHub invitation (link and access permissions).*
- *Guidance on handling Postman-related queries.*
- *Recommendations for testing in a cloud-native environment versus a local host.*

Security Risks to be Implemented (In-Class)

- *SQL Injection:*
  - *Develop and implement a test case focusing on SQL injection attacks, particularly targeting text input fields (e.g., logging in as an admin) to ensure that sensitive data remains secure and that the system is immune to such attacks.*
- *Cross-Site Scripting (XSS):*
  - *Focus on vulnerable areas such as URLs and category fields.*
  - *Implement escaping techniques as discussed in class (see Slides 24 and 25) and determine if any additional security vulnerabilities need testing.*

Performance and Load Testing Considerations

- *Web Performance Testing:*
  - *Utilize www.webpagetest.org to capture performance screenshots of your URL. (Implementation is not required; only documentation of the results is needed.)*
- *Load Testing:*
  - *Use JMeter (or develop custom client scripts) to simulate up to 500 concurrent users interacting with the REST API, ensuring that the system can handle high loads efficiently.*

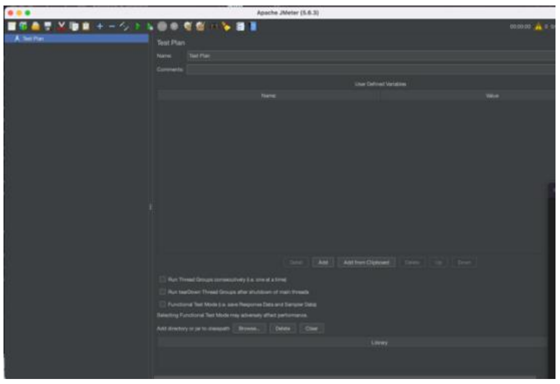
Wednesday March 19th *Time: 3:00 PM (30 minutes)*	*Nafis, Vashavi, Nargis*	*Progress* • *Adding Review functionality successful.* • *Improvements to the JWT token so that userID will not have to be explicitly put in JSON bodies.* • *Modified most backend classes for ensuring a smoother connection with the frontend.* *TODO:* • *Report (Update Package Diagram)* • *Update Design Patterns* • *Finish TDD of Admin, Chatbot and Orders, Guest*
Thursday March 20th *Time: 10:00 PM (35 minutes)*	*Nargis, Nafis, Vashavi*	*Updates:* • *Backend Completed (40 endpoints pass)* • *TDD updated the endpoints* *In Progress:* • *Connecting Backend with Frontend* • *Package Diagram for Report* *To Do:* • *Report* • *Add the meeting logs properly* • *Need to Add the TDD* • *Package Diagram* • *Invite the TA with the new branch named D2* • *Upload the database (as sql file)* *Things to do for Demo:* • *Continue with chatbot implementation - Vashavi* • *Frontend side: - Nafis* o *Add review button to Car models page* • *Work on admin analytics page - Nargis* • *Cloud deployment - Adrian*
Thursday March 20th *Time: 10:40 PM (15 minutes)*	*Nafis , Adrian*	*Cloud Deployment discussion.*
March 10th - March 21st	*Adrian*	*Assignments and Progress: (CLOUD)* • *Cross Site Scripting: Will use input sanitization to defend against XSS attacks.* • *Deploying DNS records caused a validation issue. Issue was solved by authentication of certificate user as well as root authority. Furthermore, DNS zoning needed to be reconfigured for better performance.* • *Cloud Database deployment became invalid due to schema error. Error was due to missing syntax on db file export from local machine and needed refactoring.* • *XSS will be mitigated by using User input sanitation by regex as well as by using the http header "Content-Security-Policy"(CSP) to prevent inline scrips and controls where scrips can load.* • *Troubleshooting backend and connection issues with Nafis.*

		SQL Injection: To mitigate SQL injections, we will implement sanitation as well as deploy a AWS web application firewall (WAF) to reduce the attack surface. In addition, we will eliminate using any user inputs as SQL queries but rather pass the appropriate data into prefabricated SQL queries.
Friday March 21st *Time: 9:00pm (20 minutes)*	*All team members present*	*Final report and code discussion for Deliverable 2* *Next meeting: Sunday – 1pm*
DELIVERABLE 3		
Sunday March 23rd *Time: 1pm (30 minutes)*	*All team member present*	*CHECKOUT IS NOW SUCCESSFUL !!! ☺* *Deliverable 3 of the team project* *REMAINING TODO for DEMO:* *1-The system is cloud-native (4%)* *5- The system is secure (https) (3%)* *13-It is possible to use the chatbot (3%)* *Total = 10%* *Connection Parts FrontEnd:* • *Admin Page, Admin Analytics* • *Admin adding a new vehicle (url image?)* • *Review button that opens Reviews* • *Logout buttons and navigation* • *Dynamic color changes amongst multiple pages through customizations* • *Updated taxed price put in or base price?* • *Model X Page and Model 3 Page needs to be updated via token passes for real time rendering.* • *User's order history page via my profile button if possible.* • *Make the Registration form cleaner.* • *Compare Vehicle (new page showing vehicle comparisons??)* *Admin Page will have the following:* • *User List Button* • *Adding a Vehicle Button* • *Generate Sales Report Button* • *Generate Application Usage* • *Admin Page also needs to have confirm order (based on orders placed that are pending from the user)* • *Continue as Guest Works* *TODO:* *-Focus on account locked if failed attempts > 5* *-Continue as Guest (but need to sign in for checkout)* *-Error checks and input validations* *Frontend:* *-Admin Confirms pending orders* *-Admin gets sales report (done)* *-Admin analytics (application usage)*

