AMBULANCE TRACKING SYSTEM

A project report submitted to Andhra University in

partial fulfilment for the Award of the Degree of

MASTER OF COMPUTER APPLICATIONS

Submitted by

PILLALA VENKATESH

Regno: ***323225660060***

Under the Guidance of

G.G.N. ALEKHYA RANI

NOBLE INSTITUTE OF SCIENCE & TECHNOLOGY

Affiliated to Andhra University, Visakhapatnam

(Approved by AICTE, New Delhi, India)

VISAKHAPATNAM

(2023-2025)

NOBLE INSTITUTE OF SCIENCE & TECHNOLOGY

Affiliated to Andhra University, Visakhapatnam

(Approved by AICTE New Delhi, India)

CERTIFICATE

This is to certify that the project work entitled "AMBULANCE TRACKING SYSTEM**" is being submitted by PILLALA VENKATESH, to the Noble Institute of Science and Technology, Visakhapatnam in partial fulfillment for the Award of the Degree of "MASTER OF COMPUTER APPLICATIONS" has been carried out under my guidance and Supervision.**

Principal **Project Guide**

G.G.N. ALEKHYA RANI

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and appreciation to my project guide “G.G.N. ALEKHYA RANI” who have contributed to the successful completion of this project documentation. Without your support and expertise, this project would not have been possible. First and foremost, I extend my heartfelt thanks to “G.G.N. ALEKHYA RANI” for your guidance, mentorship, and valuable insights throughout the project. Your unwavering support and encouragement played a pivotal role in shaping this documentation. I am grateful to the Noble Institute of Science and Technology for providing the necessary resources, infrastructure, and support that enabled the successful execution of this project. Their commitment to innovation and excellence has been a driving force behind my achievements. Lastly, I want to express my gratitude to my friends and family for their unwavering support, understanding, and encouragement throughout the project. Their belief in my abilities and constant motivation provided the strength needed to overcome challenges and persevere. To all those who have contributed directly or indirectly, I offer my sincerest thanks. Your dedication, collaboration, and expertise have made this project documentation a comprehensive and meaningful document. It is with great appreciation that I acknowledge all of my supporters, and I look forward to continue many projects like this in future also.

Thank you.

PILLALA VENKATESH
Roll No. 323225660060

DECLARATION

I hereby declare that the project entitled "AMBULANCE TRACKING SYSTEM" is submitted by me to the Department of Computer Science, Noble Institute of Science and Technology, Visakhapatnam, in partial fulfilment for the Award of Degree of Master of Computer Applications is entirely based on my own study and findings and is being submitted for the first time. It has not been submitted or published earlier for the Award of any Degree or Diploma of this University or any other University.

Date:
Place: Visakhapatnam

Name of the Student
PILLALA VENKATESH
Reg. No. 323225660060

ABSTRACT

The **Quick Ambulance Tracking** system is a real-time, location-based emergency response platform designed to connect users with nearby available ambulances efficiently. Developed using **Microservices Architecture**, it ensures modularity, scalability, and independent deployment of services. The backend is implemented in **Spring Boot 3.4.3** and divided into four core microservices: **Quick-Ambulance-Database** handles all CRUD operations and stores key entities such as User, Driver, Ambulance, DriverLogs, TrackDetails, and LiveLocation in a centralized **MySQL** database. **Quick-Ambulance-Location** enables real-time location tracking using **WebSocket (SockJS + STOMP)** and calculates nearest drivers based on geolocation. **Quick-Ambulance-Login-Logout** microservice uses **Spring Security** with **JWT** for user authentication and session tracking, while also maintaining login/logout activity logs. The **Quick-Ambulance-API-Gateway** acts as a unified access point, routing requests, managing CORS, and handling security validations. The **frontend**, built with **React JS**, provides role-based dashboards for Admins, Drivers, and Users. Admins manage system data and monitor activity; Drivers receive alerts and send location updates; Users book ambulances and track drivers on a live map. Inter-service communication is enabled through **Feign Clients**, and **Eureka Server** is used for service discovery and load balancing. The system operates efficiently under high demand, offers fault tolerance, and is designed for future integration with features such as WhatsApp/SMS alerts, OAuth authentication, hospital locators, and available doctor listings. Quick Ambulance provides a smart, scalable, and responsive solution to improve emergency medical transportation, reduce response time, and enhance patient outcomes in critical situations.

Keywords

Quick Ambulance Tracking, Microservices Architecture, Spring boot 3.4.3, Quick-Ambulance-Database, MySQL, Quick-Ambulance-Location, WebSocket, Quick-Ambulance-Login-Logout, Spring Security, JWT, Quick-Ambulance-API-Gateway, React JS, Feign Clients, Eureka Server.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 INTRODUCTION

In emergency medical situations, timely access to ambulance services can be the difference between life and death. However, in many regions, locating and dispatching the nearest available ambulance remains a challenge due to a lack of real-time systems. To address this problem, this project titled **"Quick Ambulance – A Real-Time Ambulance Booking and Tracking System"** aims to provide a seamless, real-time platform for users to book ambulances, track their location, and communicate with drivers efficiently.

This application is built using a **Microservices Architecture**, ensuring modularity, scalability, and independent service deployment. The backend is developed using **Spring Boot 3.4.3** with **Java 17**, and is divided into four core microservices:

- **Quick-Ambulance-Database** (handles all database operations),
- **Quick-Ambulance-Location** (manages real-time location tracking),
- **Quick-Ambulance-Login-Logout** (manages authentication using JWT),
- **Quick-Ambulance-API-Gateway** (routes all requests and handles security at entry level).

The system uses **MySQL** for persistent data storage and **Eureka Server** for dynamic service registration and discovery. Inter-service communication is handled via **Feign Clients**, and the entire system is protected using **Spring Security** with **JWT (JSON Web Token)** for authentication and authorization.

For real-time communication between users and drivers, **WebSocket with SockJS and STOMP protocol** is used, allowing the driver's live location to be pushed to the user's dashboard every few seconds. This ensures up-to-date route tracking and accurate ETAs.

The **React JS** frontend provides an interactive, user-friendly interface for three types of users:

- **Admin**: Manages ambulances, drivers, and users.
- **Driver**: Receives booking requests, shares live location, and views ride history.
- **User**: Books nearby ambulances and tracks their arrival on a live map.

Overall, this system not only digitizes and streamlines the ambulance booking process but also improves transparency, speed, and coordination in emergency response systems.

1.1.1 PURPOSE OF THE PROJECT

The primary purpose of the **Quick Ambulance** system is to provide a **fast, reliable, and real-time ambulance booking and tracking platform** that can assist users during medical emergencies. Traditional ambulance dispatch systems often suffer from inefficiencies such as delayed response, lack of real-time updates, manual booking processes, and poor communication between users and drivers. This project aims to overcome these challenges through a modern, location-aware, and role-based web application.

Key objectives behind the purpose of this project include:

1. **Reducing Response Time in Emergencies** By connecting users to the nearest available ambulances based on real-time location data, the system helps in minimizing delays and ensures faster medical assistance.
2. **Providing Real-Time Tracking** The integration of **WebSocket-based live location tracking** allows users and drivers to monitor each other's locations and the route in real-time, improving transparency and coordination.
3. **Ensuring Role-Based Access and Control** The system provides dedicated dashboards for different roles—Admin, Driver, and User—allowing each to perform actions based on their permissions and responsibilities.
4. **Automating Booking and Dispatch** The system calculates distances using geolocation data and automatically notifies nearby drivers, removing the need for manual intervention in ambulance allocation.
5. **Enhancing Data Management and Security** Using **Microservices Architecture** and **JWT-based authentication**, the application ensures secure, modular, and

scalable management of user and system data.

6. **Promoting Scalable and Maintainable Design** Microservices help in independent deployment, easy maintenance, and system extensibility, making the project suitable for real-world implementation and future enhancements.

1.1.2 PROJECT SCOPE

The Quick Ambulance system is designed as a real-time, scalable ambulance booking and live tracking solution that addresses the limitations of traditional emergency response systems. Built using a modern technology stack—React JS for the frontend and Spring Boot Microservices for the backend—the system ensures fast, interactive, and secure communication between users, drivers, and administrators. It integrates location-based services and real-time data transfer, making it suitable for rapid emergency response scenarios. The system emphasizes secure, role-based access through JWT-based authentication and modular design via microservices architecture.

The functional scope of the project includes several critical features: secure user registration and login, role-based dashboards for Admin, Driver, and User, and a comprehensive ambulance booking workflow. Admins can manage users, ambulances, and driver information while monitoring live activities across the system. Drivers can accept or reject requests, share their live location, and manage ambulance assignments. Users can view nearby ambulances within a 20 km range, initiate bookings, and track the driver's location in real-time. Booking and location services are fully integrated with backend data logging to ensure transparency and accountability.

The technical scope highlights the use of Spring Boot 3.4.3, Java 17, and React JS, providing a maintainable and high-performance software base. The backend is organized into decoupled services registered through Eureka Server and communicating via Feign Clients. The API Gateway provides a unified access point, handling security, route mapping, and CORS configurations. JWT-based authentication ensures stateless and secure interactions, while WebSocket (SockJS + STOMP) allows real-time updates of live location data every 3 seconds, ensuring responsiveness and reduced latency in the system.

Looking toward the future, the system offers high scalability potential. The architecture is capable of supporting the integration of mobile applications, payment gateways, and hospital or clinic dashboards. It can also be extended to incorporate emergency analytics, driver

performance tracking, traffic-aware routing, and GPS integration for enhanced ambulance navigation. This design ensures that the platform is not only effective in its current form but also adaptable for broader, real-world deployment in large-scale healthcare networks.

1.1.3 PROJECT OVERVIEW

Quick Ambulance is a real-time, location-aware web application designed to connect users in medical emergencies with nearby available ambulances efficiently and securely. The system uses a microservices-based architecture to handle various modules such as authentication, live location tracking, database operations, and API routing, making it modular, scalable, and easy to maintain.

The backend is developed using Spring Boot 3.4.3 and Java 17, divided into four core microservices:

- Quick-Ambulance-Database – Handles CRUD operations and manages MySQL database records.
- Quick-Ambulance-Location – Manages live location tracking using WebSocket and sends updates every 3 seconds.
- Quick-Ambulance-Login-Logout – Handles user authentication and authorization using JWT and Spring Security.
- Quick-Ambulance-API-Gateway – Manages route forwarding, JWT validation, and CORS configuration for all services.
- The frontend is developed using React JS, offering role-based dashboards for Admin, Driver, and User. Each role has a unique interface and capabilities:
- Admin: Manages ambulances, drivers, and users. Monitors system activities and live locations.
- Driver: Accepts bookings, shares live location, and maintains ride history.
- User: Registers/logins, views nearby ambulances, books rides, and tracks the assigned driver in real time.
- The system stores persistent data such as user details, driver logs, booking history, and real-time locations in a MySQL database, ensuring reliability and performance.
- Through the integration of WebSocket (SockJS + STOMP) for real-time communication, JWT for secure access control, and Eureka-based microservices for scalability, the Quick Ambulance application provides a complete end-to-end emergency medical dispatch solution.

1.2 PROBLEM STATEMENT

In emergency situations, every second counts. Unfortunately, the traditional ambulance booking systems in many regions still rely on manual call-based dispatching, lack real-time tracking, and suffer from poor communication between patients and ambulance drivers. These inefficiencies often result in delayed response times, miscommunication, and unavailability of nearby ambulances—all of which can risk patient lives.

The absence of:

- A real-time booking platform
- Live location tracking
- Role-based access control
- Automated allocation of the nearest available ambulance
- Secure and scalable architecture
- creates a critical gap in the healthcare emergency response system.
- Therefore, there is a strong need for a technology-driven solution that allows users to:
- Instantly locate and book the nearest available ambulance,
- Track the driver's real-time location,
- Seamlessly communicate during transit,
- And manage the entire workflow efficiently and securely.

The Quick Ambulance project aims to solve these problems by building a web-based application using Microservices Architecture, Spring Boot, React JS, JWT security, and WebSocket-based live tracking, providing a fast, reliable, and scalable emergency response platform.

1.3 PROPOSED SYSTEM

The Quick Ambulance system addresses the inefficiencies of traditional ambulance dispatch methods by offering a modern, real-time, web-based platform for emergency ambulance booking and live tracking. Developed using Microservices Architecture, the backend leverages Spring Boot, JWT-based Spring Security, and WebSocket for real-time communication, while the frontend is built using React JS for a dynamic and responsive user interface. The system supports three roles—Admin, Driver, and User—each with distinct

access rights. Admins can manage users, ambulances, and drivers; Drivers can accept booking requests, share their live location, and review ride history; and Users can register, log in, view nearby ambulances on a map, and book one in emergencies.

Security and responsiveness are key pillars of the system. Authentication is handled using JWT (JSON Web Tokens), allowing for secure, stateless communication between the frontend and backend. Once a user logs in, their identity and role are embedded in a token, which is then used for authorization in every subsequent request. The system also ensures real-time live tracking of ambulances using WebSocket with SockJS and STOMP, which enables the backend to send driver location updates to the user every 3 seconds post-booking. The booking logic uses the driver's coordinates to calculate distances and sends alerts to all available drivers within a 20 km radius. The first driver to accept the request is automatically assigned, and all other alerts are cleared to prevent confusion.

The architecture is cleanly divided into four microservices: (1) Quick-Ambulance-Database handles all database-related CRUD operations; (2) Quick-Ambulance-Location manages booking logic and real-time location updates; (3) Quick-Ambulance-Login-Logout oversees user authentication and driver login/logout tracking; and (4) Quick-Ambulance-API-Gateway acts as the entry point for all services and handles route security and CORS. All data is stored in a centralized MySQL database, including user/driver information, ambulance records, live locations, login/logout logs, and booking history. The React JS frontend offers role-based dashboards, live maps, and smooth navigation, ensuring a seamless experience for all users. The system's modular and scalable design allows for easy upgrades, maintenance, and reliable performance in critical situations.

1.4 MODULES

The Quick Ambulance system is architected using a modular microservices approach, ensuring that each major functionality is handled independently and efficiently. The Authentication & Authorization Module is managed by the Quick-Ambulance-Login-Logout microservice. This service handles user login and logout via JWT-based authentication. Upon successful login, it validates user credentials through a Feign client connected to the database and generates a secure JWT token that includes the username and role. It also tracks login/logout activity for drivers and manages token invalidation upon logout, contributing to secure and stateless session management.

The API Gateway Module—implemented using Spring Cloud Gateway in the Quick-

Ambulance-API-Gateway microservice—acts as the single entry point for all HTTP requests. It ensures that each incoming request is securely routed to the appropriate backend service after validating the attached JWT token. The gateway also distinguishes between public and private routes and configures CORS policies as needed. This centralized control mechanism provides both security and simplicity in request handling, supporting seamless interaction between the frontend and multiple backend services.

The User Management Module, implemented in the Quick-Ambulance-Database microservice, is responsible for handling all CRUD operations related to users, drivers, ambulances, roles, and driver logs. It also maintains booking history (Track_Details) and stores live location data. This module serves as the core data access layer and is used by other microservices through Feign Clients. By separating data management into its own microservice, the system ensures database integrity, easier maintenance, and improved scalability.

The Location & Booking Module, running in the Quick-Ambulance-Location microservice, plays a crucial role in real-time functionality. It leverages WebSocket (SockJS + STOMP) to continuously update the driver's location every 3 seconds post-login. This module calculates the geographical distance between the user and all available drivers, sending booking alerts to drivers within a 20 km radius. When a driver accepts a request, this module finalizes the booking, updates tracking details in the database, and ensures that other pending alerts are cleared, providing efficient and real-time ambulance allocation.

On the frontend, the system includes three key modules built with React JS: the Admin Dashboard, User Dashboard, and Driver Dashboard. The Admin Dashboard allows administrators to manage ambulances, drivers, users, and view live location tracking and booking history. The User Dashboard enables users to register, log in, view nearby ambulances, book rides, and track the driver in real-time. Lastly, the Driver Dashboard allows drivers to log in, automatically share location, accept/reject requests, and navigate to the user via a live map interface. These dashboards are integrated with backend APIs, offering a role-based, interactive, and real-time user experience.

2. SYSTEM ANALYSIS

2.1 PROBEM IDENTIFICATION

Traditional ambulance dispatch systems are often dependent on manual operations, such as phone-based bookings and verbal coordination between callers and dispatchers. These outdated practices lead to serious drawbacks, including delays in ambulance arrival, miscommunication during critical moments, and lack of visibility for both patients and hospitals regarding ambulance location and availability. Additionally, without a centralized platform, there is no efficient way to identify and allocate the nearest available ambulance, which can be life-threatening in emergency scenarios. These issues highlight the need for a more advanced, tech-driven solution.

2.2. OBJECTIVES AND INVESTIGATIONS

The main aim of this preliminary investigation was to determine whether building a real-time ambulance booking and tracking system would address the challenges found in existing emergency services. Specific objectives included evaluating the demand for such a platform, identifying all the stakeholders (patients, drivers, and administrators), and understanding how technologies like JWT, WebSocket, and Microservices could be leveraged for performance and scalability. The investigation also focused on reviewing competing systems to identify gaps and exploring the feasibility—both technical and financial—of implementing the proposed solution.

2.3 INFORMATION GATHERING TECHNIQUES

To ensure the investigation was grounded in real-world needs, several information gathering methods were employed. Interviews were conducted with doctors, hospital staff, and ambulance drivers to understand the operational challenges in emergency response. Questionnaires were shared with potential users to collect insights into their expectations from an emergency booking app. In addition, market research was performed to examine similar products and their limitations. Finally, field observation helped the team analyze the current workflow and spot inefficiencies in manual ambulance dispatch processes.

2.4. FINDINGS

The investigation uncovered several important findings. There is a clear lack of real-time, location-based ambulance services that are both user-friendly and operationally efficient. Users expect fast, transparent, and trackable ambulance booking experiences, while drivers prefer a system that allows them to receive requests, accept bookings, and share live location updates seamlessly. Administrators, on the other hand, require centralized tools to manage user accounts, monitor fleet movement, and review operational logs. These insights reinforced the need for a digital system that brings all stakeholders together in one streamlined workflow.

Based on the information collected, it was concluded that developing a real-time ambulance booking and tracking system is not only feasible but essential. By utilizing a microservices architecture supported by React JS, Spring Boot, JWT authentication, and WebSocket for live communication, the proposed Quick Ambulance system can significantly improve emergency response times, reduce operational inefficiencies, and enhance user experience. This early analysis validated the investment of time and resources into full-scale development of the platform.

2.5 REQUEST CLARIFICATION

Request clarification is a vital sub-phase of the preliminary investigation in the software development lifecycle. It focuses on gaining a clear, unambiguous understanding of what the user or client truly expects from the proposed system. In the case of the Quick Ambulance project, the request clarification process helped identify the exact needs of stakeholders such as patients, ambulance drivers, and hospital administrators. These users had varied expectations, ranging from real-time ambulance tracking to user-friendly booking interfaces and instant alert notifications.

During this stage, system analysts communicate directly with stakeholders through interviews, surveys, questionnaires, and group discussions. In this project, user feedback was collected to clarify several key requirements, such as the need for role-based dashboards (admin, driver, user), real-time location sharing using WebSocket, and secure login/logout using JWT tokens. Misunderstood or vaguely stated needs were further refined to ensure the final system aligns closely with actual user goals and practical constraints.

Clarifying the request also helps define the project's scope, avoid feature creep, and ensure that technical feasibility aligns with user expectations. For example, stakeholders initially

requested a simple ambulance booking platform, but through detailed clarification, it was refined into a feature-rich, microservices-based system with real-time communication, service discovery via Eureka, and modular deployment via API Gateway. This phase acted as a foundation for accurate requirement gathering and successful system design.

2.6 FEASIBILITY STUDY

The Feasibility Study is a critical phase in the System Development Life Cycle (SDLC) that evaluates whether the proposed system — *Quick Ambulance* — can be developed successfully within given constraints. This analysis focuses on four major dimensions: technical, operational, economic, and schedule feasibility. It helps ensure that the project is realistic, sustainable, and aligned with stakeholder expectations and goals.

2.6.1 Technical Feasibility: This aspect assesses whether the current technical infrastructure and expertise are adequate for building the system. The *Quick Ambulance* application leverages robust and modern technologies such as Spring Boot (Microservices), ReactJS for frontend, MySQL for database operations, and WebSocket (with SockJS and STOMP) for real-time communication. Backend services are designed using JDK 17 and Maven for efficient dependency and build management. The use of Eureka Server, Spring Cloud Gateway, and Feign Clients supports service discovery and inter-service communication, making the system scalable and maintainable. These proven technologies ensure that all required technical functionalities — including live tracking and real-time messaging — can be implemented effectively.

2.6.2. Operational Feasibility: Operational feasibility evaluates whether the system will be accepted and efficiently used in its intended environment. *Quick Ambulance* is designed to serve three user roles: Admin, Driver, and User — each with its own intuitive dashboard. Admins can manage ambulances, drivers, and monitor the entire system. Drivers receive live booking alerts and share real-time location updates. Users can view nearby ambulances, book them with one click, and track the driver's movement on the map. The role-based interface simplifies user interactions and ensures the system is aligned with user expectations and workflows.

2.6.3. Economic Feasibility: This aspect determines whether the system delivers value in proportion to its cost. All core components of the system — including Spring Boot, ReactJS, MySQL, and WebSocket — are open-source and freely available, significantly reducing software expenses. The system requires no specialized hardware or high-end infrastructure,

making it suitable for cost-conscious deployments, such as universities, startups, or regional hospitals. Furthermore, long-term benefits like improved emergency response time, better coordination, and system scalability promise high returns on investment. The system is economically feasible, especially for institutions with budget limitations.

2.6.4. Schedule Feasibility: Schedule feasibility checks if the system can be developed within a realistic and predefined timeline. *Quick Ambulance* was completed successfully within the MCA final-year project duration using an agile methodology. The microservices-based architecture allowed different modules to be developed and tested in parallel, accelerating development. Timely debugging, module integration, and real-time validations ensured that milestones were met without compromising quality. The project is schedule feasible, and its delivery timeline is achievable with proper planning.

2.7. FUNCTIONAL REQUIREMENTS

Functional requirements describe what the system should do — its core features and behavior in response to user interactions. The **Quick Ambulance** system is role-based and has three main types of users: **Admin**, **User**, and **Driver**. Each role has distinct functionalities.

2.7.1. User Module

- **User Registration**
  - A new user can register by providing a username, full name, phone, and email.
  - Role is assigned as USER by default.
- **User Login**
  - Users can log in using a valid username and password.
  - A JWT token is generated and stored in local storage upon successful login.
- **Live Location Tracking**
  - User's live location is automatically fetched and updated every 3 seconds.
- **View Nearby Ambulances**
  - Users can see all ambulances within a 20 KM radius on the map.
- **Book Ambulance**
  - Sends a booking request with location and username to the backend.
  - Waits for driver response (accept/decline).
- **Track Driver in Real Time**
  - Upon driver acceptance, user gets driver's live location, details, path on map.

2.7.2. Driver Module

- **Driver Login**
  - JWT-based login with role as DRIVER.
- **Update Login Logs**
  - On successful login, the system updates login time and status in DriverLogs table.
- **Send Live Location**
  - After login, sends current longitude and latitude to backend every 3 seconds.
- **Select Ambulance**
  - Selects one available ambulance to attach; updates DriverLogs and ambulance status.
- **Receive Booking Alerts**
  - Gets booking alerts if within 20 KM of a user request.
  - Can accept or decline the booking.
- **View User Details & Navigation**
  - Upon acceptance, gets user info and location; displays map and route to user.
- **Logout**
  - Updates logout time in DB and marks status as inactive.

2.7.3. Admin Module

- **Admin Login**
  - JWT-based login with role as ADMIN.
- **Manage Users, Drivers, Ambulances**
  - Add, update, delete operations on all entities.
  - Change driver or ambulance status.
- **Monitor Live Status**
  - View real-time location of drivers and ambulances.
  - View booking logs and track histories.

2.7.4. System Functionalities

- **JWT Authentication & Authorization**
  - Validates public/private routes using Spring Security and JWT.
  - Authenticates credentials and authorizes based on roles.

- **API Gateway Routing**
    - Routes all requests via Gateway, applies CORS, and validates JWT tokens.
- **Microservices Communication**
    - Clients interact via Feign and Eureka discovery for DB and location updates.
- **WebSocket Communication**
    - Real-time communication between user and driver during booking & tracking.
    - Live data sent every 3 seconds using SockJS + STOMP.
- **Database Operations**
    - Quick-Ambulance-Database microservice performs all CRUD operations for:
        - Roles, Users, Drivers, Ambulances
        - Live Location, Driver Logs, and Booking Records

2.8 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements define how the system should perform rather than what it should do. These include quality attributes such as performance, availability, scalability, usability, and more. For a critical application like *Quick Ambulance*, these attributes are essential to ensure the system is reliable, efficient, and user-friendly during emergency situations.

2.8.1. Availability

Availability ensures that the system remains accessible and operational whenever needed. Given the emergency nature of ambulance services, the *Quick Ambulance* platform must maintain 24/7 uptime with minimal to no downtime. The system is designed using a microservices architecture with Eureka Service Registry, which enables individual services to operate independently. For instance, even if the location tracking service fails, other services like login or database access continue functioning without interruption. Additionally, the use of an API Gateway with fallback mechanisms allows seamless rerouting of traffic, ensuring users are not affected during service outages or upgrades.

