MEENAKSHI SUNDARARAJAN

ENGINEERING COLLEGE

Kodambakkam, Chennai-600024

(An Autonomous Institution)

NM1042 – MERN STACK POWERED BY MONGODB

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROJECT TITLE: HOUSE RENTAL APPLICATION USING MERN

TEAM ID: **NM2024TMID07755**

FACULTY MENTOR: **Mrs. M. Sowmiya**

Project submitted by,

NAAN MUDHALVAN ID	NAME	REGISTER NUMBER
244E86EFDB0B8EB9DF242D30D7055553	*KARTHIGAI KANNAN. K*	*311521104024*
8AFF04CC141A8BDB0BC4341C098E7B64	*CHARAN. M*	*311521104008*
907134B6AE8D1D9B44F48E718FAE818F	*ISAAC SAMUEL. J*	*311521104018*
B6CF90AC2C95EEF9B2B4BA8C1F765F43	*PRADEEP KUMAR. G*	*311521104032*

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**HOUSE RENTAL APPLICATION USING MERN**" is the bonafide work of "**KARTHIGAI KANNAN.K (311521104024)**" "**CHARAN.M (311521104008)**" "**ISAAC SAMUEL.J (311521104018)**" "**PRADEEP KUMAR.G (311521104032)**" Naan Mudhalvan Team ID "**NM2024TMID07755**" who carried out the project work under my supervision.

SIGNATURE

Mrs. M. Sowmiya, M.E.

ASSISTANT PROFESSOR

Computer Science and Engineering

Meenakshi Sundararajan Engineering College

No. 363, Arcot Road, Kodambakkam,

Chennai -600024

SIGNATURE

Dr.S.Aarthi,M.E., Ph.D.

HEAD OF THE DEPARTMENT

Computer Science and Engineering

Meenakshi Sundararajan Engineering College

No. 363, Arcot Road, Kodambakkam,

Chennai -600024

Submitted for the project viva voce of Bachelor of Engineering in Computer Science and Engineering held on _____.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

First and foremost, we express our sincere gratitude to our Respected Correspondent **Dr. K. S. Lakshmi**, our beloved Secretary **Mr. N. Sreekanth**, Principal **Dr. S. V. Saravanan** for their constant encouragement, which has been our motivation to strive towards excellence.

Our primary and sincere thanks goes to **Dr. S. Aarthi,** Head of the Department, Department of Computer Science and Engineering, for her profound inspiration, kind cooperation and guidance.

We are grateful to **Mrs. M. Sowmiya**, Internal Guide, Assistant Professor as our project coordinator for their invaluable support in completing our project. We are extremely thankful and indebted for sharing expertise, and sincere and valuable guidance and encouragement extended to us.

Above all, we extend our thanks to God Almighty without whose grace and blessings it would not have been possible.

ABSTRACT

The House Rent Application is a modern web application designed to streamline the rental process for both tenants and landlords. It offers a user-friendly interface, real-time updates, and secure authentication, ensuring a seamless experience for all users involved in renting properties.

The frontend of the application is developed with React.js, a popular JavaScript library known for building dynamic and responsive user interfaces. React enables the creation of reusable components that offer an intuitive interface, making it easier for tenants to browse properties and for landlords to manage their listings. The React framework provides fast rendering and ensures smooth navigation within the application. The backend of the application is powered by Node.js, a robust and scalable runtime environment, with Express as the web server framework. Express simplifies the development of RESTful APIs, which handle all client requests, including creating listings, user registration, login, and property search. The backend logic is designed to ensure the efficient handling of data and secure communication between the client and server.

For data storage and management, the application uses MongoDB, a NoSQL database that provides flexibility in storing property listings, user profiles, and transaction histories. MongoDB's schema-less design allows the system to handle varying types of data, making it well-suited for the dynamic nature of real estate data. The application employs JWT (JSON Web Tokens) for secure authentication and authorization, ensuring that only authorized users can access specific resources. JWTs are used for both login and session management, allowing users to securely interact with the platform without repeatedly entering credentials. Overall, the House Rent Application aims to modernize and simplify the rental process by providing a comprehensive, secure, and easy-to-use platform for both tenants and landlords.

The House Rent Application is a web platform designed to simplify the rental process for tenants and landlords. Built with React.js for the frontend, Node.js with Express for the backend, and MongoDB for data management, it ensures a seamless user experience. The application uses JWT for secure authentication, allowing users to securely register, log in, and manage listings. Tenants can easily browse properties, while landlords can manage their listings and view tenant applications. This modern, secure system streamlines the rental process, offering a flexible and user-friendly solution for all users involved.

TABLE OF CONTENTS

CHAPTER NO.	TITE	PAGE NO.
	ABSTRACT	
	LIST OF FIGURES	
1.	**INTRODUCTION**	1
	1.1 ABOUT THE PROJECT	1
	1.2 PROJECT OVERVIEW	2
	1.3 PURPOSE	2
	1.4 EXISTING SYSTEM	2
	1.5 PROBLEM STATEMENT	3
2.	**SYSTEM ARCHITECTURE**	4
	2.1 SYSTEM ARCHITECTURE	4
	2.2 FRONTEND DEVELOPMENT	4
	2.3 BACKEND DEVELOPMENT	8
	2.4 MONGODB	10
	2.5 KEY FEATURES	10
3.	**SETUP INSTRUCTIONS**	12
	3.1 BACKEND PREREQUISITES	12
	3.2 FRONTEND PREREQUISITES	13

4.	**FOLDER STRUCTURE**	14
	4.1 CLIENT	14
	4.2 SERVER	15
5.	**SYSTEM MODELING**	16
	5.1 ENTITY RELATION MODEL	16
6.	**RUNNING THE APPLICATION**	18
7.	**API DOCUMENTATION**	20
	7.1 AUTHENTICATION / RESPONSE	20
8.	**AUTHENTICATION**	27
	8.1 TYPES OF AUTHENTICATION METHODS	27
	8.2 AUTHENTICATION FLOW	28
	8.3 AUTHENTICATION SECURITY PRACTICES	28
9.	**USER INTERFACE**	29

10.	**TESTING**	32
	10.1 UNIT TESTING	32
	10.2 INTEGRATION TESTING	32
	10.3 END-TO-END(E2E) TESTING	32
	10.4 SECURITY TESTING	33
11	**SCREENSHOTS & DEMO**	34
	11.1 HOME PAGE	34
	11.2 LOGIN & REGISTER PAGE	35
	11.2.1 REGISTER PAGE	35
	11.2.2 LOGIN PAGE	36
	11.2.3 FORGOT PASSWORD PAGE	36
	11.3 ADMIN PANEL	37
	11.4 OWNER PANEL	38
	11.5 TENANT PANEL	40
	11.6 DEMO VIDEO LINK	41
12	**FUTURE ENHANCEMENTS**	42
13	**CONCLUSION**	43
	GITHUB LINK	43

LIST OF FIGURES

FIGURE NO.	NAME OF THE FIGURE	PAGE NO.
3.2	SYSTEM ARCHITECTURE	4
4.2	ENTITY RELATION DIAGRAM	16

CHAPTER I

INTRODUCTION

1.1 ABOUT THE PROJECT :

The House Rent Application is a modern web-based platform designed to facilitate the property rental process for both tenants and landlords. It simplifies the way rental listings are managed and viewed, offering a seamless and efficient experience for users. The application is built using cutting-edge technologies, including React.js for the frontend, Node.js with Express for the backend, MongoDB for data storage, and JWT (JSON Web Tokens) for secure user authentication and authorization. These technologies are chosen to ensure scalability, performance, and security throughout the system.

On the frontend, React.js provides a fast, responsive user interface that allows tenants to browse properties effortlessly and landlords to manage listings with ease. The component-based architecture of React makes it easy to update and maintain the application, providing users with an intuitive and dynamic browsing experience. React's powerful features such as virtual DOM ensure that the application runs efficiently even with large amounts of data.

The backend is built using Node.js with Express, enabling fast and scalable server-side functionality. Express simplifies the creation of RESTful APIs that handle requests related to user registration, property listing, and communication between tenants and landlords. The backend also manages business logic, ensuring that data is properly validated and processed before being sent to the frontend.

MongoDB, a NoSQL database, is used for storing user profiles, property listings, and other relevant data. Its flexible schema allows for easy storage and retrieval of dynamic data, which is a perfect fit for the ever-changing nature of real estate listings.

Finally, the application employs JWT (JSON Web Tokens) for secure user authentication, ensuring that only authorized users can access sensitive features such as managing listings and making rental inquiries. This robust authentication mechanism guarantees data security and privacy for both tenants and landlords.

Overall, the House Rent Application integrates these technologies to create a secure, efficient, and user-friendly platform that revolutionizes the property rental process.

1.2 PROJECT OVERVIEW:

TECHNOLOGY STACK

- Frontend: React.js
- Backend: Node.js with Express
- Database: MongoDB
- Authentication: JWT (JSON Web Tokens) or similar

The goal of this project is to develop a web-based House Rent Application that allows users to search, list, and manage rental properties. The application will enable landlords to post property listings, and prospective tenants to search, view details, and inquire about properties. The MERN stack (MongoDB, Express.js, React, Node.js) will be used to build this application.

1.3 PURPOSE :

The purpose of the House Rent Application is to create a digital platform that connects property owners and potential tenants. The platform simplifies the process of renting houses by providing a secure and user-friendly interface for searching, listing, and managing rental properties.

1.4 EXISTING SYSTEM :

The existing house rent systems often rely on traditional methods for listing and managing properties, either through static websites or basic platforms with limited functionality. These systems typically lack integration between the frontend and backend, resulting in slow updates and poor user experience. Tenants may struggle to find relevant listings, and landlords often face difficulties in managing property details and tenant interactions. Additionally, many systems lack secure authentication mechanisms, exposing users to potential data breaches.

In these existing systems, frontend frameworks may be outdated or not responsive, making them less user-friendly across different devices. The backend might be built on monolithic architectures that limit scalability, causing performance issues as the number of users grows. Databases may use relational structures that don’t efficiently handle dynamic or complex data like property features and tenant details.

Moreover, authentication is often basic or non-existent, leaving user data vulnerable.

The House Rent Application, with its React.js frontend, Node.js backend, MongoDB database, and JWT authentication, addresses these challenges by offering a modern, secure, and scalable solution.

1.5 PROBLEM STATEMENT :

The traditional house rental process is often inefficient, cumbersome, and lacks modern features that enhance the user experience for both tenants and landlords. Existing platforms frequently struggle with poor user interfaces, slow performance, and limited functionalities, making property searches and management more difficult. Tenants often face challenges finding relevant and up-to-date listings, while landlords may struggle with managing property details and tenant interactions. Additionally, many systems lack secure authentication, leaving sensitive user data vulnerable to unauthorized access and breaches.

Moreover, the current systems tend to rely on outdated technologies, resulting in scalability issues as the number of users and property listings grows. Without real-time updates and integration between the frontend and backend, users experience delays in accessing critical information. Existing solutions may also lack flexibility in handling diverse property details, such as varying lease terms, amenities, or payment methods.

The House Rent Application aims to solve these issues by providing a modern solution with a React.js frontend, which ensures a responsive and dynamic user interface. The Node.js with Express backend allows for fast, scalable processing of user requests, while MongoDB enables flexible data storage that can handle complex, dynamic listings. The integration of JWT authentication ensures secure and reliable user access, protecting sensitive data. This system addresses the problems of inefficiency, security, and scalability in the current house rental landscape.