		-Admin gets list of all users (done) *-Compare Vehicles* *-Registration Form cleaner* *- Header (Add better UI) - consistency on all pages (done)* *Frontend tasks assigned to Vashavi:* *HIGH:* *-Dynamic color changes amongst multiple pages through customizations (Done)* *-Separation of writing reviews and reading reviews (Done)* *-Checkout scrollable (LOW) (Done)* *-Chatbot bottom (Done)* *-zoom (Done)* *- Compare vehicles (Done)* *-Chatbot implementation (Done)* *LOW:* *-User's order history page via my profile button (Done)* *First V1 Cloud Deployment* *Implement Security Attacks and test on V1* *Assignment of Tasks for Deliverable #3 based on Report # 3 (Check pdf)* *a) All Team Members- 10%* *b) Nargis , Vashavi 10%* *c) Nafis 5%* *d) Nargis, Vashavi 25%* *e) Nargis, Vashavi 10%* *f) All Team Members (live demo) 35%* *g) Nafis, Adrian 5%*
Sunday March 23rd *Time: 1:30pm (2 hours)*	*Nafis, Adrian*	*Deployment Phase 1*
Monday March 25th *Time: 11:30am (2 hours 30 minutes)*	*Nafis, Adrian*	*TODO:* • *Cloud V1 Deployment* • *Latest Frontend updates* • *Admin updates* • *Latest Backend updates* • *Latest endpoints* • *Cloud V2 Deployment* *Backend equivalency: http://localhost:8080/api/login* *https:api.evstoreproject4413* *Frontend equivalency: https://api.evstoreproject4413.com/api/login* *http://evstore4413-env.eba-ahawghzm.us-east-2.elasticbeanstalk.com/api/vehicles*
Thursday March 27th *Time: 9:30pm (30 minutes)*	*All team members present*	*Admin Updates* *Chatbot Updates* *Order History Updates* *Connect all endpoints for V2* *Upgrade connections to HTTPS* *Deploy V2*

		Next meeting Sunday 1pm (demo preparation)
Sunday March 30th *Time: 1pm*	*All team members present*	*How will we present the demo:* *Feature to test/check during the demo:* *1) The system is cloud-native (4%) - Adrian* *2) It is possible to register (2%) - Nafis* *3) It is possible to sign in (2%) - Nafis* *4) It is possible to sign out (2%) - Nafis* *5) The system is secure (e.g., the system is using https in its url) (3%) - Adrian* *6) It is possible to filter items (2%) - Nargis* *7) It is possible to sort items (2%) - Nargis* *8) It is possible to add items in the shopping cart (2%) - Nafis* *9) It is possible to remove items from the shopping cart (2%) - Nafis* *10) It is possible to check out items (2%) – Vashavi* *11) It is possible to write reviews on items and rate items using five stars (2%) – Vashavi* *12) It is possible to use the distinguished feature (2%) - Vashavi* *13) It is possible to use the chatbot (3%) - Vashavi* *14) It is possible to use the loan calculator (3%) - Vashavi* *15) It is possible to view hot deals (2%) – Nargis* *** ADMIN ** - confirming the order - Nargis* ***ADMIN ** functionalities – Nargis* *System Performance - Adrian* *SYSTEM IS READY FOR THE DEMO* 😊
Monday March 31st	*All team members present for the DEMO*	• *Successfully demonstrated the project to the professor.* • *Initially encountered deployment issues when migrating to the cloud.* • *Faced challenges with the admin login functionality.* • *Troubleshot and resolved both the cloud deployment and admin login issues.* • *Completed a comprehensive demo, covering all critical functionalities.* *After the demo discussed about starting the report via WhatsApp group chat – Nargis, Vashavi* *Nafis, Adrian – Will fix minor changes required in the cloud and backend.*
Tuesday April 1st *Time: 8:30pm (30 minutes)*	*Nargis, Vashavi*	*Discussed about Report Completion and Work Division:* • *Update Document Change Control* – (All team members) • *Update Sign-Off Dates* – (All team members) • *Update the Index* – (Vashavi) • *Update Chatbot references throughout the report (Vashavi)* • *Update the Deployment Diagram (Vashavi and Adrian)* • *Add the Activity Plan (Vashavi)* • *Add the Client-Side Description (Vashavi)* • *Update Chatbot Implementation – Refine the Conversation Flow (Vashavi)* • *Add Demo Section (copy/paste the table) (Vashavi)* • *Provide a Description of Strengths and Weaknesses Nafis and Adrian)*

