PROJECT REPORT

INDEX

INTRODUCTION

Project title - RideReady

Team members-

1. Aditya Dhakarwal(Frontend developer)

2.Aniruddha bhattacharjee(Frontend developer)

3.S Raghuraj(Backend developer)

4.Soumil Paseband(Backend developer)

Project Overview

- **Purpose:**

In today's fast-paced world, commuters often struggle with finding reliable, quick, and convenient transportation, especially during peak hours or in unfamiliar locations. Traditional taxi services may be unreliable or inconsistent, and many people face issues like:

- Long wait times for rides
- Lack of real-time tracking
- Unclear communication between drivers and riders
- Uncertain ride availability, especially in remote areas

RideReady was developed to tackle these problems by offering a smart, real-time cab booking solution. The app aims to simplify ride-hailing by providing a fast, secure, and intuitive platform where users can book nearby drivers, track them live, and get to their destination with ease.

RideReady bridges the gap between convenience and technology, offering a modern, efficient alternative to traditional ride services by combining real-time communication, geolocation, and user-friendly design.

- **Key Features:**

- User Authentication & Role Management:
  Secure login/signup with role-based access for riders and drivers.

- Live Location Tracking:
  Integrates geolocation and mapping APIs to track driver and user locations in real-time.

- Ride Request & Response Flow:
  Users can request rides which are broadcasted to available drivers who can accept or decline them.

- Interactive Mapping:
  Real-time maps show pickup/drop-off locations, route paths, and dynamic updates using Map APIs.

- Real-Time Communication with Socket.io:
  All ride status updates – request sent, accepted, ride started, ride ended – happen live via sockets.

- Responsive, Fast UI with Tailwind CSS & Vite:
  Provides a mobile-friendly and visually clean experience using the latest frontend tools.

- Modular Backend Architecture with Express.js:
  Handles all REST APIs, real-time socket events, and database operations efficiently.

- MongoDB Integration:
  All user data, ride history, and session data are stored in a flexible, scalable NoSQL database.

ER Diagram-

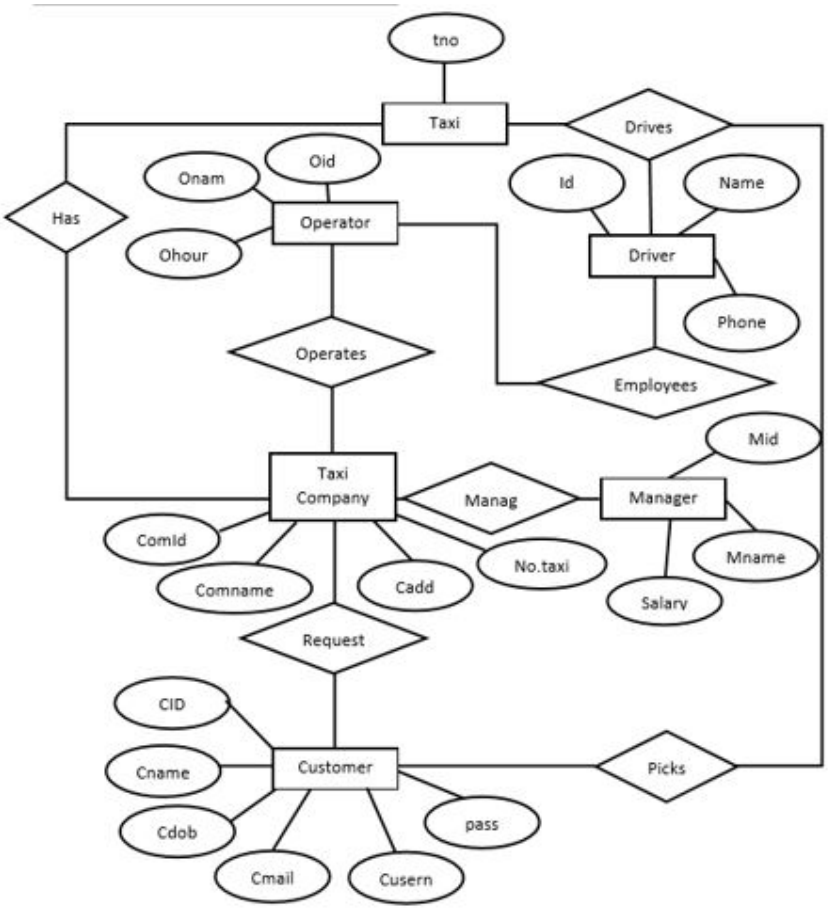


Figure 2 ER Diagram of Taxi Booking System

Architecture

RideReady is built on the powerful and modern **MERN stack**—a full-stack JavaScript solution that enables the development of robust, maintainable, and scalable applications. It combines React on the frontend, Node.js and Express.js on the backend, MongoDB for the database, and Socket.io for real-time capabilities. This architecture allows for seamless communication between all layers of the application while supporting real-time features and a responsive user experience.

□ Frontend Architecture (React + Vite + Tailwind CSS)

The frontend is developed using **React**, a component-based library that allows for the creation of reusable UI elements. It uses **Vite** as the build tool, which significantly improves development speed through faster hot-module replacement and optimized builds.

- **Component-Based Design:**
  Pages like the login screen, ride request form, driver dashboard, and real-time map view are all built using modular components. This promotes reusability and easy maintenance.

- **Routing & State Management:**
  The app handles different views for users and drivers, with routing managed either via `react-router-dom` or conditional rendering. State is managed locally and also via socket events.

- **Real-Time Updates with Socket.io Client:**
  The frontend connects to the backend using **Socket.io** to listen for real-time events like driver arrival, ride acceptance, or live movement on the map.

- **Map Integration:**
  Using services like **Google Maps API** or **Leaflet.js**, the app allows users to select pickup and drop locations, view drivers around them, and track their ride live.

- **Tailwind CSS for Styling:**

The frontend is styled with Tailwind CSS, offering a mobile-first, responsive design with a clean and modern interface. It ensures a smooth experience across different devices.

Backend Architecture (Node.js + Express + Socket.io)

The backend is built using **Node.js** and **Express.js**, providing a fast and scalable server-side environment. It's structured using the **MVC (Model-View-Controller)** pattern for organized, maintainable code.

- **API Layer with Express.js:**
  RESTful API routes handle user authentication, ride requests, ride status updates, and more. Controllers are used to process business logic, and responses are sent in JSON format.

- **Real-Time Engine with Socket.io:**
  The backend uses **Socket.io** to maintain live connections between clients (users and drivers). Events such as:

  - `rideRequest`
  - `rideAccepted`
  - `locationUpdate`
  - `rideCompleted`

  are emitted and received in real-time, enabling live tracking and immediate feedback.

- **Role-Based Functionality:**
  The backend handles two types of users: **Riders** and **Drivers**. It ensures that only valid users can perform certain actions (e.g., only drivers can accept rides).

- **Middleware & Validation:**
  Middlewares are used for authentication, error handling, and request validation to ensure security and reliability.

- **Socket Management:**
  Each client maintains a persistent connection, and the backend manages socket IDs to emit targeted events (e.g., notifying a specific driver of a new ride request).

Database Architecture (MongoDB + Mongoose)

The app uses **MongoDB** as the primary database, which offers flexibility in storing unstructured data and scales well for dynamic, real-time applications.