CHAPTER II

SYSTEM ARCHITECTURE

2.1 ARCHITECTURE:

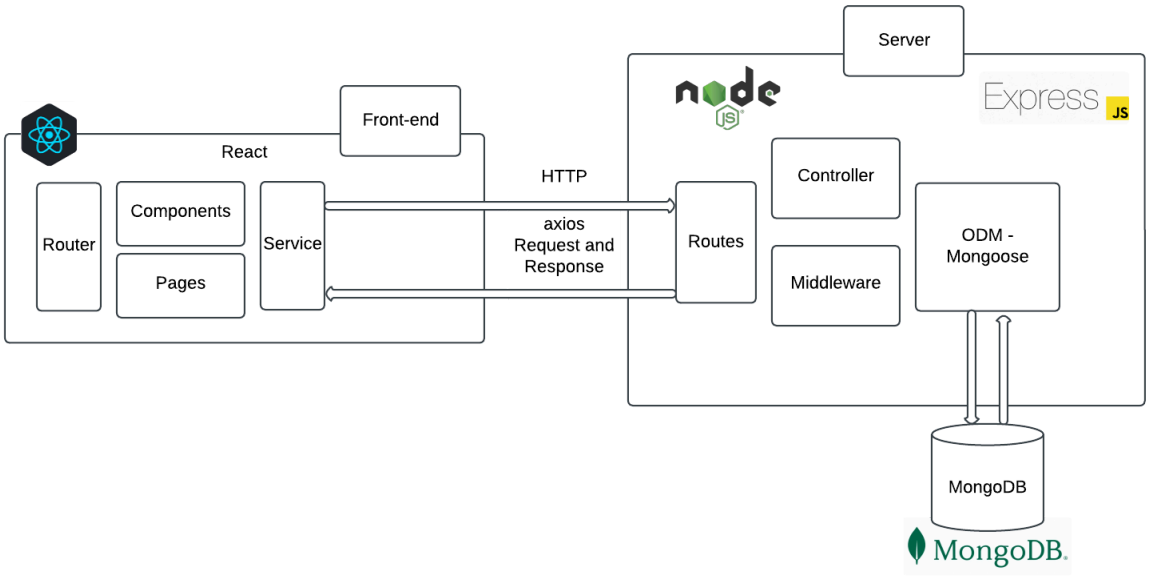


FIGURE 1. ARCHITECTURE

2.2 FRONTEND DEVELOPMENT:

Frontend Development with React.js

The frontend of the application will be developed using React.js, aiming to provide a responsive, single-page application (SPA) experience. React's component-based architecture allows for reusable components and efficient rendering. Below are the key components and functionalities of the application, as well as the state management, authentication, routing, and other details.

Key Components:

Home Component:

This component will serve as the landing page for users. It will display featured properties, which could be highlighted or curated listings. Additionally, it will offer search filters to allow users to quickly narrow down the property listings based on certain criteria such as price, location, size, etc.

Features:

Displaying featured properties (e.g., top picks or recently added listings).

Search bar with filtering options (e.g., price range, location, property type).

Promotional or news banners if needed.

Pagination or infinite scroll for property previews.

Property Listing Component:

This component will display a list of all available properties, offering users a detailed overview of each one. Users can apply additional filters like location, price, number of bedrooms, etc., to narrow down the search results.

Features:

A grid or list view of property cards with key information (name, price, location).

Filter options on the sidebar or as a dropdown menu for easy navigation.

Sorting options (e.g., by price, date, or popularity).

Pagination or infinite scroll for smoother browsing.

Property Details Component:

When a user clicks on a property from the listings, they will be directed to the Property Details Component, which will display all the information about a particular property in a comprehensive format.

Features:

Full property details such as price, location, number of rooms, amenities, etc.

Image gallery or video walkthrough of the property.

Contact form for inquiries.

Map view to show the property's location.

Reviews or ratings from previous tenants (if applicable).

User Profile Component:

This component is dedicated to users who are registered and logged in. It will allow users to manage their personal information, track their saved properties, and list any properties they may have available for rent.

Features:

Edit personal information (name, email, contact details).

View and manage listed properties (add, update, delete).

View saved rentals and saved searches for quick access.

View history of viewed properties for reference.

Account settings (change password, notification preferences).

Admin Dashboard Component:

A restricted access dashboard designed for administrators to monitor and manage the activity within the app. It provides a consolidated view of user activity, property management, and content moderation.

Features:

User management (view, edit, or deactivate user accounts).

Property management (approve, reject, or remove properties).

Analytics (track activity such as property views, user registration, etc.).

Ability to manage reports from users about properties or user behavior.

Control over content (e.g., banners, promotional materials).

Notification management to send alerts or reminders to users.

State Management:

To handle the dynamic nature of the application, state management will be required for managing the data across the application (e.g., user information, property data, filter settings, etc.). There are two main options for managing state in React:

React Context API:

A built-in state management tool within React that allows global state to be shared across components. It's relatively simple and ideal for smaller to medium-sized applications.

Use cases:

Storing user authentication state.

Sharing filter settings across components.

Passing data down the component tree without prop-drilling.

Redux:

A more robust state management solution, especially for larger applications with complex state interactions. Redux can be used to centralize the state in a single store and provides tools for managing side effects, asynchronous actions, and API responses more effectively.

Use cases:

Handling user authentication flow and storing JWT tokens.

Managing the application-wide property listings state.

Storing UI-related state (like filters, loading states).

Both approaches are feasible, and the choice depends on the scale and complexity of the application. For a medium-sized app, React Context might suffice, but for larger or more complex applications, Redux could be the better choice.

Authentication:

The application will use JWT (JSON Web Tokens) for handling user authentication. Here's how it will work:

Login:

When a user logs in, they'll provide their credentials (username and password), and upon successful authentication, the server will return a JWT.

This JWT is stored either in localStorage or HTTP-only cookies for subsequent requests.

Token Storage:

LocalStorage is an easy way to persist tokens but can be vulnerable to cross-site scripting (XSS) attacks.

HTTP-only cookies are more secure, as they can't be accessed by JavaScript, mitigating XSS risks but can be prone to cross-site request forgery (CSRF) if not implemented correctly.

API Authorization:

Every request to the backend (such as fetching user data or managing properties) will include the JWT in the request header (Authorization: Bearer <token>).

The backend will verify the token and ensure the user has appropriate permissions for the requested action.

Token Expiry and Refresh:

JWT tokens will have an expiry time. The application will need to handle token expiration and provide a refresh token to renew the JWT when needed.

Routing:

For navigation between different pages or components, React Router will be used. React Router is a powerful routing library that allows you to manage navigation and URLs in a single-page application without the need for full page reloads.

2.3 BACKEND DEVELOPMENT:

Node.js and express.js

The backend will be built using Node.js and Express.js to handle HTTP requests, API routes, and business logic. Node.js provides an event-driven, non-blocking I/O model, ideal for handling high-concurrency applications, while Express.js simplifies route management, request handling, and middleware usage. Together, they will ensure the backend is scalable, fast, and efficient, providing a solid foundation for handling complex interactions like user authentication and property management.

Restful api

A RESTful API will be built to handle client-server communication with standard HTTP methods (GET, POST, PUT, and DELETE). The API will provide endpoints for key functionalities such as user authentication, property management, and notifications. The API will return data in JSON format and will be designed to be stateless, with error handling integrated for robust communication between the frontend and backend.

User authentication

User authentication will be handled using JWT (JSON Web Tokens), allowing users to sign up, log in, log out, and reset their passwords securely:

✔ ***Signup***: Users provide email, password, and other details, and upon validation, their credentials are saved in the database with hashed passwords.

✔ ***Login:*** Upon successful login, a JWT token is issued, which is stored in local Storage or HTTP-only cookies for future authentication.

✔ ***Logout***: JWT tokens are removed from storage, and the user is logged out.

✔ ***Password Reset***: A secure link is sent to the user's email to reset their password.

The backend ensures secure handling of sensitive data, including encrypted passwords and token-based authentication.

Property management

Property management will allow CRUD (Create, Read, Update, Delete) operations on property listings:

▪ ***Create Property***: Users can submit property details (e.g., price, location, photos) to create new listings.

- ***Read Property***: Users can view property listings or details with filtering options (location, price, type).
- ***Update Property***: Property owners can update their listings (price, description, etc.).
- ***Delete Property***: Property owners can delete their listings.

Images for properties will be uploaded to a cloud service like Amazon S3, with URLs stored in the database.

Messaging system

The Messaging System will have API routes for sending and receiving messages between users (tenants, landlords, and admins). Each message will be stored in a database and associated with relevant users and properties. Routes will include:

POST /messages: For sending messages between users.

GET /messages: For retrieving conversations or specific messages.

GET /messages/:id: For viewing individual message details.

This system will allow real-time communication between tenants and landlords for inquiries or negotiations.

Admin operations

Admin Operations will include routes for managing users, properties, and reports.

Security and authorization

Security and Authorization will be managed using JWT for role-based access control. Middleware will be implemented to protect routes based on user roles (e.g., admin, tenant, landlord):

- ***JWT Authentication Middleware:*** Ensures users are logged in before accessing sensitive data.
- ***Role-based Authorization Middleware:*** Checks user roles for accessing specific routes (e.g., only admins can access user management routes).

This ensures that only authorized users can access restricted resources and perform actions based on their role.

File uploads

A dedicated endpoint for image uploads will be set up to handle property photos. The backend will allow users (landlords) to upload images for properties.

The images will be stored securely in the cloud, and URLs will be stored in the database, linked to the relevant property listings.

Database:

The database uses MongoDB with a schema defined through Mongoose. Collections are created to manage Users, Products, Orders, and Cart items, ensuring data integrity and efficient querying.

2.4 MONGODB:

Use MongoDB to store data related to users, properties, reviews, messages, and more.

Database models:

1. User Model: Stores user information such as name, email, password (hashed), role (tenant, landlord, admin), and profile details.

2. Property Model: Stores information about each property, including address, description, rent price, owner ID, images, and other details.

3. Message Model: Stores message threads between users.

4. Booking or Viewing Requests Model: Stores tenant requests for viewing or booking properties.

5. Review Model: Stores reviews submitted by tenants for properties and landlords.

6. Indexes: Ensure frequently searched fields, like location and price, are indexed for faster queries

2.5 KEY FEATURES

User Authentication:

- Sign up, login, and logout features for users (tenants and property owners).
- Role-based access control to distinguish between tenants and property owners.

- **Property Listings:**
- Property owners can add new property listings, including details like rent amount, property location, photos, and amenities.
- Tenants can browse and filter listings based on location, price, property type, and other criteria.

- **Search and Filter:**
- A search and filter option for tenants to find listings based on specific criteria (e.g., address, Ad type (rent/sale), Property type(residential/commercial/plots)).

4. Detailed Property Page:

- A dedicated page for each property with all relevant details, photos, a description, and owners contact information.
- An option for tenants to save or bookmark properties for later.

5. Booking or Viewing Requests

- Tenants can send requests to view the property or book it.
- Property owners can manage these requests and respond to tenants.

6. Admin Dashboard

- A dashboard for admins to oversee all activity on the platform, including user management, property listings, and resolving disputes.