2.8.2. Efficiency

Efficiency measures how effectively the system utilizes resources like memory, CPU, and bandwidth while maintaining performance. In this project, efficiency is achieved by using WebSocket for real-time communication instead of continuous HTTP polling, significantly

reducing server load and network traffic. The backend services are load-balanced to distribute traffic evenly across instances, preventing overload. Furthermore, database operations are optimized through Spring Data JPA and carefully designed custom queries, ensuring fast data access and minimal latency in user interactions.

2.8.3. Flexibility

Flexibility refers to the ease with which the system can adapt to new requirements or enhancements. The *Quick Ambulance* system is built with modular microservices, where each core feature—such as login, tracking, or data management—is isolated in its own service. This separation allows for straightforward updates or extensions. If new features like payment integration, ambulance reviews, or support for regional languages are required in the future, they can be added as new microservices without modifying existing components, ensuring continuous development without disruption.

2.8.4. Portability

Portability ensures that the system can run smoothly on different platforms and environments. The choice of technologies—Spring Boot for the backend and React JS for the frontend—provides cross-platform compatibility. The system is capable of running on major operating systems such as Windows, macOS, and Linux. Moreover, the entire application can be containerized using Docker, allowing seamless deployment across various cloud service providers like AWS, Google Cloud Platform, or Microsoft Azure without additional configuration changes.

2.8.5. Scalability

Scalability is the system's ability to handle growing user demands and workload without degrading performance. In the *Quick Ambulance* system, each microservice can be scaled independently based on demand. For example, if login requests increase significantly, additional instances of the login service can be deployed without affecting other components. The use of Eureka along with load balancing ensures that incoming requests are evenly distributed across available services. The WebSocket server supports many concurrent connections, enabling smooth real-time communication between users and drivers even under heavy traffic.

2.8.6. Integrity

Integrity guarantees that data remains accurate, consistent, and secure throughout its lifecycle. The application uses role-based access control to ensure that only authorized users can perform certain operations, such as assigning ambulances or accessing sensitive data. The backend enforces data integrity through MySQL constraints and relationships—for example, logs are always linked to valid ambulance and driver entries. Additionally, all user interactions are secured using JWT tokens, which prevent unauthorized data access or manipulation during API requests.

2.8.7. Usability

Usability focuses on how easy it is for users to interact with the system. The React JS-based frontend provides a clean, intuitive interface for users, drivers, and administrators. Features like login, registration, ambulance booking, and real-time tracking are presented in a simple and guided manner. Visual feedback and real-time alerts—such as “Login Failed” or “Ambulance Booked Successfully”—help users understand the system's response instantly, creating a smooth and frustration-free experience during high-stress situations.

2.8.8. Performance

Performance measures how quickly the system responds to inputs and processes user requests. The system uses WebSocket to update live ambulance locations every three seconds, delivering real-time data with minimal latency. Backend operations such as fetching nearby ambulances use optimized queries with proper indexing, ensuring rapid data retrieval. Time-consuming tasks like sending alerts to drivers are handled asynchronously, so users experience no UI lag. JWT-based authentication also speeds up repeated requests by eliminating the need to query the database for every session validation, improving overall system responsiveness.

3. SYSTEM REQUIREMENTS

3.1. HARDWARE REQUIREMENTS

The hardware requirement specifies each interface of the software elements and the hardware elements of the system. These hardware requirements include configuration characteristics.

- Operating system: windows, Linux
- Processor: minimum core i3
- Ram: minimum 4 GB
- Hard disk: minimum 216 GB

3.2. SOFTWARE REQUIREMENTS

The software requirements specify the use of all required software products like data management system. The required software product specifies the numbers and version. Each interface specifies the purpose of the interfacing software as related to this software product.

- Backend Framework: Spring boot 3.4.3, Java (JDK 17).
- Web Technologies: React JS
- Database: MySQL
- IDEs: Visual codes, Eclipse

3.3. INTRODUCTION TO SOFTWARE ENVIRONMENT

The **software environment** refers to the overall set of tools, technologies, frameworks, platforms, and runtime environments used to develop, run, and maintain the application. For a distributed, real-time system like **Quick Ambulance**, the software environment plays a crucial role in enabling smooth development, secure communication, efficient processing, and scalable deployment.

This project leverages a **microservices-based architecture** using modern technologies that ensure modularity, flexibility, and maintainability. The software environment spans across backend development, frontend user interface, inter-service communication, security mechanisms, and data persistence.

The following technologies were chosen for their robustness, community support, and suitability for real-time emergency response systems:

3.3.1 Backend Technologies

- **Java JDK 17**: Modern version of Java used to implement the business logic and microservices. Offers long-term support and modern syntax enhancements.
- **Spring Boot 3.4.3**: Primary framework used to build RESTful services, microservices, and secure endpoints. It provides auto-configuration, embedded servers, and seamless integration with other Spring modules.
- **Spring Data JPA**: Handles database operations in a clean and efficient way using object-relational mapping (ORM). Reduces boilerplate code and improves maintainability.
- **Spring Security + JWT**: Provides robust authentication and authorization mechanisms using JSON Web Tokens. Ensures that only valid users can access protected resources.
- **WebSocket with SockJS & STOMP**: Enables real-time bi-directional communication between users and drivers for live location tracking and alerts.
- **Eureka Server & Clients (Service Discovery)**: Manages registration and discovery of microservices, allowing dynamic scaling and inter-service communication.
- **Feign Client + Load Balancer**: Simplifies inter-service calls and automatically balances requests across instances for fault tolerance and scalability.

3.3.2 Frontend Technologies

- **React JS**: A popular JavaScript library used for building a responsive and dynamic single-page application (SPA). It enhances user experience with real-time updates and smooth navigation.
- **Axios**: Used in the frontend to make HTTP requests to backend services, including sending tokens and receiving responses.
- **Leaflet.js or Google Maps API** (if used): For displaying real-time maps, ambulance markers, and routes between users and drivers.

3.3.3 Database Layer

- **MySQL**: A reliable and scalable relational database used to store all persistent data such as users, drivers, ambulances, bookings, and live locations. Foreign key constraints ensure data integrity.

3.3.4 Infrastructure & Middleware

- **Spring Cloud Gateway**: Serves as the API Gateway to route incoming requests to appropriate microservices. Also handles token validation and cross-origin (CORS) issues.
- **Maven**: Build and dependency management tool that ensures consistent builds and simplifies project packaging.
- **Localhost / Docker (optional)**: The system is designed to run on localhost for development. It can be containerized for deployment on cloud or production servers.

3.3.5 Features of Operating System Interface

The Operating System (OS) interface is a vital layer that bridges the gap between the application software and the underlying hardware. It plays a critical role in providing the necessary services, control mechanisms, and runtime environment for applications like *Quick Ambulance* to operate securely, efficiently, and reliably. From process scheduling to memory management and hardware interaction, the OS ensures that the entire system functions in harmony.

One of the primary features is User Interface (UI) support, which provides either a Command Line Interface (CLI) or Graphical User Interface (GUI). This is essential for both users and developers. Developers benefit from GUI-based Integrated Development Environments (IDEs) like Eclipse or IntelliJ for writing, compiling, and deploying code, while system administrators may rely on CLI tools for system-level configurations. The OS also handles Program Execution, by loading applications into memory, creating processes, and managing their life cycle. This ensures smooth execution of backend services such as Spring Boot microservices that support login, location tracking, and booking functionalities.

The OS also facilitates File System Management, ensuring secure and organized storage and retrieval of essential files, such as configuration files, logs, and database backups. Through Device Management, the OS interacts with hardware components, including GPS modules, network adapters, and I/O peripherals, which is especially critical for real-time features like live ambulance tracking. Complementing this is Memory Management, which dynamically allocates memory to active microservices, WebSocket connections, and user sessions, ensuring efficient use of RAM and system responsiveness under load.

Security is another core responsibility, handled by Access Control Mechanisms that manage user authentication and protect sensitive data from unauthorized access. Networking Support provided by the OS allows seamless communication between distributed microservices

through protocols like HTTP, TCP/IP, and WebSockets—vital for real-time data exchange in your system. In addition, Multitasking and Scheduling enable multiple components—like the React frontend, backend services, and databases—to run concurrently while ensuring efficient CPU time sharing. The OS also provides Error Detection and Handling, which identifies and logs software or hardware malfunctions to prevent system crashes or data loss. Finally, Resource Allocation ensures optimal distribution of CPU, memory, disk I/O, and network bandwidth across all active processes, guaranteeing consistent system performance even under peak usage.

3.4 APPLICATION

Real-World Applications of the Quick Ambulance System

The **Quick Ambulance system** is a robust, real-time medical transportation solution designed to meet the urgent needs of patients, healthcare providers, and emergency responders. With its modern architecture, live GPS tracking, and seamless communication, the system offers significant value across various real-world scenarios.

Emergency Medical Services (EMS) are the core beneficiaries of this system. Hospitals and healthcare organizations can drastically improve their response times by leveraging live tracking and automatic nearest-driver selection. In critical situations, where every second matters, the Quick Ambulance system ensures timely ambulance dispatch and arrival, thereby enhancing the chances of patient survival and reducing operational chaos during emergencies.

In the domain of **Public Health Management**, local governments and health authorities can use this platform to coordinate city-wide emergency medical transportation services. Especially during health crises like pandemics or natural disasters, the system aids in resource optimization by allowing officials to track, allocate, and dispatch ambulances efficiently. Centralized dashboards can offer insights into ambulance availability and geographic coverage in real time.

For **Private Ambulance Operators**, Quick Ambulance presents an opportunity to transition from manual booking methods to digital platforms. These operators can offer seamless app-based or web-based ambulance bookings to their customers. Not only does this improve service delivery, but it also brings transparency and accountability to private ambulance operations, streamlining coordination between drivers, dispatchers, and patients.

The system aligns well with **Smart City Projects**, as it can be integrated with intelligent traffic systems and IoT-based infrastructure. Live location data from ambulances can be fed

into urban traffic control centers to dynamically alter traffic signals and create green corridors for emergency vehicles. This reduces response time and ensures ambulances reach destinations faster, even during peak traffic hours.

From a **User Convenience** perspective, the system empowers patients and their families to independently request the nearest available ambulance through a simple mobile or web interface. Once booked, users receive real-time updates on the assigned driver's name, contact number, and current location on a map, helping reduce anxiety and uncertainty during emergencies.

Driver and Fleet Monitoring is another crucial application. Administrators can monitor every vehicle's real-time status, including whether it's active, inactive, in transit, or idle. This visibility helps ensure driver accountability, prevents vehicle misuse, and allows for better planning of regular maintenance or repairs to keep the fleet in optimal condition.

Medical Institutions and Campuses, such as large hospitals, universities, and healthcare research centers, can use this system internally to facilitate the dispatch of ambulances within campus premises. It helps ensure that emergency services are always a call or click away, whether within hostels, academic buildings, or hospital departments.

Lastly, the system’s powerful **Data Logging and Analytics** capabilities allow storage of user actions, driver logs, booking history, and tracking records. This data can be used for generating analytical reports, evaluating system performance, identifying peak hours or bottlenecks, and improving service strategies. Over time, these insights can be used to predict demand patterns, optimize resource allocation, and even assist in policy-making.

4. DESIGN

4.1 DESIGN INTRODUCTION

The **Quick Ambulance** system is designed using a modular **microservices architecture** that separates concerns and enhances scalability, maintainability, and flexibility. Each microservice is responsible for a specific functionality—such as authentication, live location tracking, or database operations—and all are registered through the Eureka Server for dynamic service discovery. An API Gateway acts as the single-entry point for all frontend requests and ensures secure routing to the respective services after validating JWT tokens and handling CORS policies.

The backend is built using **Spring Boot 3.4.3** with technologies like **Spring Security**, **JWT**, **Spring Data JPA**, **WebSocket with SockJS**, and **Feign Client**. The system comprises four major services: the **Login-Logout Service** for authentication and role-based access control; the **Database Service** for handling CRUD operations on seven key entities; the **Location Service** for real-time location updates and booking workflows; and the **API Gateway** for centralized routing and security. Each service communicates through REST and Feign Clients, ensuring decoupled interaction and ease of testing.

On the frontend, the application is built using **React JS** to provide a responsive and interactive interface for users, drivers, and administrators. The UI dynamically adapts based on user roles after login and integrates real-time WebSocket communication for live tracking and booking alerts. The system's database design includes normalized tables such as Users, Drivers, Ambulances, DriverLogs, and LiveLocation, which are efficiently queried and updated by the backend services. Together, the frontend and backend create a seamless, scalable solution for emergency ambulance booking and tracking.

4.2. UML DIAGRAM

Unified Modelling Language (UML) diagrams are essential for visualizing the structure, behavior, and interactions within the Quick Ambulance System. They help stakeholders, developers, and testers understand how components interact and function across the application. Below are key UML diagrams used for the design and planning of this project:

Unified Modelling Language (UML) diagrams are essential tools in software engineering that help visualize, specify, and document the architecture and design of complex systems. In the

Quick Ambulance System, UML diagrams play a key role in representing both the static structure and dynamic behaviour of the application. They provide a clear understanding of system interactions, user roles, and component relationships, enabling smooth development and collaboration among team members.

The UML diagrams for the **Quick Ambulance** system provide a comprehensive visual representation of both the structural and behavioral aspects of the application. The **Use Case Diagram** outlines the key roles—User, Driver, and Admin—and their interactions with the system, such as booking an ambulance, tracking it in real-time, and managing resources. The **Class Diagram** details the internal structure, showcasing entities like User, Driver, Ambulance, Booking, and their attributes, operations, and relationships. It helps in understanding how the system is organized at the code level. The **Sequence Diagram** captures the time-based interaction between these entities, particularly during events like booking and tracking, illustrating the request-response flow between the user, backend services, and the driver. The **Activity Diagram** shows the step-by-step process flow from user login to ambulance booking and live tracking, highlighting decision points and parallel actions. The **Data Flow Diagram (DFD)** represents how data moves through the system—from users and drivers to backend services and databases—clearly showing input, processing, and output flows. Lastly, the **Entity Relationship Diagram (ERD)** visualizes the logical structure of the database, establishing the relationships among entities like Users, Bookings, Ambulances, and Logs. Together, these UML diagrams serve as a blueprint that helps developers, stakeholders, and testers understand, communicate, and implement the Quick Ambulance system effectively.

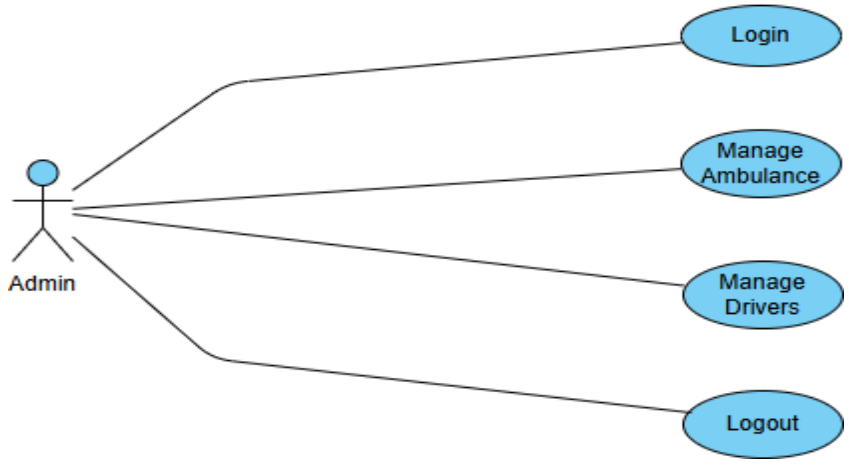

Figure:1 Admin UML Diagram

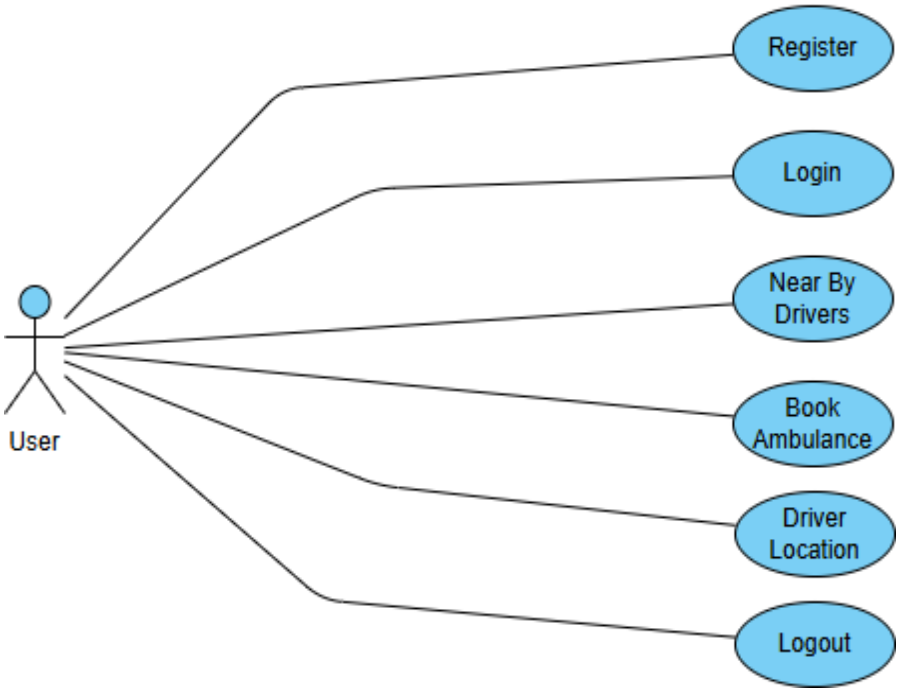

Figure:2 User UML Diagram

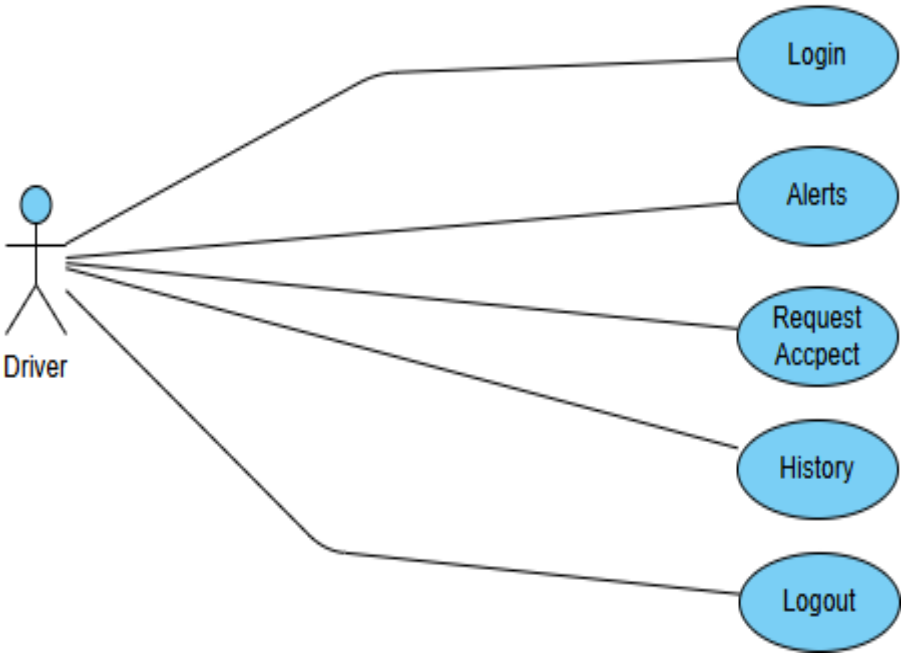

Figure:3 Driver UML Diagram

4.2.1. Use Case Diagram

The Use Case Diagram outlines the main functionalities of the Quick Ambulance System from the user's perspective. It identifies the different types of users (called actors) such as Admin, Driver, and User (Patient), and their respective interactions with the system, like booking an ambulance, accepting a ride, assigning vehicles, and tracking locations. This diagram helps developers and stakeholders visualize the system's scope and ensures all core requirements are addressed through well-defined user actions.

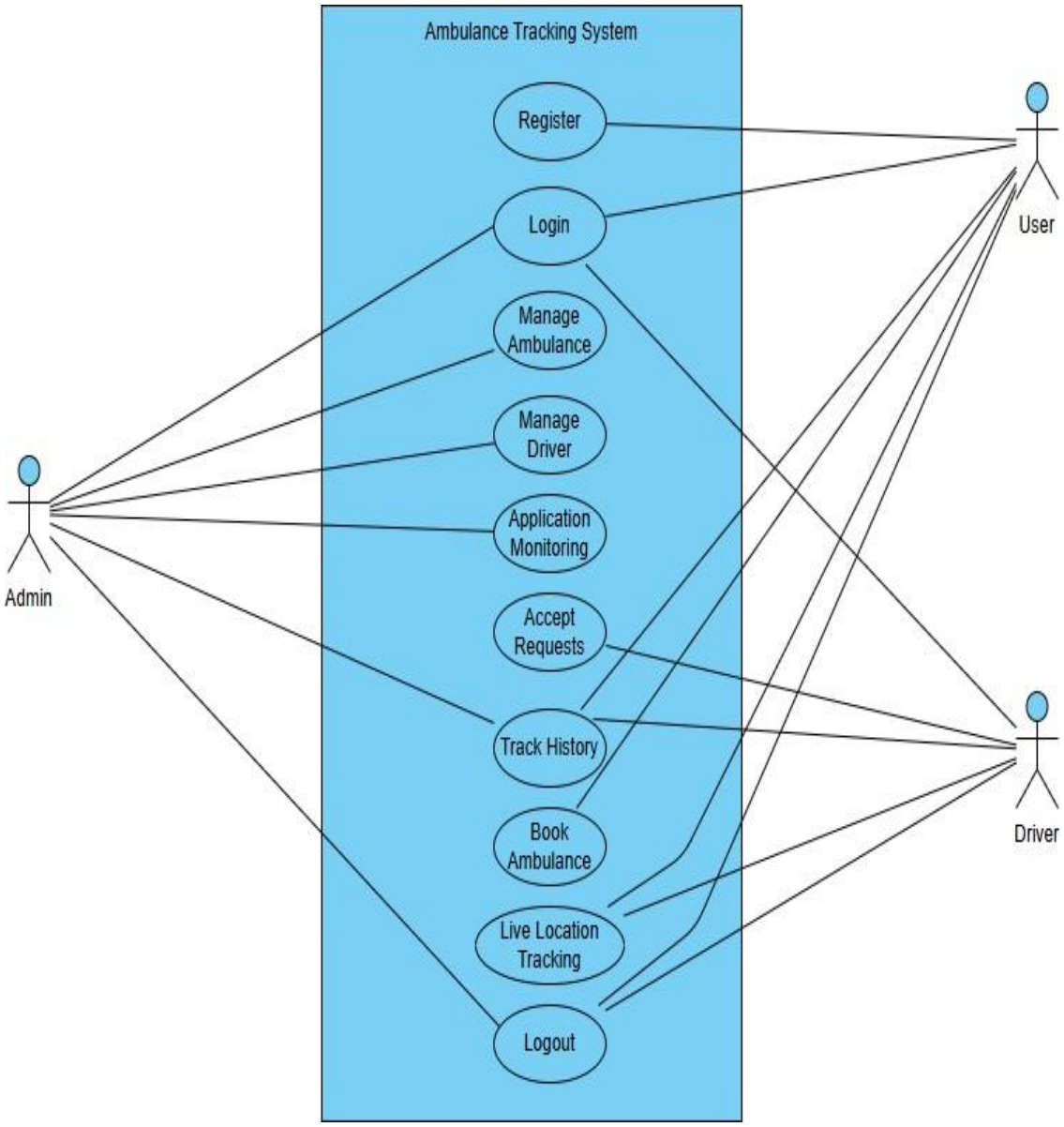

Figure:4 Use Case Diagram

4.2.2. Class Diagram

The Class Diagram presents the static structure of the system by detailing the classes (like User, Driver, Ambulance, Booking), their attributes, methods, and the relationships between them. It shows how data is organized and how different parts of the application interact with each other logically. This helps in understanding how user and driver data are stored, how bookings are created and assigned, and how location data is managed within the system, forming the foundation for database design and object-oriented development.