- **Mongoose ODM:**
  MongoDB is accessed using **Mongoose**, an Object Document Mapper that defines schemas for consistent and structured data storage.

- **Key Collections:**

  - `users`: Stores profile info, role (rider/driver), login credentials.

  - `rides`: Contains ride data including pickup/drop locations, status (`requested`, `accepted`, `completed`), timestamps, and assigned driver.

  - `locations`: Stores recent coordinates for live tracking (optional if not handled solely via sockets).

- **Data Relationships & Queries:**
  Rides are linked to users and drivers by reference IDs. The backend performs efficient queries to match available drivers with incoming ride requests.

How It All Connects

1. **User registers or logs in**, role is determined (driver or rider).
2. The **rider requests a ride** → API call to backend + socket event sent.
3. **Nearby drivers receive the ride request** via sockets.
4. When a **driver accepts**, a real-time update is sent back to the user.
5. The **rider tracks the driver** live on the map using continuous socket-based location updates.
6. Once the ride is completed, the **status is updated** in MongoDB, and both rider and driver are notified.

This architecture ensures RideReady can support:

- Real-time performance
- Scalability for high user load
- Clean, modular code for easier maintenance

Setup

Prerequisites

Before setting up the Uber clone application, ensure you have the following software and tools installed on your development machine:

Required Software

1. **Node.js (v14 or higher)**
    - The application backend is built on Node.js, which provides the runtime environment for executing JavaScript server-side.

- Download from nodejs.org and follow the installation instructions for your operating system.
- Verify installation by running `node -v` in your terminal.

2. **MongoDB**
   - The application uses MongoDB as its primary database.
   - Install MongoDB Community Edition from mongodb.com.
   - Alternatively, you can use MongoDB Atlas, a cloud-based MongoDB service. ○ Ensure the MongoDB service is running before starting the application.
3. **npm (Node Package Manager)**
   - This comes bundled with Node.js installation.
   - Used for managing application dependencies. ○ Verify with `npm -v` in your terminal.
4. **Google Maps API Key**
   - Required for location services, distance calculation, and mapping features.
   - Create a project in the Google Cloud Platform Console.
   - Enable the following APIs:
     - Google Maps JavaScript API
     - Google Places API
     - Google Directions API
     - Google Distance Matrix API
     - Google Geocoding API
   - Generate an API key with appropriate restrictions for security.
5. **Git**
   - For version control and cloning the repository.
   - Download from git-scm.com and follow installation instructions.

Development Tools (Recommended but Optional)

1. **Visual Studio Code** or any preferred code editor
2. **Postman** for testing API endpoints
3. **MongoDB Compass** for visual database management
4. **Redux DevTools** browser extension if using Redux for state management

Installation and Configuration

1. Clone the Repository bash

```
# Clone the repository to your local machine git

clone [repository-url]


# Navigate to the project root directory cd

uber-clone
```

2. Backend Setup

The backend provides APIs for authentication, ride management, and real-time communication.

bash

```
# Navigate to the backend directory cd

backend


# Install all required dependencies

npm install
```

Environment Variables Configuration

Create a `.env` file in the backend directory with the following variables:

```
# Database Connection

DB_CONNECT=mongodb://localhost:27017/uber-clone

# Or your MongoDB Atlas connection string

#
DB_CONNECT=mongodb+srv://<username>:<password>@cluster.mongodb.net
/uber-clone


# JWT Configuration

JWT_SECRET=your_jwt_secret_key_here

# Set a strong, random string for JWT token generation


# Google Maps API

GOOGLE_MAPS_API_KEY=your_google_maps_api_key_here


# Server Configuration

PORT=3000

NODE_ENV=development
```

```
# Optional: For production
# NODE_ENV=production
```

Database Initialization

The application will automatically connect to MongoDB when started as configured in the `db.js` file. It uses Mongoose to create schemas and models as defined in the models directory.

3. Frontend Setup

The frontend is built with React and uses Vite as its build tool.

bash

```
# Navigate to the frontend directory
cd ../frontend

# Install all required dependencies
npm install
```

Environment Variables Configuration
Create a `.env` file in the frontend directory with the following variables:

```
# API Configuration
VITE_API_URL=http://localhost:3000
# Update this if your backend runs on a different port

# Google Maps
VITE_GOOGLE_MAPS_API_KEY=your_google_maps_api_key_here
# Use the same API key as in the backend

# Socket Configuration
```

```
VITE_SOCKET_URL=http://localhost:3000 #

This should match your backend server
```

4. Database Setup and Configuration

The application uses MongoDB with Mongoose for object modeling. The following models are defined:

1. **User Model**: Stores passenger information including name, email, and authentication details
2. **Captain Model**: Stores driver information including personal details, vehicle information, and location
3. **Ride Model**: Manages ride information including pickup, destination, fare, and status
4. **BlacklistToken Model**: Handles logged-out tokens to prevent unauthorized access

The database schemas are automatically created when the application connects to MongoDB. Refer to the models directory in the backend code for detailed schema definitions.

5. Building the Application

Backend Build

No build step is required for the backend as it uses Node.js directly.

Frontend Build

bash

```
# In the frontend directory npm

run build


# This creates a 'dist' directory with optimized production files
```

6. Running the Application

Development Mode

Backend Server bash

```
# In the backend directory npm

run dev
```

```
# Or 'node server.js' if no dev script is defined
```

Frontend Development Server bash

```
# In the frontend directory npm
run dev
```

Production Mode

Backend Server bash

```
# In the backend directory npm
start
# Assuming the start script is configured in package.json
```

Frontend Serving

In production, you would typically serve the frontend build using a static file server like Nginx or through the Express backend.

7. Troubleshooting Common Setup Issues

MongoDB Connection Problems

- Ensure MongoDB service is running
- Verify connection string in `.env` file
- Check for network connectivity issues if using MongoDB Atlas

API Access Issues

- Verify Google Maps API key is correctly configured
- Ensure all required Google APIs are enabled in your Google Cloud Console ● Check for any API quota limitations

Socket.io Connection Problems

- Ensure CORS configuration in backend matches frontend origin
- Check network connectivity and firewall settings
- Verify socket URL in frontend configuration

JWT Authentication Issues

- Ensure JWT_SECRET is properly set in the backend `.env` file
- Check token expiration settings in the authentication middleware

8. Security Considerations

- Store API keys and secrets securely in environment variables
- Implement rate limiting on sensitive endpoints
- Set appropriate CORS policies to prevent unauthorized access
- Use HTTPS in production environments
- Regularly update dependencies to patch security vulnerabilities

By following these comprehensive setup instructions, you'll have the Uber clone application running in your development environment, ready for exploration, testing, and further development.

Folder structure

Folder Structure

The Uber clone application follows a well-organized project structure that separates frontend and backend code, making it easier to maintain and scale. The project employs the MERN stack (MongoDB, Express.js, React, Node.js) architecture with clear separation of concerns. Let's explore the detailed folder structure of both the client and server sides.

Client (Frontend) Structure

The frontend is built using React with Vite as the build tool, offering a modern, fast development experience. The application structure follows best practices for React applications with a component-based architecture.