CHAPTER III

SETUP INSTRUCTIONS

3.1 BACKEND PREREQUISITES:

1. **Prerequisites:**

- ❖ Node.js and npm: Download and install from Node.js.
- ❖ MongoDB: You can set up a local MongoDB instance or use MongoDB Atlas for a cloud database.
- ❖ Git: For version control (optional but recommended).

2 .Installation:

Navigate to the backend directory: cd backend

✔ **Install the backend dependencies:**
bash npm install express mongoose dotenv jsonwebtoken bcryptjs cors

✔ **.Install development dependencies for easier development:**
bash npm install --save-dev nodemon

3. Create a .env file for environment variables in the backend directory and add the following variables (with appropriate values for your setup):

makefile

MONGO_URI=your_mongodb_connection_string

JWT_SECRET=your_jwt_secret

PORT=3000

4. Start the backend server in development mode:

npm start

5.This will run the backend on http://localhost:3000 .

3.2 FRONTEND PREREQUISITES:

- **Install the frontend dependencies:**

npm install react react-dom react-router-dom axios

- **Start the React application:**

npm start

This will start the frontend server on http://localhost:3000.

- **Install Frontend Dependencies**

Navigate to the client directory:

cd frontend

CHAPTER IV

FOLDER STRUCTURE

4.1 CLIENT:

```
frontend/
│
├── public/            # Static assets (e.g., index.html, images)
├── src/               # React components, state management, and hooks
│   ├── components/    # Reusable UI components
│   ├── pages/         # Individual pages (e.g., Home, Login, Register)
│   ├── services/      # API calls to backend
│   ├── modules/       # Organized feature-specific components
│   │   ├── admin/     # Admin-related components (e.g., AdminHome, AllUsers)
│   │   ├── common/    # Common components (e.g., Login, Register)

│   │   ├── user/      # User-related components
│   │   │   ├── Owner/  # Owner-specific components (e.g., AddProperty)
│   │   │   └── Renter/ # Renter-specific components (e.g., RenterHome)
│   ├── App.css        # Global styles
│   ├── App.js         # Main app component
│   └── index.js       # Entry point for React app
├── package.json       # Frontend dependencies and scripts
└── README.md          # Project documentation
│   ├── App.js         # Main app component
│   └── index.js       # Entry point for React app
└── package.json       # Frontend dependencies and scripts
```

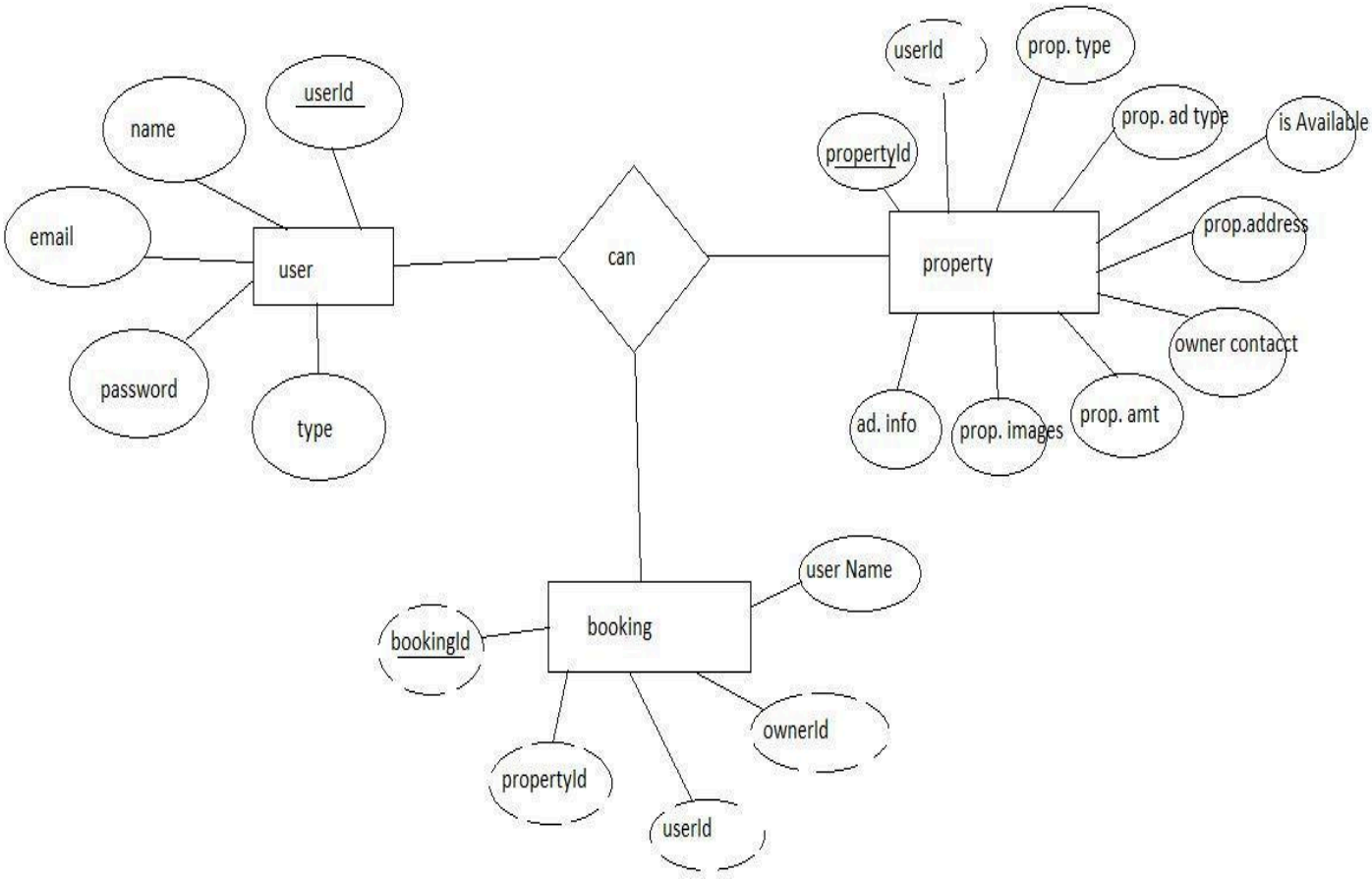
4.2 SERVER:

```
Backend/
│
├── config/            # Configuration files (e.g., DB connection)
│   └── connect.js     # Database connection setup
├── controllers/       # Business logic for each route
│   ├── adminController.js
│   ├── ownerController.js
│   └── userController.js
├── middlewares/       # Authentication, error handling, etc.
│   └── authMiddleware.js
├── routes/            # API routes (e.g., /auth, /users, /properties)
│   ├── adminRoutes.js
│   ├── ownerRoutes.js
│   └── userRoutes.js
├── schemas/           # Mongoose models
│   ├── bookingModel.js
│   ├── propertyModel.js
│   └── userModel.js
├── uploads/           # Folder for file uploads (e.g., images)
├── server.js          # Main entry point for the server
├── .env               # Environment variables
├── package.json       # Backend dependencies and scripts
└── README.md          # Server documentation
```

CHAPTER V

SYSTEM MODELING

5.1 ENTITY RELATION DIAGRAM:



Here there is 3 collections namely users, property, and booking which have their own fields in

Users:

1. _id: (MongoDB creates by unique default)
2. name
3. email

4. password
5. type

Property:

1. userID: (can be act as foreign key)
2. _id: (MongoDB creates by unique default)
3. prop.Type
4. prop.AdType
5. isAvailable
6. prop.Address
7. owner contact
8. prop.Amt
9. prop.images
10. add.Info

Booking:

1. _id: (MongoDB creates by unique default)
2. propertId
3. userId
4. ownerId
5. username

CHAPTER VI

RUNNING THE APPLICATION

Prerequisites

Install Node.js (latest LTS version recommended).

Install MongoDB and ensure it's running locally or use a cloud-based MongoDB service like MongoDB Atlas.

Ensure you have npm or yarn installed (comes with Node.js).

Clone the repository to your local machine:

git clone https://github.com/Kannannair07/Residence-Aura_Naan-Mudhalvan_Project.git

cd Residence Aura+

Backend Setup

Navigate to the backend directory:

cd backend

Install dependencies:

npm install

Create a .env file in the backend folder and add the following environment variables:

PORT=5000

MONGO_URI=<your_mongodb_connection_string>

JWT_SECRET=<your_jwt_secret>

Replace <your_mongodb_connection_string> with your MongoDB connection string (e.g., from MongoDB Atlas or your local MongoDB instance).

Replace <your_jwt_secret> with a strong secret key for JWT token generation.

Start the backend server:

npm start

Frontend Setup:

Navigate to the frontend directory:

cd ../frontend

Install dependencies:

npm install

Start the React development server:

npm start

The frontend server should start at http://localhost:3000.

Access the Application

Open http://localhost:3000 in your browser.

Directory Overview

backend: Contains server-side code, API endpoints, and MongoDB schema models.

frontend: Contains React-based client-side code for the user interface.

CHAPTER VII

API DOCUMENTATIONS

7.1 AUTHENTICATION / RESPONSE

Authentication:

1. Login:

URL: /auth/login

Method: POST

Description: Authenticates a user and returns a JWT token.

Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Response:

Success (HTTP 200):

```
{
  "message": "Login successful",
  "token": "your_jwt_token"
}
```

Failure (HTTP 401):

```
{
```

"message": "Invalid email or password"
}

2. **Register**

URL: /auth/register

Method: POST

Description: Registers a new user.

Request Body:

{
"name": "Roy",
"email": "user@example.com",
"password": "password123",
"role": "owner" // or "renter"
}

Response:

Success (HTTP 201):

{
"message": "User registered successfully"
}

Failure (HTTP 400):

{
"message": "Email already exists"
}

Properties

3.**Add Property**
URL: /properties/add
Method: POST
Description: Allows an owner to add a new property.
Headers:
Authorization: Bearer <your_token>
Request Body:

```
{
  "title": "3 BHK Apartment",
  "description": "A beautiful apartment in the city center.",
  "price": 25000,
  "address": "123 Street Name, City",
  "image": "base64_image_data"
}
```

Response:
Success (HTTP 201):

```
{
  "message": "Property added successfully"
}
```

Failure (HTTP 400):

```
{
  "message": "Failed to add property"
}
```

4. **Get All Properties**
URL: /properties
Method: GET
Description: Fetches all properties for renters to view.
Response:

Success (HTTP 200):

```
[
  {
    "id": "123",
    "title": "3 BHK Apartment",
    "description": "A beautiful apartment in the city center.",
    "price": 25000,
    "address": "123 Street Name, City",
```

```
    "image": "image_url"
  }
]
```

5**. Delete Property**
URL: /properties/:id
Method: DELETE
Description: Allows an owner to delete a property.
Headers:
Authorization: Bearer <your_token>
URL Params:
id: Property ID
Response:
Success (HTTP 200):

```
{
  "message": "Property deleted successfully"
}
```

Failure (HTTP 404):

```
{
  "message": "Property not found"
}
```

Bookings

6. **Create Booking**

URL: /bookings/create

Method: POST

Description: Allows a renter to book a property.

Headers:

Authorization: Bearer <your_token>

Request Body:

```
{
  "propertyId": "123",
  "startDate": "2024-11-25",
```

"endDate": "2024-11-30"
}

Response:

Success (HTTP 201):

{
"message": "Booking created successfully"
}

Failure (HTTP 400):

{
"message": "Failed to create booking"
}

7. **Get All Bookings**

URL: /bookings

Method: GET

Description: Fetches all bookings for the logged-in user.