		• *Creating web design using Figma (Nargis)* • *Update the README file and create a txt file (Nargis)* • *Create a GitHub Link to repository and the deployed page to AWS with credentials -> pdf file (Nargis)* • *Push AdminDashboard (final) branch to Main branch (Nargis)* • *Final review of the report*
Thursday April 3rd	*All team member present (WhatsApp Chat)*	*Todo's:* • *Add the Chatbot flowchart URL - Nargis* • *Update the latest GitHub URL of the database – Nargis* • *Go over the Cloud integration and deployment parts and make edits if needed – Adrian* • *Add the Strengths, Weaknesses and Insights of the System (Nafis and Adrian)* • *GANTT chart – Nargis* • *Web Design using Figma - Nargis* • *Update the README file and create a txt file (Nargis)* • *Create a GitHub Link to repository and the deployed page to AWS with credentials -> pdf file (Nargis)* • *Push AdminDashboard (final) branch to Main branch (Nargis)* • *Review the report and make edits wherever needed – Nafis*
Friday April 4th *Time: 11am*	*Nargis, Vashavi*	*Done:* • *Add the Chatbot flowchart URL - Nargis* • *Update the latest GitHub URL of the database – Nargis* • *GANTT chart – Nargis* • *Web Design - Nargis* • *Create a GitHub Link to repository and the deployed page to AWS with credentials -> pdf file (Nargis)* *Push AdminDashboard (final) branch to Main branch (Nargis)*
Friday April 4th *Time: 3pm*	*Nafis Adrian*	• *Conduct full Analysis of the System and update final report Sections 3.4 and 7.4* • *Explanation and Implementation of Security of our system.* • *Review & Work on the final version of the report* • *Conduct a performance test in terms of computer networking and application protocols of our website. (EECS 3214)* • *Analyze Performance. (Response Time, Throughput, Utilization)* • *Install Apache JMeter* • *Simulate 10-100 concurrent users hitting /login, /cart, /checkout* • *Metrics: requests per second, avg response time, error % under load*

		 Apache JMeter (5.6.3) Test Plan Test Plan Name: Test Plan Comments: User Defined Variables Name: Value Add Add from Clipboard Run Thread Groups consecutively (i.e. one at a time) Run tearDown Thread Groups after shutdown of main threads Functional Test Mode (i.e. save Response Data and Sampler Data) Selecting Functional Test Mode may adversely affect performance. Add directory or jar to classpath Browse... Delete Clear Library • • *Conduct endpoint tests using Apache JMeter* *Conduct test using: https://www.webpagetest.org/*
Saturday April 5th *Time: 3 pm (40 minutes)*	*All team members present*	*Did a thorough review of the report and code.* *Discussed final change to be implemented.* *Report and the EV Store e-commerce site are now fully ready for submission as Deliverable 3* 🥳

10 Test Driven Development

Test ID	*TID1*
Category	*User Registration & Authentication*
Requirements Coverage	*UC1: Register,* *UC4: Authenticate-Users*
Initial Condition	*The system is running, and the registration page is accessible.*
Procedure	*1. Open the registration page and enter valid details.* *2. Click "Register", and the system validates and stores user credentials.* *3. Proceed to the login page, enter valid credentials, and click "Login".* *4. The Authentication Service verifies credentials and grants access.*
Expected Outcome	*1. Registration is successful, and the system displays "Registration complete. Please log in."* *2. Login is successful, and the user is redirected to the homepage/dashboard.* *3. A welcome message "Welcome, [User]!" is displayed.*
Notes	· *Ensure passwords are hashed before storing.* · *Prevent duplicate registrations using the same email* · *Ensure authentication service queries the database securely.*

WEB API	*POST http://localhost:8080/api/register*
	
	JSON Body: { "firstName": "Vashavi", "lastName": "Shah", "dateOfBirth": "1990-01-01", "email": "vashavi.shah@evstore.com", "password": "WeakPassword123", "region": "North America", "role": "USER" }
	
	POST http://localhost:8080/api/login
	JSON Body: { "email": "vashavi.shah@evstore.com", "password": "WeakPassword123" }

Test ID	*TID2*
Category	*Failed Login Attempts & Security*
Requirements Coverage	*UC2: Sign-In,* *UC4: Authenticate-Users*
Initial Condition	*The system is running, and the user enters incorrect login details*
Procedure	*1. Open the login page and enter an invalid username or password.* *2. Click "Login", and the Authentication Service checks credentials.* *3. If credentials do not match, the system logs the failed attempt.* *4. After multiple failed attempts, the system temporarily locks the account.*
Expected Outcome	*1. The system displays an error message: "Invalid username or password. Please try again."* *2. If multiple failed attempts occur, an additional message appears: "Too many failed attempts. Please try again later."* *3. No session or JWT token is created, and the user remains on the login page.*
Notes	· *Implement brute-force protection to block excessive failed attempts.* · *The system should not disclose if the username exists for security reasons.*