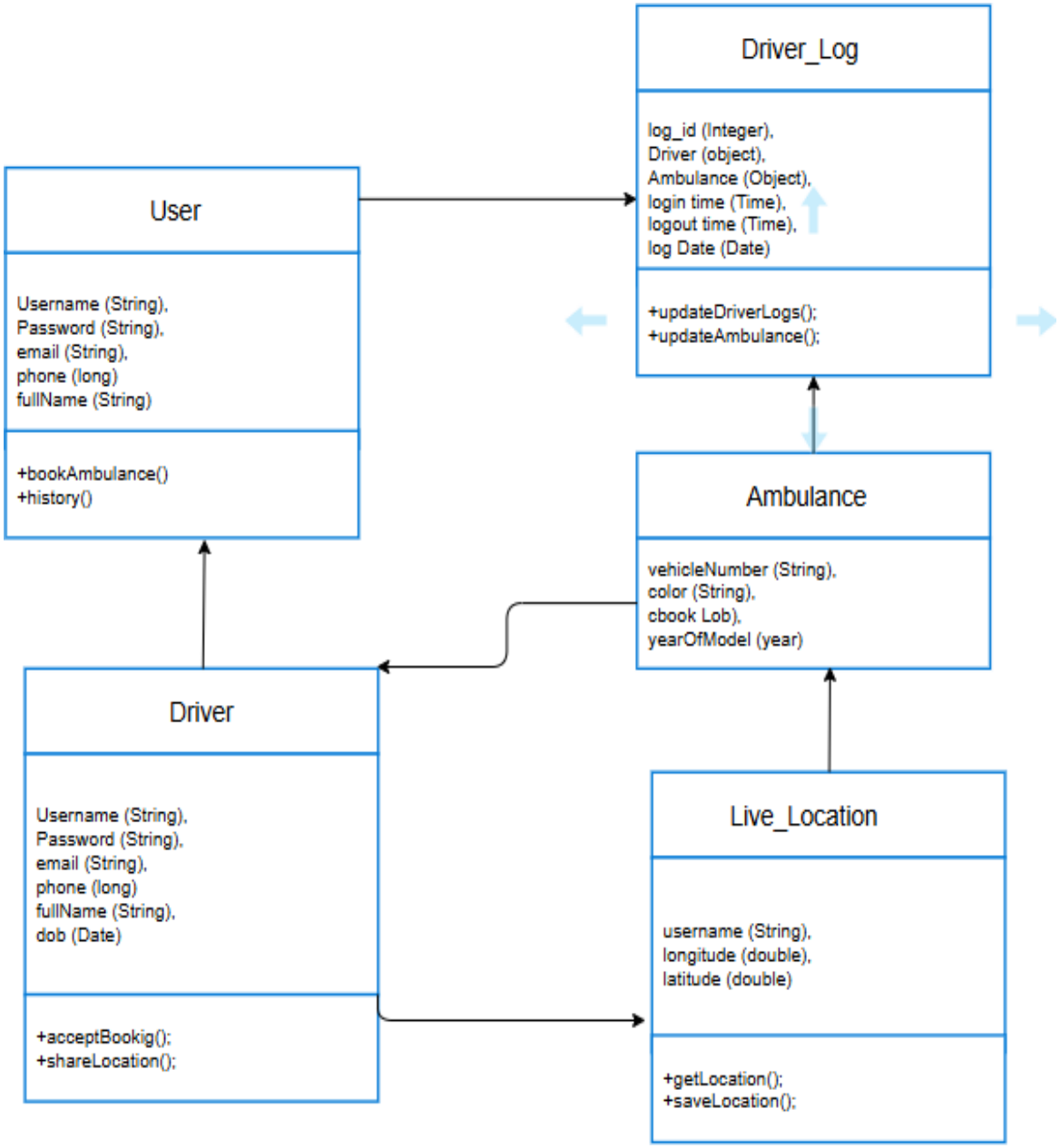

Figure:5 Class Diagram

4.2.3. Sequence Diagram

The Sequence Diagram shows the step-by-step flow of interactions between different system components during specific scenarios, such as the ambulance booking process. It illustrates how messages are exchanged between objects like the frontend, backend microservices, and WebSocket layer, and in what order. This is especially useful in the Quick Ambulance System to map out real-time operations—such as when a user books an ambulance, the alert is sent to drivers, and the first responder is confirmed—ensuring that real-time workflows are accurately implemented.

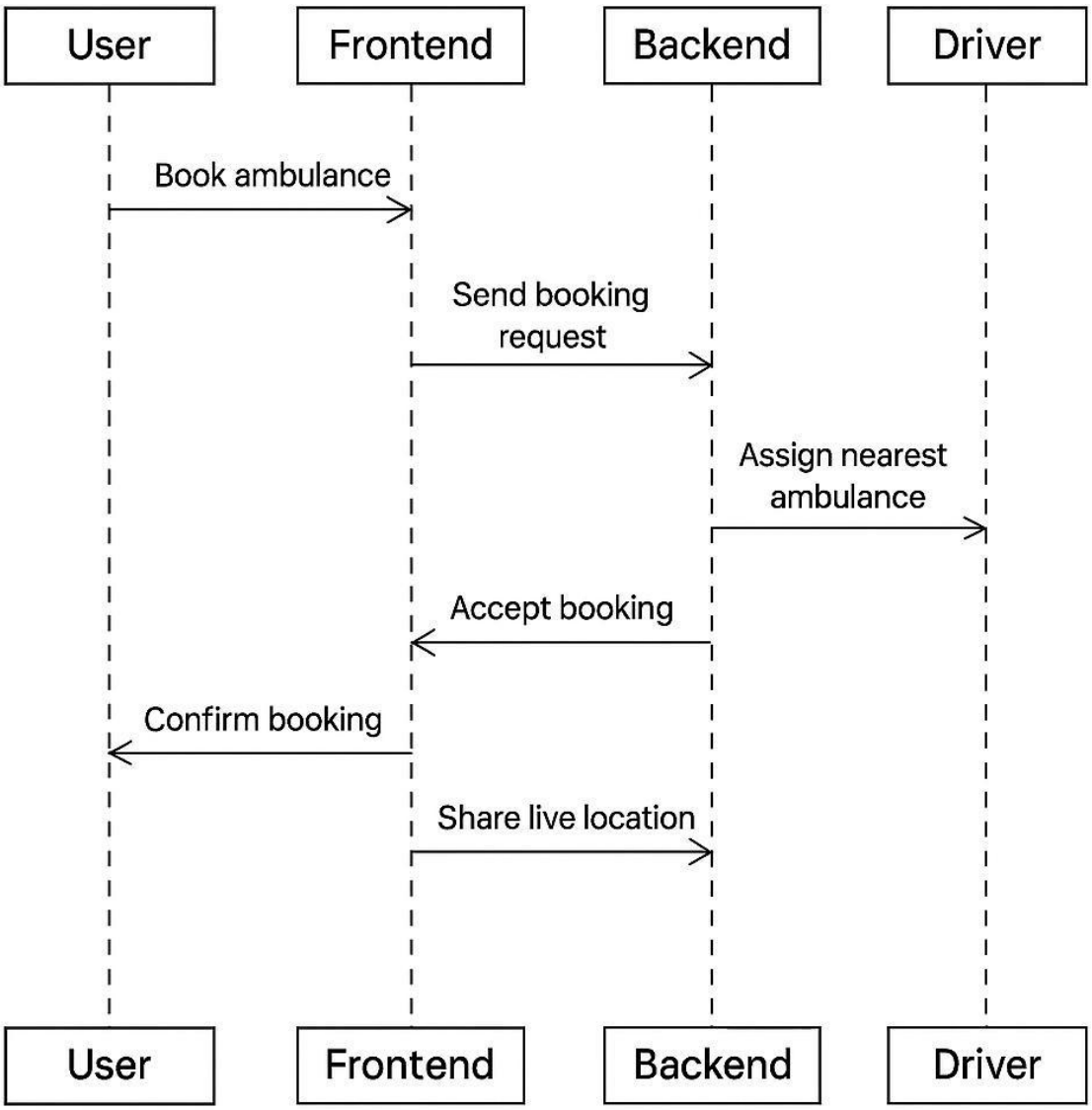

Figure:6 Sequence Diagram

4.2.4. Data Flow Diagram

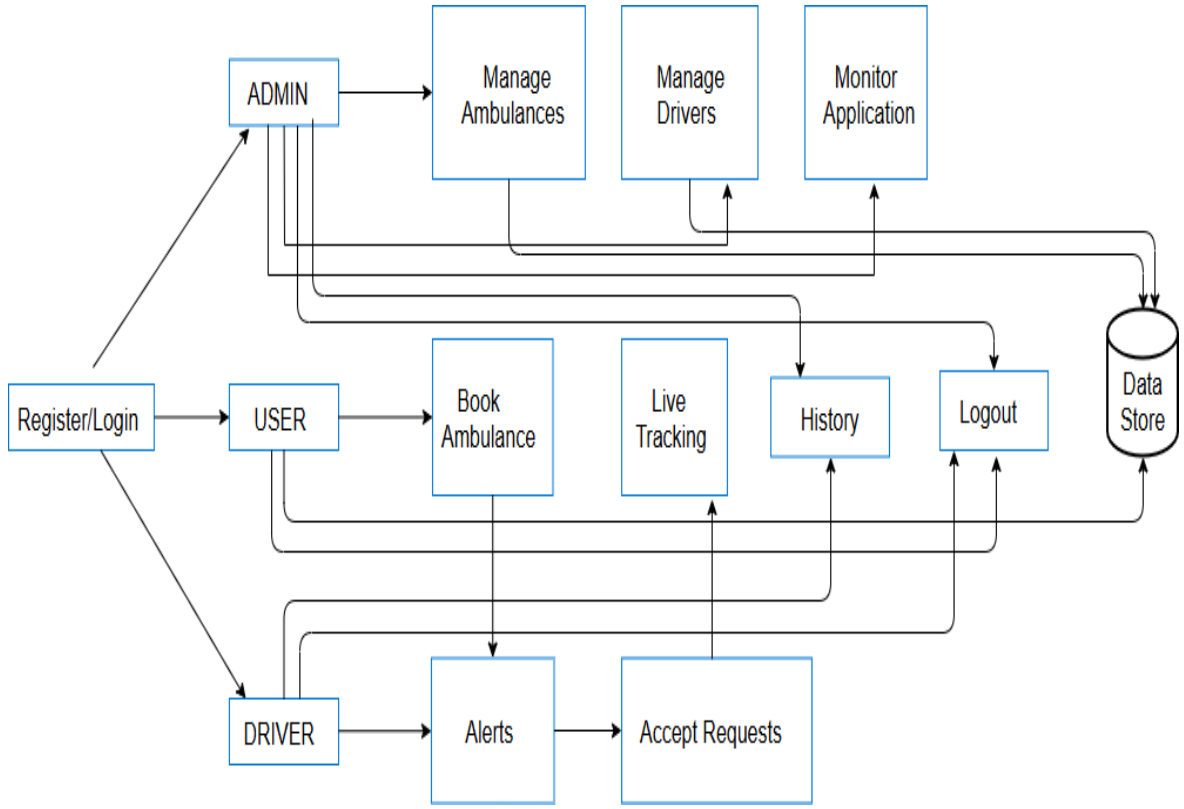

Figure:7 Data Flow Diagram

4.3. DATABASE DESIGN:

The database design of the Quick Ambulance System plays a crucial role in ensuring data integrity, fast access, and seamless real-time interactions between users, drivers, admins, and system components. It is structured to support modular microservices that interact with the database independently, while maintaining consistency through carefully planned relationships.

The database follows a **relational model**, with clearly defined tables such as Users, Drivers, Ambulances, Bookings, Live_Locations, Login_Logs, and Notifications. Each table includes a **primary key** for unique identification and **foreign keys** to maintain relationships. This relational structure ensures that each user, driver, ambulance, and booking record is interconnected appropriately.

The Users table stores information about all registered users, including their name, email, mobile number, address, password (encrypted), and role (e.g., USER or ADMIN). This table is essential for login authentication and booking history. It supports scalability, allowing

thousands of users to register and manage their ambulance requests.

The Drivers table contains personal and contact details of ambulance drivers, along with their availability status, last known location, and a reference to the assigned ambulance (foreign key to Ambulances). This table enables the system to identify nearby drivers during the booking process and update their availability in real-time.

The Ambulances table includes vehicle details such as ambulance number, type (basic/life support), capacity, and current status (available, on-duty, or maintenance). Each ambulance is mapped to a specific driver, helping the system track which vehicle is currently active and who is operating it.

The Bookings table is central to the system, capturing each ambulance request made by a user. It includes booking ID, user ID, driver ID, ambulance ID, pickup and destination locations (latitude and longitude), booking time, status (pending, accepted, completed, or canceled), and timestamps. This table helps in logging the complete booking lifecycle and tracking performance metrics.

To support **real-time location tracking**, a Live_Locations table records frequent updates of the driver's location during an active booking. Each entry includes the booking ID, driver ID, timestamp, and coordinates. This table enables the system to stream live GPS data to the user interface every few seconds via WebSocket, ensuring accurate tracking.

The Login_Logs table keeps track of login and logout activities of all users and drivers. Each log entry stores username, role, login time, logout time, IP address, and token ID (if using JWT). This table is crucial for security auditing, session tracking, and understanding system usage patterns.

For communication, the Notifications table stores messages sent to users and drivers, including alert types, message content, timestamps, and read status. It helps deliver booking confirmation messages, cancellation alerts, and driver assignment notifications efficiently using WebSocket or fallback SMS services.

Finally, the database schema is normalized up to the **third normal form (3NF)** to eliminate redundancy and improve performance. Indexing is applied on commonly searched columns (like email, mobile, and booking status), and all sensitive information such as passwords is encrypted using secure hashing algorithms like bcrypt.

This well-structured and optimized database design ensures that the Quick Ambulance System can handle thousands of concurrent users, manage real-time operations, and scale as needed. It supports future extensions such as analytics, hospital integrations, and emergency response coordination without compromising data consistency or speed.

MYSQL tables:

Ambulance Table:

VEHICLE_NUMBER	CBOOK	COLOR	COMPANY	STATUS	YEAR_OF_MODEL
AP09IU7654	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	RED	Hyndai	FALSE	2021
AP32TD4321	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2023
AP41TF3456	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2019
TS45TG4356	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2022
AP39TG7654	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2015
AP35TW3216	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2014
TS41FR3459	4749463839610100010080000000000ffffff2c0000000001000100000 2014c003b	WHITE	Tata	FALSE	2012

Figure:8 Ambulance Table

User Table

USERNAME	EMAIL	FULL_NAME	PHONE
siva@gmail.com	siva@gmail.com	Siva K	9876543210
sai@gmail.com	sai@gmail.com	Sai S	7896541230
hitesh@gmail.com	hitesh@gmail.com	Hitesh A	8974563210
vidya@gmail.com	vidya@gmail.com	Vidya A	8596741230
prasad@gmail.com	prasad@gmail.com	Prasad P	8695741523

Figure:9 User Table

Driver_Logs

LOGID	LOGIN	LOGIN_DATE	LOGOUT	VEHICLE_NUMBER	DRIVER_NAME
1	11:01:28.329	2025-07-20	11:05:01.235	TS41FR3459	dilliswar@gmail.com
2	11:06:36.779	2025-07-20	11:08:39.993	AP32TD4321	basha@gmail.com
3	11:09:04.976	2025-07-20	11:10:10.918	AP41TF3456	chinna@gmail.com
4	11:12:26.375	2025-07-20	11:13:33.421	AP35TW3216	laxman@gmail.com

Figure:10 Driver_Logs Table

Roles Table

USERNAME	PASSWORD	ROLE
siva@gmail.com	$2a$10$FDoWLo3uGshJQmcrQF4nr.2ndhnrq8gJhInFI1HqyUBTniYkK.eYW	USER
sai@gmail.com	$2a$10$kaVizGaWlucuIVb7UfgeyuHipjEosmOkMGmJS7o/zCChVdxSbaZ1.	USER
hitesh@gmail.com	$2a$10$N9BgJ/A/H.imOEr99obL8OyhcTY/jATfS3xPnTmohit3grKqSckBC	USER
vidya@gmail.com	$2a$10$aYCcLAweXFxOmkB8EpWBhOzyQy3P/OYjJ1HFRIINrfNgzXfsPXVsG	USER
prasad@gmail.com	$2a$10$zhsUq/D9nzcgkPOeVabOfeiMY9GNRCjPhpfgLnT47efh0LzHdUHZa	USER
eswar@gmail.com	$2a$10$jaYA6.0OnzGz2kx8QIEUqOE6X40wiBpkno7zsaLnOZYFrqzPcFQNi	USER
dilliswar@gmail.com	$2a$10$qGFn9ZxCWLeT3P4xVnBstO4g7c5RsGfQhB2vrMJ2YA80VVxWcomDC	DRIVER
basha@gmail.com	$2a$10$H4b9PkVTwPQ2NaJerKXMH.N4HhfOwhdpIpEYrx0JeSpSABnVIkSby	DRIVER
chinna@gmail.com	$2a$10$TC9Rf7fTcXvUbF0Z9Qif1e3JjLPHGVfcteGWZYe6hkDKWEjoQzPem	DRIVER
laxman@gmail.com	$2a$10$p03XVbsbdtElj/YI98Fb9OGn/qdpgm9117IyIKSCJOhW/DGufjRda	DRIVER

Figure:11 Roles Table

Driver Table

USERNAME	DOB	DRIVER_NAME	EMAIL	LICENSE	PHONE	STATUS
basha@gmail.com	1998-12-10	Bashs B	basha@gmail.com	47494638396101000100800000000000ffffff2c000000000100010000002014c003b	8523697410	FALSE
chinna@gmail.com	1992-11-30	Chinna P	chinna@gmail.com	47494638396101000100800000000000ffffff2c000000000100010000002014c003b	7896541230	FALSE
krishna@gmail.com	1989-01-01	Krishna P	krishna@gmail.com	47494638396101000100800000000000ffffff2c000000000100010000002014c003b	8523697014	FALSE
ganesh@gmail.com	1932-05-28	Ganesh L	ganesh@gmail.com	47494638396101000100800000000000ffffff2c000000000100010000002014c003b	8745693210	FALSE
mani@gmail.com	2003-04-28	Mani A	mani@gmail.com	47494638396101000100800000000000ffffff2c000000000100010000002014c003b	7854123690	FALSE

Figure:12 Driver Table

Driver Live_Location Table

LATITUDE	LONGITUDE	USERNAME
16.56669926144508	80.54724915320668	basha@gmail.com

Figure:13 Live_Location Table

Track_Details Table

BOOK_DATE	BOOK_TIME	TRACK_ID	DRIVER	PICKUP	USER_NAME	VEHICLE_NUMBER
2025-07-20	12:28:01.14	1	basha@gmail.com	null null Ibrahimpatnam Ibrahimpatnam	venkatesh	AP45TG4321

Figure:14 Track Details

5. IMPLEMENTATION

5.1. IMPLEMENTATION

The implementation phase marks the transition from design to execution, where the planned architecture, database models, and system workflows are translated into working code. For the Quick Ambulance System, implementation was carried out using a **Microservices Architecture**, allowing independent deployment and management of different modules such as user authentication, ambulance booking, real-time tracking, and admin management.

The **backend** was developed using **Spring Boot (version 3.4.4)**, where each microservice was designed to handle a specific functionality. The login-logout service manages user and driver authentication using **Spring Security** with session-based or JWT-based validation, depending on the environment. The booking service handles user requests, finds nearby drivers, and assigns ambulances accordingly. The **Feign Client** and **Eureka Server** were used to enable seamless communication between services, and **Spring Cloud Gateway** acts as the API gateway that routes frontend requests to appropriate services securely.

For real-time communication, especially during the booking and tracking process, **WebSocket** with **STOMP over SockJS** was implemented. This allowed continuous two-way communication between the frontend and backend for features such as live location tracking and driver-user alert messages. JavaScript on the client side sends and receives real-time location updates every few seconds, which are processed by the Spring Boot WebSocket controller and stored in the database for reference.

The **frontend** was developed using **ReactJS**, providing a dynamic, single-page application (SPA) experience. React components were developed for each role: admin, user, and driver. Axios was used for API calls, while @stomp/stompjs was used for establishing WebSocket connections. The user interface includes login pages, booking forms, real-time maps using Google Maps API, and dashboards showing active requests. Each role has controlled access, enforced by React route guards and backend role-based authentication.

The **database layer** was implemented using **MySQL**, with JPA/Hibernate handling ORM (Object Relational Mapping) between Java classes and database tables. All critical operations

such as booking creation, user registration, driver assignment, and live location logs were managed using well-optimized repository and service layers. Complex SQL operations like joins and filters were abstracted behind Spring Data JPA queries or custom native queries when required.

Testing was done at multiple levels — unit tests using JUnit and Mockito, integration tests for services, and real-time system tests for WebSocket functionality. During development, the application was continuously tested using tools like Postman, browser WebSocket clients, and frontend simulations to ensure correct data flow and performance. Errors and logs were captured using Spring Boot's built-in logging framework and displayed to developers for debugging.

In deployment scenarios, Docker containers or traditional WAR packaging can be used depending on the hosting environment. The microservices can be hosted independently or orchestrated using Docker Compose or Kubernetes for scalability. Security was a major concern, and HTTPS was configured along with role validation, SQL injection protection, and proper session/token handling to secure the system.

5.2 IMPLEMENTATION PROCEDURES

The implementation of the **Quick Ambulance System** followed a modular and phased approach using microservices. The project began with **environment setup**, where Java (JDK 17+), Spring Boot, MySQL, ReactJS, and supporting tools like Postman, Eclipse/IntelliJ, and Visual Studio Code were configured. A separate codebase was maintained for each microservice, and GitHub was used for version control and team collaboration.

The first core module implemented was the **User and Driver Authentication Service**. It was built using Spring Security in the login-logout microservice. The backend validates credentials using either session-based or JWT-based authentication (as per project need). Login details are fetched via Feign clients from the database microservice, and successful logins are recorded in the logs table. For security, roles (USER, DRIVER, ADMIN) were assigned and verified for protected routes.

Once authentication was ready, the **Ambulance Booking Service** was developed. This service accepts a user's booking request and finds nearby available ambulances within a 20 km radius. The system uses latitude and longitude values to calculate distance using the Haversine formula. Upon matching a driver, it stores the booking in the database and sends real-time alerts to drivers using WebSocket.

Next, the **Live Location Tracking Module** was implemented using WebSocket and STOMP protocol. When a user books an ambulance, a continuous WebSocket connection is established. The frontend JavaScript sends the driver's location every 3 seconds after login. The backend receives these updates and broadcasts them to the user. This bi-directional flow enables real-time tracking with minimal latency.

The **Frontend Development** was handled using ReactJS. Separate dashboards were created for users, drivers, and admins. Components included login forms, booking panels, driver tracking maps, and ambulance status boards. Google Maps API was integrated to show the live position of drivers and users. Axios was used for secure API calls, while @stomp/stompjs managed WebSocket connections.

The **Database Layer** was implemented using MySQL. Entities like User, Driver, Ambulance, Booking, and DriverLogs were created using JPA annotations. Primary keys were auto-generated, and foreign key relationships (e.g., between Driver and Ambulance) were maintained to ensure data consistency. Repositories were created for all major operations like finding ambulances, saving logs, and updating booking statuses.

In the **Admin Module**, features like driver approval, booking status tracking, and location logs were implemented. The admin could accept or reject driver registrations, monitor active bookings, and track ambulance fleet availability. Role-based access control was enforced so that only admin accounts could access these endpoints.

Finally, **Testing and Debugging** were performed using Postman for APIs, browser DevTools for WebSocket inspection, and console logging. Unit tests were written using JUnit, and integration tests ensured the services worked correctly when connected. Logs were monitored using Spring Boot's logging framework for detecting service failures or performance issues.

The completed system was packaged for deployment. Optionally, **Docker** was used to containerize microservices. Each service could be independently started and scaled. The final setup included Eureka Server (for service discovery), API Gateway (for routing and security), microservices (login, booking, tracking), and a React frontend served either via local server or a cloud-based service.

The implementation phase is where theoretical design translates into working software. This process for the Quick Ambulance System involved setting up a distributed, real-time ambulance booking platform using a microservices architecture. Various components were implemented step by step—ranging from user authentication, real-time booking, and live location tracking, to data management and admin control. Spring Boot was used for backend services, ReactJS for the frontend, WebSocket for real-time communication, and MySQL for persistent data storage. Each module and service was carefully developed, tested, and integrated to ensure the system met functional and non-functional requirements.

5.2.1. Environment Setup

The project environment was set up using modern and scalable tools. For the backend, the Spring Tool Suite (STS) was used to create multiple Spring Boot projects, each representing a microservice. Dependencies like Spring Web, Spring Security, Spring Data JPA, and WebSocket were added via Maven. For the frontend, ReactJS was initialized using create-react-app, and additional libraries like axios, react-router, and @stomp/stompjs were installed. MySQL was installed as the database engine, and DBeaver was used for managing DB schema and queries. Git was used for version control, and each module was developed on separate branches to avoid conflicts during integration.

5.2.2. Authentication Module

Security was implemented using Spring Security for backend and token/session-based login for the frontend. All users—drivers, admins, and normal users—register through dedicated endpoints, and their credentials are stored securely using BCrypt password hashing. The backend validates login attempts and, upon success, returns role-specific responses. Admins can log in to manage drivers, while drivers can only log in after approval. Spring Security also protects routes so that only authorized users can access certain endpoints. Post-login and

logout success URLs are redirected for DB logging, helping maintain an audit trail of all user activities.

5.2.3. User and Driver Management

The system provides separate user interfaces and endpoints for registering as a user or driver. User data such as name, phone, address, and emergency notes are collected and stored. Drivers must submit their license and ambulance details for admin approval. The admin reviews the driver data and sets their status as ACTIVE or INACTIVE. Only active drivers are eligible to receive booking alerts. Drivers also have a dashboard where they can update their availability status, location, and view booking history. All user and driver data is stored in respective tables (users, drivers), with unique identifiers for proper mapping.