Headers:

Authorization: Bearer <your_token>

Response:

Success (HTTP 200):

[
{
"id": "123",
"propertyId": "123",
"startDate": "2024-11-25",

```
    "endDate": "2024-11-30",
    "status": "confirmed"
  }
]
```


Admin Operations

8. **Get All Users**

URL: /admin/users

Method: GET

Description: Fetches all users (admin only).

Headers:

Authorization: Bearer <your_token>

Response:

Success (HTTP 200):


```
[
  {
    "id": "123",
    "name": "John Doe",
    "email": "user@example.com",
    "role": "owner"
  }
]
```


9. **Delete User**

URL: /admin/users/:id

Method: DELETE

Description: Allows the admin to delete a user.

Headers:

Authorization: Bearer <your_token>

URL Params:

id: User ID

Response:

Success (HTTP 200):

```
{
  "message": "User deleted successfully"
}
```

Error Handling

400 Bad Request: When invalid data is provided.

401 Unauthorized: When a token is missing or invalid.

404 Not Found: When a resource does not exist.

500 Internal Server Error: When something unexpected occurs on the server.

CHAPTER VIII

AUTHENTICATION

Authentication is the process of verifying the identity of a user or system to ensure that the individual or entity is who they claim to be. It is a critical aspect of web application security, ensuring that only authorized users can access certain resources or functionalities.

8.1 TYPES OF AUTHENTICATION METHODS:

1. **Basic Authentication**:

Description: Involves sending a username and password in the HTTP header. This method is simple but not secure on its own, as credentials are often transmitted in plaintext unless the connection is secured (e.g., over HTTPS).

Use Cases: Often used for quick, low-level access where security isn't a major concern or in legacy systems.

2. Form-Based Authentication:

Description: Involves a login form where users enter their credentials (username and password), which are then validated by the server. This method can be enhanced with additional features like session management (cookies) and encryption.

Use Cases: Commonly used in web applications, especially those that require user registration and login.

3. Token-Based Authentication (e.g., JWT):

Description: The server generates a token (usually a JSON Web Token, JWT) after the user logs in. The token is then sent with each request, typically in the HTTP Authorization header. This method is stateless and doesn't require the server to store session data.

Use Cases: Ideal for RESTful APIs and single-page applications (SPAs), especially for handling distributed, scalable systems.

4. Session-Based Authentication:

Description: After a successful login, the server creates a session and stores session information, typically in a cookie. The session is used to track user interactions and maintain login state without requiring the user to re-enter credentials.

Use Cases: Most traditional web applications use session-based authentication for maintaining user sessions.

5. **API Key Authentication**:

Description: Involves sending an API key, which is a unique identifier, in requests to authenticate API calls. The server verifies the key and grants access to the requested resources.

Use Cases: Common in APIs and microservices, especially for public or internal APIs.

8.2 AUTHENTICATION FLOW:

User Login: The user submits credentials (username and password) via a login form or API endpoint.

Server Validation: The server checks the credentials against a database. If the credentials are valid, the server generates a session or token.

Session/Token Creation: For session-based authentication, the server creates a session and stores it in memory or a database. For token-based authentication (like JWT), a token is generated and sent to the client.

Accessing Resources: The client sends the session ID or authentication token with each request to access protected resources.

Logout/Session Expiry: On logout, the session is destroyed, or the token expires, requiring the user to authenticate again.

8.3 AUTHENTICATION SECURITY PRACTICES:

- **Use HTTPS**: Always use HTTPS to encrypt credentials during transmission, ensuring that sensitive data like usernames and passwords aren't exposed during transit.
- **Password Hashing**: Store passwords securely by hashing them (using algorithms like bcrypt or Argon2) instead of storing them in plaintext.
- **Avoid Storing Plaintext Tokens**: If using tokens like JWT, ensure they are signed and optionally encrypted to prevent tampering.
- **Limit Login Attempts**: Implement mechanisms to prevent brute-force attacks by limiting the number of login attempts per user.
- **Session Timeout**: Implement session expiration to reduce the risk of unauthorized access in case a session is hijacked.
- **Secure Cookie Attributes**: Set cookies with secure attributes such as HttpOnly, Secure, and SameSite to prevent cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

CHAPTER IX

USER INTERFACE

The User Interface (UI) plays a crucial role in allowing users to interact with a web application, particularly when it comes to logging in and authenticating their identity. In the context of a House Rent Application using the MERN stack, the login process and integration with a MongoDB database ensure that user credentials are stored and securely managed.

User Login Process

- **Login UI**: The login page allows users to input their credentials (username and password). This page typically includes a form with fields for the user's username/email and password, along with a button to submit the form and attempt to log in. Additional options, such as a forgot password link or a signup link for new users, might also be available.
- **Submitting Credentials**: Once the user enters their information, the form data is sent to the backend for verification. The backend checks the credentials against the stored user data in the database (MongoDB).

BACKEND AUTHENTICATION PROCESS

MongoDB Database*:*

- MongoDB stores user data, including usernames, email addresses, and hashed passwords. The database is structured as a collection of documents, each representing a user's profile.
- When a user submits their login credentials, the backend queries the MongoDB database to find the document that matches the entered username or email.

Password Verification:

The backend compares the entered password with the hashed version stored in MongoDB. This ensures that passwords are not stored in plaintext, offering better security for the user's sensitive data.

Session or Token Creation*:*

Once the credentials are validated, the server generates a session or an authentication token (e.g., JWT - JSON Web Token) that is sent back to the frontend. This token is used to authenticate future requests from the user without needing to re-enter credentials.

STORING DATA IN MONGODB

User Data Storage:

- After user registration (sign-up), the credentials (username, email, password) are stored in MongoDB. The password is hashed before it is saved, ensuring that sensitive data is protected.
- MongoDB provides flexibility in storing various types of data, including user profiles, preferences, and other personalized settings.

Data Retrieval:

- Once the user is logged in, their session or token allows them to access personalized resources, such as viewing rental listings, updating their profile, or managing other data specific to the application.
- The backend queries MongoDB to retrieve the necessary data for the authenticated user.

SESSION MANAGEMENT AND ACCESS CONTROL

Session Handling:

- After successful login, a session is created on the server, or a token is issued. This token is stored on the client side (usually in local storage or cookies) and is included in subsequent requests to authenticate the user.
- Sessions or tokens ensure that users remain logged in across different pages or actions within the application.

Access Control:

The backend checks the session or token with each request to confirm that the user is authorized to access certain resources. For example, a user can view their rental listings or update their profile, but they cannot access another user's data unless they have the appropriate permissions.

User Interface Feedback

Successful Login:

Upon successful authentication, the UI typically redirects the user to the main dashboard or home page. The user can now interact with the application, such as browsing rental listings, submitting inquiries, or updating personal information.

Error Handling:

If the login attempt fails (e.g., incorrect credentials), the UI will show an error message, prompting the user to check their username and password or try resetting their password.

SECURITY CONSIDERATIONS

Data Protection:

Sensitive information such as passwords is never stored in plain text. Instead, they are securely hashed before being saved in MongoDB. This helps prevent unauthorized access if the database is compromised.

Token Expiry:

Tokens used for authentication often have an expiration time. This helps prevent unauthorized access if a session is left open for too long, enhancing security.

CHAPTER X

TESTING

When developing a rental home application using the MERN stack (MongoDB, Express.js, React.js, Node.js), it's crucial to implement various testing strategies to ensure that the application works as expected and meets the requirements. Each part of the MERN stack has its own testing needs, and there are different tools and methodologies you can use for each layer of the stack. Below is a detailed breakdown of the testing strategies and the tools you can use for a rental home application.

10.1 UNIT TESTING

Objective:

Ensure that individual units or components of your application (such as functions, classes, or React components) work as expected in isolation.

Tools:

Frontend (React.js):

Jest: A popular testing framework for JavaScript that works well with React. It provides utilities to mock functions and check if a component behaves as expected.

React Testing Library: This helps you test React components by rendering them and interacting with them as a user would (e.g., clicking buttons, entering input).

Enzyme (Alternative to React Testing Library): Another testing utility for React that allows you to shallow render components, simulate events, and make assertions on component behaviour

Backend (Node.js/Express.js):

Jest or **Mocha + Chai**: These tools allow you to test API endpoints, business logic, and database interaction in isolation. You can mock dependencies like database calls to test your functions without interacting with the actual database.

10.2 INTEGRATION TESTING

Objective:

Ensure that multiple components or services work together as expected. This includes interactions between the frontend and backend, as well as between backend services like database operations.

Tools:

Supertest: A popular library for testing HTTP endpoints. It can be used to test how well the frontend interacts with the backend.

Jest or Mocha + Chai for integration testing.

MongoDB Memory Server: For testing the backend in isolation without affecting the real database. It creates an in-memory MongoDB instance to run tests.

10.3 END-TO-END(E2E) TESTING:

Objective:

Simulate real-world user scenarios and test the entire application stack from start to finish. E2E tests verify the user experience across all layers of the app.

Tools:

Cypress: A popular end-to-end testing framework that allows you to interact with your app in a real browser and simulate user interactions. It supports features like automatic waiting, debugging, and screenshot capturing.

Selenium: An alternative to Cypress, useful for testing in multiple browsers, but more complex to set up.

Playwright: Another end-to-end testing framework similar to Cypress, but it allows for testing across multiple browsers (Chrome, Firefox, WebKit).

10.4 SECURITY TESTING:

Objective:

Identify vulnerabilities and ensure that the application handles sensitive data securely.

Tools:

OWASP ZAP (Zed Attack Proxy): A tool for finding security vulnerabilities in web applications. It can be used for penetration testing and scanning for common vulnerabilities such as XSS, SQL injection, and CSRF.

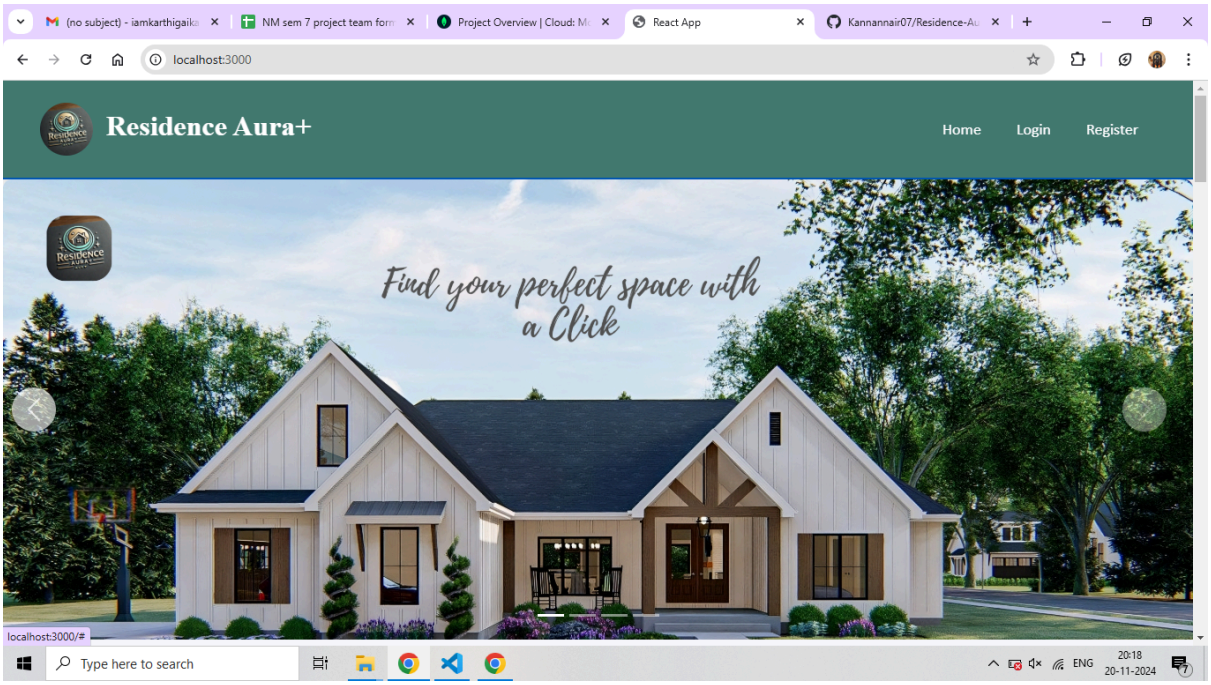
Burp Suite: Another tool for security testing and vulnerability scanning.