WEB API	*POST http://localhost:8080/api/login* *JSON Body: { "email": "e.g@z.com", "password": "Password12" }*

Test ID	*TID3*
Category	*User Logout & Session Handling*
Requirements Coverage	*UC3: Sign-Out,* *UC4: Authenticate-Users*
Initial Condition	*The user is logged in with an active session*
Procedure	*1. Click the "Logout" button on the dashboard.* *2. Confirm logout when prompted.* *3. The system invalidates the session or JWT token and redirects to the login page.* *4. Attempting to access restricted pages redirects back to the login screen.*
Expected Outcome	*1. The user is successfully logged out.* *2. The system displays a confirmation message: "You have been logged out successfully."* *3. Session data is fully cleared, and restricted pages are inaccessible.*
Notes	· *Ensure JWT/session tokens are completely invalidated.* · *Users should not be able to navigate back to protected pages after logout.*
WEB API	*POST http://localhost:8080/api/logout*

Test ID	*TID4*
Category	*List Electric Vehicles*
Requirements Coverage	*UC4: List Vehicles*
Initial Condition	*System running, vehicles exist in the database*
Procedure	*1. The user logs in and navigates to the Vehicle Catalog page.* *2. The system fetches and displays all available vehicles.* *3. The user scrolls through the list of available vehicles.*
Expected Outcome	*1. Users can browse all available vehicles in a paginated list..*
Notes	• *Ensure vehicles are loaded dynamically for performance.* • *Maintain proper image loading to prevent broken links.*
WEB API	*POST http://localhost:8080/api/login* *JSON Body: { "email": "e.g@z.com", "password": "Password123" }* *GET http://localhost:8080/api/vehicles*

Test ID	*TID5*
Category	*Sort Electric Vehicles*
Requirements Coverage	*UC5: Sort Vehicles*
Initial Condition	*The system is running, and the database has multiple vehicles.*
Procedure	*1. The user navigates to the Vehicle Catalog page.* *2. The user selects a sorting option (e.g., Price: Low to High, High to Low, Mileage).* *3. The system processes the request and rearranges the vehicle list.* *4. The user verifies if the sorting order is correct.*
Expected Outcome	*1. Vehicles appear in the correct order (ascending or descending).*
Notes	• *Ensure sorting should be fast and accurate across different filters.* • *Users should be able to toggle sorting options dynamically.*
WEB API	*GET http://localhost:8080/api/vehicles?sort=price_asc* *GET http://localhost:8080/api/vehicles?sort=price_desc*

Test ID	*TID6*
Category	*Filter Electric Vehicles*
Requirements Coverage	*UC6: Filter Vehicles*
Initial Condition	*The system has multiple vehicles with different models, years, and price ranges.*
Procedure	*1. The user navigates to the Vehicle Catalog page.* *2. The user applies filters (Brand, Price Range, Model Year, Features).* *3. The system filters and displays only matching vehicles.*
Expected Outcome	*1. The user sees only vehicles that match the selected filters.*
Notes	• *Ensure that multiple filters can be applied at once.* • *Filters should not reset when switching pages.*
WEB API	*GET http://localhost:8080/api/vehicles?brand=Tesla&year=2025* *GET http://localhost:8080/api/vehicles?price=59999*

Test ID	*TID7*
Category	*VIew Electric Vehicles Details*
Requirements Coverage	*UC7: VIew Vehicles Details*
Initial Condition	*Vehicles exist in the database, and detailed specifications are stored.*
Procedure	*1. The user navigates to the Vehicle Catalog page.* *2. The system retrieves and displays the vehicle details.* *3. The user reviews information such as price, mileage, specifications and images.*
Expected Outcome	*1. The detailed page loads successfully with all specifications.*
Notes	• *Ensure all images load correctly.* • *Provide an error message if vehicle details are unavailable.*

WEB API	*GET http://localhost:8080/api/vehicles/7* *GET http://localhost:8080/api/vehicles/11*

Test ID	*TID8*
Category	*VIew Hot Deals*
Requirements Coverage	*UC8: VIew Hot Deals*
Initial Condition	*The system is running, and discounted vehicles exist.*
Procedure	*1. The user clicks on the Hot Deals section.* *2. The system retrieves discounted vehicle listings.* *3. The user can view the discount percentage and final price.*
Expected Outcome	*1. The user can see all available discounts.*
Notes	• *Ensure expired deals do not appear.* • *Highlight vehicles with the highest discounts.*
WEB API	*GET http://localhost:8080/api/vehicles/hot-deals*

Test ID	*TID9*
Category	*Compare Electric Vehicles*
Requirements Coverage	*UC10: Compare Vehicles*
Initial Condition	*The system has multiple vehicles with varying specifications.*
Procedure	*1. The user selects two or more vehicles for comparison.* *2. The system retrieves specifications and displays a side-by-side view.*
Expected Outcome	*1. The user can easily compare vehicle features, pricing, and specifications.*
Notes	• *Ensure responsive layout for proper comparison.* • *Allow users to remove a vehicle from comparison dynamically.*
WEB API	*POST http://localhost:8080/api/vehicles/compare* *JSON Body: { "vehicleIds": [7, 9] }*

