My Riding App

Design Document

Case Study 2: Ride-Hailing Solution

Submitted by:

ABDUR REHMAN

Submitted to:

jeeny

Contents

Chapter 1

Tech Stack

1.1 Tech Stack Used

The application was built using **React.js for the frontend** and **Node.js for the backend**, with additional tools like Socket.IO and JWT. These technologies were chosen because I had already used them during my internship and final year project (FYP), so I was familiar with how they work in real-world web applications.

1.1.1 Frontend — React.js

React.js was used for the frontend because it is based on reusable components, makes the user interface fast using a virtual DOM, and helps build responsive and easy-to-manage web pages. It was used to create the main screens for both passengers and captains.

1.1.2 Backend — Node.js with Express.js

Node.js and Express.js were selected for the backend because they handle many user requests at the same time efficiently. Since I had worked with them in my previous projects, it made the development process faster and easier to manage.

1.1.3 Database — MongoDB with Mongoose

MongoDB was used to store data like users, captains, vehicles, and rides. It allows flexible and dynamic data storage. Mongoose was used to define and validate the structure of that data.

1.1.4 Realtime Communication — Socket.IO

Socket.IO was added to support live communication between users and captains. It was important for features like sending ride requests instantly to nearby captains.

1.1.5 Authentication and Security

JWT (JSON Web Tokens) was used to handle login and authentication without saving session data on the server. Passwords were hashed using Bcrypt for security. An OTP system was also added to make sure the ride only starts when the captain reaches the passenger and gets the OTP from them. This confirms that the correct captain is at the right location.

1.1.6 Functioning Code Context

According to the allowed options, this project uses a **web-based frontend with React.js** and a **backend-only API using Node.js and Express**. This setup separates the user interface from the backend logic, which is a common structure in modern applications.

Chapter 2

Assumptions

2.1 Assumptions Made

During the development of this project, the following assumptions were made to simplify the system and focus on its core features:

- The application can run without an internet connection because mock data is used instead of live APIs like Google Maps.
- Location, distance, and fare calculations are estimated using static or predefined values, not based on real-time GPS data.
- A captain can accept only one ride at a time, and a user can request only one ride at a time.
- The OTP system works offline — the OTP is generated when the ride is created, and the user shares it with the captain only after the captain reaches their location.
- All necessary data (users, captains, rides) is stored locally using MongoDB, and no external APIs or cloud services are used.
- The user and captain interact through the application interface without needing third-party services like Google Maps, Twilio, or Firebase.
- A payment method button is included in the interface, but it is not connected to any real payment system; it is assumed that the user pays the fare directly.

- Both the frontend and backend are assumed to run on the same machine or local network during development.
- No deep performance or security testing is done; features like JWT and Bcrypt are used to demonstrate secure design only.

Chapter 3

Entity Relationship Diagram

3.1 Data Model Overview

The following diagram shows the overall data model of the system. It illustrates how users, captains, and rides are related in the backend database design. The relationships and data structures are visually represented in **Figure** 3.1.

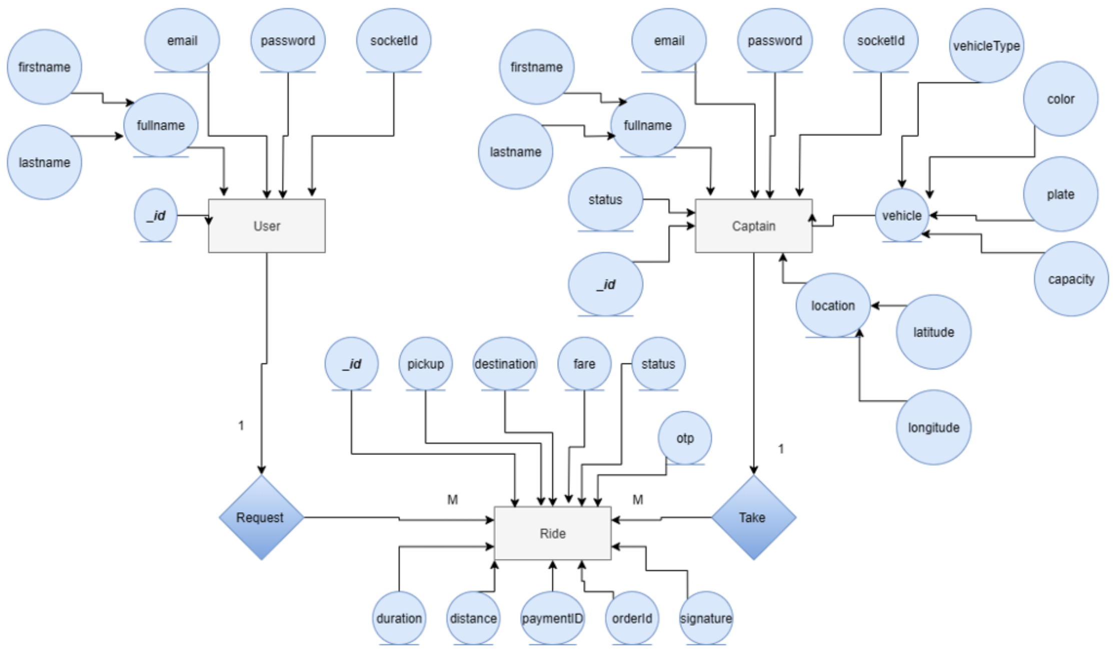

FIGURE 3.1: Entity Relationship Diagram for the Ride Booking Application

3.2 ERD Explanation

The Entity Relationship Diagram (ERD) illustrates how data is organized in the ride-booking system. It highlights the relationships between the main entities: **User**, **Captain**, and **Ride**. Below is a detailed explanation of each entity and how they are connected.

3.2.1 User Entity

The `User` entity represents passengers who can log in to the application, request rides, and interact with captains. Each user has the following key attributes:

- **Full Name** (First and Last Name)
- **Email Address**
- **Password**
- **Socket ID** (for real-time communication)

A user can book multiple rides, establishing a **one-to-many relationship** with the `Ride` entity.

3.2.2 Captain Entity

The `Captain` entity represents drivers who can accept ride requests. Each captain has personal and vehicle-related information, including:

- **Full Name** (First and Last Name)
- **Email Address**
- **Password**
- **Socket ID**
- **Status** (active or inactive)
- **Vehicle Information**:
  - Vehicle Type (car, auto, motorcycle)
  - Color

  - Plate Number
  - Capacity
- **Location** (Latitude and Longitude)

A captain can complete many rides, resulting in a **one-to-many relationship** with the `Ride` entity.

3.2.3 Ride Entity

The `Ride` entity represents a transaction between a user and a captain. It stores all information about the ride such as:

- **User ID** (foreign key referencing `User`)
- **Captain ID** (foreign key referencing `Captain`)
- **Pickup and Destination Locations**
- **Fare**
- **Status** (pending, accepted, ongoing, completed, cancelled)
- **OTP Code** (for secure ride start)
- **Duration and Distance**
- **Payment Information**

3.2.4 Relationships Summary

- One **User** can request multiple **Rides**.
- One **Captain** can complete multiple **Rides**.
- Vehicle and location details are embedded within the `Captain` entity.

This ERD structure supports the main business logic of the ride-booking system: from ride creation and driver assignment to OTP verification and ride completion.