Root Structure

```
frontend/
├── .gitignore           # Git ignore configuration
├── eslint.config.js     # ESLint configuration for code linting
├── index.html           # Entry HTML file
├── package.json         # Dependencies and scripts configuration
├── postcss.config.js    # PostCSS configuration for CSS processing
├── tailwind.config.js   # Tailwind CSS configuration
```

```
├── vite.config.js       # Vite build tool configuration
└── src/                 # Source code directory
```

Source Directory Structure

```
src/
├── assets/              # Static assets (images, icons, etc.)
├── components/          # Reusable UI components
│   ├── common/          # Shared components used across multiple
pages
│   │   ├── Button.jsx   # Custom button component
│   │   ├── Input.jsx    # Form input component
│   │   ├── Loader.jsx   # Loading indicator component
│   │   └── ...
│   ├── layout/          # Layout components
│   │   ├── Header.jsx   # Application header
│   │   ├── Footer.jsx   # Application footer
│   │   ├── Sidebar.jsx  # Navigation sidebar │
│   └── ...
│   ├── maps/            # Map-related components
│   │   ├── GoogleMap.jsx         # Main map component
│   │   ├── LocationMarker.jsx    # Map marker component
│   │   ├── RouteDisplay.jsx      # Route visualization component
│   │   └── ...
│   ├── user/            # User-specific components
│   │   ├── Profile.jsx           # User profile component │
│   ├── RideHistory.jsx       # Ride history display
│   │   └── ...
│   └── captain/         # Captain (driver) specific components │
├── Dashboard.jsx         # Captain dashboard
│       ├── VehicleInfo.jsx       # Vehicle information display
│       └── ...
├── contexts/            # React context providers │
├── AuthContext.jsx  # Authentication context
│   ├── RideContext.jsx  # Ride management context
│   ├── SocketContext.jsx # Socket.io connection context
│   └── ...
├── hooks/               # Custom React hooks
│   ├── useAuth.js       # Authentication hook
│   ├── useRide.js       # Ride management hook
│   ├── useLocation.js   # Location services hook
│   ├── useSocket.js     # Socket connection hook
│   └── ...
├── pages/               # Application pages
│   ├── Home.jsx         # Home/landing page
│   ├── Login.jsx        # User login page
│   ├── Register.jsx     # User registration page
│   ├── RideBooking.jsx  # Ride booking page
```

```
│   ├── CaptainLogin.jsx # Captain login page
│   ├── CaptainRegister.jsx # Captain registration page │
├── RideTracking.jsx # Ride tracking page
│   └── ...
├── services/            # API service modules
│   ├── api.js           # Base API configuration
│   ├── authService.js   # Authentication API methods
│   ├── rideService.js   # Ride-related API methods
│   ├── mapService.js    # Maps and location API methods
│   └── ...
├── utils/               # Utility functions
│   ├── validation.js    # Form validation utilities
│   ├── formatters.js    # Data formatting utilities │
├── constants.js     # Application constants
│   └── ...
├── App.jsx              # Main application component
├── main.jsx             # Application entry point
└── index.css            # Global styles (with Tailwind)
```

Component Organization

The frontend follows a modular component architecture that promotes reusability and maintainability:

1. **Common Components**: These are basic UI building blocks used throughout the application, such as buttons, inputs, modals, and loading indicators.
2. **Layout Components**: These define the overall structure of the application, including headers, footers, and navigation elements.
3. **Feature-Specific Components**: Organized by feature (maps, user, captain) to keep related functionality together.
4. **Page Components**: These represent complete views composed of multiple smaller components, corresponding to different routes in the application.

State Management

The application likely uses React's Context API for state management, with separate contexts for:

- Authentication state (login status, user/captain data)
- Ride state (current ride details, history)
- Real-time communication via Socket.io
- Location and map state

Server (Backend) Structure

The backend follows a modular architecture that separates concerns and promotes maintainability. It's built with Node.js and Express.js, using MongoDB as the database.

Root Structure

```
backend/
├── .gitignore           # Git ignore configuration
├── package.json         # Dependencies and scripts configuration
├── app.js               # Express application setup
├── server.js            # Server entry point
├── socket.js            # Socket.io setup and event handlers
├── db.js                # Database connection configuration
└── ...                  # Other configuration files
```

Functional Structure

```
backend/
├── controllers/         # Request handlers
│   ├── user.controller.js       # User-related controllers
│   ├── captain.controller.js    # Captain-related controllers
│   ├── ride.controller.js       # Ride-related controllers
│   ├── map.controller.js        # Map and location controllers
│   └── ...
├── models/              # Database models
│   ├── user.model.js            # User schema and model
│   ├── captain.model.js         # Captain schema and model
│   ├── ride.model.js            # Ride schema and model
│   ├── blackListToken.model.js  # Token blacklist model
│   └── ...
├── routes/              # API route definitions
│   ├── user.routes.js           # User API routes
│   ├── captain.routes.js        # Captain API routes
│   ├── ride.routes.js           # Ride API routes
│   ├── maps.routes.js           # Maps API routes
│   └── ...
├── middleware/          # Express middleware
│   ├── auth.middleware.js       # Authentication middleware
│   ├── validation.middleware.js # Request validation middleware
│   └── ...
├── services/            # Business logic services
│   ├── user.service.js          # User-related services
│   ├── captain.service.js       # Captain-related services
│   ├── ride.service.js          # Ride-related services
│   ├── maps.service.js          # Maps and location services
│   └── ...
├── utils/               # Utility functions
│   ├── helpers.js               # Helper functions
│   ├── constants.js             # Application constants
│   └── ...
└── config/              # Configuration files     ├──
environment.js           # Environment-specific
configuration
```

```
    ├── database.js             # Database configuration
└── ...
```

Architectural Pattern

The backend follows the Model-View-Controller (MVC) pattern with some additional layers:

1. **Models**: Define the data structure and database schema using Mongoose. These include:
    - `user.model.js`: Handles user data and authentication methods
    - `captain.model.js`: Manages captain (driver) data including vehicle information
    - `ride.model.js`: Stores ride details, status, and relationships
    - `blackListToken.model.js`: Manages invalidated tokens
2. **Controllers**: Handle HTTP requests and responses, invoking appropriate services:
    - `user.controller.js`: Manages user registration, login, profile access, and logout
    - `captain.controller.js`: Handles captain registration, login, profile access, and logout
    - `ride.controller.js`: Manages ride creation, confirmation, starting, and ending
    - `map.controller.js`: Handles location-based operations like getting coordinates and distances
3. **Services**: Contain business logic separate from request handling:
    - These would implement the core functionality like fare calculation, captain matching, etc.
4. **Routes**: Define API endpoints and connect them to controllers:
    - `user.routes.js`: User-related endpoints
    - `captain.routes.js`: Captain-related endpoints
    - `ride.routes.js`: Ride management endpoints
    - `maps.routes.js`: Location service endpoints
5. **Middleware**: Implements cross-cutting concerns:
    - `auth.middleware.js`: Handles authentication and authorization
    - Input validation middleware

Database Schema Design

The database models are carefully designed to represent the domain entities:

1. **User Model**:
    - Personal information (name, email)
    - Authentication details (password)
    - Socket connection ID for real-time communication

2. **Captain Model**:
    - Personal information (name, email)
    - Authentication details (password)
    - Vehicle information (color, plate, capacity, type)
    - Current location coordinates
    - Status (active/inactive)
    - Socket connection ID for real-time updates
3. **Ride Model**:
    - References to user and captain
    - Pickup and destination addresses
    - Fare amount
    - Status (pending, accepted, ongoing, completed, cancelled)
    - Distance and duration details
    - OTP for ride verification
4. **BlacklistToken Model**:
    - Storage for invalidated tokens
    - Automatic expiration mechanism

API Organization

The backend exposes a RESTful API organized by resource:

1. `/users/*`: User registration, authentication, and profile management
2. `/captains/*`: Captain registration, authentication, and profile management
3. `/rides/*`: Ride booking, tracking, and management
4. `/maps/*`: Location services, distance calculation, and suggestions

Communication Pattern

The application implements bidirectional real-time communication using Socket.io:

1. Socket connections are established when users and captains log in
2. Real-time events include:
    - Location updates from captains
    - Ride status changes
    - New ride notifications to nearby captains
    - Ride confirmation notifications to users

This comprehensive folder structure demonstrates a well-organized, maintainable codebase that follows modern development practices for a full-stack MERN application. It provides clear separation of concerns, promotes code reuse, and facilitates collaboration among development team members.