5.2.4. Ambulance Booking Flow

The booking flow was implemented as the system's core module. When a user initiates a booking, their geolocation is captured via the browser and sent to the backend. The backend then calculates which ambulances are within a 20 km radius by applying the Haversine formula to GPS coordinates. These ambulances (drivers) receive booking alerts via WebSocket. The first driver to accept the request is assigned, and all others' alerts are automatically cancelled. Once the booking is confirmed, both the driver and user receive each other's details, and a WebSocket session begins for continuous location updates.

5.2.5. Real-Time Location Tracking

WebSocket-based real-time location tracking was implemented using STOMP over SockJS. The driver app continuously sends their location to the backend every 3 seconds, which then forwards it to the corresponding user's frontend in real time. The user's screen updates dynamically, showing the moving ambulance on a map (Google Maps or Leaflet). Similarly, the user's location is shared with the driver. The STOMP protocol manages message routing through publish-subscribe topics such as /topic/user-location/{id} and /topic/driver-location/{id}, ensuring secure and efficient communication. Backend services store historical locations for analytics and auditing.

5.2.6. Frontend Development (ReactJS)

The frontend was developed using ReactJS and styled with CSS and Bootstrap. Different dashboards were created for admins, drivers, and users. The admin panel provides driver management, booking logs, and system monitoring features. The driver dashboard includes status toggles, location tracking, and booking acceptance UI. The user dashboard includes booking forms, live tracking views, and booking status updates. React Router enables role-based navigation, while Axios is used to connect with backend APIs. Real-time map views are rendered using the Google Maps API or Leaflet.js, with real-time location updates from WebSocket listeners.

5.2.7. Database Design & Data Handling

The database schema was carefully structured to ensure efficient data retrieval and integrity. Major tables include users, drivers, ambulances, bookings, locations, and activity_logs. Primary keys were auto-incremented IDs, and foreign key relationships were established—for example, the bookings table contains user_id and driver_id as foreign keys. This allows for fast joins and reporting. The locations table stores timestamped latitude and longitude points for real-time and historical tracking. Indexing was used on frequently queried columns like status, email, and booking_time to improve query performance.

5.2.8. Admin Control Panel

The admin module allows centralized control of the entire system. Admins can view registered drivers, approve or reject applications, monitor ongoing bookings, and access system logs. The admin dashboard provides graphical reports on driver activity, number of bookings, and most active regions. Additionally, the admin can deactivate users or drivers, delete suspicious accounts, and generate CSV reports. This module also integrates with the audit log system to track login/logout events, booking timestamps, and driver performance, ensuring transparency and accountability across the platform.

5.2.9. Microservices Communication

The system is divided into services like user-service, driver-service, booking-service, login-logout-service, and api-gateway. Each service is independently deployable and registered with the Eureka Server for service discovery. Communication between services is handled

using Feign clients, which automatically create REST clients for inter-service API calls. The API Gateway routes external requests to the correct service and handles security checks, token validation, and logging. For instance, the login-service verifies credentials and issues tokens, which the gateway then validates before passing requests to the protected endpoints in other services.

5.2.10. Testing & Debugging

The application was tested at both the module and system levels. Backend APIs were tested using Postman and Swagger, while frontend interactions were tested across different browsers and devices. Unit tests were written using JUnit and Mockito for services like login validation and booking confirmation. End-to-end testing involved simulating a complete booking—from user login, location capture, driver alert, booking acceptance, to real-time tracking. Bugs related to location delay, socket disconnection, and CORS issues were identified and fixed. Error logs were recorded in a logs table with timestamps and stack traces for easier debugging.

5.3 OPERATIONAL DOCUMENT

Here is a detailed **Operational Document** for your **Quick Ambulance System**. This document outlines how to deploy, configure, and operate the system efficiently in a real-world environment. It's written in paragraph form with clear points to help guide users, admins, and developers.

Operational Document for Quick Ambulance System

The **Quick Ambulance System** is a real-time, microservices-based ambulance booking platform designed to facilitate emergency medical transportation. This operational document serves as a practical guide for installing, configuring, launching, and maintaining the system in a live environment. It covers key operations from user roles to system administration and data handling.

5.3.1. System Deployment and Installation

The system consists of multiple Spring Boot microservices and a ReactJS frontend. Each microservice (e.g., user-service, driver-service, booking-service, location-service, and login-logout-service) should be built using Maven and deployed either locally or on a cloud platform like AWS, Azure, or Heroku. The Eureka Server and Spring Cloud API Gateway

should be up and running before launching any dependent services. MySQL needs to be installed with preconfigured databases and user privileges. Each service has its own application.properties or YAML file to configure ports, database credentials, and Eureka URLs.

5.3.2. User Roles and Access Control

There are three primary roles in the system: **User**, **Driver**, and **Admin**. Each role has distinct permissions and dashboards. Users can register, log in, and book ambulances. Drivers can receive booking requests and send live location updates. Admins manage registrations, bookings, and system logs. Spring Security ensures that endpoints are accessible only to authorized roles using session-based or token-based authentication mechanisms. Role checks and route guards are also implemented in the frontend using React Router.

5.3.3. Booking Flow Operation

Users initiate bookings by sharing their geolocation, which is captured via the frontend using the browser's Geolocation API. The backend service receives the coordinates and identifies nearby drivers using Haversine distance calculations. All eligible drivers receive a real-time booking alert via WebSocket. The first driver to accept the booking is assigned, and both parties receive confirmation. This flow is fully automated and optimized for real-time emergency response.

5.3.4. WebSocket and Real-Time Communication

WebSocket is implemented using STOMP over SockJS for live tracking. The system sets up dedicated channels for users and drivers (/topic/user-location/{id} and /topic/driver-location/{id}) so both sides can share and receive location data every 3 seconds. This ensures dynamic updates on the map without page reloads. If a WebSocket session is disconnected, the system automatically attempts to reconnect using fallback mechanisms. Backend controllers listen for messages, process coordinates, and forward them to the corresponding clients.

5.3.5. Admin Panel and Monitoring

The admin panel is a secure dashboard built using ReactJS. It provides interfaces to review and approve driver applications, monitor real-time system usage, and analyze performance metrics. Admins can activate or deactivate user and driver accounts, manage ambulances,

view booking logs, and export reports. A logging mechanism records all sensitive actions such as login attempts, booking confirmations, and error traces. This panel is essential for operational oversight and security.

5.3.6. Database Operations and Data Integrity

All data related to users, drivers, ambulances, and bookings is stored in a MySQL relational database. Tables are linked using primary and foreign keys to maintain relational integrity. For instance, each booking entry has references to both user and driver IDs. Triggers or scheduled jobs can be configured to purge old records or archive them. Daily backups should be enabled for disaster recovery. Database normalization and indexing ensure optimal read/write performance even during high-load scenarios.

5.3.7. Error Handling and System Logs

Each microservice includes a global exception handler that catches and logs errors with timestamps and request details. These logs are saved in the backend and optionally sent to monitoring tools like ELK Stack or Prometheus + Grafana. Common failures like server downtime, invalid login, WebSocket disconnection, and database timeout are automatically detected and reported. Admins can view and analyze logs from their dashboard or through direct database queries.

5.3.8. Security and Data Privacy

User data such as phone numbers, names, and locations are protected through HTTPS, encrypted storage, and Spring Security mechanisms. Passwords are hashed using BCrypt, and role-based access control (RBAC) is implemented to prevent unauthorized access. Sensitive operations such as ambulance assignment, admin actions, and driver updates are logged and require authentication. The system is also designed to comply with basic data privacy standards (such as not exposing live location outside authenticated sessions).

5.3.9. Maintenance and Updates

System updates (backend service upgrades, bug fixes, frontend enhancements) can be rolled out independently due to the microservices architecture. Docker can be used for containerizing services for easier deployment. Each microservice runs independently on separate ports or containers, making scaling and updates more manageable.

5.3.10. User and Driver Support

For operational continuity, a help desk or support module can be integrated where users and drivers can report issues or request support. Notifications can be sent via email or SMS during system downtime or booking failures. Admins can manually intervene to reassign ambulances or verify location mismatches. Support contact details should be displayed on the frontend along with an FAQ section for common questions about booking, cancellation, and tracking.

5.4 SYSTEM MAINTENANCE

System maintenance is a critical aspect of ensuring the ongoing functionality, security, and performance of the **Quick Ambulance System**, especially since it handles real-time operations, live tracking, and emergency medical data. A well-maintained system minimizes downtime, prevents data loss, and ensures a seamless experience for users, drivers, and administrators.

To begin with, **preventive maintenance** should be scheduled regularly. This includes checking system logs for abnormal activities, cleaning up unused sessions or cache, optimizing database queries, and monitoring service performance metrics such as CPU usage, memory consumption, and API response times. Preventive steps help avoid performance degradation over time and prepare the system for peak usage periods.

Corrective maintenance is performed when the system encounters unexpected issues like bugs, failed booking assignments, or location mismatches. These problems must be identified through monitoring tools and addressed through hotfixes or patch releases. Version control and CI/CD pipelines can streamline patch deployments without bringing the entire system offline.

For **adaptive maintenance**, the system must be updated to stay compatible with evolving technologies. For example, if a new version of Spring Boot or ReactJS is released with security or performance improvements, the services should be refactored accordingly. Similarly, WebSocket libraries or map APIs (like Google Maps) should be updated to maintain compatibility and avoid deprecated functionality.

Database maintenance is also crucial. Regular database backups should be automated to protect against data loss. Indexes should be monitored and restructured to keep query speeds optimized. Unused or obsolete records such as completed bookings older than a year can be archived or purged. Additionally, database schema should be reviewed periodically to

incorporate new features or optimize existing relationships between tables.

Security maintenance involves updating user authentication protocols, applying patches for known vulnerabilities, and reviewing access control policies. Firewalls, HTTPS certificates, and encrypted data storage must be validated periodically. Logging and alert systems should be tested to ensure they trigger the necessary notifications on suspicious activities or server failures.

Lastly, **documentation and support maintenance** ensures that all operational, deployment, and troubleshooting guides remain up to date. Admins and developers must document new modules, service changes, or API revisions. User-facing help sections must also be updated as the platform evolves, ensuring continued usability and smooth onboarding for new users or drivers.

5.5 CODE

5.5.1 Code Structure and Architecture

The codebase for the Quick Ambulance System is structured using a **microservices architecture**, where each feature—such as user login, ambulance booking, live location tracking, and driver alert—is implemented as an independent service. Each microservice is built using **Spring Boot** and communicates via REST APIs or WebSocket for real-time operations. This modular approach ensures better code maintainability, scalability, and fault isolation. Services are registered using **Eureka Discovery Server**, allowing dynamic registration and inter-service communication through the **API Gateway**, which also handles routing and security filtering.

5.5.2 Backend Code and API Development

The backend code is written in **Java (Spring Boot)** and follows a clean **MVC (Model-View-Controller)** pattern. Models represent database entities, Repositories use Spring Data JPA for seamless data access, and Controllers expose RESTful endpoints. For example, the UserController handles endpoints like /register, /login, and /logout, while the BookingController manages ambulance requests. Each service also has a Service layer that encapsulates business logic, making the code cleaner and easier to test. HTTP methods are well-structured using @GetMapping, @PostMapping, and secured via Spring Security.