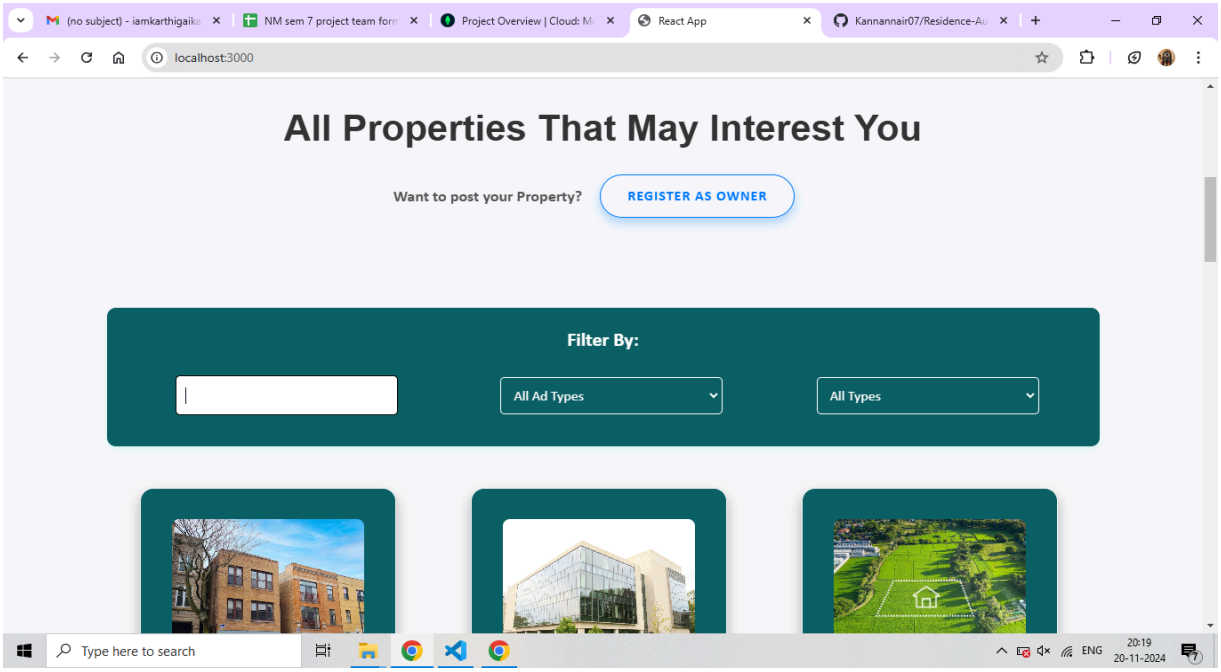
Jest (for testing API authentication): Ensure that unauthorized users cannot access protected routes.