Running application

Running **RideReady** locally involves setting up both the **frontend (client)** and **backend (server)** environments independently, ensuring they communicate over defined ports and interact seamlessly in real-time. Below is a comprehensive guide to executing and understanding the startup flow.

□ Local Development Setup: Overview

RideReady is split into two separate environments:

- **Frontend (React + Vite)** – handles UI/UX and interacts with the backend via REST APIs and sockets.
- **Backend (Node.js + Express + Socket.io)** – manages the business logic, real-time communication, and database operations.

Both servers run independently and communicate over specified ports (e.g., frontend on `localhost:5173`, backend on `localhost:5000`).

Starting the Frontend Server

Location:

Navigate to the `frontend` directory using your terminal.

bash CopyEdit
```
cd
frontend
```

Install Dependencies:

bash CopyEdit
```
npm
install
```

This installs packages listed in `package.json`, including:

- `react` • `vite`
- `tailwindcss`
- `socket.io-client`
- `axios`

Start the Frontend Dev Server:

bash CopyEdit
```
npm run
dev
```

What Happens:

- Vite starts the local development server (usually on port `5173`).
- The browser opens automatically showing the login or homepage.
- The app connects to the backend using Axios for API calls and Socket.io client for real-time communication.

Live Reloading:

Vite supports Hot Module Replacement (HMR), so any changes in components or styles update instantly in the browser without requiring a full page reload.

Starting the Backend Server

Location:

Navigate to the `Backend` directory.

bash
CopyEdit `cd Backend`

Install Dependencies:

bash CopyEdit
`npm install`
Packages
installed
include:

- `express` – to create the API routes
- `mongoose` – for MongoDB integration
- `cors` – to allow cross-origin requests
- `dotenv` – for managing environment variables
- `socket.io` – for real-time WebSocket communication

Create a `.env` file (if not already provided):

env
CopyEdit

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/rideready
```

Start the Backend Server:

bash
CopyEdit
```
npm start
```

What Happens:

- The server starts on `localhost:5000`.
- It connects to MongoDB and starts listening for API requests and socket connections.
- Logs appear in the terminal showing that the server is running and connected to the database.

Live Communication Between Frontend & Backend

The two servers communicate in two primary ways:

1. **HTTP Requests via Axios:**
    - Used for authentication (login/register), requesting rides, and storing ride status.
    - Follows REST architecture (`POST /api/auth/login`, `GET /api/rides`, etc.).
2. **WebSocket Events via Socket.io:**
    - Enables real-time features like:
        - Sending ride requests to all online drivers.
        - Notifying the user when a driver accepts a ride.
        - Live location tracking (driver moving on the map).
        - Ride status updates in real-time (started, ended, cancelled).

Example events:

js
CopyEdit
```
socket.emit("ride-request", { ...data }); socket.on("ride-accepted", (data) => { ... });
```

Development Notes & Debugging Tips

- **Check Port Conflicts:**

  Make sure no other app is running on ports `5000` or `5173`.

Socket Connection Errors:
Ensure both frontend and backend are running when testing real-time features. Add console logs to track connection status:

js
CopyEdit `socket.on("connect", () => console.log("Connected to socket"));`

-

CORS Issues:
The backend must have proper CORS middleware enabled to allow requests from the frontend port:

js
CopyEdit `app.use(cors({ origin: "http://localhost:5173" }));`

-
- **Database Connectivity:**

  MongoDB must be installed and running. Use `mongod` to start the MongoDB server.

  Ensure your URI in `.env` is correct.

- **Environment Variables:**

  Make sure your `.env` file exists and is correctly configured. Avoid committing it to version control.

Why the App Runs This Way

- **Modularity:**
  Keeping frontend and backend separate allows independent development and deployment (you could even host them on different servers later).

- **Real-Time Infrastructure:**

Socket.io needs continuous connection between the client and server, which is why both need to be running concurrently.

- **Security & Performance:**
  Running separate dev servers allows better control over logging, debugging, and testing API endpoints without interfering with the UI.

API Documentation

The RideReady backend exposes a set of RESTful APIs to handle key operations such as user authentication, ride management, and real-time interaction support. These endpoints are consumed by the frontend using Axios and complement Socket.io for bi-directional, event-driven communication.

The API is organized modularly using Express.js routes, controllers, and middleware for validation and authentication. All endpoints return data in JSON format and follow conventional HTTP status codes.

Base URL

bash CopyEdit

```
http://localhost:5000/api/
```

Authentication APIs

POST `/api/auth/register`

Registers a new user (either rider or driver).

Request Body:

json
CopyEdit

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "123456",
  "role": "rider"  // or "driver" }
```

Response:

json
CopyEdit

```
{
  "message": "User registered successfully",
  "user": {
    "id": "64532e...",
    "name": "John Doe",
    "role": "rider",
    "token": "jwt_token_here"
  } }
```

POST `/api/auth/login`

Logs in an existing user.

Request Body:

json
CopyEdit

```
{
  "email": "john@example.com",
  "password": "123456" }
```

Response:

json
CopyEdit

```
{
  "message": "Login successful",
  "user": {
    "id": "64532e...",
    "name": "John Doe",
    "role": "rider",
    "token": "jwt_token_here"
  } }
```

Ride Management APIs

POST `/api/rides/request`

Rider requests a new ride.
Request Body:

json
CopyEdit

```
{
  "pickup": "MG Road, Bangalore",
  "destination": "Electronic City",
  "userId": "64532e..." }
```

Response:

json
CopyEdit

```
{
  "message": "Ride requested successfully",
  "ride": {
    "id": "64abcd...",
    "status": "requested",
    "assignedDriver": null
  } }
```

GET `/api/rides/available`

Driver fetches a list of ride requests they can accept.

Response:

json
CopyEdit

```
[
  {
    "id": "64abcd...",
    "pickup": "MG Road",
    "destination": "Electronic City",
    "status": "requested"
  } ]
```

POST `/api/rides/accept/:rideId`
Driver accepts a ride. Headers:

```
Authorization: Bearer <jwt_token>
```

Response:

json
CopyEdit

```
{
  "message": "Ride accepted",
  "ride": {
    "id": "64abcd...",
    "driver": "64789c...",
"status": "accepted"
  } }
```

PATCH `/api/rides/start/:rideId`

Driver starts the ride.

Response:

json
CopyEdit

```
{
  "message": "Ride started",
  "ride": {
    "id": "64abcd...",
    "status": "in_progress"
```

```
} }
```

PATCH `/api/rides/complete/:rideId`

Completes the ongoing ride.