Quick-Ambulance-Database Module:

```
package com.database.comp;
import org.springframework.context.annotation.Bean;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;
@Component
public class Components {
	@Bean
	public PasswordEncoder passwordEncoder() {
		return new BCryptPasswordEncoder();
	}
}
package com.database.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.database.entity.Ambulance;
import com.database.service.IAmbulanceService;
@RestController
@RequestMapping("/ambulance")
public class AmbulanceController {
	@Autowired
	private IAmbulanceService service;
	@GetMapping
	public ResponseEntity<Ambulance> findVehicleByVehicleNumber(@RequestParam
String vehicleNumber){
		Ambulance ambulance = service.findByVehicleNumber(vehicleNumber);
		return new ResponseEntity<Ambulance>(ambulance, HttpStatus.OK);
	}
	@GetMapping("/find-all")
	public ResponseEntity<List<Ambulance>> findAllAmbulanceDetails(){
		List<Ambulance> list = service.findAllAmbulanceDetails();
		return new ResponseEntity<List<Ambulance>>(list, HttpStatus.OK);
	}
	@DeleteMapping
	public ResponseEntity<String> deleteVehicleByVehicleNumber(@RequestParam
String vehicleNumber){
		String status = service.deleteAmbulanceDetails(vehicleNumber);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
```

```
	@PostMapping
	public ResponseEntity<String> saveVehicleDetails(@RequestBody Ambulance ambulance){
		String status = service.saveAmbulance(ambulance);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@PutMapping
	public ResponseEntity<String> updateVehicleDetails(@RequestBody Ambulance ambulance){
		String status = service.updateAmbulanceDetails(ambulance);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@PostMapping("/update-status")
	public ResponseEntity<String> updateAmbulanceStatus(@RequestParam String vehicleNumber,
		@RequestParam boolean status){
		System.out.println(vehicleNumber+" "+status);
		String result = service.updateStatus(vehicleNumber, status);
		return new ResponseEntity<String>(result, HttpStatus.OK);
	}
	@GetMapping("/available")
	public ResponseEntity<List<String>> getAvailableAmbulances(){
		List<String> list = service.getAvailableAmbulance();
		return new ResponseEntity<List<String>>(list, HttpStatus.OK);
	}
}

package com.database.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.commons.dto.DriverDTO;
import com.database.entity.Driver;
import com.database.service.IDriverService;
import com.database.service.ISaveAndDeleteService;
@RestController
@RequestMapping("/driver")
public class DriverController {
	@Autowired
	private ISaveAndDeleteService sadDriverService;
	@Autowired
```

```
	private IDriverService driverService;
	@PostMapping
	public ResponseEntity<String> saveDriver(@RequestBody DriverDTO driverDto){
		String status = sadDriverService.saveDriver(driverDto);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@DeleteMapping
	public ResponseEntity<String> deleteDriver(@RequestParam String username){
		String status = sadDriverService.deleteDriver(username);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@GetMapping("/find-all")
	public ResponseEntity<List<Driver>> findAllDrivers(){
		List<Driver> list = driverService.findAllDriverDetails();
		return new ResponseEntity<List<Driver>>(list, HttpStatus.OK);
	}
	@GetMapping
	public ResponseEntity<Driver> findByUsername(@RequestParam String username){
		Driver driver = driverService.findDriverByUsername(username);
		return new ResponseEntity<Driver>(driver, HttpStatus.OK);
	}
	@PutMapping
	public ResponseEntity<String> updateDriverDetails(@RequestBody Driver driver){
		String status = driverService.updateDriverDetails(driver);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@PostMapping("/update-status")
	public ResponseEntity<String> updateDriverStatus(@RequestParam String
username, @RequestParam boolean status){
		String stat = driverService.updateDriverStatus(username, status);
		return new ResponseEntity<String>(stat, HttpStatus.OK);
	}
}

package com.database.controller;
import java.time.LocalDate;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.database.entity.DriverLogs;
import com.database.projection.IVehicleAndPhoneProjection;
import com.database.service.IDriverLogsService;
```

```
@RestController
@RequestMapping("/driver-logs")
public class DriverLogsController {
	@Autowired
	private IDriverLogsService driverLogService;

	@GetMapping("/username")
	Public ResponseEntity<List<DriverLogs>>
getDriverLogsByUsername(@RequestParam String username){
		List<DriverLogs> list = driverLogService.findByUsername(username);
		return new ResponseEntity<List<DriverLogs>>(list,HttpStatus.OK);
	}
	@GetMapping("/vehicle-number")
	Public ResponseEntity<List<DriverLogs>>
getDriverLogsByVehicleNumber(@RequestParam String vehicleNumber){
		List<DriverLogs> list =
driverLogService.findByVehicleNumber(vehicleNumber);
		return new ResponseEntity<List<DriverLogs>>(list, HttpStatus.OK);
	}
	@GetMapping("/username-date")
	Public ResponseEntity<DriverLogs>
getDriverLogsByUsernameAndDate(@RequestParam String username, @RequestParam
LocalDate date){
		DriverLogs dl = driverLogService.findByUsesrnameAndDate(username,
date);
		return new ResponseEntity<DriverLogs>(dl, HttpStatus.OK);
	}
	@GetMapping("/vehicleNumber-date")
	public ResponseEntity<DriverLogs>
getDriverLogsByVehicleNumberAndDate(@RequestParam String vehicleNumber,
@RequestParam LocalDate date){
		DriverLogs dl =
driverLogService.findByVehicleNumberAndDate(vehicleNumber, date);
		return new ResponseEntity<DriverLogs>(dl, HttpStatus.OK);
	}
	@GetMapping("/username-vehicleNumber-date")
	public ResponseEntity<DriverLogs>
getByUsernameVehicleNumberAndDate(@RequestParam String username,
@RequestParam String vehicleNumber, @RequestParam LocalDate date){
		DriverLogs dl =
driverLogService.findByUsernameVehicleNumberAndDate(username, vehicleNumber,
date);
		return new ResponseEntity<DriverLogs>(dl, HttpStatus.OK);
	}
	@GetMapping("/date")
	public ResponseEntity<List<DriverLogs>> getByDate(@RequestParam LocalDate
date){
		List<DriverLogs> list = driverLogService.findByDate(date);
		return new ResponseEntity<List<DriverLogs>>(list, HttpStatus.OK);
	}
```

```
	@PostMapping("/update-vehicle")
	public ResponseEntity<Integer> updateVehicleNumber(@RequestParam String username, @RequestParam String vehicleNumber){
		Integer status = driverLogService.updateVehicleNumberByUsername(username, vehicleNumber);
		return new ResponseEntity<Integer>(status, HttpStatus.OK);
	}
	@PostMapping("/login")
	public ResponseEntity<Long> newLogin(@RequestParam String username){
		Long status = driverLogService.updateLogin(username);
		return new ResponseEntity<Long>(status, HttpStatus.OK);
	}
	@PostMapping("/logout")
	public ResponseEntity<Long> updateLogout(@RequestParam String username){
		Long status = driverLogService.updateLogoutByUsername(username);
		return new ResponseEntity<Long>(status, HttpStatus.OK);
	}
	@GetMapping("/vehicle-phone")
	public ResponseEntity<Map<String, Object>> getDriverPhoneAndVehicleNumber(@RequestParam String username){
		Map<String, Object> response = new HashMap<String, Object>();
		IVehicleAndPhoneProjection vp = driverLogService.getVehicleNumberAndPhone(username);
		response.put("vehicleNumber", vp.getVehicleNumber());
		response.put("phone", vp.getPhone());
		return new ResponseEntity<Map<String,Object>>(response, HttpStatus.OK);
	}
	@GetMapping("/vehicle")
	public ResponseEntity<String> getVehicleNumberByUsername(@RequestParam String username){
		String vehicleNumber = driverLogService.getVehicleNumberByDriverUsername(username);
		return new ResponseEntity<String>(vehicleNumber, HttpStatus.OK);
	}
	}
package com.database.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.database.entity.LiveLocation;
import com.database.service.ILiveLocationService;
@RequestMapping("/driver-live-location")
```

```
@RestController
public class LiveLocationController {
	@Autowired
	private ILiveLocationService service;
	@PostMapping
	public ResponseEntity<String> saveDriverLiveLocation(@RequestBody
LiveLocation driverLiveLocation){
		String status =
service.saveAndUpdateDriverLiveLocation(driverLiveLocation);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@GetMapping
	public ResponseEntity<List<LiveLocation>> getAllDriverLiveLocations(){
		List<LiveLocation> dll =  service.getAllDriverLiveLocation();
		return new ResponseEntity<List<LiveLocation>>(dll, HttpStatus.OK);
	}
	@DeleteMapping
	public ResponseEntity<String> deleteDriverLiveLocation(@RequestParam String
username){
		String status = service.deleteDriverLocation(username);
		return new ResponseEntity<String>(username, HttpStatus.OK);
	}
}

package com.database.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.database.entity.Roles;
import com.database.service.IRolesService;
@RestController
@RequestMapping("/roles")
public class RolesController {
	@Autowired
	private IRolesService roleService;

	public ResponseEntity<String> updatePassword(Roles role){
		String status = roleService.updatePasswordByUsername(role);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@PostMapping
	public ResponseEntity<String> saveRole(@RequestBody Roles role){
		String status = roleService.saveRoles(role);
		return new ResponseEntity<String>(status, HttpStatus.OK);
```

```
	}
	@GetMapping
	public ResponseEntity<Roles> getRolesByUsername(@RequestParam String
username){
		Roles role = roleService.findRolesByUsername(username);
		return new ResponseEntity<Roles>(role, HttpStatus.OK);
	}
}

package com.database.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.commons.dto.TrackDetailsDTO;
import com.database.entity.TrackDetails;
import com.database.service.ITrackDetailsService;
@RestController
@RequestMapping("/track-details")
public class TrackDetailsController {
	@Autowired
	private ITrackDetailsService service;
	@PostMapping
	public ResponseEntity<Long> saveTrackDetails(@RequestBody TrackDetailsDTO
dto){
		Long status = service.saveBookingDetails(dto);
		return new ResponseEntity<Long>(status, HttpStatus.OK);
	}
	@GetMapping("/user-history")
	public ResponseEntity<List<TrackDetails>> getUserHistory(@RequestParam String
username){
		List<TrackDetails> list = service.userHistory(username);
		return new ResponseEntity<List<TrackDetails>>(list, HttpStatus.OK);
	}
	@GetMapping("/driver-history")
	public ResponseEntity<List<TrackDetails>> getDriverHistory(@RequestParam
String username){
		List<TrackDetails> list = service.driverHistory(username);
		return new ResponseEntity<List<TrackDetails>>(list, HttpStatus.OK);
	}
}
package com.database.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import com.commons.dto.UserDTO;
import com.database.entity.User;
import com.database.service.ISaveAndDeleteService;
import com.database.service.IUserService;
@RestController
@RequestMapping("/user")
public class UserController {
	@Autowired
	private ISaveAndDeleteService sadUserService;
	@Autowired
	private IUserService userService;
	@PostMapping
	public ResponseEntity<String> saveUser(@RequestBody UserDTO userDto){
		String status = sadUserService.saveUser(userDto);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@DeleteMapping
	public ResponseEntity<String> deleteUser(@RequestParam String username){
		String status = sadUserService.deleteUser(username);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
	@GetMapping("/find-all")
	public ResponseEntity<List<User>> findAllUsers(){
		List<User> list = userService.findAllUserDetails();
		return new ResponseEntity<List<User>>(list, HttpStatus.OK);
	}
	@GetMapping
	public ResponseEntity<User> findByUsername(@RequestParam String username){
		User user = userService.findUserByUsername(username);
		return new ResponseEntity<User>(user, HttpStatus.OK);
	}
	@PutMapping
	public ResponseEntity<String> updateUserDetails(@RequestBody User user){
		String status = userService.updateUserDetailsByUsername(user);
		return new ResponseEntity<String>(status, HttpStatus.OK);
	}
}

package com.database.dao;
import java.util.List;
```

```
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.transaction.annotation.Transactional;
import com.database.entity.Ambulance;
public interface IAmbulanceDao extends JpaRepository<Ambulance, String> {
	@Modifying
	@Transactional
	@Query("UPDATE Ambulance a SET a.status = :status WHERE a.vehicleNumber =
:vehicleNumber")
	Integer updateStatusByVehicleNumber(String vehicleNumber, boolean status);
	@Modifying
	@Transactional
	@Query("SELECT a.vehicleNumber FROM Ambulance a WHERE a.status = false")
	Optional<List<String>> findAllAvailableAmbulanceNumbers();
}
package com.database.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.transaction.annotation.Transactional;
import com.database.entity.Driver;
public interface IDriverDao extends JpaRepository<Driver, String> {
@Modifying
@Transactional
@Query("UPDATE Driver d SET d.status = :status WHERE d.username = :username")
Integer updateDriverStatusByUsername(String username, boolean status);
}

package com.database.dao;
import java.time.LocalDate;
import java.util.List;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import com.database.entity.DriverLogs;
import com.database.projection.IVehicleAndPhoneProjection;
public interface IDriverLogsDao extends JpaRepository<DriverLogs, Long> {
Optional<List<DriverLogs>> findByDriverUsername(String username);
Optional<List<DriverLogs>> findByAmbulanceVehicleNumber(String vehicleNumber);
Optional<DriverLogs> findByDriverUsernameAndLoginDate(String username, LocalDate
date);
Optional<DriverLogs> findByAmbulanceVehicleNumberAndLoginDate(String
vehicleNumber, LocalDate date);
Optional<DriverLogs>
findByDriverUsernameAndAmbulanceVehicleNumberAndLoginDate(String username,
                                                String vehicleNumber, LocalDate date);
Optional<List<DriverLogs>> findByLoginDate(LocalDate date);
```

```
                                    Optional<DriverLogs>
findByDriverUsernameAndAmbulanceIsNullAndLogoutIsNull(String username);
                                    Optional<DriverLogs>
findByDriverUsernameAndLogoutIsNull(String username);
@Query("SELECT d.ambulance.vehicleNumber AS vehicleNumber, d.driver.phone AS
phone " + "FROM DriverLogs d " + "WHERE d.driver.username = :username AND d.logout
IS NULL AND d.ambulance.vehicleNumber IS NOT NULL")
Public                                              IVehicleAndPhoneProjection
findVehicleInfoByUsernameAndDate(@Param("username") String username);
@Query("SELECT l.ambulance.vehicleNumber FROM DriverLogs l " +"WHERE
l.driver.username = :username AND l.logout IS NULL")
Optional<String>
findAmbulanceVehicleNumberByDriverUsernameAndLogoutIsNull(@Param("username")
String username);
}

package com.database.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import com.database.entity.LiveLocation;
public interface ILiveLocationDao extends JpaRepository<LiveLocation, String>{}
package com.database.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import com.database.entity.Roles;
public interface IRolesDao extends JpaRepository<Roles, String> {}

package com.database.dao;
import java.util.List;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import com.database.entity.TrackDetails;
public interface ITrackDetailsDao extends JpaRepository<TrackDetails, Long> {
Optional<List<TrackDetails>> findByUserUsername(String username);
Optional<List<TrackDetails>> findByDriverUsername(String username);
}
package com.database.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import com.database.entity.User;
public interface IUserDao extends JpaRepository<User, String> {}
package com.database.exceptionhandler;
import java.time.LocalDateTime;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import com.commons.dto.ErrorResponseDTO;
import  com.commons.exception.DuplicateResourceException;
import com.commons.exception.ResourceNotFoundException;
@RestControllerAdvice
public class GlobalExceptionHandler {
@ExceptionHandler(ResourceNotFoundException.class)
```

```
public ResponseEntity<ErrorResponseDTO>
handleNotFoundException(ResourceNotFoundException ex){
ErrorResponseDTO error = new
ErrorResponseDTO(HttpStatus.NOT_FOUND.value(), ex.getMessage(),
LocalDateTime.now());
return new ResponseEntity<ErrorResponseDTO>(error,
HttpStatus.NOT_FOUND);
}
@ExceptionHandler(DuplicateResourceException.class)
public ResponseEntity<ErrorResponseDTO>
handleDuplicateResourceException(DuplicateResourceException ex){
ErrorResponseDTO error = new
ErrorResponseDTO(HttpStatus.CONFLICT.value(), ex.getMessage(),
LocalDateTime.now());
return new ResponseEntity<ErrorResponseDTO>(error,
HttpStatus.CONFLICT);
}
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponseDTO> handleGlobalException(Exception ex){
ErrorResponseDTO error = new
ErrorResponseDTO(HttpStatus.INTERNAL_SERVER_ERROR.value(), ex.getMessage(),
LocalDateTime.now());
return new ResponseEntity<ErrorResponseDTO>(error,
HttpStatus.INTERNAL_SERVER_ERROR);
}
}
package com.database.service;
import java.util.Base64;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import  com.commons.exception.DuplicateResourceException;
import com.commons.exception.ResourceNotFoundException;
import com.database.dao.IAmbulanceDao;
import com.database.entity.Ambulance;
@Service
public class AmbulanceServiceImple implements IAmbulanceService {
@Autowired
private IAmbulanceDao dao;
private Ambulance amb;
private String status;
@Override
public String saveAmbulance(Ambulance ambulance) {
if (ambulance != null) {
dao.findById(ambulance.getVehicleNumber()).ifPresent(a -> {
throw new DuplicateResourceException(
"AMBULANCE_EXISTED_VEHICLE_NUMBER " +
ambulance.getVehicleNumber());
});
ambulance.setVehicleNumber(ambulance.getVehicleNumber(
```

```
).toUpperCase());
                        ambulance.setStatus(false);
                        ambulance.setCbook(this.convertBase64ToBytes(ambulance.g
etBook()));
                                amb = dao.save(ambulance);
                                status = amb.getVehicleNumber();
                        } else
                                throw                                   new
RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN.....!");
                        return status;
                }
@Override
public Ambulance findByVehicleNumber(String vehicleNumber) {
if (vehicleNumber != null) {
amb = dao.findById(vehicleNumber.toUpperCase()).orElseThrow(
                                                        ()          ->          new
ResourceNotFoundException("VEHICLE_NOT_FOUND_WITH_VEHICLE_NUMBER " +
vehicleNumber));
                        }
                        return amb;
                }
                @Override
                public List<Ambulance> findAllAmbulanceDetails() {
                        return dao.findAll();
                }
                @Override
                public        String        updateAmbulanceDetails(Ambulance
ambulance) {
                        if (ambulance != null) {
                                amb                                             =
dao.findById(ambulance.getVehicleNumber())
                                                        .orElseThrow(()         ->          new
ResourceNotFoundException(
                                "RESOURCE_NOT_FOUND_WITH_VEHICLE_NUMBER
 " + ambulance.getVehicleNumber()));
                        }
                        if (amb != null) {
                                Ambulance                       amb                     =
this.updateNullValues(ambulance);
                                status = dao.save(amb).getVehicleNumber();
                        } else
                                throw                                   new
RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");

                        return status;
                }
                @Override
                public String deleteAmbulanceDetails(String vehicleNumber)
{
                        amb                                                     =
```

```
dao.findById(vehicleNumber.toUpperCase()).orElseThrow(
                                        () -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_VEHICLE_NUMBER
" + vehicleNumber));
                                if (amb != null) {
                                        dao.delete(amb);
                                        status = vehicleNumber;
                                } else
                                        throw new
RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");
                                return status;
                        }
                        private byte[] convertBase64ToBytes(String file) {
                                return Base64.getDecoder().decode(file);
                        }
                        private Ambulance updateNullValues(Ambulance ambulance)
{
                                amb =
dao.findById(ambulance.getVehicleNumber()).orElseThrow(() -> new
ResourceNotFoundException(
                                "VEHICLE_NOT_FOUND_WITH_VEHICLE_NUMBER "
+ ambulance.getVehicleNumber()));
                                if (ambulance.getStatus() == null)
                                        ambulance.setStatus(amb.getStatus());
                                if (ambulance.getCbook() == null)
                                        ambulance.setCbook(amb.getCbook());
                                if (ambulance.getColor() == null)
                                        ambulance.setColor(amb.getColor());
                                if (ambulance.getCompany() == null)
                                        ambulance.setCompany(amb.getCompany());
                                if (ambulance.getVehicleNumber() == null)
                        ambulance.setVehicleNumber(amb.getVehicleNumber());
                                if (ambulance.getYearOfModel() == null)
                                ambulance.setYearOfModel(amb.getYearOfModel());
                                return ambulance;
                        }
                        @Override
                        public String updateStatus(String vehicleNumber, boolean s) {
                        dao.findById(vehicleNumber.toUpperCase()).orElseThrow(
                                        () -> new
ResourceNotFoundException("VEHICLE_NOT_FOUND_WITH_VEHICLE_NUMBER " +
vehicleNumber));
                                Integer result =
dao.updateStatusByVehicleNumber(vehicleNumber.toUpperCase(), s);
                                if (result != 0)
                                        status = vehicleNumber;
                                else
                                        throw new
RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");
                                return status;
```

```
	}
	@Override
	public List<String> getAvailableAmbulance() {
		return
dao.findAllAvailableAmbulanceNumbers().filter(logs -> !logs.isEmpty())
.orElseThrow(() -> new
ResourceNotFoundException("AMBULANCES_ARE_NOT_AVAILABLE"));
	}
}
package com.database.service;
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.commons.exception.ResourceNotFoundException;
import com.database.dao.IAmbulanceDao;
import com.database.dao.IDriverDao;
import com.database.dao.IDriverLogsDao;
import com.database.entity.Ambulance;
import com.database.entity.Driver;
import com.database.entity.DriverLogs;
import com.database.projection.IVehicleAndPhoneProjection;
@Service
public class DriverLogsServiceImple implements IDriverLogsService {
	@Autowired
	private IDriverLogsDao dao;
	@Autowired
	private IDriverDao driverDao;
	@Autowired
	private IAmbulanceDao ambulanceDao;
	@Override
	public List<DriverLogs> findByUsername(String username) {
		return
dao.findByDriverUsername(username.toLowerCase())
.filter(logs -> !logs.isEmpty())
.orElseThrow(
() -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME
"+username)
);
	}
	@Override
	public List<DriverLogs> findByVehicleNumber(String
vehicleNumber) {
		return
dao.findByAmbulanceVehicleNumber(vehicleNumber.toUpperCase())
.filter(logs -> !logs.isEmpty())
.orElseThrow(
```

```
                                        () ->new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_VEHICLE_NUMBER
"+vehicleNumber)
                                        );
                        }

                        @Override
                        public DriverLogs findByUsesrnameAndDate(String
username, LocalDate date) {
                                return
dao.findByDriverUsernameAndLoginDate(username.toLowerCase(), date).orElseThrow(
                                        () -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME_AND_D
ATE "+username+" "+date)
                                        );
                        }

                        @Override
                        public DriverLogs findByVehicleNumberAndDate(String
vehicleNumber, LocalDate date) {
                                return
dao.findByAmbulanceVehicleNumberAndLoginDate(vehicleNumber.toUpperCase(),
date).orElseThrow(
                                        () -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_VEHICLE_NUMBER_
AND_DATE "+vehicleNumber+" "+date)
                                        );
                        }
                        @Override
                        public DriverLogs
findByUsernameVehicleNumberAndDate(String username, String vehicleNumber,
LocalDate date) {
                                return
dao.findByDriverUsernameAndAmbulanceVehicleNumberAndLoginDate(username.toLowe
rCase(), vehicleNumber.toUpperCase(), date).orElseThrow(
                                        () -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME_VEHICL
E_NUMBER_AND_DATE "+username+" "+vehicleNumber+" "+date)
                                        );
                        }
                        @Override
                        public List<DriverLogs> findByDate(LocalDate date) {
                                return dao.findByLoginDate(date)
                                        .filter(logs -> !logs.isEmpty())
                                        .orElseThrow(