Test ID	*TID10*
Category	*Customize Electric Vehicles*
Requirements Coverage	*UC11: Customize Vehicles*
Initial Condition	*The system has vehicles with available customization options.*
Procedure	*1. The user navigates to a vehicle and looks for "Customizations Available".* *2. The system loads available color and wheel choices.* *3. The user makes selections and confirms.*
Expected Outcome	*1. The system applies customizations and updates the final price.*
Notes	• *Ensure customization saves to the cart for later checkout.*

WEB API	*PUT http://localhost:8080/api/vehicles/7/customize* *JSON Body: { "exteriorColour": "Ultra Red", "tires": "Premium" }*

Test ID	*TID11*
Category	*Add to Cart & Edit/Remove from Cart*
Requirements Coverage	*UC12: Shopping Cart, UC13: Modify/Remove from Cart*
Initial Condition	*The user has browsed the catalog and selected at least one vehicle for purchase.*
Procedure	*1. The user selects a vehicle and clicks "Add to Cart".* *2. The system verifies vehicle availability and adds it to the shopping cart.* *3. he user navigates to the cart page to review their selection.* *4. The user updates the quantity, removes an item, or clears the entire cart.* *5. The user updates the quantity, removes an item, or clears the entire cart.*
Expected Outcome	*1. The selected vehicle appears in the cart after adding.* *2. The user can edit the quantity or remove items successfully.* *3. The total price updates dynamically based on cart modifications.*
Notes	• *Ensure vehicles cannot be added if out of stock.* • *Cart should persist across sessions (if applicable).* • *Users should receive confirmation messages when adding, editing, or removing items.*
WEB API	*POST http://localhost:8080/api/cart/add* *JSON Body: {{ "cartId": 3, "vehicleId": 7, "quantity": 2, "price": 59990.00, "customizations": { "color": "red", "wheels": "sport" } }*

Test ID	*TID12*
Category	*Checkout Process*
Requirements Coverage	*UC14: Payment & Checkout*
Initial Condition	*The user has at least one vehicle in the cart.*
Procedure	*1. Click "Checkout", enter payment details, and confirm.* *2. The payment processes successfully.*
Expected Outcome	*1. The order is successfully placed.* *2. You get a order confirmation message via email*
Notes	• *Ensure secure payment gateway integration.* • *Provide an error message via email if the payment fails.*
WEB API	*POST http://localhost:8080/api/cart/checkout*

Test ID	*TID13*
Category	*Use Loan Calculator*
Requirements Coverage	*UC9: Loan Estimation*
Initial Condition	*Loan API is available, and vehicles have listed prices.*
Procedure	*1. The user navigates to a vehicle page.* *2. Enter loan details (price, down payment, interest rate, term, selects province (to calculate Tax rate)).* *3. Click "Calculate".*

Expected Outcome	*1. The system displays accurate loan estimates with monthly, biweekly and weekly payments.*
Notes	• *Ensure interest rates are correctly applied.*
WEB API	*POST http://localhost:8080/api/loan/provinceCalculate* *JSON Body: { "vehicle_price": 59990.00, "term_months": 60, "down_payment": 5000, "interest": 3.5, "province": "ON" }*

Test ID	*TID14*
Category	*360-Degree View*
Requirements Coverage	*UC17: 3D & Zoom Feature*
Initial Condition	*The system has 3D models available.*
Procedure	*1. Click "360° View".* *2. Rotate and zoom in on different sections.*
Expected Outcome	*1. The interactive 3D model loads properly.*
Notes	• *Provides fallback options if 3D models fail.*
UI	*Can be achieved by selecting a vehicle model to "Order Now" via the UI and use the "Zoomed In View"*

Test ID	*TID15*
Category	*Write Review & Rating*
Requirements Coverage	*UC15: Customer Reviews*
Initial Condition	*The user has purchased a vehicle.*
Procedure	*1. Click "Write a Review", enter feedback, and submit.*
Expected Outcome	*1. The review is saved and displayed publicly.*
Notes	• *Ensure users can edit or delete reviews.*
WEB API	*POST http://localhost:8080/api/reviews/add* *JSON Body: { "vehicleId": 7, "rating": 5, "reviewText": "Absolutely love this car! Smooth ride and great mileage." }*

Test ID	*TID16*
Category	*Chatbot Assistance*
Requirements Coverage	*UC16: Chatbot Support*
Initial Condition	*The system has an active chatbot.*
Procedure	*1. The user types a question about vehicles or checkout.* *2. The chatbot retrieves relevant answers.*
Expected Outcome	*1. The chatbot responds with helpful information.*
Notes	• *Ensure chatbot integrates with live support if needed.*

Additional 16 Test Cases:

Test ID	*TID17*
Category	*User Registration Validation*
Requirements Coverage	*UC1 - Register*
Initial Condition	*The system is running, and the registration API is accessible.*
Procedure	*1. Send a POST request to /api/register with valid user details.* *2. The system validates and stores user credentials in the database.* *3. A response with HTTP 201 (Created) is returned.*
Expected Outcome	*1. The new user is successfully registered and saved in UserRepository.* *2. The response contains a success message and user ID.*
Notes	• *Ensure password hashing before storing.* • *Prevent duplicate registrations with the same email.*
WEB API	*POST http://localhost:8080/api/register* *JSON Body: { "firstName": "Ruba", "lastName": "Omari", "dateOfBirth": "1990-01-01", "email": "ruba.omari@evstore.com", "password": "WhoAmI123", "region": "North America", "role": "USER" }*

Test ID	*TID18*
Category	*Failed User Registration*
Requirements Coverage	*UC1 - Register*
Initial Condition	*The registration API is running.*
Procedure	*1. Send a POST request to /api/register with missing or invalid fields.* *2. The system validates input and returns an error message.*
Expected Outcome	*1. The response contains a 400 Bad Request status.* *2. Error message: "Email is required" or "Password must be at least 8 characters"*
Notes	• *Ensure backend validation rules prevent invalid input.*
WEB API	*POST http://localhost:8080/api/register* *JSON Body: { "firstName": "Ruba", "lastName": "Omari", "dateOfBirth": "1990-01-01", "email": "", "password": "", "region": "North America", "role": "USER" }*

Test ID	*TID19*
Category	*User Login with Valid Credentials*
Requirements Coverage	*UC2 - Sign In*
Initial Condition	*A registered user exists in UserRepository*
Procedure	*1. Send a POST request to /api/login with valid credentials.* *2. The system verifies the user and returns a JWT token.*
Expected Outcome	*1. The login is successful, and the user gets an authentication token.* *2. The response contains "Welcome, [User]!".*

Notes	• *Ensure JWT tokens are properly generated and returned.*
WEB API	*POST http://localhost:8080/api/login* *JSON Body: { "email": "e.g@z.com", "password": "Password123" }*

Test ID	*TID20*
Category	*Fetching Products from Database*
Requirements Coverage	*UC4 - List Electric Vehicles*
Initial Condition	*At least one product exists in ProductRepository.*
Procedure	*1. Send a GET request to /api/products* *2. The system retrieves and returns all available products.*
Expected Outcome	*1. Products are fetched and returned as JSON.* *2. The response contains a list of electric vehicles.*
Notes	• *Implement pagination to limit results per request..*
WEB API	*GET http://localhost:8080/api/vehicles*

Test ID	*TID21*
Category	*Fetch Single Product Details*
Requirements Coverage	*UC7 - View Vehicle Details*
Initial Condition	*A valid product exists in ProductRepository.*
Procedure	*1. Send a GET request to /api/products/{id}.* *2. The system fetches and returns the product details.*
Expected Outcome	*1. The response contains detailed specifications of the product.* *2. If the product does not exist, return 404 Not Found.*
Notes	• *Implement error handling for missing product IDs.*
WEB API	*GET http://localhost:8080/api/vehicles/7*

Test ID	*TID22*
Category	*Adding a Product to Cart*
Requirements Coverage	*UC12 - Add to Cart*
Initial Condition	*A valid user is logged in, and the product exists.*
Procedure	*1. Send a POST request to /api/cart/add with a valid product ID.* *2. The system adds the product to the user's shopping cart.*
Expected Outcome	*1. The product appears in CartItemRepository.* *2. A success message: "Item added to cart successfully!".*
WEB API	*POST http://localhost:8080/api/cart/add* *JSON Body: {{ "cartId": 3, "vehicleId": 7, "quantity": 2, "price": 59990.00, "customizations": { "color": "red", "wheels": "sport" } }*

Test ID	*TID23*
Category	*Removing a Product from Cart*
Requirements Coverage	*UC13 - Edit/Remove from Cart*
Initial Condition	*A cart item already exists.*
Procedure	*1. Send a DELETE request to /api/cart/remove/{id}.* *2. The system removes the item from the CartItemRepository.*
Expected Outcome	*1. The cart updates and the item is removed.* *2. If the item does not exist, return 404 Not Found.*
Notes	• *Implement real-time cart updates.*
WEB API	*LETE http://localhost:8080/api/cart/remove/7*

Test ID	*TID24*
Category	*Checkout & Payment Processing*
Requirements Coverage	*UC14 - Checkout*
Initial Condition	*The user has items in the cart and a valid payment method.*
Procedure	*1. Send a POST request to /api/checkout with payment details.* *2. The system processes payment and creates an order.*
Expected Outcome	*1. The order appears in OrderRepository.* *2. A payment confirmation is sent via email.*
Notes	• *Ensure secure payment processing using encrypted data.*
WEB API	*POST http://localhost:8080/api/cart/checkout*