Response:

json
CopyEdit

```
{
  "message": "Ride completed",
  "ride": {
    "status": "completed"
  } }
```

User APIs

GET `/api/users/:userId`

Fetch user profile details.

Response:

json
CopyEdit

```
{
  "id": "64532e...",
  "name": "John Doe",
  "email": "john@example.com",
  "role": "rider" }
```

Error Handling

All APIs return structured error messages with status codes:

Example Error (Validation Failed):

json
CopyEdit

```
{
```

```
  "error": "Email already exists" }
```

Example Error (Unauthorized):

json
CopyEdit

```
{
  "error": "Token is invalid or expired"
}
```

Authentication & Security

- All protected endpoints require a JWT token in the `Authorization` header.
- Middleware checks token validity before allowing access to sensitive endpoints (ride actions, user profile, etc.).

Socket.io Events (Complementing API)

While REST APIs handle data persistence and primary actions, real-time events are managed via Socket.io:

Event Name	Triggered By	Description
`ride-request`	Rider	When a ride is requested
`Ride-available`	Server → Drivers	Broadcasts new ride to all nearby drivers
`ride-accepted`	Driver	When driver accepts the ride
`location-update`	Driver	Sends live location to user
`ride-completed`	Driver	When the trip is marked completed

Best Practices Followed:

- RESTful endpoint design.

- Role-based route access via middleware.
- Use of HTTP verbs (`GET`, `POST`, `PATCH`, etc.) aligned with resource behavior.
- Token-based auth with JWT.
- Input validation to prevent malformed data.
- Socket.io used where real-time performance is essential.

Authentication

Authentication is a crucial part of RideReady, ensuring that only verified users (riders or drivers) can access and interact with the system. It prevents unauthorized access to protected APIs, ensures role-based interaction, and plays a vital role in secure real-time communication between clients and the backend.

RideReady uses **JWT (JSON Web Tokens)** for stateless, token-based authentication, allowing scalability, easy session management, and secure client-server communication.

Authentication Flow Overview

1. **User Registers or Logs In**
    - Provides email, password, and role (rider/driver).
    - If credentials are valid, a **JWT token** is generated and returned.
2. **Frontend Stores the Token**
    - The JWT token is stored on the client-side (typically in local storage or memory).
    - The token is attached to every authenticated API request via the `Authorization` header.
3. **Backend Verifies the Token**
    - A middleware checks the token on protected routes.
    - If valid, the request proceeds; if not, an error is returned.

JWT (JSON Web Token) in RideReady

RideReady uses **JWT tokens** to maintain a stateless, secure session between the client and server.

Token Generation

Upon successful login or registration, the server generates a token using a secret key and payload containing:

- User ID
- User Role (driver or rider)
- Expiration timestamp

js
CopyEdit
```
const token =
jwt.sign(
  { userId: user._id, role: user.role },
  process.env.JWT_SECRET,
  { expiresIn: '1h' } );
```

Token Structure

A JWT has three parts:

- **Header:** Contains the algorithm and token type.
- **Payload:** Contains claims like `userId` and `role`.
- **Signature:** Used to verify that the payload has not been tampered with.

Authentication Middleware

Protected routes use an `authMiddleware.js` function to:

- Check for the presence of the `Authorization` header.
- Verify the JWT token using the secret key.
- Decode the user info and pass it to the request object.

Example:

js
CopyEdit

```
const token = req.header("Authorization")?.split(" ")[1]; const
decoded = jwt.verify(token, process.env.JWT_SECRET); req.user
= decoded;
```

If invalid or expired:

json
CopyEdit

```
{
  "error": "Unauthorized access. Invalid or expired token." }
```

Role-Based Access Control (RBAC)

Role-based permissions are enforced across endpoints. For example:

- Only **riders** can request rides.
- Only **drivers** can accept or complete rides.
- Certain Socket.io events are filtered based on roles.

This is handled either in middleware or within controller logic by checking:

js
CopyEdit

```
if (req.user.role !== "driver") {
  return res.status(403).json({ error: "Forbidden action for this
role." }); }
```

Session Management

- Since JWT is **stateless**, the server doesn't store sessions in memory.
- This makes it easier to **scale horizontally** without maintaining a central session store.
- Each request is self-contained with its token, ensuring high performance.

Authentication in Socket.io

Socket.io also respects authentication:

1. **Token is sent during socket connection:**

js
CopyEdit
```
const socket = io("http://localhost:5000", {
auth: { token: "user_jwt_token" } });
```

2. **Server verifies token before establishing the connection:**

js
CopyEdit
```
io.use((socket, next) => {
  const token = socket.handshake.auth.token;   const user
= jwt.verify(token, process.env.JWT_SECRET);
  socket.user = user;
next(); });
```

3. **If invalid, the socket is rejected:**

js
CopyEdit `next(new Error("Authentication failed."));`

Security Practices Implemented

- **Password Hashing:**
  All passwords are securely hashed using **bcrypt** before storing in MongoDB.

- **Environment Variables:**
  JWT secrets, DB URIs, and other sensitive values are stored in a `.env` file and never hardcoded.

- **Token Expiry:**
  Tokens expire after 1 hour to reduce security risks. Users must re-authenticate after expiration.

- **Error Handling:**
  Proper status codes (`401`, `403`, `500`) and error messages are returned for all auth failures.

Sample Authorization Header

Every protected API call includes this header:

makefile
CopyEdit
```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...
```

This token is verified on the backend, and the decoded user details are injected into the request object for further processing.

Benefits of the Approach

- **Security:** Tokens are signed and cannot be tampered with without the secret.
- **Scalability:** No need to store sessions in memory or database.
- **Flexibility:** Easy to implement role-based logic and expand to include refresh tokens later.
- **Reusable in Realtime & HTTP Layers:** The same JWT token authenticates both API requests and real-time socket connections.

User Interface

he **User Interface (UI)** of RideReady is built with a focus on simplicity, responsiveness, and real-time interactivity. It caters to two distinct user roles — **Riders** and **Drivers**, each having tailored dashboards and workflows. The UI is developed using **React.js**, **Vite**, and **Tailwind CSS**, ensuring rapid development and a clean, modern look across all devices.

Design Goals & Principles

The design philosophy behind RideReady emphasizes:

- **Clarity & Ease of Use** – Minimalist layout with intuitive navigation.
- **Role-Based UI Flow** – Different screens for riders and drivers depending on login.
- **Real-Time Feedback** – Visual feedback on ride status, live location, and notifications.
- **Responsiveness** – Mobile-first design using Tailwind's utility classes.
- **Separation of Concerns** – Modular components and clearly defined page responsibilities.