                                        () -> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_DATE "+ date)
                                        );
                        }
                        @Override
```

```
public Integer updateVehicleNumberByUsername(String username, String vehicleNumber) {
		Integer status = 0;
		LocalDate date = LocalDate.now();
		DriverLogs dl = dao.findByDriverUsernameAndAmbulanceIsNullAndLogoutIsNull(username.toLowerCase()).orElseThrow(
				() -> new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME "+username+" "+vehicleNumber+" "+date)
				);
		Ambulance amb = ambulanceDao.findById(vehicleNumber.toUpperCase()).orElseThrow(
				()->{
					throw new ResourceNotFoundException("RESOURCE_NOT_AVAILABLE_WITH_VEHICLE_NUMBER "+vehicleNumber);
				});

		if (dl != null) {
			dl.setAmbulance(amb);
			dao.save(dl);
			status = 1;
		}
		else
			throw new RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");
		return status;
	}
	@Override
	public Long updateLogoutByUsername(String username) {
		Long status = 0L;
		DriverLogs dl = dao.findByDriverUsernameAndLogoutIsNull(username.toLowerCase()).orElseThrow(
				() -> new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME "+username+" ")
				);
		if (dl != null) {
			dl.setLogout(LocalTime.now());
			status = dao.save(dl).getLogid();
		}
		else
			throw new RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");
		return status;
	}
	@Override
	public Long updateLogin(String username) {
		Driver driver = null;
```

```
                DriverLogs dl = null;
                Long status = 0L;
                driver =
driverDao.findById(username.toLowerCase()).orElseThrow(
                        ()-> new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME
"+username)
                        );
                Optional<DriverLogs> opt =
dao.findByDriverUsernameAndLogoutIsNull(username);
                if(opt.isPresent()) {
                    dl = opt.get();
                    String vehicleNumber =
dl.getAmbulance().getVehicleNumber();
                    if(vehicleNumber != null) {

                ambulanceDao.findById(vehicleNumber).ifPresent(a->{

                ambulanceDao.updateStatusByVehicleNumber(vehicleNumbe
r, false);
                        });
                    }
                    dl.setLogout(LocalTime.now());
                    dao.save(dl);
                }
                if (driver != null) {
                    dl = new DriverLogs();
                    dl.setLoginDate(LocalDate.now());
                    dl.setDriver(driver);
                    dl.setLogin(LocalTime.now());
                    status = dao.save(dl).getLogid();
                }
                else
                    throw new
RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN...!");
                return status;
            }
            @Override
            public IVehicleAndPhoneProjection
getVehicleNumberAndPhone(String username) {
                IVehicleAndPhoneProjection vp =
dao.findVehicleInfoByUsernameAndDate(username);
                if(vp == null)
                    throw new
ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH  "+username);
                return vp;
            }
            @Override
            public String getVehicleNumberByDriverUsername(String
username) {
```

```
                String vehicleNumbner = dao.findAmbulanceVehicleNumberByDriverUsernameAndLogoutIsNull(username)
                        .orElseThrow(
                                () -> new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH "+username)
                                );
                return vehicleNumbner;
        }
}

package com.database.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import  com.commons.exception.DuplicateResourceException;
import com.commons.exception.ResourceNotFoundException;
import com.database.dao.IDriverDao;
import com.database.entity.Driver;

@Service
public class DriverServiceImple implements IDriverService {
        @Autowired
        private IDriverDao dao;
        private String status;
        private Driver driv;
        @Override
        public String saveDriverDetails(Driver driver) {
        dao.findById(driver.getUsername().toLowerCase()).ifPresent(
a -> {
                throw new DuplicateResourceException("RESOURCE_EXISTED_WITH_USERNAME " + driver.getUsername());
                });
                driv = dao.save(driver);
                if (driv != null)
                        status = driv.getUsername();
                else
                        throw new RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN.....!");
                return status;
        }
        @Override
        public Driver findDriverByUsername(String username) {
                driv = dao.findById(username.toLowerCase()).orElseThrow(
                        ()-> new ResourceNotFoundException("DRIVER_NOT_AVAILABLE_WITH_ID  "+username)
                        );
                return driv;
        }
```

```
	@Override
	public List<Driver> findAllDriverDetails() {
		return dao.findAll();
	}
	@Override
	public String updateDriverDetails(Driver driver) {
		Driver driv = null;
		if (driver != null) {
			driv = this.updateNullValues(driver);
			status = dao.save(driv).getUsername();
		}
		return status;
	}
	@Override
	public String deleteDriverByUsername(String username) {
		Driver driv = dao.findById(username.toLowerCase()).orElseThrow(
				()-> new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME "+username)
				);
		if(driv != null) {
			dao.delete(driv);
			status = username;
		}
		else {
			throw new RuntimeException("SOMETHING_WENT_WRONG_TRY_AGAIN.....!");
		}
		return status;   }
	@Override
	public String updateDriverStatus(String username, boolean st) {
		Integer result = dao.updateDriverStatusByUsername(username.toLowerCase(), st);
		if(result != null)
			status = username;
		else
			throw new RuntimeException("SOMETHING_WENT_WRONG......!");
		return status;
	}
	private Driver updateNullValues(Driver driver) {
		Driver driv = dao.findById(driver.getUsername().toLowerCase()).orElseThrow(
				()-> new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_USERNAME "+driver.getUsername())
				);
		if(driv != null) {
```

```
                                        if(driver.getStatus() == null)
                                                driver.setStatus(driv.getStatus());
                                        if(driver.getDob() == null)
                                                driver.setDob(driv.getDob());
                                        if(driver.getDriverName() == null)
                                    driver.setDriverName(driv.getDriverName());
                                        if(driver.getEmail() == null)
                                                driver.setEmail(driv.getEmail());
                                        if(driver.getLicense() == null)
                                                driver.setLicense(driv.getLicense());
                                        if(driver.getPhone() == null)
                                                driver.setPhone(driv.getPhone());
                                        if(driver.getUsername() == null)
                                                driver.setUsername(driv.getUsername());
                                    }
                                    return driver;
                                }
}

package com.database.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.commons.exception.ResourceNotFoundException;
import com.database.dao.ILiveLocationDao;
import com.database.entity.LiveLocation;
@Service
public class LiveLocationServiceImple implements ILiveLocationService {
                                @Autowired
                                private ILiveLocationDao dao;
                                @Override
                                public                                                  String
saveAndUpdateDriverLiveLocation(LiveLocation driverLiveLocation) {
                                    String                      status                  =
dao.save(driverLiveLocation).getUsername();
                                    return status;
                                }
                                @Override
                                public List<LiveLocation> getAllDriverLiveLocation() {
                                    return dao.findAll();
                                }
                                @Override
                                public String deleteDriverLocation(String username) {
                                dao.findById(username.toLowerCase()).orElseThrow(()->
new ResourceNotFoundException("RESOURCE_NOT_FOUND_WITH_"+username));
                                    dao.deleteById(username.toLowerCase());
                                    return username;
                                }
}
```

```
package com.database.service;
import java.util.Base64;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.commons.dto.DriverDTO;
import com.commons.dto.UserDTO;
import com.database.entity.Driver;
import com.database.entity.Roles;
import com.database.entity.User;
import jakarta.transaction.Transactional;
@Service
@Transactional
public class SaveAndDeleteServiceImple implements ISaveAndDeleteService {
                                        @Autowired
                                        private IUserService userService;
                                        @Autowired
                                        private IDriverService driverService;
                                        @Autowired
                                        private IRolesService roleService;
                                        private String status;
                                        private Driver driver;
                                        private User user;
                                        private Roles role;
                                        @Override
                                        @Transactional
                                        public String saveUser(UserDTO userDto) {
                                                user = new User();
                                                role = new Roles();
                                                String use = null;
                                                String rol = null;
                                                if (userDto != null) {
                                                        user.setEmail(userDto.getEmail());
                                                        user.setFullName(userDto.getFullName());
                                                        user.setPhone(userDto.getPhone());
                                        user.setUsername(userDto.getUsername().toLowerCase());
                                                        role.setPassword(userDto.getPassword());
                                        role.setUsername(userDto.getUsername().toLowerCase());
                                                        role.setRole("USER");
                                                        use = userService.saveUserDetails(user);
                                                        rol = roleService.saveRoles(role);
                                                }
                                                if ((use != null && rol != null) && use.equals(rol))
                                                        status = use;
                                                return status;
                                        }
                                        @Override
                                        @Transactional
                                        public String saveDriver(DriverDTO driverDto) {
                                                driver = new Driver();
```

```
                role = new Roles();
                String driv = null;
                String rol = null;
                if (driverDto != null) {
                        driver.setDob(driverDto.getDob());
                driver.setDriverName(driverDto.getDriverName());
                        driver.setEmail(driverDto.getEmail());
                        driver.setPhone(driverDto.getPhone());
                        driver.setStatus(false);
                driver.setUsername(driverDto.getUsername().toLowerCase());
                driver.setLicense(this.convertBase64ToBytes(driverDto.getLi
cense()));
                        role.setPassword(driverDto.getPassword());
                role.setUsername(driverDto.getUsername().toLowerCase());
                        role.setRole("DRIVER");
                        driv = driverService.saveDriverDetails(driver);
                        rol = roleService.saveRoles(role);
                }
                if((rol != null && driv != null) && rol.equals(driv))
                        status = rol;
                return status;
        }
        @Override
        @Transactional
        public String deleteUser(String username) {
                String                  use                     =
userService.deleteUserByUsername(username.toLowerCase());
                String                  rol                     =
roleService.deleteRoleByUsername(username.toLowerCase());
                if ((use != null && rol != null) && (use.equals(rol)))
                        status = use;
                return status;
        }
        @Override
        @Transactional
        public String deleteDriver(String username) {
                String                  driv                    =
driverService.deleteDriverByUsername(username.toLowerCase());
                String                  rol                     =
roleService.deleteRoleByUsername(username.toLowerCase());
                if ((driv != null && rol != null) && (driv.equals(rol)))
                        status = driv;
                return status;
        }
        private byte[] convertBase64ToBytes(String file) {
                return Base64.getDecoder().decode(file);
        }
}
```

5.5.3. Real-Time Communication via WebSocket

A major portion of the code focuses on **real-time features** like ambulance driver alerts and live location tracking. These are implemented using **Spring WebSocket with STOMP protocol** and **SockJS**. On the frontend, JavaScript or React clients establish a WebSocket connection and periodically send the current location every 3 seconds. The backend handles messages using @MessageMapping, broadcasting updates to subscribed users via @SendTo. This part of the code ensures timely and smooth interactions without overwhelming the server, as would happen with constant HTTP polling.

Quick-Ambulance-Location Module:

```
package com.location.controller;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Controller;
import com.location.comp.BookStore;
import com.commons.dto.BookingInfo;
import com.commons.dto.DriverTransferDTO;
import com.commons.dto.LiveLocation;
import com.commons.dto.TrackDetailsDTO;
import com.commons.dto.UserTransferDTO;
import com.location.service.IDriverLogsService;
import com.location.service.IDriverService;
import com.location.service.ILiveLocationService;
import com.location.service.ILocationService;
import com.location.service.IRolesService;
import com.location.service.ITrackDetailsService;
import com.location.service.IUserService;
@Controller
public class LocationController {
                                        @Autowired
                                        private ILocationService locationService;
                                        @Autowired
                                        private SimpMessagingTemplate messagingTemplate;
                                        @Autowired
                                        private IRolesService roleService;
                                        @Autowired
                                        private ILiveLocationService liveLocationService;
                                        @Autowired
                                        private BookStore bookStore;
                                        @Autowired
                                        private IDriverLogsService driverLogsService;
                                        @Autowired
                                        private ITrackDetailsService trackService;
```

```
@Autowired
private IUserService userService;
@Autowired
private IDriverService driverService;
@MessageMapping("/test")
public void testMethod(LiveLocation location) {
    messagingTemplate.convertAndSend("/topic/test/" +
location.getUsername(), location);
}
@MessageMapping("/live-location")
public void sendLocationName(LiveLocation location) {
    System.out.println(location);
    try {
        String username =
liveLocationService.saveDriverLiveLocation(location);
    } catch (Exception e) {

messagingTemplate.convertAndSend("/topic/error/" +
location.getUsername(),
                                    "RESOURCE_NOT_FOUND " +
location.getUsername());
    }
}
@MessageMapping("/driver-locations")
public void nearByAmbulance(LiveLocation
userLiveLocation) {
    try {
        List<LiveLocation> nearByDrivers =
liveLocationService.getNearByDrivers(userLiveLocation);

messagingTemplate.convertAndSend("/topic/nearby-
ambulance/" + userLiveLocation.getUsername(),
                                    nearByDrivers);
    } catch (Exception e) {

messagingTemplate.convertAndSend("/topic/error/" +
userLiveLocation.getUsername(),
                                    "RESOURCE_NOT_FOUND " +
 userLiveLocation.getUsername());
    }
}

@MessageMapping("/book")
public void bookAmbulance(LiveLocation userLiveLocation)
{

    try {
        System.out.println("User location
"+userLiveLocation);
```

```
                    bookStore.save(userLiveLocation.getUsername(),
userLiveLocation);
                        List<LiveLocation> nearByDrivers =
liveLocationService.getNearByDrivers(userLiveLocation);
                        System.out.println("Near by drivers
"+nearByDrivers);
                        for (LiveLocation driverLiveLocation :
nearByDrivers) {
                            String message =
userLiveLocation.getUsername();
                            String topic = "/topic/alert/" +
driverLiveLocation.getUsername();

                    messagingTemplate.convertAndSend("/topic/alert/" +
driverLiveLocation.getUsername(), message);
                            System.out.println("Message sent to
"+driverLiveLocation.getUsername());
                        }
                    } catch (Exception e) {
                        e.printStackTrace();

                    messagingTemplate.convertAndSend("/topic/error/" +
userLiveLocation.getUsername(),
                                                "RESOURCE_NOT_FOUND " +
userLiveLocation.getUsername());
                    }

                }
                @MessageMapping("/accept")
                public void acceptBooking(BookingInfo driverLiveLocation)
{
                    try {
                        LiveLocation userLiveLocation =
(LiveLocation) bookStore.get(driverLiveLocation.getUserUsername());
                        System.out.println(userLiveLocation);
                        Map<String, Object> response =
driverLogsService.getDriverVehicleNumberAndPhone(
                    driverLiveLocation.getDriverUsername()
                                            );
                        String vehicleNumber =
(String)response.get("vehicleNumber");
                        Long driverPhone =
(Long)response.get("phone");
                        Long userPhone =
userService.findUserByUsername(userLiveLocation.getUsername()).getPhone();
                        Double distance =
liveLocationService.calculateDistance(userLiveLocation, this.convert(driverLiveLocation))
                                            / 1000.0;
                        Long status = 0L;
                        if
```

```
(userLiveLocation.getUsername().equals(driverLiveLocation.getUserUsername()))  {
                                                TrackDetailsDTO trackDetailsDto = new
TrackDetailsDTO();
                                trackDetailsDto.setDriverName(driverLiveLocation.getDriver
Username());
                                trackDetailsDto.setUsername(userLiveLocation.getUsername(
));
                                trackDetailsDto.setVehicleNumber(vehicleNumber);
                                trackDetailsDto.setPickup(this.convertLocationToString(user
LiveLocation));
                                                status                                  =
trackService.saveBookingDetails(trackDetailsDto);
                                        }
                                        UserTransferDTO userDTO = null;
                                        DriverTransferDTO driverDTO = null;

                                        if (status != 0) {
                                                userDTO                 =               new
UserTransferDTO(driverLiveLocation.getDriverUsername(), vehicleNumber,
                                                driverLiveLocation.getLongitude(),
driverLiveLocation.getLatitude(), driverPhone, distance);
                                                driverDTO               =               new
DriverTransferDTO(userLiveLocation.getUsername(), userLiveLocation.getLongitude(),
                                                        userLiveLocation.getLatitude(),
userPhone, distance);                   }

                                        if (userDTO != null && driverDTO != null) {
                                                String  userTopic  =  "/topic/user/"  +
userLiveLocation.getUsername();
                                                String  driverTopic  =  "/topic/driver/"  +
driverLiveLocation.getDriverUsername();
                                messagingTemplate.convertAndSend(userTopic,     userDTO);
                                        messagingTemplate.convertAndSend(driverTopic,
driverDTO);
                                                System.out.println("sent");
                                        }
                                } catch (Exception e) {
                                        e.printStackTrace();
                                messagingTemplate.convertAndSend("/topic/error/"            +
driverLiveLocation.getUserUsername(),
                                                        "RESOURCE_NOT_FOUND " +
driverLiveLocation.getUserUsername());
                                messagingTemplate.convertAndSend("/topic/error/"            +
driverLiveLocation.getDriverUsername(),
                                driverLiveLocation.getDriverUsername());
                                        }
                                }
                                @MessageMapping("/driver-live-updates")
                                public void sendDriverLive(BookingInfo driverLiveLocation)
{
```

```
                try {
                        LiveLocation userLiveLocation =
(LiveLocation) bookStore.get(driverLiveLocation.getUserUsername());
                        Double distance =
liveLocationService.calculateDistance(userLiveLocation, this.convert(driverLiveLocation));
                        UserTransferDTO userDTO = null;
                        DriverTransferDTO driverDTO = null;
                        userDTO = new
UserTransferDTO(driverLiveLocation.getDriverUsername(), null,
                driverLiveLocation.getLongitude(),
driverLiveLocation.getLatitude(), null, distance / 1000.0);
                        driverDTO = new
DriverTransferDTO(userLiveLocation.getUsername(), userLiveLocation.getLongitude(),
                                        userLiveLocation.getLatitude(),
null, distance / 1000.0);
                        if (userDTO != null && driverDTO != null) {
                                String userTopic = "/topic/user/" +
userLiveLocation.getUsername();
                                String driverTopic = "/topic/driver/" +
driverLiveLocation.getDriverUsername();
                messagingTemplate.convertAndSend(userTopic, userDTO);
                        messagingTemplate.convertAndSend(driverTopic,
driverDTO);
                                }
                                if (Math.abs(distance - 1.0) < 0.1)
                        bookStore.remove(userLiveLocation.getUsername());
                        } catch (Exception e) {
                messagingTemplate.convertAndSend("/topic/error/" +
driverLiveLocation.getUserUsername(),
                                        "RESOURCE_NOT_FOUND " +
driverLiveLocation.getUserUsername());

                messagingTemplate.convertAndSend("/topic/error/" +
driverLiveLocation.getDriverUsername(),
                driverLiveLocation.getDriverUsername());
                        }
                }
                private LiveLocation convert(BookingInfo info) {
                        LiveLocation live = new LiveLocation();
                        live.setLatitude(info.getLatitude());
                        live.setLongitude(info.getLongitude());
                        live.setUsername(info.getDriverUsername());
                        return live;
                }
                private String convertLocationToString(LiveLocation live) {
                        String pickup =
locationService.getFullAddress(live).getAddresses().get(0).getAddress().getStreetNumber() +
" " +
locationService.getFullAddress(live).getAddresses().get(0).getAddress().getStreetName() + "
" +
```

```
locationService.getFullAddress(live).getAddresses().get(0).getAddress().getMunicipality() +
" "                                                                    +
locationService.getFullAddress(live).getAddresses().get(0).getAddress().getMunicipality();
                                                    return pickup;
                                                }}
```