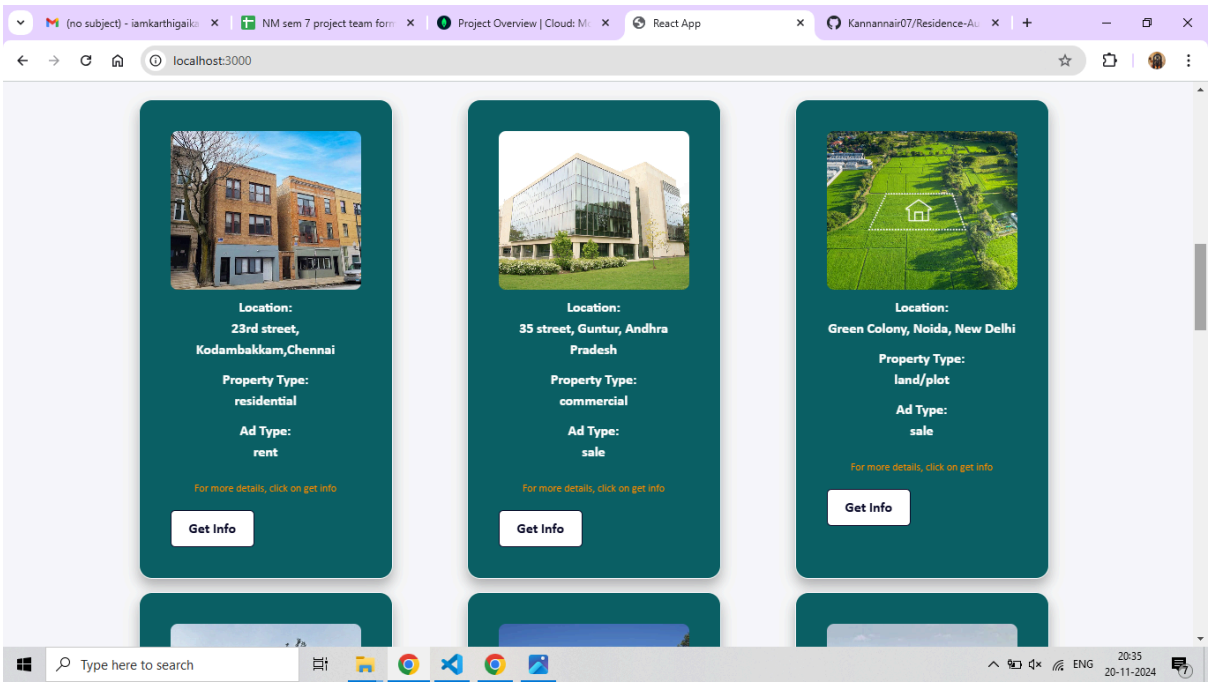
CHAPTER XI

SCREENSHOTS & DEMO

11.1 HOME PAGE :

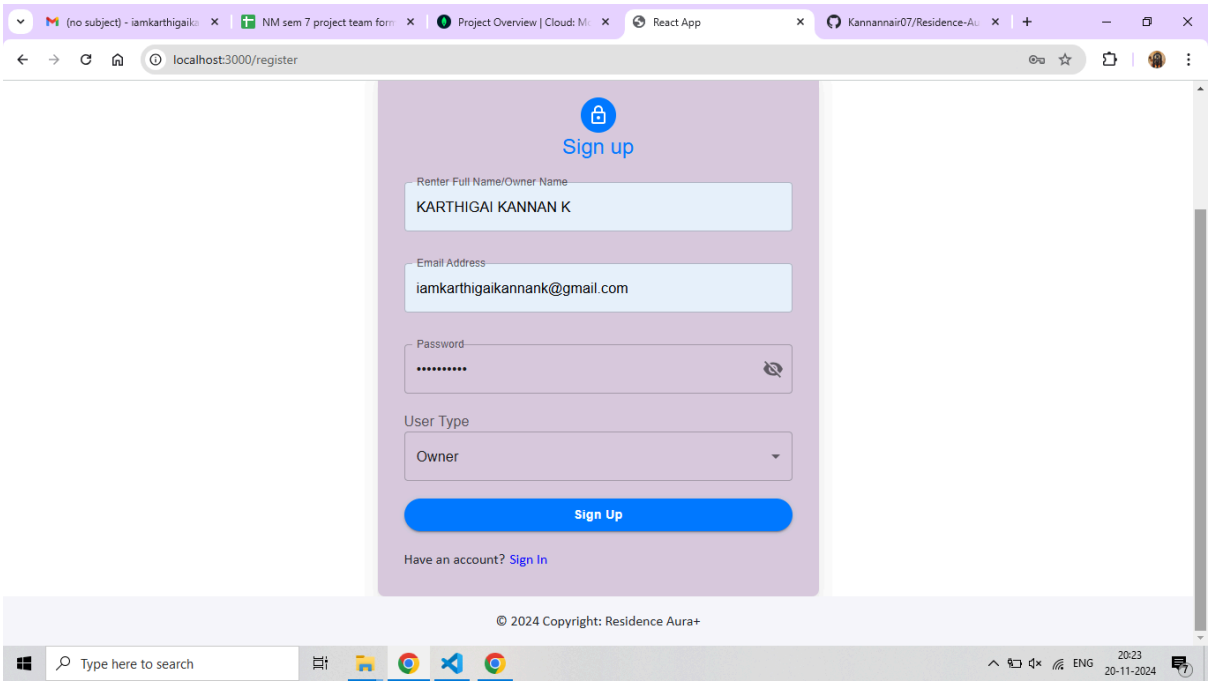




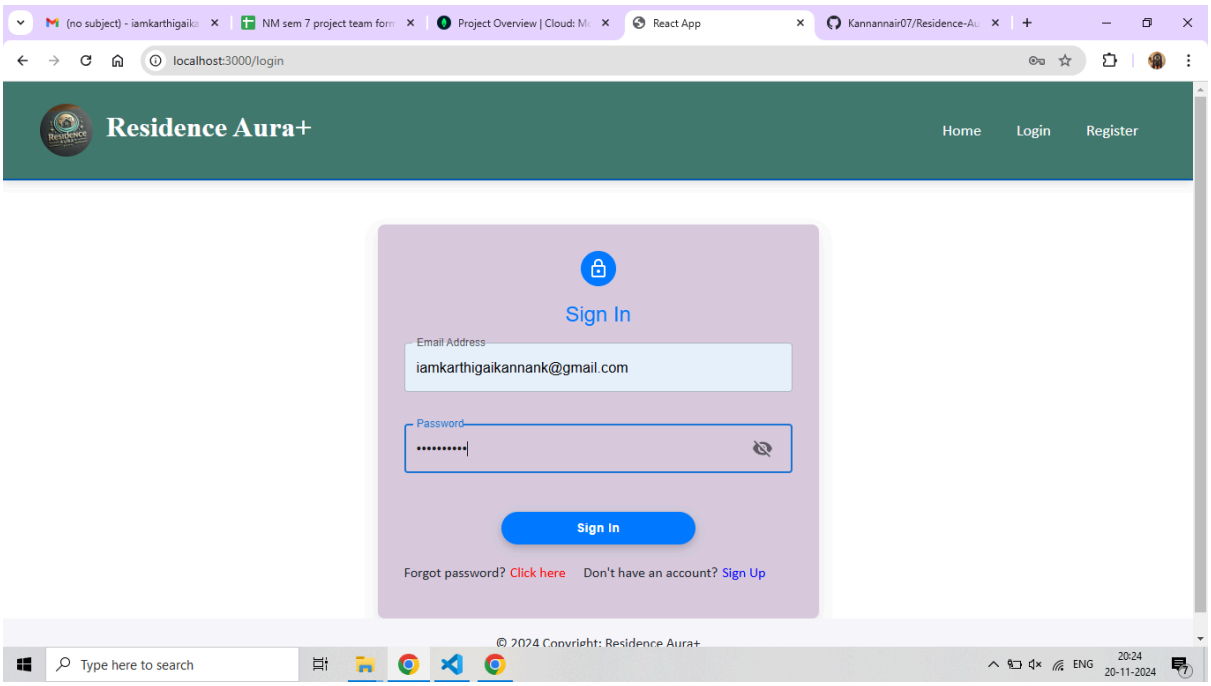


11.2 LOGIN & REGISTER PAGE:

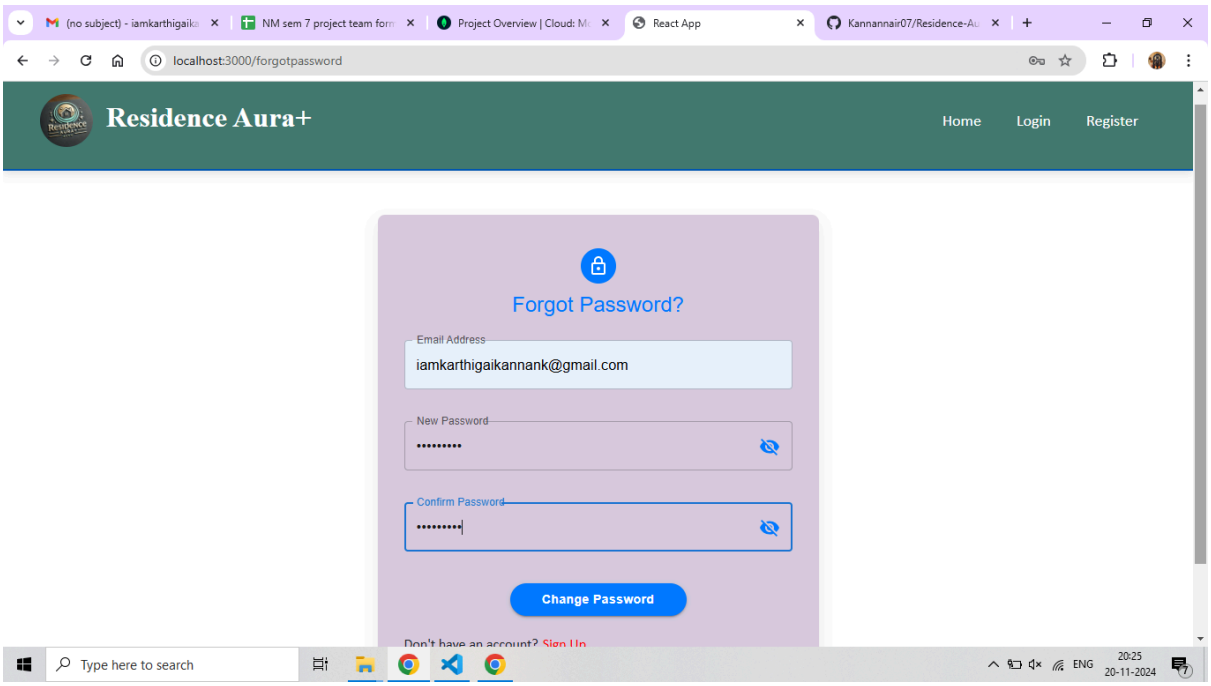
11.2.1 REGISTER:



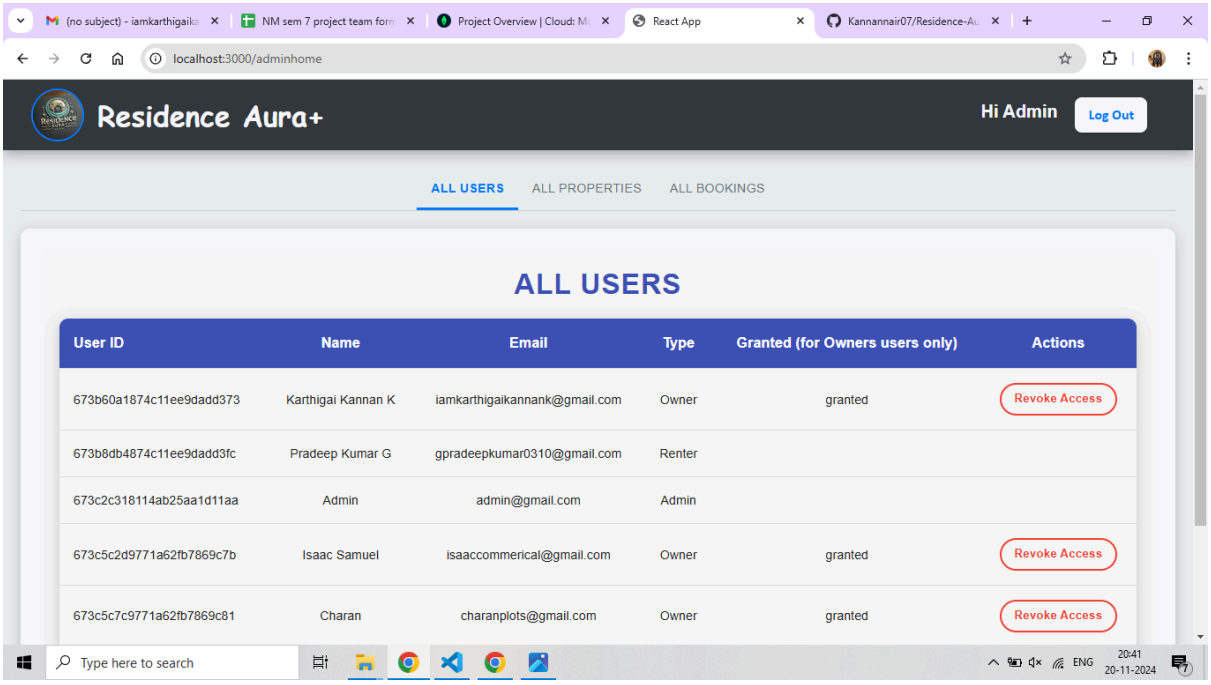
11.2.2 LOGIN:

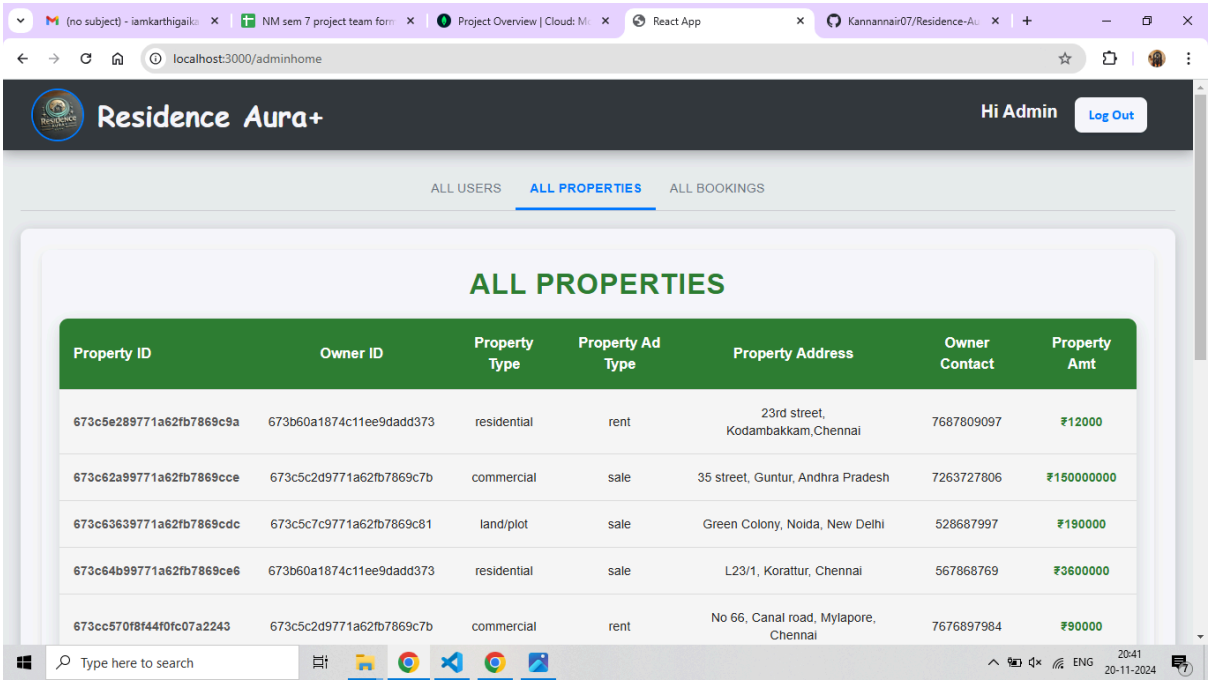


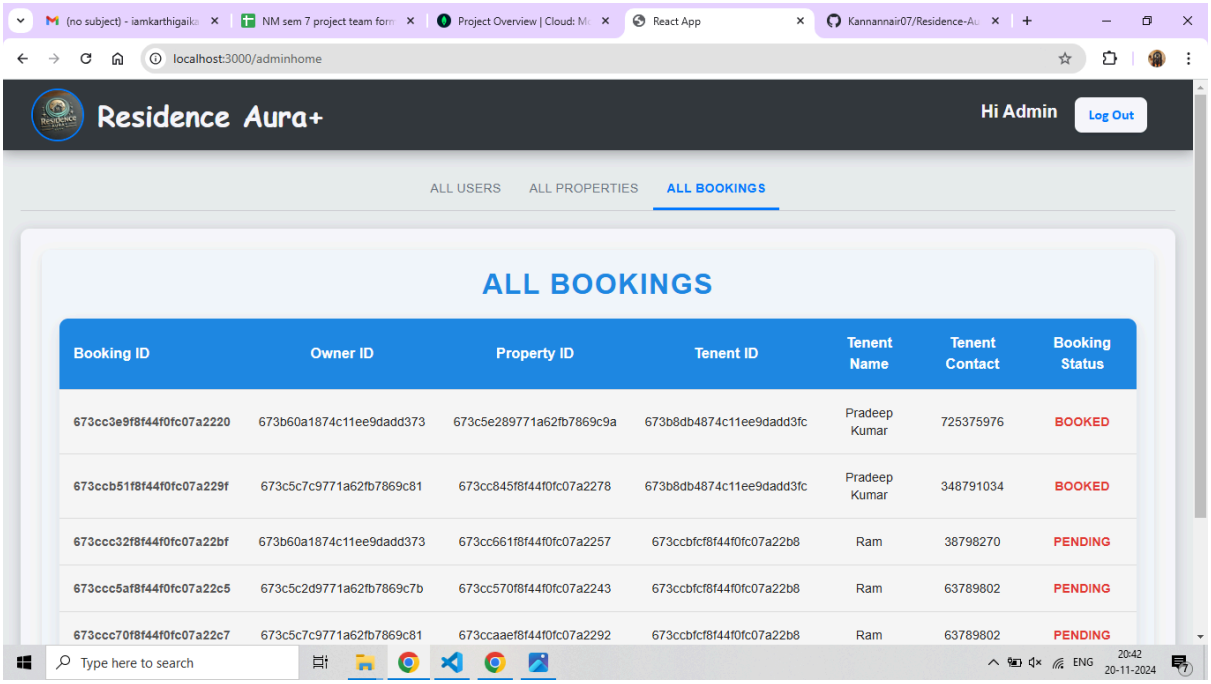
11.2.3 FORGOT PASSWORD:



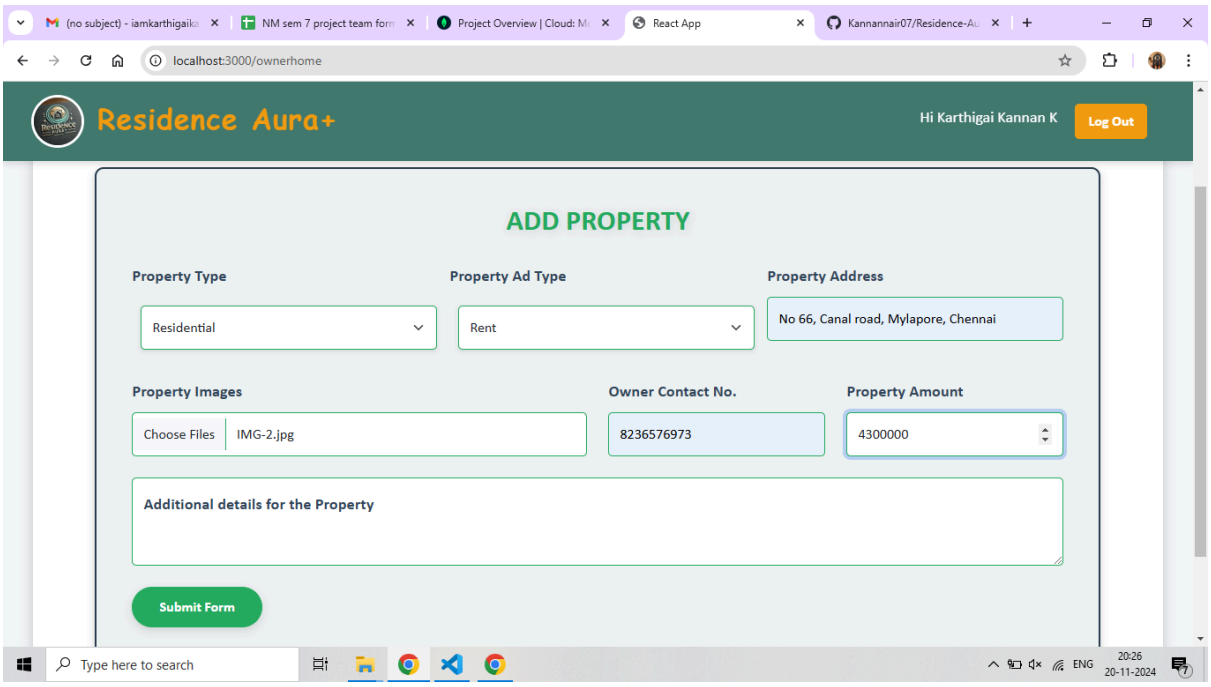
11.3 ADMIN PANEL:

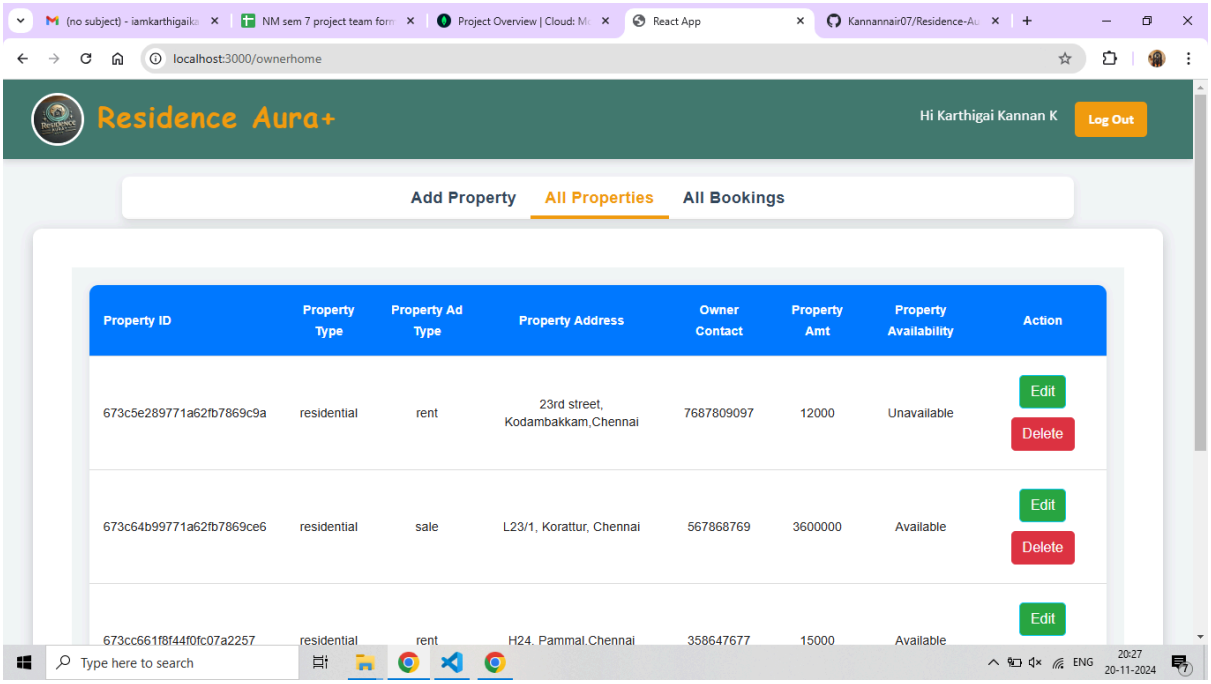


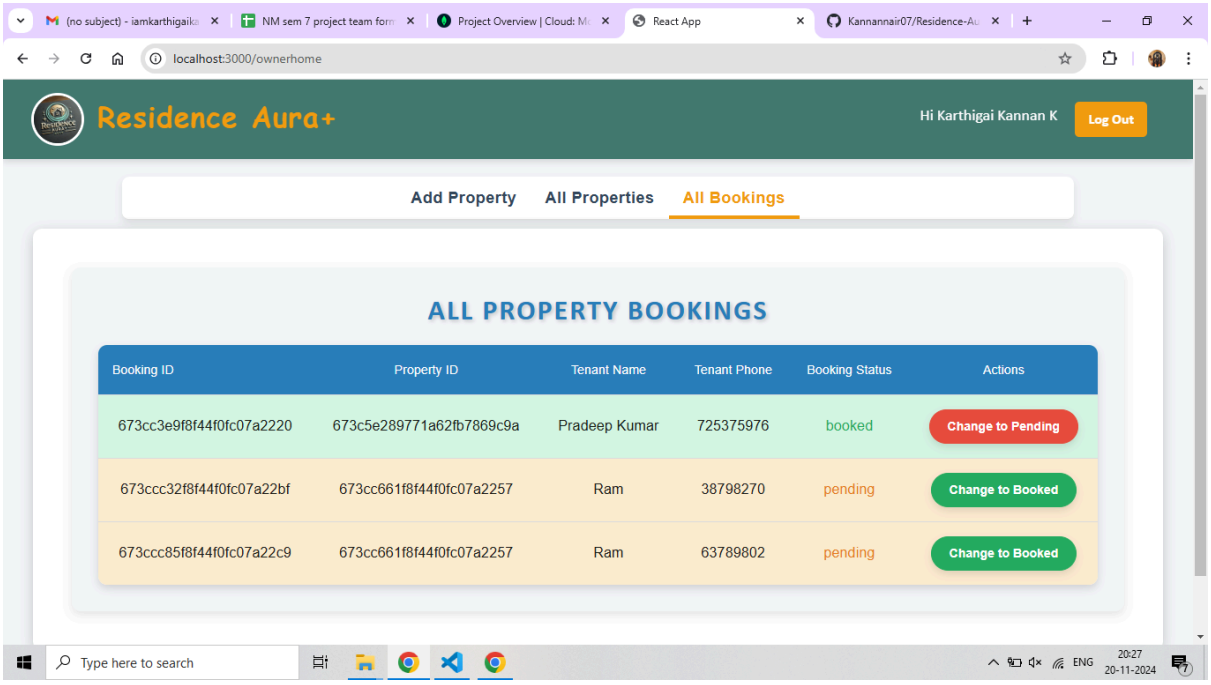




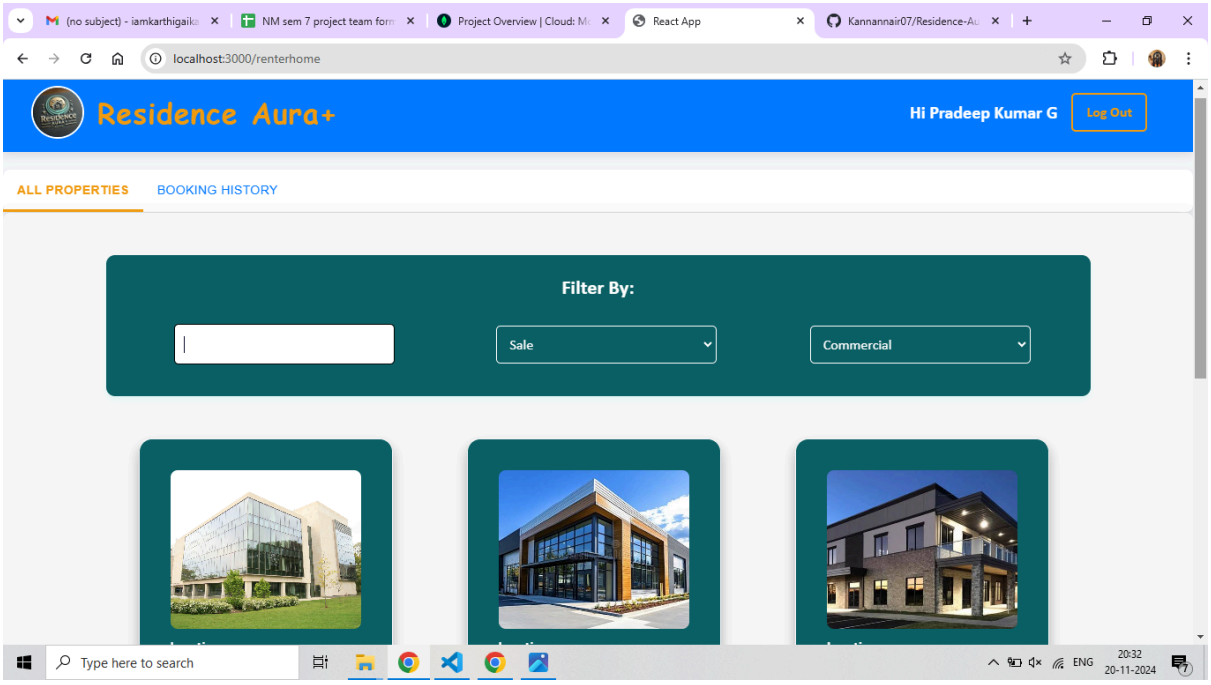
11.4 OWNER PANEL:

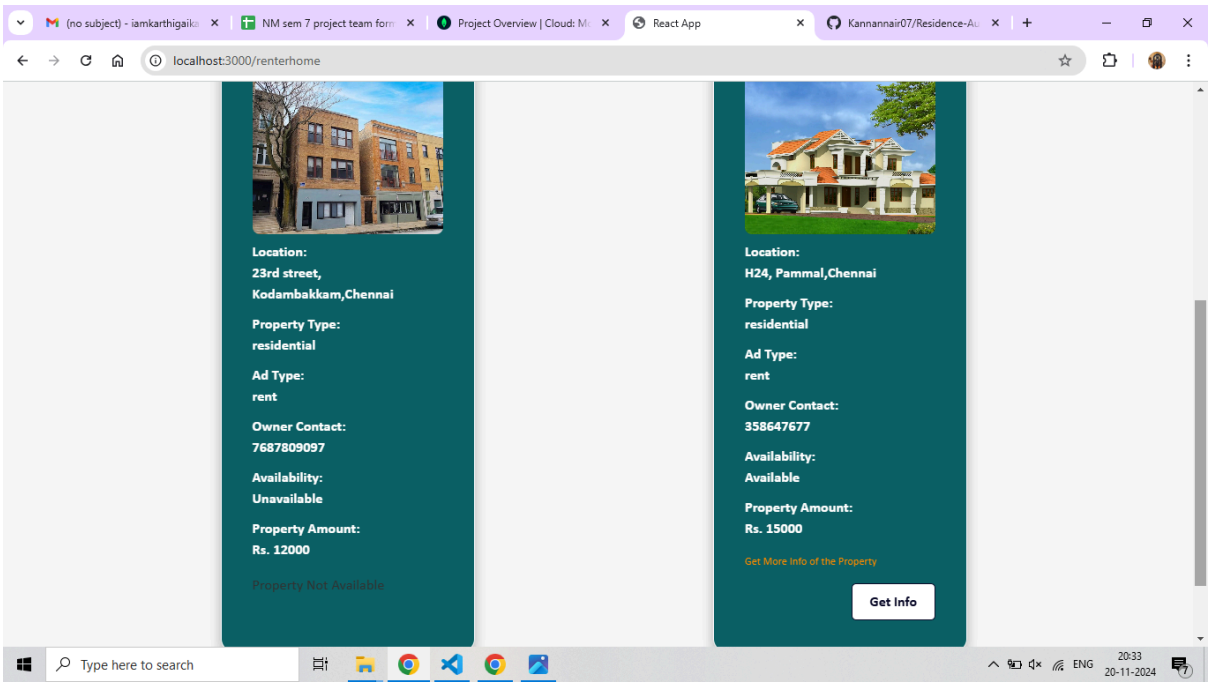


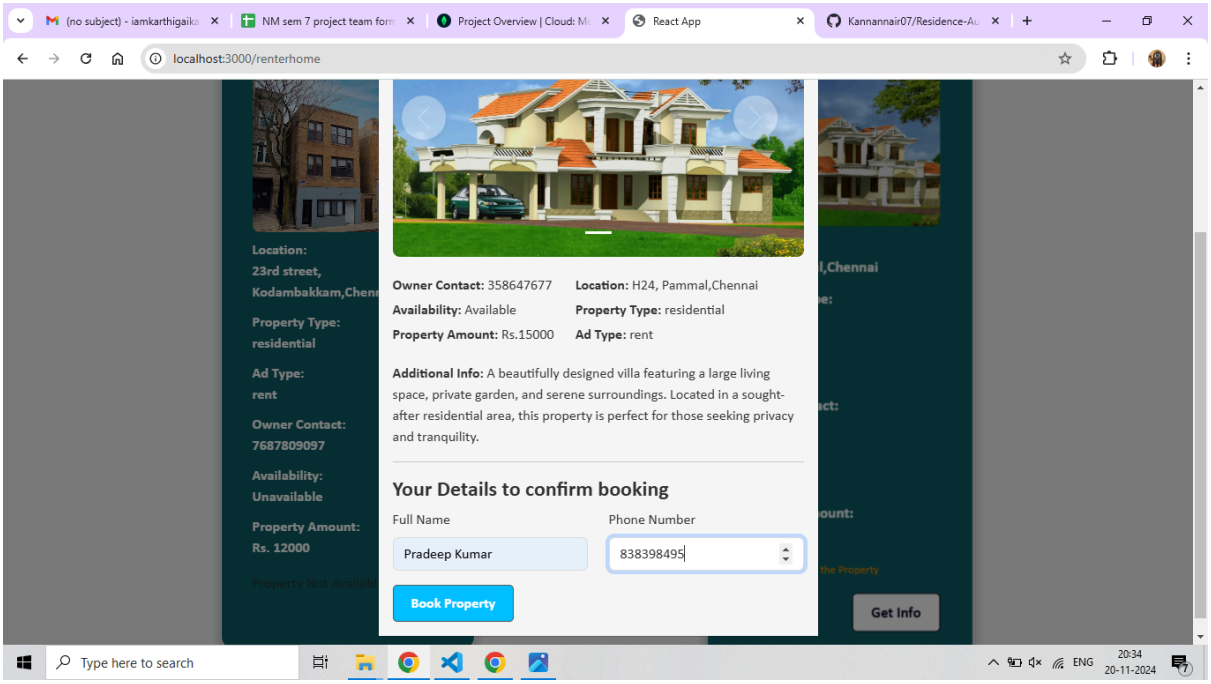

Residence Aura+
Hi Karthigai Kannan K
Log Out
Add Property
All Properties
All Bookings
Property ID
Property Type
Property Ad Type
Property Address
Owner Contact
Property Amt
Property Availability
Action
673c5e289771a62fb7869c9a
residential
rent
23rd street, Kodambakkam,Chennai
7687809097
12000
Unavailable
Edit
Delete
673c64b99771a62fb7869ce6
residential
sale
L23/1, Korattur, Chennai
567868769
3600000
Available
Edit
Delete
673cc661f8f44f0fc07a2257
residential
rent
H24, Pammal,Chennai
358647677
15000
Available
Edit

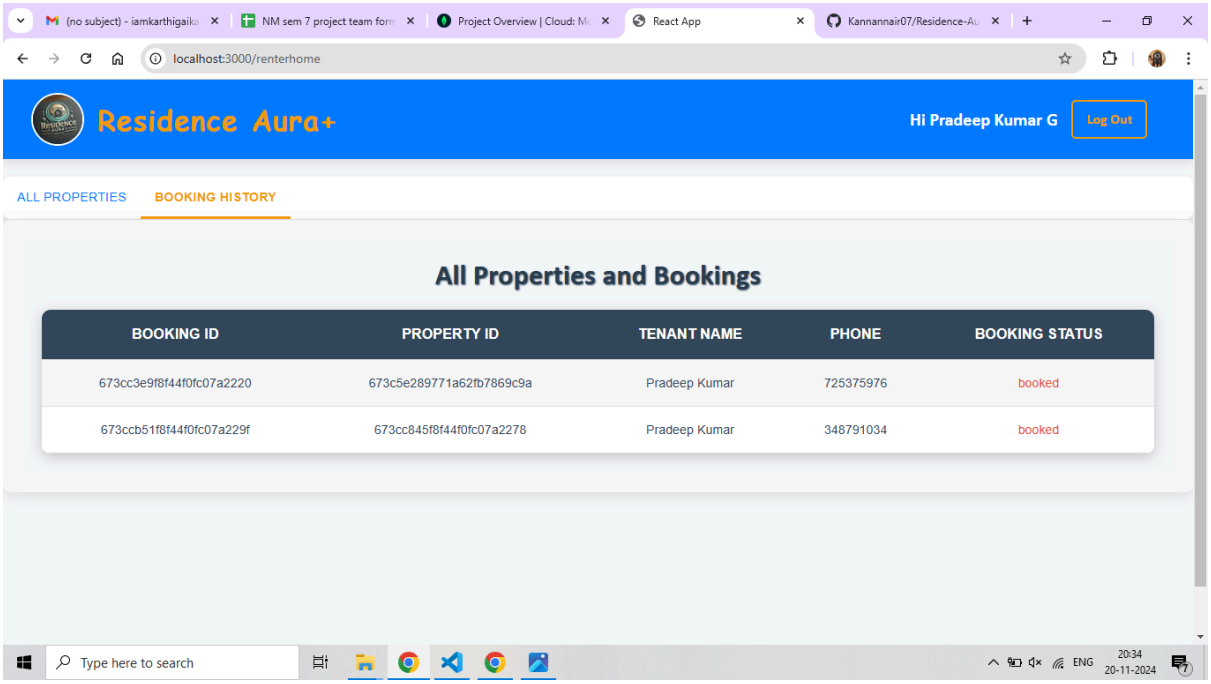

Residence Aura+
Hi Karthigai Kannan K
Log Out
Add Property
All Properties
All Bookings
ALL PROPERTY BOOKINGS
Booking ID
Property ID
Tenant Name
Tenant Phone
Booking Status
Actions
673cc3e9f8f44f0fc07a2220
673c5e289771a62fb7869c9a
Pradeep Kumar
725375976
booked
Change to Pending
673ccc32f8f44f0fc07a22bf
673cc661f8f44f0fc07a2257
Ram
38798270
pending
Change to Booked
673ccc85f8f44f0fc07a22c9
673cc661f8f44f0fc07a2257
Ram
63789802
pending
Change to Booked

11.5 TENANT PANEL:









11.6 DEMO VIDEO LINK: [RESIDENCE AURA+]

CHAPTER XII

FUTURE ENHANCEMENTS

Browser Compatibility:

Different browsers may interpret web elements slightly differently, leading to inconsistencies. Tests may pass on one browser but fail on another, requiring extra handling for cross-browser compatibility.

Dynamic Content Handling:

Selenium struggles with dynamic or JavaScript-heavy websites. Elements may not be immediately available or intractable, leading to errors like ElementNotVisibleException or ElementNotFoundException. Proper waits and handling of AJAX requests are necessary.

Flaky Tests:

Tests might intermittently pass or fail due to timing issues, especially in asynchronous environments. This can be caused by race conditions or slow network responses. Proper synchronization using WebDriverWait can mitigate this.

WebDriver Crashes:

Occasionally, WebDriver instances may crash or fail to initiate, often due to mismatches between browser versions and the corresponding WebDriver (e.g., ChromeDriver and Chrome).

Slow Execution:

Selenium's real-browser interactions make it slower compared to headless testing tools or mocking frameworks. This can be an issue when running large test suites or when tests need to run quickly in CI environments.

Limited Mobile Testing Support:

Although Appium extends Selenium's capabilities to mobile testing, setting up and executing mobile automation tests can be complex and less stable than web automation.

CHAPTER XIII

CONCLUSION

In conclusion, the House rent application offers a comprehensive, scalable, and user-friendly platform for managing house rentals, built using the MERN stack (MongoDB, Express.js, React, and Node.js). The platform caters to tenants, landlords, and admins, providing essential features such as property listings, advanced search and filtering, real-time messaging, and an intuitive admin panel. Tenants can easily search for and inquire about rental properties, while landlords can list, manage, and update their properties with ease. The inclusion of a secure authentication system, responsive design, and user-friendly interface ensures a smooth experience across devices.

The use of MongoDB as a NoSQL database, along with Express and Node.js for the backend, ensures scalability and flexibility, enabling the application to grow with added features like payment integration or machine learning-based recommendations. Additionally, the platform is designed with security in mind, implementing JWT-based authentication and robust data validation to protect user information. The admin panel enhances control, offering powerful tools for content moderation, analytics, and user management, ensuring smooth operations.

GITHUB LINK: **HOUSE RENT APP**