Navigation Flow

Upon login, users are directed to a role-specific dashboard:

Rider Flow:

1. **Home/Dashboard**
    - Displays map with current location
    - "Request Ride" form with pickup & drop locations
2. **Ride Status Screen**

- Waiting for driver acceptance
- Driver details shown upon acceptance

3. **Live Tracking Screen**
    - Live updates of driver's location on the map
4. **Ride Completed Screen**
    - Trip summary & feedback (optional)

Driver Flow:

1. **Driver Dashboard**
    - Shows available ride requests in real-time
    - Map with own location
2. **Incoming Ride Request**
    - Prompt to accept/reject ride
3. **Ride Status**
    - Accept → Start → End buttons appear sequentially
    - Live tracking by rider shown via sockets
4. **Completion Screen**
    - Confirmation and readiness for next ride

Component Architecture

The frontend is organized into reusable **React components**. Some key components include:

- `MapView.js` – Embeds and manages the interactive map
- `RideRequestForm.js` – Pickup & destination selection
- `RideStatusCard.js` – Displays status (requested, accepted, in progress)

- `DriverListItem.js` – Shows drivers/riders in list form
- `Navbar.js` – Role-based dynamic navigation bar
- `SocketHandler.js` – Handles all socket interactions at component level
- `RoleRouter.js` – Redirects users to appropriate dashboards after login

---

Responsive Design

Using **Tailwind CSS**, every UI element adjusts fluidly across devices:

- **Grid & Flexbox** layouts ensure clean structure.
- Buttons, inputs, and cards scale automatically.
- Conditional rendering hides irrelevant content on smaller screens.
- Uses `min-h-screen`, `w-full`, `overflow-scroll`, `p-4`, etc., for scrollable, fluid sections.

Screens are tested on:

- ☑Mobile (320–768px)
- ☑Tablet (768–1024px)
- ☑Desktop (>1024px)

Map & Geolocation Integration

The map is central to the user interface:

- Shows **current location**, **pickup & drop**, and **real-time tracking**
- Integrated via **Google Maps API** or **Leaflet.js**
- Riders can visually confirm the driver's movement toward them

- Drivers get directions to the pickup and drop-off points
- Animated markers enhance clarity of location updates

Real-Time UI Updates

Socket.io client is integrated within components to handle live updates:

- A rider sees driver status updates immediately after ride acceptance.
- Driver's live location is continuously reflected on the rider's map.
- No page refresh needed — UI re-renders in response to socket events.

Example UI states:

- "Waiting for driver…" (spinner with status card)
- "Driver is en route" (live location with name/photo/vehicle info)
- "Ride Completed" (trip info + success message)

Visual Styling Highlights

- **Buttons:** Large, rounded, green primary buttons with hover effects (`hover:bg-green-700`)
- **Cards:** Soft shadows, rounded corners, and responsive content blocks
- **Forms:** Minimalist input fields, validation feedback (`border-red-500` on errors)
- **Icons:** SVGs or Feather Icons used for location pins, vehicles, status indicators
- **Toasts/Alerts:** Informational messages for ride updates using in-app alerts or libraries like `react-toastify`

User Experience Enhancements

- **Loading States:** Shimmer or spinner indicators during async events (e.g., fetching ride status)
- **Role-Based Redirection:** Users are automatically redirected after login based on their role (Driver or Rider)
- **Error Handling:** Friendly messages like “No drivers nearby” or “Ride not found”
- **Feedback Loop:** Clear call-to-action messages and buttons after ride completion

---

Sample Screenshots to Include in Report:

- Login & Registration Page (Desktop + Mobile)
- Rider Dashboard with Map and Ride Request
- Driver Ride Acceptance Screen
- Live Ride Tracking Page
- Ride Completed Summary Page

(You can capture these using tools like Snipping Tool or use emulators for mobile preview.)

Why This UI Works Well

- It's **fast** (thanks to Vite), **clean**, and **highly responsive**.
- Role-based design ensures clarity and removes clutter.
- Real-time updates make the app feel alive and interactive.
- Minimalist Tailwind-based styling makes it look professional without being overwhelming.

Testing

Testing plays a critical role in ensuring the **stability**, **functionality**, and **performance** of RideReady — a real-time, full-stack ride-booking platform. Since the project involves both RESTful APIs and real-time socket communication, a hybrid testing strategy was adopted combining **manual testing**, **automated test cases**, and **live socket interaction simulations**.

Testing Strategy

RideReady uses a **multi-layered testing approach**:

1. **Unit Testing** – Ensures each individual function, service, or logic unit behaves as expected.

2. **Integration Testing** – Verifies interaction between backend modules like controllers, models, routes, and middleware.

3. **End-to-End (E2E) Testing** – Simulates real user actions across the full stack to validate complete workflows.

4. **Manual UI & UX Testing** – Verifies frontend usability, responsiveness, and real-time interactivity.

5. **Socket Testing** – Tests the bidirectional real-time communication between clients and the server.

⚙Tools and Frameworks Used

Type	Tool/Library	Purpose

Unit Testing	**Jest**	JavaScript testing framework for backend logic
Integration	**Supertest**	Testing Express.js APIs
E2E Testing	**Postman**, **Thunder Client**	Manual REST API tests with headers/body
UI Testing	**Chrome DevTools**	Manual testing on different screen sizes
Real-Time Testing	**Socket.io Test Clients**	Simulate driver/rider socket events

---

Backend Testing

1. Unit Tests (Jest):

- User registration/login service
- JWT token verification logic
- Ride creation logic
- Role-based access middleware
- MongoDB model hooks (e.g., password hashing)

js

CopyEdit

```
test('should hash user password before saving', async () => {
  const user = await User.create({ email: "...", password: "123456"
});
expect(user.password).not.toBe("123456");
});
```

2. API Testing (Supertest + Jest):

- Auth endpoints: `/api/auth/register`, `/api/auth/login`
- Ride endpoints: `/api/rides/request`, `/api/rides/accept`
- Middleware error handling
- Token validation scenarios (expired, malformed, missing)

js

CopyEdit
```
request(app)
  .post("/api/rides/request")
  .set("Authorization", `Bearer ${token}`)
.send({ pickup: "...", destination: "..." })
  .expect(200);
```

3. Postman Collections:

- Full set of API requests with saved environment variables
- Used for regression testing after major backend changes

Frontend Testing

Manual UI Testing Focused On:

- Form validation: Empty inputs, invalid credentials
- Role-based routing (riders vs drivers)
- Map component loading under poor internet conditions
- Real-time updates on ride request and driver status
- Responsive layout across mobile, tablet, and desktop

Example Scenarios:

- Rider books a ride and receives real-time driver location.

- Driver logs in and sees ride requests immediately.
- Pages redirect correctly after login/logout.
- Errors like "No drivers available" show proper alerts.

Socket.io Testing

Real-time features were tested manually and using simulated socket clients:

Test Scenarios:

- Rider requests a ride → Broadcast reaches all drivers.
- Driver accepts ride → Rider receives update instantly.
- Driver shares live location → Rider map updates in real-time.
- Unauthorized socket connections are rejected correctly.