5.3.4. Security Implementation

Security is handled using **Spring Security**, with role-based access control and authentication mechanisms. The code verifies user credentials from the database and issues a session or JWT token depending on the configuration. All protected APIs include proper role checks (hasRole('USER'), hasRole('ADMIN'), etc.), ensuring only authorized users can access specific operations like accepting bookings or tracking ambulances. Security filters are implemented to handle CORS, CSRF, and unauthorized access attempts.

Quick-Ambulance-Login-Logout module:

```
package com.loginlogout.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import com.commons.dto.Roles;
import com.loginlogout.service.AuthService;
import com.loginlogout.service.IAmbulanceService;
import com.loginlogout.service.IDriverLogsService;
import com.loginlogout.service.IDriverService;
import com.loginlogout.service.ILiveLocationService;
import com.loginlogout.util.JwtUtil;
import jakarta.servlet.http.HttpServletRequest;
import java.util.HashMap;
import java.util.Map;
@RestController
@RequestMapping("/auth")
public class AuthController {
                                        @Autowired
                                        private IDriverService driverService;
                                        @Autowired
                                        private IDriverLogsService driverLogsService;
                                        @Autowired
                                        private IAmbulanceService ambulanceService;
                                        @Autowired
                                        private ILiveLocationService liveLocationService;
                                        @Autowired
                                        private AuthService authService;
                                        @Autowired
                                        private JwtUtil jwtUtil;
                                        @PostMapping("/login")
                                        public ResponseEntity<?> login(@RequestBody Roles role) {
```

```
                Map<String, Object> response = null;
                String rol = null;
                if (role != null) {
                        response = authService.authenticateUser(role);
                }
                if (response != null)
                        rol = (String) response.get("role");
                if (rol != null) {
                        if (rol.equalsIgnoreCase("DRIVER")) {
                driverService.updateDriverStatus(role.getUsername(),
true);
                driverLogsService.saveDriverLoginDetails(role.getUsername(
));
                        }
                }
                return new ResponseEntity<Map<String,
Object>>(response, HttpStatus.OK);
                }
                @GetMapping("/logout")
                public ResponseEntity<?> logout(HttpServletRequest request)
{
                String authHeader =
request.getHeader("Authorization");
                String role = null;
                String token = null;
                String username = null;
                Map<String, Object> response = new HashMap<String,
Object>();
                if (authHeader != null &&
authHeader.startsWith("Bearer "))
                        token = authHeader.substring(7);
                if (token != null) {
                        role = jwtUtil.extractRole(token);
                        username = jwtUtil.extractUsername(token);
                }
                if (role != null && username != null &&
role.equalsIgnoreCase("DRIVER")) {
                        driverService.updateDriverStatus(username,
false);
                ambulanceService.updateAmbulanceStatus(driverLogsService

                .findVehicleNumberByUsername(username),
                                        false);
                        driverLogsService.logoutDriver(username);

                //liveLocationService.deleteDriverLiveLocation(username);
                }
                response.put("status", "logout success");
                response.put("username", username);
                return ResponseEntity.ok(response);
```

```
                }
                @PostMapping("/update-vehicle")
                public ResponseEntity<?> updateVehicle(HttpServletRequest
request, @RequestParam String vehicleNumber) {
                        String authHeader =
request.getHeader("Authorization");
                        String username = null;
                        String role = null;
                        String token = null;
                        System.out.println("Update vehicle");
                        Map<String, Object> response = new HashMap<String,
Object>();
                        if (authHeader != null &&
authHeader.startsWith("Bearer ")) {
                                token = authHeader.substring(7);
                        } else {
                                response.put("status", "Unauthorized");
                        }
                        if (token != null) {
                                role = jwtUtil.extractRole(token);
                                username = jwtUtil.extractUsername(token);
                                System.out.println("ROle and Username
"+role+" "+username);
                        }

                        if (role != null && username != null &&
role.equalsIgnoreCase("DRIVER")) {
                                Integer user =
driverLogsService.updateVehicle(username, vehicleNumber);
                                System.out.println("Vehicle number updated
success "+user);
                                String v =
ambulanceService.updateAmbulanceStatus(vehicleNumber, true);
                                System.out.println("ambulance status updated
success "+v);                           response.put("status", "Vehicle Updated");
                        } else {
                                response.put("status", "Unauthorized");
                        }
                        return ResponseEntity.ok(response);
                }
}

package com.loginlogout.filter;
import java.io.IOException;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.GrantedAuthority;
```

```
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import com.loginlogout.service.CustomUserDetailsService;
import com.loginlogout.util.JwtUtil;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
@Component
public class JwtFilter extends OncePerRequestFilter{
                @Autowired
                private JwtUtil jwtUtil;
                @Autowired
                private ApplicationContext context;
                @Override
                protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
                                throws ServletException, IOException {
                        String token = null;
                        String username = null;
                        String role = null;
                        String path = request.getRequestURI();
                        String method = request.getMethod();
                        if((path.equals("/auth/login")                          &&
method.equalsIgnoreCase("POST"))) {
                                filterChain.doFilter(request, response);
                                return;
                        }
                        String                  authHeader                      =
request.getHeader("Authorization");
                        if(authHeader           !=          null          &&
authHeader.startsWith("Bearer ")) {
                                token = authHeader.substring(7);
                                username = jwtUtil.extractUsername(token);
                                role = jwtUtil.extractRole(token);
                        }
                        if(authHeader           !=          null          &&
SecurityContextHolder.getContext().getAuthentication() == null) {
                                UserDetails             userDetails             =
context.getBean(CustomUserDetailsService.class).loadUserByUsername(username);
                                if(jwtUtil.validateUser(userDetails, token)) {
                                        List<GrantedAuthority>     authority     =
List.of(new SimpleGrantedAuthority(role));
                                        UsernamePasswordAuthenticationToken
authToken = new UsernamePasswordAuthenticationToken(username, null, authority);
```

```
                                                    authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                                SecurityContextHolder.getContext().setAuthentication(authTo
ken);
                                                }
                                        }
                                        filterChain.doFilter(request, response);
                                }
}
```

5.3.5. Frontend Integration

The frontend code is written in **ReactJS**, and it interacts with backend APIs using Axios. The React components are split logically—login forms, booking panels, maps for tracking, and admin dashboards. State management is done using React hooks, and WebSocket messages are handled with JavaScript event listeners to update the UI in real time. The frontend is styled using responsive CSS, ensuring a smooth user experience across devices. Error messages like "Login Failed" or "Ambulance Unavailable" are handled gracefully through dynamic UI updates.

```
import React from 'react'
import { FaAmbulance, FaUser, FaRoute, FaCalendarCheck } from 'react-icons/fa'
const StatCard = ({ icon: Icon, title, value }) => (
  <div className="bg-white p-6 rounded-lg shadow-md border border-zinc-200">
    <div className="flex items-center space-x-4">
      <div className="bg-zinc-100 p-3 rounded-full">
        <Icon className="text-zinc-700 text-2xl" />
      </div>
      <div>
        <h3 className="text-zinc-500 text-sm font-medium">{title}</h3>
        <p className="text-zinc-900 text-2xl font-semibold">{value}</p>
      </div>
    </div>
  </div>
)
const Admin = () => {
  const stats = [
    { icon: FaAmbulance, title: 'Total Ambulances', value: '24' },
    { icon: FaUser, title: 'Active Drivers', value: '18' },
    { icon: FaRoute, title: 'Ongoing Trips', value: '5' },
    { icon: FaCalendarCheck, title: 'Today\'s Bookings', value: '12' }
  ]
  return (
    <div className="flex-1  p-6">
      <div className="max-w-7xl mx-auto">
        <div className="mb-8">
          <h1 className="text-3xl font-bold text-zinc-900">Dashboard</h1>
```

```
        <p className="text-zinc-600 mt-1">Welcome to the admin dashboard</p>
      </div>
      <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-6 mb-8">
        {stats.map((stat, index) => (
          <StatCard key={index} {...stat} />
        ))}
      </div>
      <div className="grid grid-cols-1 lg:grid-cols-2 gap-6">
        <div className="bg-white p-6 rounded-lg shadow-md border border-zinc-200">
          <h2 className="text-xl font-semibold text-zinc-900 mb-4">Recent Bookings</h2>
          <div className="space-y-4">
            {/* Placeholder for recent bookings list */}
            <div className="flex justify-between items-center p-3 bg-zinc-50 rounded">
              <div>
                <p className="font-medium text-zinc-900">Emergency Booking #1242</p>
                <p className="text-sm text-zinc-600">City Hospital → Metro Hospital</p>
              </div>
              <span className="px-3 py-1 rounded-full text-sm bg-green-100 text-green-800">In Progress</span>
            </div>
          </div>
        </div>
        <div className="bg-white p-6 rounded-lg shadow-md border border-zinc-200">
          <h2 className="text-xl font-semibold text-zinc-900 mb-4">Available Ambulances</h2>
          <div className="space-y-4">
            <div className="flex justify-between items-center p-3 bg-zinc-50 rounded">
              <div>
                <p className="font-medium text-zinc-900">Ambulance #A-123</p>
                <p className="text-sm text-zinc-600">Type: Advanced Life Support</p>
              </div>
              <span className="px-3 py-1 rounded-full text-sm bg-blue-100 text-blue-800">Available</span>
            </div>
          </div>
        </div>
      </div>
    </div>
  )
}
export default Admin
import React, { useState,useEffect} from 'react'
import { FiUploadCloud } from "react-icons/fi";
import api from '../api';
import { useNavigate,useParams } from 'react-router-dom';
const AmbulanceForm = () => {
  const [formData,setFormData] = useState({
    vehicleNumber:'',
    color:'',
```

```
    yearOfModel:",
    company:",
    book :null
});
const navigate = useNavigate();
const { id } = useParams();
useEffect(() => {
  const fetch = async () => {
    try {
      const res = await api.get(`/ambulance?vehicleNumber=${id}`);
      const { status, ...updatedData } = res.data;
      setFormData(prev => ({
        ...prev,
        ...updatedData,
      }));
      if (res.status === 200) {
        console.log("Ambulance Data Fetched");
      }
    } catch (err) {
      console.error("Found an error:", err);
    }
  }
  if (id) fetch();
}, [id])
const toBase64 = (file) =>
  new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.readAsDataURL(file);
    reader.onload = () => resolve(reader.result.split(',')[1]);    // This will return full base64
string including metadata
    reader.onerror = (error) => reject(error);
  });
  const handleChange = async (e) => {
    const { name, value, files } = e.target;
    if (name === "book" && files.length > 0) {
      const file = files[0];
      const base64 = await toBase64(file); // convert file to base64
      setFormData((prev) => ({ ...prev, [name]: base64 }));
    } else if (name === "yearOfModel") {
      setFormData((prev) => ({ ...prev, [name]: parseInt(value) }));
    } else {
      setFormData((prev) => ({ ...prev, [name.trim()]: value }));
    }
  };
  const handleAddAmbulance = async (e) => {
    e.preventDefault();
    try {
      const res = await api.post("/ambulance", formData, {
        headers: {
          "Content-Type": "application/json",
```

```
      },
    });
    console.log("Success:", res);
    if(res.status===200){
    navigate("/admin/ambulance");
    }
  } catch (err) {
    console.error("Found an error:", err.response?.data || err.message);
  }
};
const handleDelete = async () => {
  try {
    const res = await api.delete(`/ambulance?vehicleNumber=${id}`);
    console.log("Success:", res);
    if (res.status === 200) {
      console.log("Driver Deleted");
      navigate("/admin/ambulance");
    }
  } catch (err) {
    console.error("Found an error:", err);
  }
};
const handleUpdation = async (e) => {
  e.preventDefault();
  try {
    const res = await api.patch("/ambulance", formData, {
      headers: {
        "Content-Type": "application/json",
      },
    });
    console.log("Success:", res);
    if (res.status === 200) {
      console.log("Ambulance Updated");
      navigate("/admin/ambulance");
    }
  } catch (err) {
    console.error("Found an error:", err);
  }
};
return (
  <div className='flex-1  m-2 flex justify-center'>
     <div className='mt-8 md:m-8 w-[90vw] md:w-[50vw] flex flex-col gap-5'>
        <h1 className='text-4xl font-medium'>Add New Ambulance</h1>
        <div>
          <form
          onSubmit={id?handleUpdation:handleAddAmbulance}
          className='text-[1.1rem] text-zinc-600 font-medium flex flex-col gap-2'>
            <div className='w-full flex flex-col'>
               <label htmlFor="vehicleNumber " className=' m-1' >vehicle number </label>
               <input type="text" name="vehicleNumber " id="vehicleNumber"
```

```
                value={formData.vehicleNumber}
                 onChange={handleChange}
                 className='px-2 py-2 border-2 border-zinc-200 rounded-[5px]' />
              </div>
              <div className='w-full flex flex-col'>
                <label htmlFor="color" className='m-1' >Color</label>
                <input type="text"
                name="color" id="color"
                value={formData.color}
                onChange={handleChange}
                className='px-2 py-2 border-2 border-zinc-200 rounded-[5px]' />
              </div>
              <div className='w-full flex flex-col '>
                <label htmlFor="yearOfModel" className=' m-1' >Year of Model</label>
                <input type="number"
                value={formData.yearOfModel}
                onChange={handleChange}
                name="yearOfModel" id="yearOfModel"
                className='px-2 py-2 border-2 border-zinc-200 rounded-[5px]' />
              </div>
              <div className='w-full flex flex-col '>
                <label htmlFor="company" className=' m-1'>Company</label>
                <input type="text"
                onChange={handleChange}
                name="company" id="company"
                value={formData.company}
                className='px-2 py-2 border-2 border-zinc-200 rounded-[5px]' />
              </div>
              <div className='w-full flex flex-col '>
                <label htmlFor="license-upload" className=' m-1'>
                  Upload cbook
                </label>
                <label htmlFor="cbook-upload" className='border-2 border-dashed p-4 border-
zinc-200 rounded-[5px] flex justify-center'>
                  <div className='flex flex-col items-center gap-3'>
                    <FiUploadCloud size="40"/>
                    <div>Choose an image</div>
                    <div className='text-sm'>JPEG,JPG, and PNG formats</div>
                    <div className="btn w-fit p-3 border-2 rounded-2xl cursor-pointer">Browse
file</div>
                  </div>
                </label>
                <input
                  id="cbook-upload"
                  type="file"
                  onChange={handleChange}
                  name='book'
                  className='invisible'
                />
              </div>
```

```
            {formData.book&&<div            className='bg-zinc-300            p-2            rounded-
xl'>{formData.book.split("\\")[2]}</div>}
            <div className="flex justify-between">
           {id ? <button type="submit"
            className='mb-[30vw]      sm:my-3      text-2xltext-zinc-600      hover:bg-zinc-800
hover:text-[#fff] hover:font-normal hover:border-zinc-700 font-medium px-3 py-2.5 border-2
border-zinc-500 rounded-2xl w-fit cursor-pointer'>
            Update
           </button> :
            <button type="submit"
             className='mb-[30vw]      sm:my-3      text-2xltext-zinc-600      hover:bg-zinc-800
hover:text-[#fff] hover:font-normal hover:border-zinc-700 font-medium px-3 py-2.5 border-2
border-zinc-500 rounded-2xl w-fit cursor-pointer'>
              Add Ambulance
            </button>
           }
           {id && <button
            onClick={handleDelete}
            className='mb-[30vw]      sm:my-3      text-2xltext-zinc-600      hover:bg-zinc-800
hover:text-[#fff] hover:font-normal hover:border-zinc-700 font-medium px-3 py-2.5 border-2
border-zinc-500 rounded-2xl w-fit cursor-pointer'>
            Delete
           </button>}
          </div>
           </form>
          </div>
       </div>
    </div>
  )
}
export default AmbulanceForm
import React, { useEffect, useState } from 'react'
import { USERNAME,ROLE } from '../constants';
import api from '../api';
import { Navigate } from 'react-router-dom';
const AuthRedirectRoute = ({children}) => {
  const [isAuthenticated,setIsAuthenticated] = useState(null);
  const [role,setRole] = useState('');
  useEffect(()=>{
    auth();
  },[])
  const auth = async ()=>{
    try{
      const username = localStorage.getItem(USERNAME)||'';
      const role = localStorage.getItem(ROLE)||'';
      if(username===''  || role==='') {
      setIsAuthenticated(false);
      }else{
        setIsAuthenticated(true);
      }
```

```
    if(role==="ADMIN") setRole("admin");
      else if(role==="DRIVER") setRole("driver");
      else if(role==="USER") setRole("app");
   }catch(e){
    console.log("Found an Error : ",e);
   }
  }
  return isAuthenticated?<Navigate to={`/${role}`}/>:children;
}
export default AuthRedirectRoute
import React from 'react'
const Booking = ({address,addressError,handleBook,isConnected}) => {
  return (
    <div className="form mx-3 my-4 flex justify-center">
          <form action="" className='w-[90vw] m-5 p-2 flex flex-col sm:flex-row gap-5
justify-between'>
            <div className=' sm:w-[50vw]'>
              <h5 className='w-full text-[16px] text-zinc-500'>From Address</h5>
              <input type="text"
                name="fromAddress" id="fromAddress"
                placeholder='From '
                className='w-full px-2 py-2 border-2 border-zinc-200 rounded-[5px]'
                value={address} />
                <div className="error text-[15px] text-red-600">{addressError}</div>
            </div>
            {/* <div>
              <h5 className='text-[15px] text-zinc-500'>To Address</h5>
              <input type="text" name="toAddress" id="toAddress"
                placeholder='To'
                className='w-full px-2 py-2 border-2 border-zinc-200 rounded-[5px]' />
            </div> */}
            <button type="submit"
              onClick={(e)=>handleBook(e)}
              disabled={!isConnected}
              className='border-1 px-3 py-[8px] border-zinc-100 bg-zinc-800 text-xl text-zinc-
100 rounded-xl'>
              Book Ambulance
            </button>
          </form>
        </div>
  )
}
export default Booking
import React, { useState, useEffect,useRef } from 'react';
import {
  connectDriverSocket,
  sendDriverLiveLocation,
  disconnectDriverSocket,
  acceptRide,
  subscribeToAcceptance
```

```
} from '../utils/Socket';
import driveAmb from '../assests/Amb_driver.png'
import { TbAlertSquareRounded } from "react-icons/tb";
import { USERNAME } from '../constants';
import { ToastContainer,toast } from 'react-toastify';
import { MapContainer, TileLayer, Marker, Popup } from 'react-leaflet';
import RoutingMachine from '../components/RoutingMachine';
import 'leaflet/dist/leaflet.css';
const Driver = () => {
  const [incomingBooking, setIncomingBooking] = useState(null);
  const [location, setLocation] = useState(null);
  const [userLocation,setUserLocation] = useState([]);
  const [userId,setUserId] = useState(null);
  const [distance,setDistance] = useState(null);
  const username = localStorage.getItem(USERNAME);
  const alertsRef = useRef(null);
  useEffect(() => {
    if (window.location.hash === '#alerts' && alertsRef.current) {
      alertsRef.current.scrollIntoView({ behavior: 'smooth' });
    }
  }, []);
  useEffect(() => {
    const fetchLocation = async () => {
      try {
        if (navigator.geolocation) {
          navigator.geolocation.getCurrentPosition(
          (position) => {
              const { latitude, longitude } = position.coords;
              setLocation([latitude, longitude]);
            },
            (error) => {
              console.error("+ Error getting location:", error);
            }
          );
        } else {
          console.log('+ Geolocation is not supported by this browser.');
        }
      } catch (error) {
        console.error('+ Error getting location:', error);
      }
    };
    fetchLocation();
    return () => {
      clearInterval(interval);
      disconnectDriverSocket();
    }
  useEffect(() => {
    if (!location) return;
    connectDriverSocket({
    driverUsername: username,
```

```
    onError: (err) => alert("⚠ Location failed to save: " + JSON.stringify(err)),
    onAlert: (data) => {
      setUserId(data);
      const k = "New booking alert from "+data;
      toast.success(k);
      console.log("New booking alert from user:", data);
    },
    onConnect: () => {
      console.log("🟢 Connected as driver:", username);
    }
  });
  const interval = setInterval(() => {
    sendDriverLiveLocation({
    username:username,
      latitude: location[0],
      longitude: location[1]
    });
  }, 1000*60*2);
  return () => {
    clearInterval(interval);
    disconnectDriverSocket();
  };
 }, [username, location]);
 const handleRide = (e)=>{
 acceptRide({
    userUsername: userId,
    driverUsername: username,
    latitude: location[0],
    longitude: location[1]
  });
      subscribeToAcceptance({
    driverUsername:   username,
    onDriverUpdate: (data) => {
      setUserLocation([data.latitude,data.longitude]);
      setDistance(data.distance);
      console.log("User info received after accepting ride:", data);
      toast.success("Sent Acceptance request to user");
    }
  });
 }
 return (
  <div>
    <div className='sm:m-8 text-zinc-800 flex justify-center sm:p-3'>
      <div className="container max-w-[1200px] flex justify-between gap-5">
        <div className="banner w-full sm:h-[80vh] flex flex-col sm:flex-row justify-
between ">
          <div className='font-roboto sm:w-[35vw] px-5 flex flex-col ' >
            <div className='text-4xl sm:text-6xl pt-11 sm:pt-24 font-bold'> Be the hero
someone needs—on your schedule.</div>
            <div className='my-4 sm:my-2 sm:p-3 text-zinc-600 text-[1rem] sm:text-
```

```
[1.1rem] font-medium'>
                    Be There When It Counts
                  </div>
                  <div className='flex flex-col sm:flex-row sm:justify-baseline'>
                    <a
                    href='#alerts'
                    className='px-4 sm:px-4 rounded-2xl border-2 hover:bg-zinc-800 hover:text-
slate-100 py-4 flex items-center gap-6 text-zinc-600 decoration-solid cursor-pointer'>
                      Check Active Bookings
                    </a>
                  </div>
                </div>
                <div className=' px-3 py-5 sm:px-7  h-[80vh]' >
                  <img  src={driveAmb}  alt=""  className="w-[30vw]  h-full  object-cover
rounded-md"  />
                </div>
              </div>
            </div>
          </div>
           <div className='flex justify-center'>
            <div className='container max-w-[1200px] '>
              <div className=' flex flex-col min-h-[100vh]' id="alerts" ref={alertsRef} >
              <h2 className="text-6xl m-5">Driver Dashboard</h2>
              {userId && userLocation.length <= 0 ? (
                <div className="mt-4 px-5 py-4 shadow-md rounded-2xl w-[50vw]">
                  <div className='w-full text-end flex justify-end'>
                    <div className='bg-zinc-900 w-5 h-5 rounded-full'/>
                  </div>
                  <div className='flex gap-5 px-6 py-3 '>
                    <div className='text-2xl font-medium'>User :</div>
                    <div className="text-2xl">{userId}</div>
                  </div>
                  <div className='w-full text-end flex justify-end'>
                    <button
                      type="button"
                      onClick={handleRide}
                      className='w-fit px-4 py-2 bg-green-500 text-2xl rounded-2xl'>
                      Accept
                    </button>
                  </div>
                </div>
              ) : (
              <div>
                {(userLocation.length <= 0) && <div>
                  <div className="flex flex-col items-center justify-center text-center py-12 px-4">
                    <TbAlertSquareRounded  className="w-40 h-40 mb-6" />
                    <h2 className="text-xl font-semibold text-gray-800">You're all caught up!</h2>
                    <p className="text-gray-500 mt-2 mb-6">
                      No incoming bookings at the moment. Stay online to get notified instantly.
                    </p>
```

```
            <button className="bg-red-600 hover:bg-red-700 text-white px-5 py-2 rounded-lg">
              Refresh
            </button>
          </div>
          {/* <p className='flex sm:justify-center text-2xl sm:mt-24'>No incoming bookings yet.</p> */}
        </div>}
      </div>
      )}
    </div>
  </div>
  {/*  Route Map */}
  {location && userLocation.length > 0 && (
   <div className='my-6'>
    <MapContainer
     center={location}
     zoom={13}
     style={{ height: "400px", width: "80%", margin: "0 auto", borderRadius: "10px" }}
    >
     <TileLayer url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
     <Marker position={location}>
      <Popup>Driver Location</Popup>
     </Marker>
     <Marker position={userLocation}>
      <Popup>User Pickup Location</Popup>
     </Marker>
     <RoutingMachine from={location} to={userLocation} />
    </MapContainer>
   </div>
  )}
  <ToastContainer />
 </div>
);
};
export default Driver;
import React,{useState} from 'react'
import { Marker,Popup,useMapEvents} from 'react-leaflet'

function LocationMarker({setUserLocation}) {
  const [position, setPosition] = useState(null);
  const map = useMapEvents({
   click(e) {
    const { lat, lng } = e.latlng;
    setPosition(e.latlng);
    setUserLocation([lat,lng]);
    map.flyTo(e.latlng, map.getZoom());
   },
   locationfound(e) {
```

```
        setPosition(e.latlng)
        map.flyTo(e.latlng, map.getZoom())
      },
    })
  }
  export default LocationMarker;
import { useEffect } from "react";
import L from "leaflet";
import "leaflet-routing-machine";
import { useMap } from "react-leaflet";
const RoutingMachine = ({ from, to }) => {
  const map = useMap();
  useEffect(() => {
    if (!from || !to) return;
    const routingControl = L.Routing.control({
      waypoints: [L.latLng(from[0], from[1]), L.latLng(to[0], to[1])],
      routeWhileDragging: false,
      addWaypoints: false,
      draggableWaypoints: false,
      show: false,
      fitSelectedRoutes: true,
      createMarker: function (i, wp) {
        return L.marker(wp.latLng, {
        icon: L.icon({
            iconUrl:
              i === 0
                ? "https://cdn-icons-png.flaticon.com/512/10523/10523013.png"
                : "https://cdn-icons-png.flaticon.com/512/684/684908.png",
            iconSize: [32, 32],
            iconAnchor: [16, 32],
          }),
        });
      },
    }).addTo(map);

    return () => {
      map.removeControl(routingControl);
    };
  }, [from, to, map]);

  return null;
};

export default RoutingMachine;
import React, { useState, useEffect } from 'react'

import LocationMarker from './LocationMarker';
import Booking  from './Booking';
import RoutingMachine from './RoutingMachine';
```

```
import 'leaflet/dist/leaflet.css';
import '../styles/App.css'
import { MapContainer, TileLayer, useMap, Marker, Popup } from 'react-leaflet'
import { Icon } from "leaflet"
import LocomotiveScroll from 'locomotive-scroll';
import { ToastContainer,toast } from 'react-toastify';
import {USERNAME} from '../constants';
import { connectUserSocket ,bookRide,subscribeToAcceptance } from '../utils/Socket';
const UserHome = () => {
  const [userLocation, setUserLocation] = useState(null);
  const [address, setAddress] = useState("");
  const [addressError, setAddressError] = useState(null);
  const [markers,setMarkers] = useState([]);
  const username = localStorage.getItem(USERNAME);
  useEffect(() => {
    const scroll = new LocomotiveScroll();
  }, []);
  const CustomIcon = new Icon({
    iconUrl: "https://cdn-icons-png.flaticon.com/512/10523/10523013.png",
    iconSize: [48, 48]
  })
  const handleReverseGeocode = async (latitude, longitude) => {
    try {
      if (!latitude || !longitude) {
        setAddress('Please provide both latitude and longitude.');
        return;
      }
      const response = await fetch(

`https://nominatim.openstreetmap.org/reverse?lat=${latitude}&lon=${longitude}&format=js
on`
      );
      if (!response.ok) {
        setAddressError('Error fetching geocode data');
        throw new Error('Error fetching geocode data');
      }
      const data = await response.json();
      if (data.error) {
        setAddressError('No address found for these coordinates.');
      } else {
        const place = data.display_name;
        setAddress(place);
      }
    } catch (error) {
      console.error('Error fetching geocode data', error);
      setAddressError('Failed to fetch address.');
    }
  };
  useEffect(() => {
    const fetchLocation = async () => {
```

```
    try {
      if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
        (position) => {
            const { latitude, longitude } = position.coords;
            setUserLocation([latitude, longitude]);
          },
          (error) => {
            console.log(error);
          }
        );
      } else {
        console.log('Geolocation is not supported by this browser.');
      }

    } catch (error) {
      console.error('Error getting location:', error);
    }
  };

  fetchLocation();
}, []);

const MapFlyTo = () => {
  const map = useMap();
  useEffect(() => {
    if (userLocation) {
      map.flyTo(userLocation, 15, { animate: true });
    }
  }, [userLocation, map]);

  return null;
};
useEffect(() => {
  userLocation ? handleReverseGeocode(userLocation[0], userLocation[1]) : null;
}, [userLocation]);
const [drivers, setDrivers] = useState(null);
const [isConnected, setIsConnected] = useState(false);
const [driverInfo, setDriverInfo] = useState(null);
const [bookingSent, setBookingSent] = useState('');
const handleBook = (e) => {
  e.preventDefault();
    if (!isConnected) {
      alert("Please wait for WebSocket connection.");
      return;
    }
    bookRide({
      username:username,
      latitude:userLocation[0],
      longitude:userLocation[1],
```

```
    });
    toast.success("Your Booking is successful. ")
    setBookingSent(true);
  };
  const [driverLocation, setDriverLocation] = useState(null);
  useEffect(() => {
   connectUserSocket({
    username,
    onConnect: (frame) => {
     console.log('Connected to WebSocket', frame);
     setIsConnected(true);
     // 🟩 Safe to subscribe only after connection is established
     subscribeToAcceptance({
      userUsername: username,
      onUserUpdate: (data) => {
       console.log('🚨 User received driver acceptance:', data);
       toast.success(`Driver ${data?.Driverusername} accepted your ride!`);
       setDriverInfo(data);
       setMarkers([{
           geoCode: [data.latitude, data.longitude],
           PopUp: `${data.vehicleNumber}`
        }]);
       setBookingSent(false);
      },
     });
    },
    onError: (error) => {
     console.error('WebSocket error:', error);
     setIsConnected(false);
    },
    onMessage: (data) => {
     console.log('Drivers Data:', data);
     setDrivers(data);
    },
   });
  }, [username]);




  return (
   <div>
    {!bookingSent?( <div  className='relative sm:mx-8 text-zinc-800 flex justify-center p-3'>
     {/* Map Container */}
     <div data-scroll data-scroll-speed="-0.8" className="container max-w-[1250px] flex
justify-between gap-5">
     <MapContainer
      center={[17.6868, 83.2185]}
      zoom={10}
      scrollWheelZoom={true}
```

```
      className="map-container"
    >
      <TileLayer
        attribution="Ambulance Booking System"
        url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
      />
      {markers.map((marker, index) => (
        <Marker key={index} position={marker.geoCode} icon={CustomIcon}>
          <Popup>{marker.PopUp}</Popup>
        </Marker>
      ))}
      {userLocation && (
        <Marker position={userLocation}>
          <Popup>Your Location</Popup>
        </Marker>
      )}
      <LocationMarker setUserLocation={setUserLocation} />
      <MapFlyTo />
      {/* Add this for path display */}
      {userLocation && driverInfo && (
        <RoutingMachine
          from={userLocation}
          to={[driverInfo.latitude, driverInfo.longitude]}
        />
      )}
    </MapContainer>
    </div>
    {/* User Ambulance Booking */}
    <div     className="content absolute max-w-[1300px] bottom-5 left-1/2 transform -
translate-x-[50%] w-full h-[20vh] sm:h-[70px] z-[1000] ">
      <div className="h-[100vh] mt-7 bg-white rounded-t-3xl p-2 shadow-md z-[1000]">
        {/* line bar */}
        <div className="bar mt-4 flex justify-center items-center">
          <div  className="sm:w-[10%]  w-[25%]  border-t-6  border-zinc-500  rounded-2xl
"></div>
        </div>
        <div className="overflow-y-auto h-full">
          {driverInfo?(
            <div className=' w-[40vw]'>
              <div className="p-4  flex justify-between items-center">
                <div className="text-2xl ">Name:</div>
                <div className='text-xl' >{driverInfo.driverUsername}</div>
                </div>
              <div className="p-4  flex justify-between items-center">
                <div className="text-2xl">Vechile Number:</div>
                <div className='text-xl' >{ driverInfo.vehicleNumber}</div>
                 </div>
              <div className="p-4 flex justify-between items-center ">
                <div className="text-2xl">Phone Number:</div>
                <div className='text-xl' >{ driverInfo.phone}</div>
```

```
            </div>
            <div className="p-4 flex justify-between items-center">
              <div className="text-2xl">Distance:</div>
              <div className='text-xl'>{ driverInfo.distance} </div>
              </div>
            </div>
          ):(
            <div>
              {/* Booking Form */}
              <Booking address={address} addressError={addressError}
handleBook={handleBook} isConnected={isConnected}/>
              {/* Near By Amblances */}
              <div className='mx-2 sm:mx-8'>
                <h1 className='text-3xl'>Near by Ambulances</h1>
                {drivers ?(
                  <div className="shadow-md my-3 mx-1 p-3 rounded-2xl flex gap-4">
                  <div className="img w-[80px] h-[80px] bg-zinc-400 rounded-full "></div>
                  <div className="ambulance-details">
                    <h3 className='text-xl'>Driver Name : {drivers}</h3>
                    <div className="ambulance-no">Amubulance no : ######</div>
                    <div className="location">Hyderabad</div>
                  </div>
                </div>
                ): <div className='text-2xl font-medium text-center my-6 py-6'>There are no
nearby Ambulances</div>}
              </div>
              </div>
          )}
        </div>
      </div>
    </div>
  </div>):( <div className="m-4 p-4 border border-blue-300 bg-blue-50 rounded shadow">
  <h3 className="text-blue-600 font-semibold">Booking Request Sent</h3>
  <p className="mt-2">Please wait while we connect you to the nearest available
driver.</p>
  <p className="text-sm text-gray-500 mt-1">You'll receive driver details once your
request is accepted.</p>
  <div className="mt-4 animate-pulse text-blue-500">
    <span className="inline-block w-2 h-2 bg-blue-500 rounded-full mr-1"></span>
    <span className="inline-block w-2 h-2 bg-blue-500 rounded-full mr-1"></span>
    <span className="inline-block w-2 h-2 bg-blue-500 rounded-full"></span>
  </div>
 </div>)}
 <ToastContainer />
  </div>
 )
}

export default UserHome;
import React from 'react'
```

```
import { FaHospitalAlt, FaShieldAlt, FaCity, FaAmbulance } from 'react-icons/fa'

const audiences = [
  {
    icon: FaHospitalAlt,
    title: 'Hospitals',
    desc: 'Seamless patient handoff with real-time ambulance data.'
  },
  {
    icon: FaShieldAlt,
    title: 'Emergency Responders',
    desc: 'Smart dispatching and routing for faster on-ground coordination.'
  },
  {
    icon: FaCity,
    title: 'Government Bodies',
    desc: 'Monitor city-wide emergency vehicle activity and optimize resource allocation.'
  },
  {
    icon: FaAmbulance,
    title: 'Private Ambulance Operators',
    desc: 'Manage fleets efficiently and build trust with real-time updates.'
  }
]

const WhoIsItFor = () => {
  return (
    <section className="bg-zinc-900 py-12 px-4 md:px-0">
      <div className="max-w-5xl mx-auto text-center mb-10">
        <h2 className="text-3xl md:text-4xl font-bold text-zinc-100 mb-2">Who Is It For?</h2>
        <p className="text-zinc-50 text-lg">Our system is tailored for every part of the emergency response chain.</p>
      </div>
      <div className="grid grid-cols-1 sm:grid-cols-2 gap-8 max-w-4xl mx-auto">
        {audiences.map((aud, idx) => (
          <div
            key={idx}
            className="bg-white border border-zinc-200 rounded-xl p-7 flex flex-col items-center shadow-sm transition-transform hover:scale-105 hover:shadow-lg duration-200"
          >
            <div className="bg-zinc-100 p-4 rounded-full mb-4">
              <aud.icon className="text-zinc-700 text-3xl" />
            </div>
            <h3 className="text-xl font-semibold text-zinc-900 mb-2">{aud.title}</h3>
            <p className="text-zinc-600 text-base">{aud.desc}</p>
          </div>
        ))}
      </div>
    </section>
```

```
  )
}

export default WhoIsItFor
import React from 'react'
import Navbar from '../components/Navbar.jsx'
import Banner from '../components/Banner.jsx'
import Feature from '../components/Feature.jsx'
import Footer from '../components/Footer.jsx'
import WhoIsItFor from '../components/WhoIsItFor.jsx'
import Testimonials from '../components/Testimonials.jsx'
const Home = () => {
  return (
    <div>
      <Navbar/>
      <Banner/>
      <Feature/>
      <WhoIsItFor />
      <Testimonials />
      <Footer/>
    </div>
  )
}
export default Home
import React from 'react'
import UserNav from '../components/UserNav'
import UserHome from '../components/UserHome'
const User = () => {
  return (
    <div>
        <UserNav/>
        <UserHome/>
    </div>
  )
}

export default User
```