Test ID	*TID25*
Category	*Failed Payment Handling*
Requirements Coverage	*UC14 - Checkout*
Initial Condition	*The user attempts a checkout with insufficient funds.*
Procedure	*1. Send a POST request to /api/checkout with incorrect payment details.* *2. The system rejects the transaction.*
Expected Outcome	*1. A 402 Payment Required response is returned.* *2. Error message: "Transaction declined due to insufficient funds".*
Notes	• *Ensure secure handling of payment failures.*

Test ID	*TID26*
Category	*Admin Deletes a Vehicle*
Requirements Coverage	*UCA1*
Initial Condition	*Admin removes a vehicle from the catalogue.*
Procedure	*1. Send a DELETE request to /api/admin/vehicles/delete/11*

	2. The chatbot retrieves and returns an answer.
Expected Outcome	*Vehicle with VehicleID 11 is removed from the catalogue.*
WEB API	*DELETE http://localhost:8080/api/admin/vehicles/delete/11*

Test ID	*TID27*
Category	*Admin Updates Order Status from “Pending” to “Completed”*
Requirements Coverage	*UCA1*
Initial Condition	*Admin verifies payment status and then updates the order status.*
Procedure	*1. Admin logs in and sees the table of all orders.* *2. Admin updates the selected order that the payment is successful.*
Expected Outcome	*Order Status with OrderID gets changed.*
WEB API	*PUT http://localhost:8080/api/admin/reports/*orders/{orderid}/status *Params:* *Key: Status* *Value: Completed* *PUT http://localhost:8080/api/admin/reports/orders/7/status?status=Completed*

Test ID	*TID28*
Category	*Admin Access to Sales Reports*
Requirements Coverage	*UC18 - Run Reports*
Initial Condition	*The admin is logged in.*
Procedure	*1. Send a GET request to /api/admin/reports/sales.* *2. The system returns a detailed sales report.*
Expected Outcome	*1. Sales reports are retrieved and displayed.*
Notes	• *Ensure only admins have access.*
WEB API	*GET http://localhost:8080/api/admin/reports/sales*

Test ID	*TID29*
Category	*Admin gets to see the list of all users.*
Requirements Coverage	*UC A1*
Initial Condition	*The admin is logged in and sees the list of all users of the system.*
Procedure	*1. Send a GET request to /api/users* *2. The system returns the list of all users.*
WEB API	*GET http://localhost:8080/api/*users
Notes	• *Ensure only admins have access.*

Test ID	*TID30*
Category	*Order Confirmation Notification*

Requirements Coverage	*UC22 - Send Notifications*
Initial Condition	*A customer has successfully placed an order.*
Procedure	*1. The system detects a new order in OrderRepository.* *2. The NotificationService triggers a purchase confirmation email.*
Expected Outcome	*1. The customer receives an email/SMS confirming their order.* *2. The message contains order ID, product details, and expected delivery date.*
Notes	• *Ensure email templates are correctly formatted.* • *Implement error handling for failed notifications.*

Test II	*TID31*
Categ	*Adding/Deleting a New Product*
Requi	*UC18 - Manage Vehicle Listings*
Initial	*The admin is logged in and has access to ProductService.*
Proce	*1. The admin sends a POST request to /api/admin/products/add(or delete) with product details.* *2. The system validates and stores the product in ProductRepository.*
Expec	*1. The new vehicle is added to the database and appears in the store.* *2. If the product ID already exists, return 409 Conflict.*
Notes	• *Ensure input validation for required fields (price, model)*
WEB A	*POST http://localhost:8080/api/admin/vehicles/add* *JSON Body: { "type": "New", "model": "Model S", "payment": "Finance", "trim": "Long I* *Drive", "paint": "Black", "wheels": "Standard", "interior": "Black", "performanceUpgrac* *true, "price": 139990.00, "year": "2025", "mileage": 755, "hotDeal": true, "exteriorCo* *Blue", "tires": "Standard", "range": "700 km", "topSpeed": "211 km/h", "seat_layout": "F* *Interior", "cover_image": "https://static-* *assets.tesla.com/configurator/compositor?context=design_studio_2&options=$MT356,$PPSB,$W3* *T34&model=m3&size=1920&size=720&bkba_opt=2&crop=1150,580,390,250&overlay=0&" }*

Test ID	*TID32*
Category	*Loan Calculator API Validation*
Requirements Coverage	*UC9 - Use Loan Calculator*
Initial Condition	• *The Loan Calculator API is deployed and running.* • *The customer selects a vehicle and enters loan parameters (price, interest rate, down payment, duration).*
Procedure	*1. The customer navigates to the Loan Calculator page.* *2. They enter the loan parameters (e.g., price, down payment, interest rate, and loan duration).* *3. The system sends a POST request to the Loan Calculator API.* *4. The API validates the inputs and calculates the monthly installment and total loan amount.* *5. If the API responds successfully, the system displays the calculated loan details.* *6. If invalid inputs are provided, the API returns an error message, and the system prompts the user to correct them.*
Expected Outcome	*1. If valid loan parameters are entered, the system displays an accurate loan breakdown (e.g., monthly installment, total cost).* *2. If invalid inputs are provided, the system displays an error message: "Invalid input. Please enter a valid loan amount, interest rate, or duration."*