Socket Debugging Tools:

- Used `socket.io-client` test script
- Enabled verbose logging: `localStorage.debug = '*'` in browser console
- Created separate Node clients to simulate driver & rider socket interactions

---

Cross-Device Testing

The app was tested across:

- Chrome, Firefox, and Brave browsers
- Android phone (via local IP testing)
- Chrome DevTools mobile emulation

Bug Fixes Identified During Testing

Issue Detected	Resolution
Map not rendering when reloaded	Added loading guard for map script
Rider receiving ride-completed prematurely	Added status validation before event emission
Form submission without fields	Frontend form validation with user feedback
CORS error in socket handshake	Proper `cors` configuration added in server setup
Socket events firing multiple times	Fixed socket reinitialization inside component

Test Case Summary

Module	Type	Test Cases Written	Passed	Notes
Auth	Unit/API	10		JWT, login, register
Rides	API + Socket	12		Accept, request, update

Socket Events	Integration	8	Live tracking, broadcast
Frontend Screens	Manual	10+ scenarios	Map, form, UI feedback

Why This Testing Approach Works

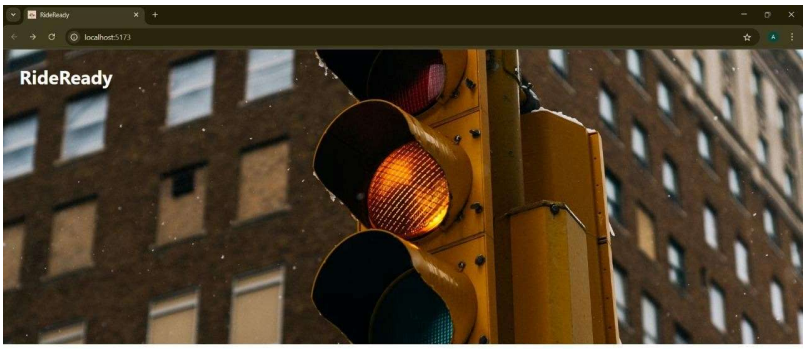
- It validates both **stateless API logic** and **stateful real-time behavior**.
- Manual + automated testing ensures both logic and experience are reliable.
- Role-specific flows and socket-based behavior are verified at runtime.
- Easy to replicate bugs using Postman and socket scripts.

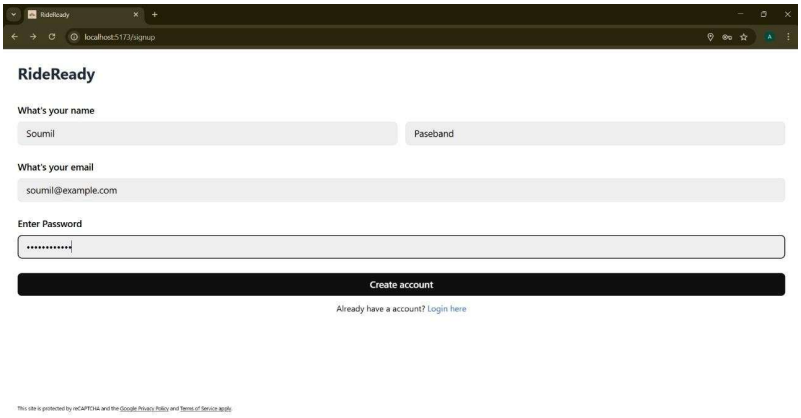
Demo

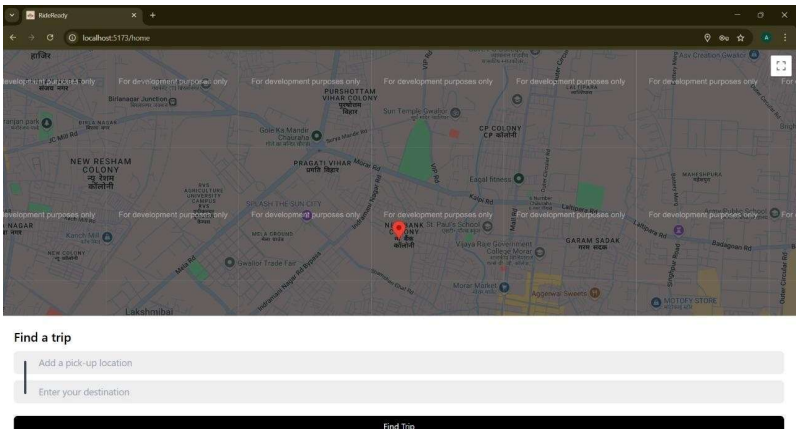
Git link -

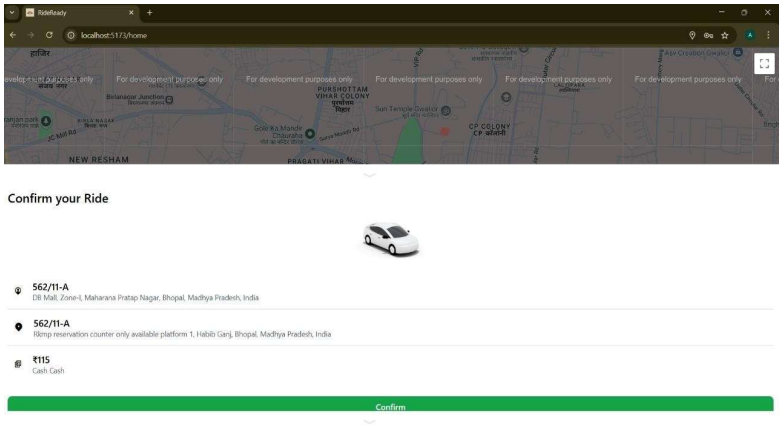
https://github.com/anibjee/RideReady-Cab-Booking-App **Video demo** https://drive.google.com/file/d/1gMV6vwGW4AONU04kviZuVNyi_zJvljFX/view?usp=sharing

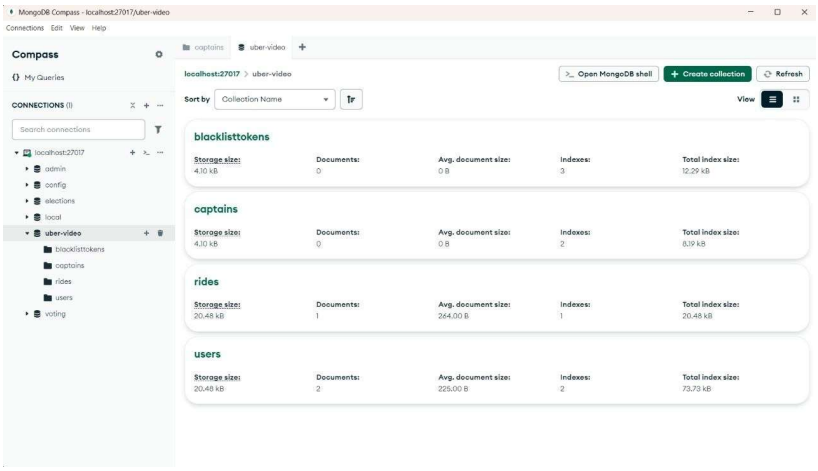
Below given are the screenshots of the project created by our team-






RideReady
localhost:5173/home
Find a trip
Add a pick-up location
Enter your destination
Find Trip


RideReady
localhost:5173/home
Confirm your Ride
562/11-A
DB Mall, Zone-I, Maharana Pratap Nagar, Bhopal, Madhya Pradesh, India
562/11-A
Nkmp reservation counter only available platform 1, Habib Ganj, Bhopal, Madhya Pradesh, India
₹115
Cash Cash
Confirm


MongoDB Compass - localhost:27017/uber-video
Compass
My Queries
CONNECTIONS (1)
localhost:27017
admin
config
elections
local
uber-video
blacklisttokens
captains
rides
users
voting
localhost:27017 > uber-video
Open MongoDB shell
Create collection
Refresh
Sort by
Collection Name
View
blacklisttokens
captains
rides
users

Known Issues

Despite its robust architecture and smooth functionality, **RideReady**, like all real-time web applications in active development, comes with a few **known issues and limitations**. These issues are not necessarily bugs, but rather challenges that are either under optimization or flagged for future enhancement.