6. SCREENSHOTS

Home

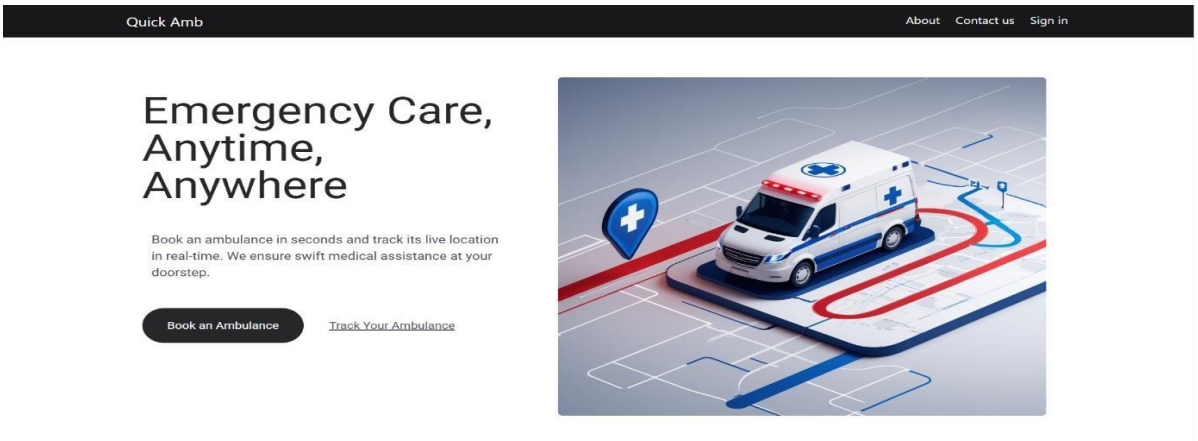

Figure:15 Home Page

Sign up

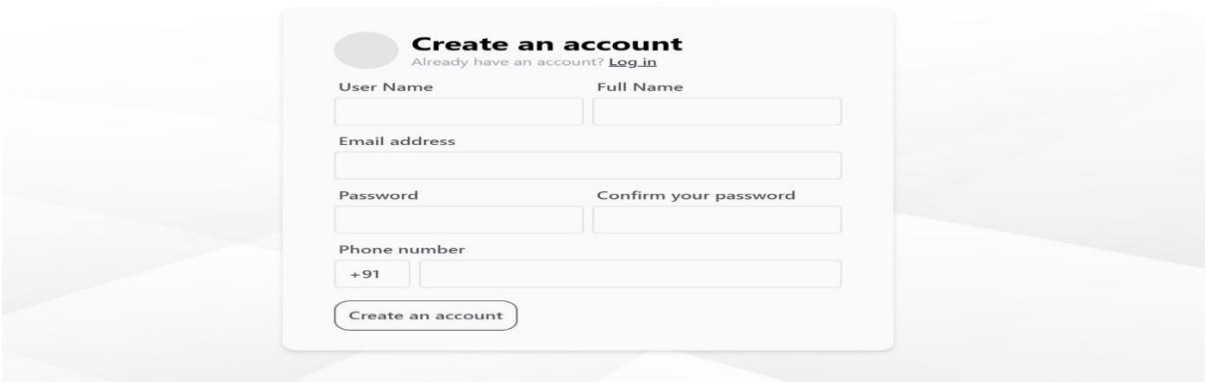

Figure:16 Sign up page

Login

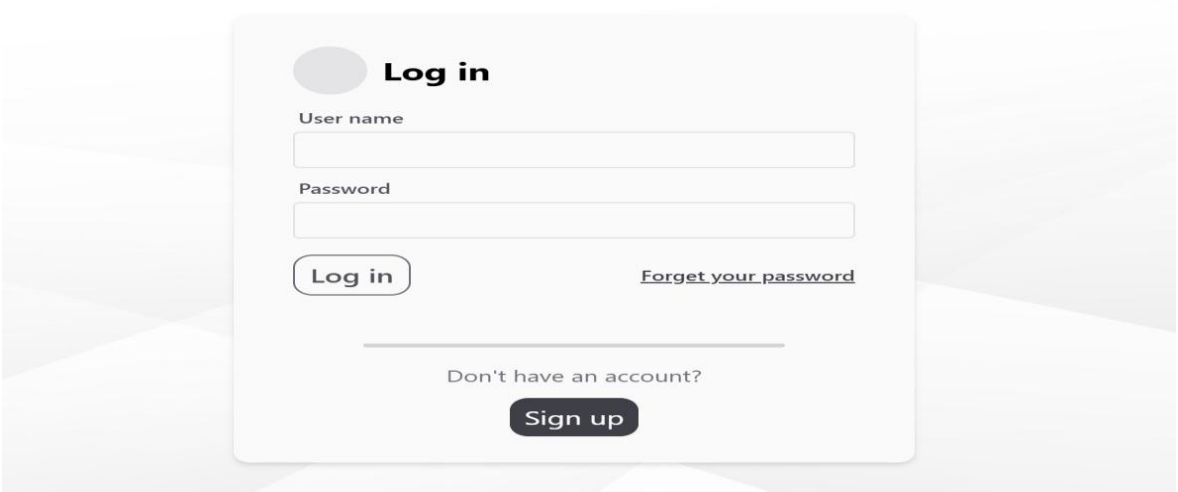

Figure:17 Login Page

Admin Dashboard

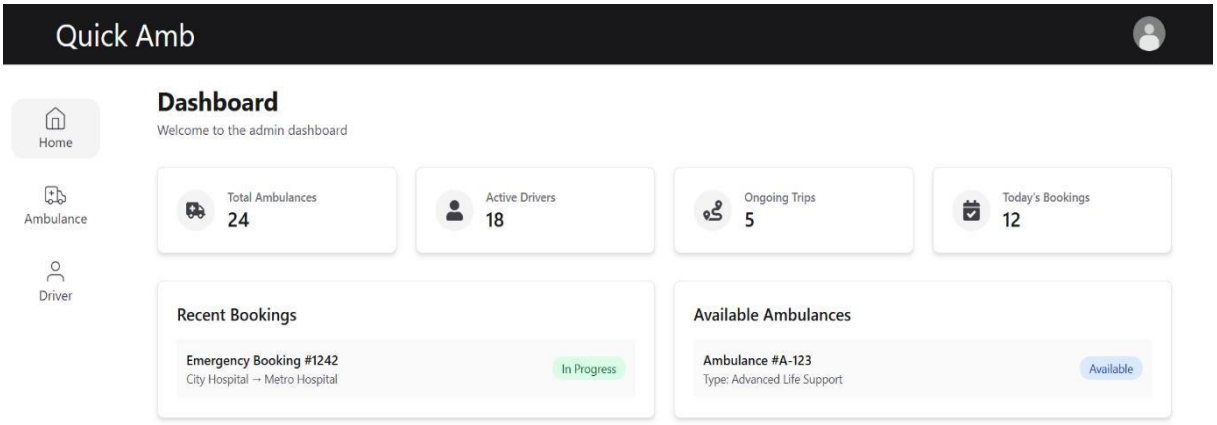

Figure:18 Admin Dashboard

Add Ambulance

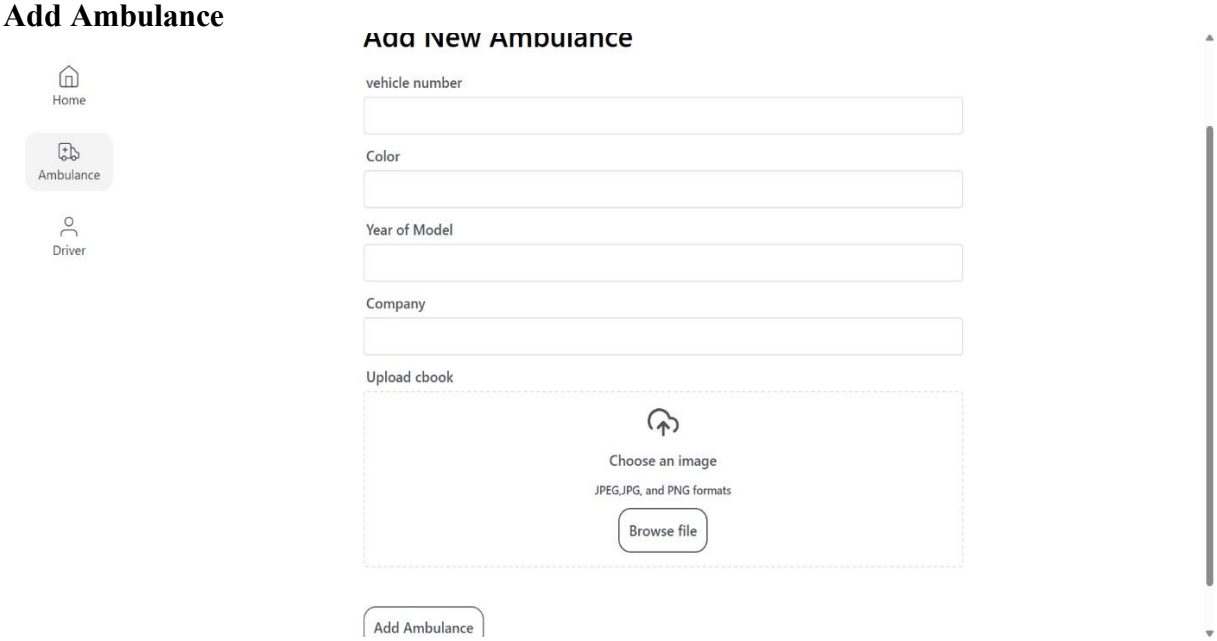

Figure:19 Add Ambulance

Add Driver

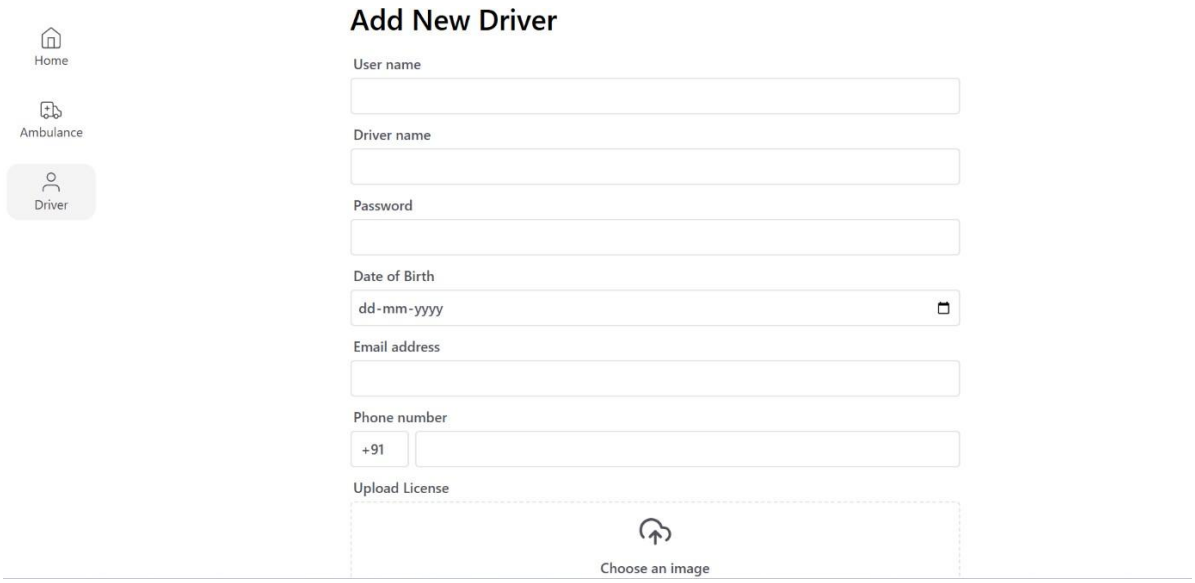

Figure:20 Add Driver

Drivers List

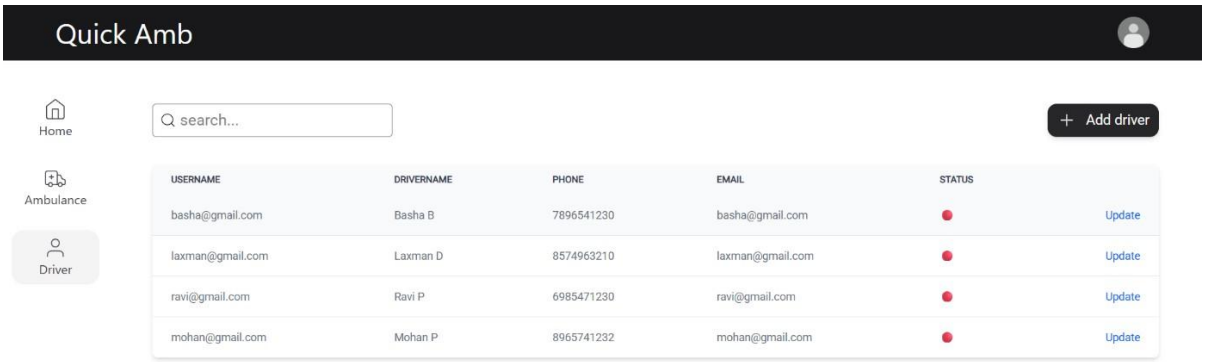

Figure:21 List of Drivers

Ambulance List

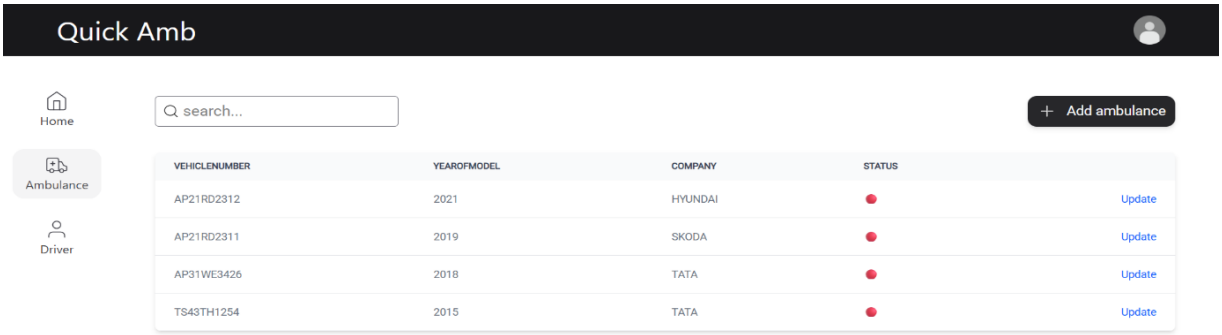

Figure:22 List of Ambulances

User Live Location

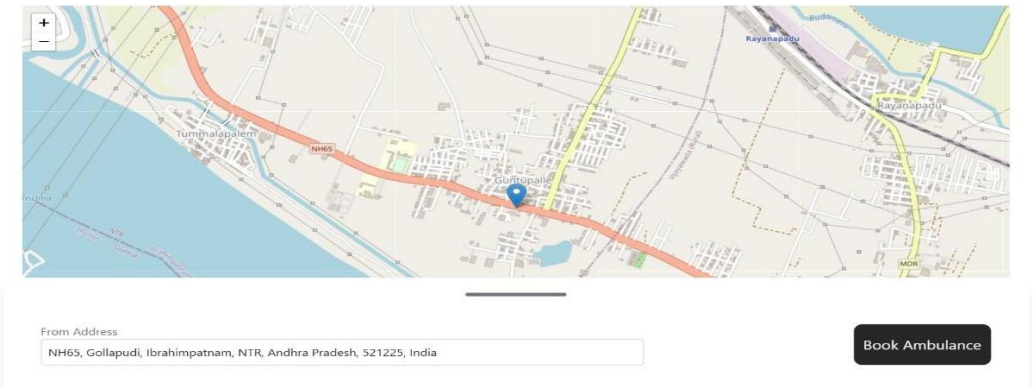

Figure:23 User Live Location

All Available Ambulances List

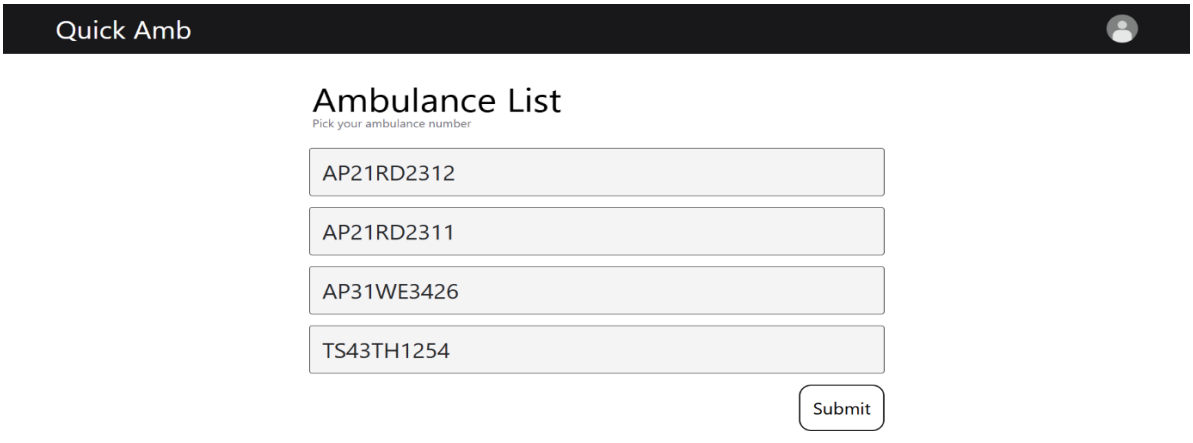

Figure:24 Available Ambulances

Request Notification

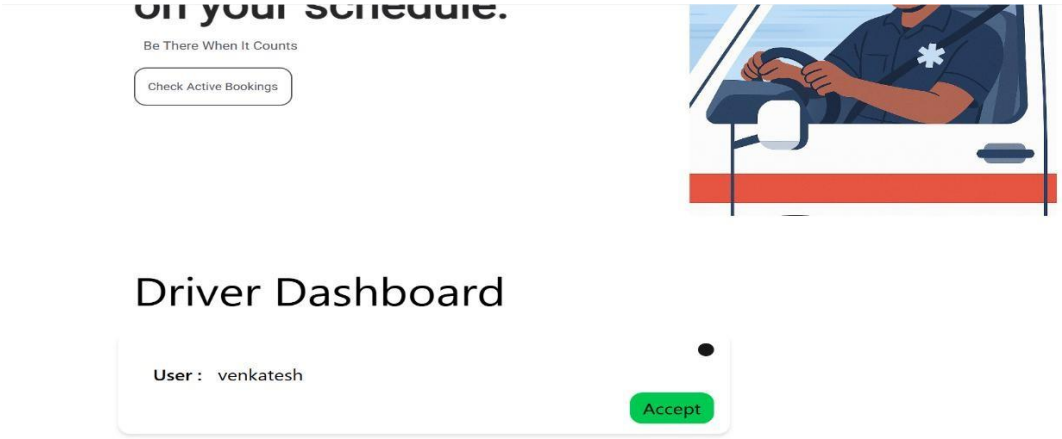

Figure:25 Notifications

Send Booking Request

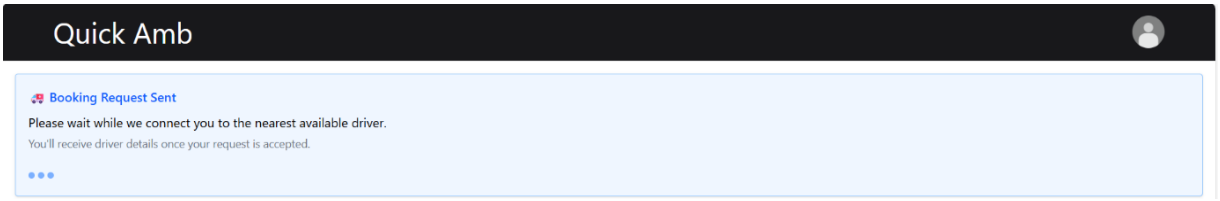

Figure:26 Send Request

Driver-User

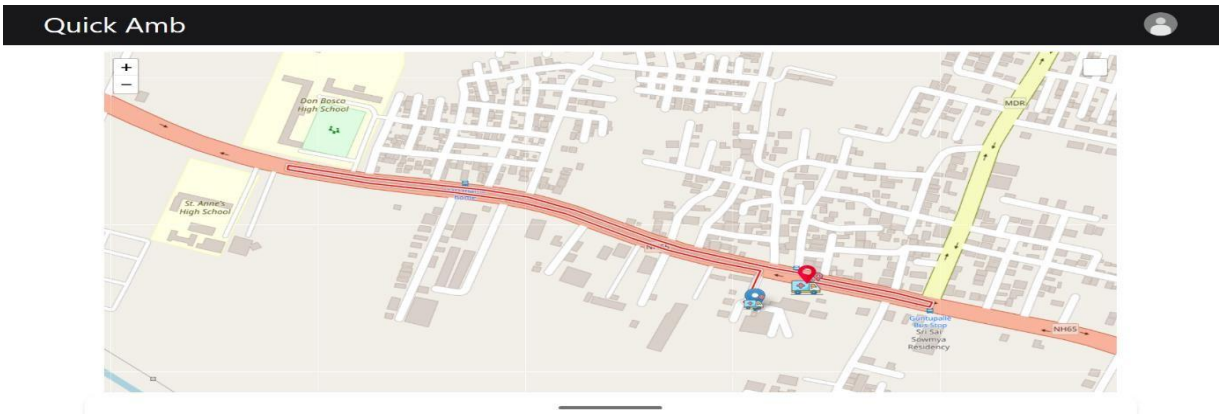

Figure:27 Driver-User

User-Driver

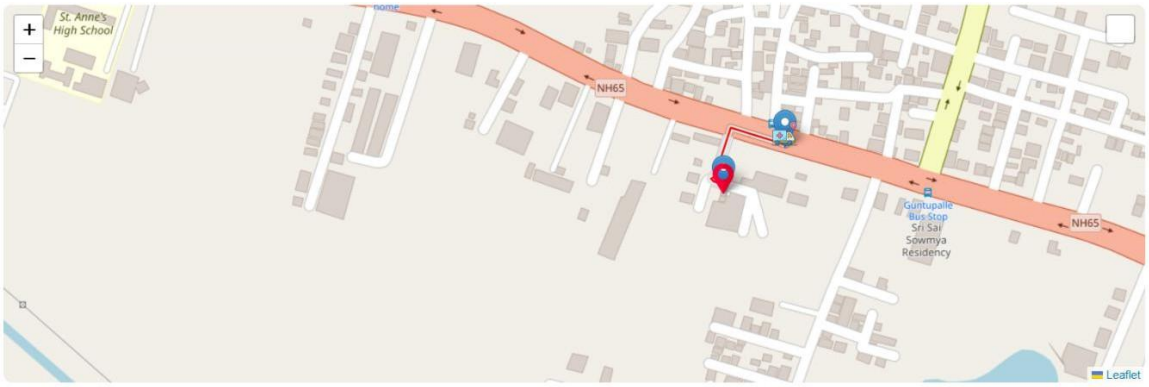

Figure:28 User-Driver

7. SYSTEM TESTING

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, subassemblies, assemblies and/or a finished product It is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

System testing in the Quick Ambulance project ensures that all integrated components—frontend, backend microservices, database, and WebSocket communication—work together as expected in a real-world environment. This phase validates the entire system's functionality, security, and performance by simulating real user actions, such as user login/logout, ambulance booking, live tracking, and driver acceptance. The system is tested end-to-end, including role-based access (Admin, User, Driver), token-based authentication, and interaction between microservices via the API Gateway and Eureka Server. Special focus is given to real-time updates, session handling, and data synchronization. The goal is to confirm that all modules perform correctly and meet the specified requirements before deployment.

7.1. UNIT TESTING

Unit testing involves testing individual components or functions in the application in isolation. Each function, class, or method is tested independently to ensure it performs correctly on its own. It usually targets the smallest pieces of functionality, such as a service method that calculates the estimated arrival time or fetches available ambulances within a certain radius.

In the Quick Ambulance system, unit testing would be applied to service layer logic, repository interfaces, utility functions, and controller-level endpoints. For instance, the function that checks if an ambulance is available within 20 km or the method that fetches a user's role after login would be ideal for unit tests. Frameworks like JUnit and Mockito are typically used to simulate inputs and test behaviours without needing the full system.

Unit testing helps developers catch bugs early, speeds up development by validating code

during changes, and allows safe refactoring. It improves code quality, increases developer confidence, and contributes to a maintainable and scalable system by ensuring that each piece of the application works as intended before integrating it with others.

7.2. INTEGRATION TESTING

Integration testing focuses on how different components of the system work together. Rather than testing isolated methods, it evaluates if modules like the database, backend logic, and APIs collaborate correctly. In a microservice architecture, this type of testing ensures that services like LoginService, BookingService, and NotificationService interact without failure.

For example, integration tests in Quick Ambulance might validate whether the driver data fetched from the DriverService is correctly passed to the BookingService, or whether the location tracking module sends accurate data to the frontend. REST endpoints and WebSocket channels are common targets for integration testing, especially where inter-service data flow is critical.

This testing is crucial because, even if individual units work, their integration might produce unexpected issues due to serialization, data format mismatches, or communication failures. Integration testing ensures seamless coordination between modules and that the system functions well in real-time situations like ambulance booking and tracking.

7.3. SYSTEM TESTING

System testing evaluates the application as a whole to verify that it meets the requirements and works correctly in a complete and integrated environment. It tests the overall system functionality, performance, reliability, and user interaction to confirm everything works from start to finish.

In Quick Ambulance, system testing would simulate the entire user journey—from registration and login to ambulance booking and live driver tracking. It includes all components like the frontend (React), backend (Spring Boot), database (MySQL), WebSocket communications, and Google Maps integration. The objective is to ensure the system performs as expected under normal usage conditions.

System testing is the final gate before deployment. It provides confidence that the product fulfils user needs, is stable under real-world use, and supports full functionality. It can reveal issues not caught during unit or integration tests and is critical for verifying end-to-end workflows in emergency medical scenarios.

7.4. FUCTIONAL TESTING

Functional testing verifies that each feature behaves according to the specified requirements. It involves providing specific inputs and validating whether the outputs match expectations. These tests are generally based on user stories or use cases and often performed manually or using automation tools.

For this project, functional testing would cover scenarios like user login, ambulance request submission, driver acceptance, and real-time updates. It would also include verifying admin actions such as adding ambulances, viewing logs, and managing users. Each action is treated as a function to test whether it performs accurately in different conditions.

Functional testing ensures that business logic is correctly implemented and user-facing features work as promised.

7.5. PERFORMANCE TESTING

Performance testing assesses how well the application performs under expected and extreme load conditions. It evaluates metrics like response time, system throughput, server CPU and memory usage, and scalability. This type of testing is critical for applications that expect real-time responses.

In Quick Ambulance, performance testing would examine how quickly booking requests are processed, how the system behaves when multiple users are connected via WebSocket, or how long it takes to track and update ambulance locations. It also evaluates backend service response times and database read/write performance under load.

A performance-tested system avoids slow response times, lag in location updates, or timeouts during booking. This is essential for healthcare-related applications, where delays can be life-threatening. Tools like Apache JMeter or Gatling can simulate concurrent users and monitor how the app behaves under real-world pressure.

7.6. SECURITY TESTING

Security testing is a critical phase in the Quick Ambulance project to ensure that the entire system—including all microservices and communication layers—is protected from potential threats and vulnerabilities. Since the platform deals with sensitive information like personal details, real-time locations, and medical emergency data, a robust security framework is implemented and thoroughly tested. JWT (JSON Web Token)-based authentication is tested to verify secure login/logout processes for users and drivers, token integrity, proper expiration, and signature validation. Penetration testing is conducted to detect and prevent

vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). The API Gateway is tested for proper request filtering, route protection, and security header enforcement. Additionally, role-based access control (RBAC) is validated to ensure that only authorized users can perform specific actions—like admin access to driver management or users booking an ambulance. Encryption mechanisms for storing credentials and transmitting data are verified to maintain confidentiality and integrity. WebSocket channels, used for live location sharing, are tested for secure communication to prevent unauthorized interception or tampering. All these measures contribute to making the system compliant with security best practices and ensure the protection of user data in real-time emergency scenarios.

7.7. USER ACCEPTANCE TEST

User Acceptance Testing (UAT) is conducted by the end users or stakeholders to confirm that the system meets real-world expectations and business requirements. It focuses more on usability, accuracy, and whether the application delivers value as intended.

In Quick Ambulance, UAT would involve admins, patients, and ambulance drivers interacting with the system to simulate real-life scenarios. For example, a user would try booking an ambulance, a driver would respond to an alert, and an admin would monitor requests. Feedback from this testing is used to fix usability issues or make adjustments before deployment.

UAT provides final approval before launch. It ensures that the system not only works technically but also provides a good user experience, aligns with workflow expectations, and is ready for public use in emergencies.

7.8. REGRESSION TESTING

Regression testing involves re-running test cases after a code update or bug fix to ensure that previously working functionalities still operate correctly. It helps identify unintended side effects of code changes.

For Quick Ambulance, regression testing might involve re-testing login functionality, ambulance booking, and driver acceptance after adding a new notification feature or modifying database schema. Automated test suites can be used to speed up this repetitive but essential process.

Maintaining application stability over time is vital for user trust, especially in a life-critical system.

7.9. COMPATIBILITY TESTING

Compatibility testing checks whether the application behaves consistently across different devices, operating systems, screen sizes, and browsers. It ensures a uniform experience for all users.

For the Quick Ambulance system, this means testing the React frontend on Chrome, Firefox, Edge, Safari, and mobile browsers. It also includes verifying WebSocket functionality on Android and iOS and ensuring GPS tracking works across devices. Backend services must also be tested for compatibility with deployment environments.

By performing compatibility testing, you reduce user complaints due to browser bugs or inconsistent UI behavior. In a system that might be used on-the-go via mobile during emergencies, cross-device functionality is a must.

7.10. DATABASE TESTING

Database testing ensures that data operations (insert, update, delete, retrieve) are accurate, secure, and reliable. It also verifies database schema integrity, primary and foreign key relationships, indexing, and stored procedures.

In this project, database testing includes checking that booking records are correctly stored, deleted on cancellation, and updated on driver acceptance. It also verifies foreign key relationships between user and ambulance tables, uniqueness of booking IDs, and performance of frequently used queries.

Database testing is crucial in ensuring the consistency and accuracy of medical and location data, which form the backbone of the Quick Ambulance system. It guarantees that the backend supports smooth operation and avoids data corruption or integrity issues.

8. CONCLUSION AND FUTURE SCOPE

8.1. CONCLUSION

The **Quick Ambulance System** represents a powerful, real-time solution designed to address the critical need for fast and reliable emergency medical transportation. Built using microservices architecture, live location tracking, WebSocket communication, and a secure authentication system, the platform ensures quick response times, transparent booking, and seamless coordination between users, drivers, and administrators. By leveraging modern technologies such as Spring Boot, React, MySQL, and WebSockets, the application enhances both operational efficiency and user experience.

This system is not only suitable for hospitals and private ambulance operators but also highly scalable for use by governments in public health management, especially during crises or large-scale emergencies. With features like driver alerting, distance-based filtering, and real-time data exchange, Quick Ambulance significantly improves ambulance dispatch accuracy and helps reduce critical response times. Moreover, the system provides robust tools for administrators to monitor fleet health, log system activity, and analyze performance over time.

In conclusion, the Quick Ambulance system stands as a practical, scalable, and life-saving technological innovation. It showcases how smart systems and real-time tracking can transform traditional healthcare logistics into an efficient and responsive service. With future enhancements such as AI-based dispatch prediction, multilingual support, or integration with smart city infrastructure, the system can become an even more vital part of emergency healthcare ecosystems around the world.

8.2. SCOPE FOR FUTURE ENHANCEMENT

The Quick Ambulance system holds immense potential for expansion through several advanced and essential future enhancements. One of the core improvements includes integrating **real-time WhatsApp and SMS alerts** to notify users and drivers about booking confirmations, live locations, and emergency updates—even without internet access—enhancing communication and response speed. To strengthen security and user experience, implementing **OAuth 2.0-based authentication** will allow login through trusted platforms like Google, Facebook, or national identity services, making the system both secure and user-friendly. A powerful upgrade is the addition of **real-time hospital locator services**, helping

users identify the nearest hospital with the shortest travel time, current bed availability, and critical care support. Complementing this, displaying the **availability of doctors by specialization in each hospital** would allow ambulances to route patients to the most suitable facility based on their emergency needs. The system can also expand its accessibility by supporting **multilingual interfaces and voice command capabilities**, especially beneficial for users in rural regions or those with disabilities. Advanced **AI-based dispatch algorithms** can be integrated to auto-assign ambulances based on proximity, driver availability, traffic data, and emergency severity, thus reducing response time. The introduction of an **SOS emergency mode** will let users book ambulances with a single tap or voice command, while also notifying emergency contacts and hospitals instantly. **Integration with wearable health devices** can enable real-time transmission of vital signs like heart rate and blood pressure directly to hospitals, preparing them in advance for treatment. A centralized **hospital dashboard** will allow emergency rooms to update bed status, doctor availability, and handle bookings efficiently. For system admins, **advanced analytics** can track usage patterns, performance metrics, and regional demand, aiding data-driven decisions. A **driver rating and performance system** can encourage better service delivery and accountability. On the technical side, **fleet monitoring with GPS and telematics** will track ambulance fuel levels, maintenance needs, and usage stats. The system can incorporate **cloud synchronization** for real-time data access across multiple hospitals and regions. A **multi-tier admin portal** will allow hospital-level, city-level, and national-level views and control of ambulance operations. **Offline booking support**, through lightweight apps or SMS-based services, ensures functionality in low-network zones. **Web portals for patient relatives** can display live tracking and driver details. Integration with **smart traffic systems** can allow ambulances to request traffic signal clearance for faster movement. Future versions may include **inter-state or district ambulance dispatch**, making the system scalable across regions. In emergency logistics, **drone support for critical supply delivery** could also be considered. Compliance with **data privacy regulations** will be enhanced through role-based access control, secure audit logs, and token management. This system, designed with a modular microservices architecture, is highly adaptable for public healthcare, private providers, and smart city initiatives, positioning it as a next-generation, technology-driven emergency response platform.

9. REFERENCES

REFERENCES

1. **Spring boot 3.4.3 Documentation -Spring Boot 3.4.3 available now**
2. **Spring boot Data JPA–** Spring Data JPA
3. **Spring boot Security -** Spring Security
4. **Spring web socket -** WebSockets :: Spring Framework
5. **Microservices -** Spring | Microservices
6. **JWT –** JSON Web Tokens - jwt.io
7. **React JS -** React
8. **MySQL Official Website** – MySQL :: MySQL Documentation
9. **Bootstrap Documentation** – https://getbootstrap.com
10. **W3Schools HTML/CSS/JS Tutorials** – https://www.w3schools.com
11. **Stack Overflow** – https://stackoverflow.com (for resolving development issues)
12. **GitHub** – https://github.com (for open-source examples and code references)