	3. If the API is down or unresponsive, the system shows an error: "Loan service is currently unavailable. Please try again later."
Notes	• *Ensure the Loan Calculator API is tested with various input combinations to handle edge cases.* • *Validate error handling for incorrect values (e.g., negative interest rate, zero loan duration).* • *Implement graceful failure handling if the Loan API is down.*
WEB API	*POST http://localhost:8080/api/loan/provinceCalculate* *JSON Body: { "vehicle_price": 59990.00, "term_months": 60, "down_payment": 5000, "interest": 3.5, "province": "ON" }*

Other endpoints Test Cases:

Test ID	*TID33*
Category	*Get My Reviews and Get Reviews by Vehicle ID*
Requirements Coverage	*UC17 – View Customer Reviews*
Initial Condition	• *The user is logged in and has submitted reviews.*
Procedure	*1. Send a GET request to /api/reviews/my-reviews with valid authentication.* *2. Send a GET request to /api/reviews/vehicle/10 to retrieve reviews for the vehicle with ID 10.*
Expected Outcome	• *The first call returns a JSON list of all reviews submitted by the user.* • *The second call returns a JSON list of reviews for vehicle ID 10.*
Notes	• *If no reviews exist, return an empty list or a friendly message.*
WEB API	*GET http://localhost:8080/api/reviews/my-reviews* *GET http://localhost:8080/api/reviews/vehicle/10*

Test ID	*TID34*
Category	*View My Reviews*
Requirements Coverage	*UC17 – View Customer Reviews*
Initial Condition	• *The user is logged in and has previously submitted reviews.*
Procedure	*1. Send a GET request to /api/reviews/my-reviews with valid credentials.*
Expected Outcome	*A JSON list showing all the reviews submitted by the user is returned.*
Notes	• *This test ensures that the "See My Reviews" feature accurately displays the user's reviews.*
WEB API	*GET http://localhost:8080/api/reviews/my-reviews*

Test ID	*TID35*
Category	*Order Processing*
Requirements Coverage	*UC14 – Checkout & Order Confirmation*
Initial Condition	• *The user has items in the cart and has provided valid order/payment details.*
Procedure	*1. Send a POST request to /api/orders/confirm with the required order details in the payload.*

Expected Outcome	• *The system confirms the order and returns a success message along with order details.*
Notes	• *Validate that the order is recorded correctly in the database and duplicate orders are not created if re-sent.*
WEB API	*POST http://localhost:8080/api/orders/confirm*

Test ID	*TID36*
Category	*Get Orders for the Logged-In User*
Requirements Coverage	*UC14 – Checkout & Order Confirmation*
Initial Condition	• *The user is authenticated and has placed one or more orders..*
Procedure	*1. Send a GET request to /api/orders/my-orders with valid authentication.*
Expected Outcome	• *A JSON list of the user's orders is returned.*
Notes	• *Implement pagination if needed.*
WEB API	*GET http://localhost:8080/api/orders/my-orders*

Test ID	*TID37*
Category	*User Profile Authentication*
Requirements Coverage	*Retrieve Authenticated User*
Initial Condition	• *The user is logged in with a valid JWT token.*
Procedure	*1. Send a GET request to /api/orders/me with proper authentication.*
Expected Outcome	• *The system returns a JSON response containing the authenticated user's details (e.g., name, email, role).*
Notes	• *Ensure that sensitive information is not exposed.*
WEB API	*GET http://localhost:8080/api/orders/me*

Test ID	*TID38*
Category	*Order Confirmation*
Requirements Coverage	*UC14 – Checkout & Order Confirmation*
Initial Condition	• *The user has completed the checkout process with valid payment details.*
Procedure	*1. Send a POST request to /api/orders/confirm again to test idempotency.*
Expected Outcome	• *The system returns a confirmation without duplicating the order.*
Notes	• *Verify that re-confirming does not create a new order.*
WEB API	*POST http://localhost:8080/api/orders/confirm*

Test ID	*TID39*
Category	*Shopping Cart Management*

Requirements Coverage	*UC12 – Shopping Cart*
Initial Condition	• *The user is logged in and has added items to their cart..*
Procedure	*1. Send a GET request to /api/cart/my-cart with valid authentication.*
Expected Outcome	• *A JSON response displaying the current contents of the user's cart is returned.*
Notes	• *Verify that re-confirming does not create a new order.*
WEB API	*GET http://localhost:8080/api/cart/my-cart*

Test ID	*TID40*
Category	*Admin Authentication*
Requirements Coverage	*Admin Login*
Initial Condition	• *An admin account exists with the specified credentials.*
Procedure	*1.Send a POST request to /login with the admin's email and password.*
Expected Outcome	• *The admin successfully logs in and receives a JWT token.*
Notes	• *Ensure that the system correctly differentiates between admin and regular user logins*

WEB API	*POST http://localhost:8080/login*
	{
	  "email": "nargis.rafie@admin",
	  "password": "BeginnerPassword123"
	}