The following outlines all known technical, functional, and UI/UX-related limitations encountered during development and testing.

---

1. No Persistent Ride History

- **Issue:** Currently, ride data is stored in MongoDB but not displayed on the frontend post-completion.
- **Impact:** Users (both riders and drivers) cannot view their past rides.
- **Status:** Backend already supports it; frontend UI for historical ride view is under development.

---

2. Real-Time Location Glitches Under Poor Internet

- **Issue:** Socket-based real-time tracking depends on strong internet. With slow or intermittent networks, live driver location updates may freeze or delay.
- **Impact:** Rider may not see real-time updates accurately.
- **Status:** Consider integrating socket reconnection strategies and throttling updates to reduce jitter.

---

3. Token Expiry Without Refresh Handling

- **Issue:** JWT tokens expire after 1 hour, and there is currently no **refresh token** mechanism in place.

- **Impact:** Users must log in again after token expiry, which disrupts long session rides or background usage.
- **Status:** Planned improvement: implement refresh tokens or silent re-authentication.

---

4. Duplicate Socket Event Listeners on Page Refresh

- **Issue:** Refreshing the frontend may reinitialize the same socket multiple times.
- **Impact:** Can cause duplicate events to fire (e.g., two ride accepted messages).
- **Status:** Resolved in later iterations by cleaning up socket instances with `socket.off()` during component unmount.

---

5. Hardcoded Map API Key (Security Risk)

- **Issue:** The frontend map service (Google Maps or Leaflet) may use a temporarily hardcoded API key during development.
- **Impact:** Exposes the key to frontend users, risking quota exhaustion or abuse.
- **Status:** Needs to be moved into `.env` file and securely accessed using environment variables during build.

---

6. No Cancellation Flow Yet

- **Issue:** Neither drivers nor riders can cancel a ride once it's requested or accepted.
- **Impact:** Inflexible user experience if someone changes their mind or makes a mistake.
- **Status:** Planned feature in the next release — implementing cancel buttons and socket cancellation events.

-

7. Lack of Concurrent Driver Filtering

Issue: All drivers receive all ride requests regardless of distance.

- **Impact:** Inefficient matching system in populated areas.
- **Status:** To be optimized with geo-fencing or proximity-based filtering (e.g., filter drivers within 5km).

---

8. Mobile Offline Handling

- **Issue:** No fallback UI or offline detection implemented in the frontend.
- **Impact:** Sudden loss of connection may not display alerts or retry mechanisms to the user.
- **Status:** Will be improved with offline event listeners and alert modals.

---

9. Minimal Input Validation on Location Fields

- **Issue:** The ride request form accepts blank or poorly formatted location text.
- **Impact:** May cause backend errors or map failures.
- **Status:** Needs frontend form enhancements and stricter backend input sanitization.

---

10. Ride Lifecycle Not Displayed After Completion

- **Issue:** Once the ride ends, there's no UI summary (e.g., trip duration, distance, driver name).
- **Impact:** Reduces feedback loop and user trust.
- **Status:** The backend supports this; frontend UI is planned for the next iteration.

-

11. Socket Disconnection Unnoticed

Issue: When the socket disconnects (e.g., due to backend restart), the user isn't informed.

- **Impact:** Riders may assume they're still being tracked when they are not.
- **Status:** Implementing a toast/alert system for socket disconnection and reconnection attempts.

---

12. No Automated Frontend Tests

- **Issue:** While backend APIs are tested with Jest and Supertest, there are no unit or UI tests for React components.
- **Impact:** UI regressions can go unnoticed.
- **Status:** Future plan includes adding **React Testing Library** or **Cypress** for component-level and E2E testing.

Summary of Known Issues

Issue	Area Affected	Impact	Status
No ride history	UI/UX	Low	Planned
Socket lag on poor internet	Real-time	Medium	In Progress
JWT expiry without refresh	Auth	Medium	To Do
Duplicate socket listeners	Real-time	Medium	Fixed
Hardcoded map API key	Security	High	To Do
No ride cancellation	UX	High	To Do
All drivers get all rides	Matching Logic	Medium	Planned
No offline mode	UX	Medium	To Do
Poor location validation	Form Validation	Medium	In Progress
Missing trip summary	UX	Medium	Planned

• No socket disconnection notice	UX	Medium	Planned

No React UI tests	Testing	Medium	Future Plan

Future Enhancements

As a dynamic and evolving real-time ride-booking platform, **RideReady** has the potential to scale far beyond its current MVP (Minimum Viable Product) version. This section outlines enhancements across **technical**, **functional**, **UX**, and **business** domains, turning RideReady into a production-ready platform.

---

1. Advanced Geo-Based Matching Algorithm

Current Limitation:
All online drivers receive all ride requests, regardless of proximity.

Enhancement Plan:

- Integrate Haversine formula or GeoJSON queries in MongoDB to filter drivers within a radius (e.g., 3–5 km).
- Prioritize closest driver or most available one based on ETA and previous rides.

Impact:
Faster response times, more optimized driver allocation, and reduced fuel consumption.

---

2. Ride Cancellation & Refund Logic

Current Limitation:
Neither the rider nor the driver can cancel once a ride is requested or accepted.

Enhancement Plan:

- Add cancel buttons with confirmation prompts.
- Emit socket events to inform the other party in real-time.
- Integrate cancellation fees based on timing or role (e.g., last-minute rider cancelations).

Impact:
Improves trust and control, mimics real-world user behavior.

---

3. Refresh Token Implementation

Current Limitation:
JWT tokens expire after 1 hour; no auto-renewal mechanism exists.

Enhancement Plan:

- Implement **access + refresh token** logic with silent refresh in frontend.
- Store refresh token securely (HTTP-only cookie or encrypted local storage).

Impact:
Improves user experience and prevents abrupt session drops, especially during long trips.

---

4. Ride History and Analytics Dashboard

Enhancement Plan:

- Add a new dashboard tab showing past rides, driver details, trip distance, and fare.
- Generate charts for total rides, earnings, and miles driven (for drivers).

Impact:
Improves user insight, adds transparency, and encourages repeat usage.

---

5. In-App Chat Between Driver & Rider

Enhancement Plan:

- Implement real-time messaging system using Socket.io rooms.
- Allow drivers and riders to communicate after ride acceptance.
- Include emojis, message history, and alert tones.

Impact:
Reduces miscommunication, builds connection, and improves coordination.

---

6. AI-Powered ETA and Surge Pricing

Enhancement Plan:

- Integrate ML models to estimate ride durations based on traffic and history.
- Introduce surge pricing based on time, demand, or events.
- Use Google Maps Traffic API for live traffic info.

Impact:
Modernizes RideReady into a data-driven platform and increases revenue potential.

---

7. Wallet System and Payment Gateway Integration

Enhancement Plan:

- Allow users to load money into in-app wallet.
- Integrate Razorpay, Stripe, or Paytm for ride payments.
- Generate dynamic ride invoices with breakdown of charges.

Impact:
Enables monetization, streamlines cashless payments, and boosts professionalism.

---

8. Multilingual Support

Enhancement Plan:

- Add i18n libraries (e.g., react-i18next) to support Hindi, Tamil, Bengali, etc.
- Allow language selection during login or via profile settings.

Impact:
Increases accessibility and inclusivity, especially in rural or regional areas.