House Rent App using MERN(MongoDB,Express JS, React JS,Node JS**)**

INTRODUCTION:

The house rent app using the MERN stack ensures seamless interaction between the frontend, backend, and database. React.js powers the dynamic and responsive user interface, while Node.js and Express.js handle server-side logic and API communication. MongoDB serves as the database for storing user data, property listings, and transaction details. The app follows a modular architecture, ensuring scalability and ease of maintenance. This design enables real-time updates, secure data handling, and an efficient user experience.

Project Title:

HOUSERENT MERN APP

Team Members:

- Vellaisamy A RegNo-513421104048
- MadhanKumar S RegNo-513421104019
- Kishore A RegNo-513421104015
- Aadhiesh G RegNo-513421104307

2.Project Overview:

PURPOSE:

1. For Renters:

The app aims to empower renters by simplifying the process of searching and securing rental properties. Key purposes include:

Property Browsing and Filtering: Renters can explore a wide range of properties using search filters like location, rent range, number of bedrooms, and amenities, ensuring they find options that match their preferences.

Convenient Communication: The app provides tools to contact property owners or managers directly through messaging or email, enabling clear and efficient discussions.

Booking Management: Renters can submit booking requests, track their status, and receive notifications about approvals or updates, eliminating confusion in the rental process.

2. For Property Owners

The app supports property owners in effectively managing their rental listings. Key purposes include:

Listing Management: Owners can add new properties, edit details, upload images, and specify availability, ensuring their listings are up-to-date and appealing.
Availability Updates: Owners can mark properties as available or occupied, helping renters view accurate availability information.
Communication with Renters: The app facilitates secure and direct communication between owners and potential renters, allowing negotiation of terms and conditions.

3. For Administrators

Administrators are responsible for overseeing the platform and ensuring smooth operations. Key purposes include:

User Approvals: Admins verify and approve property owners and renters to ensure the legitimacy of users.
Platform Governance: Admins monitor activities on the platform, enforce policies, and handle disputes, maintaining trust and security.
Data Oversight: Admins can review platform-wide data, including bookings, user interactions, and property listings, to manage operations effectively.

Goals:

1. Efficiency

The app is designed to streamline the rental process by:

- Reducing the time renters spend searching for properties by providing comprehensive listings with filters.
- Helping property owners manage their listings without manual paperwork.
- Allowing admin users to handle approvals and monitoring through an organized dashboard.

2. Transparency

- Ensuring a clear and trustworthy interaction among renters, owners, and admins.
- Renters can view detailed property descriptions, including rent, amenities, and images.
- Owners and renters can use the app's messaging feature to negotiate terms directly.
- Booking statuses and notifications keep all parties informed about the process.

3. Scalability

The app is built on the MERN stack, ensuring a scalable solution that can:

- Handle an increasing number of users and properties as the platform grows.
- Support future features like payment integration, lease agreement templates, or enhanced analytics.

4. User Experience
Providing a seamless experience through:

- Intuitive navigation for renters, property owners, and administrators.
- Responsive design for both mobile and web users.
- Features like filters, real-time updates, and user-friendly forms to minimize friction.

5. Security:
Ensuring data safety by:

- Using MongoDB for secure data storage and efficient retrieval.
- Implementing secure authentication and authorization mechanisms.
- Protecting sensitive user data, including contact details and booking information.

6. Flexibility:
Supporting varied user needs:

- Renters can find and book properties effortlessly.
- Owners have tools for dynamic property management.
- Admins are equipped with features for monitoring and ensuring platform integrity.

Key Features and Functionalities:

1. User Authentication and Login
Google Login with Firebase:

- Users can securely log in using their Google accounts via Firebase Authentication.
- Provides a fast, secure, and seamless login experience.

2. Property Browsing

Comprehensive Property Listings:

- Users can view detailed property listings with descriptions, images, and key details like location, price, and amenities.
- A user-friendly interface allows for effortless navigation and exploration of properties.

Search and Filters:

- Users can apply filters based on location, price range, property type, and other criteria to refine their search results.

3. Booking Management

Property Booking:

- Renters can select properties and book them by providing check-in and check-out dates.
- The system calculates the total price dynamically based on the booking duration.

Real-Time Booking Status:

- Renters can track the status of their booking requests, such as pending, confirmed, or canceled.

4. Admin Panel

Comprehensive Dashboard:

- Admins have access to a centralized dashboard to oversee all platform activities.

View All Bookings:

- Admins can monitor and manage all property bookings, including renter details, dates, and payment status.

User Management:

- Admins can view, verify, and manage all user accounts, including renters and property owners.

Property Approval:

- Admins review and approve property listings submitted by owners to ensure quality and legitimacy.

5. Owner Functionality

Property Management:

- Property owners can add, edit, and remove property listings.
- They can update the availability status of their properties dynamically.

View Booking Requests:

- Owners can see incoming booking requests and approve or reject them based on their preferences.

6. Secure Messaging

Users and property owners can communicate directly through a secure messaging feature, enabling smooth negotiation and query resolution.

7. Payment Integration (Optional/Planned)

Users can make secure payments for bookings through integrated payment gateways (upcoming feature).

8. Enhanced User Experience

Responsive Design:

A responsive layout ensures compatibility with both web and mobile platforms.

Real-Time Updates:

- Booking statuses, property availability, and user interactions update instantly.
- These features collectively provide a seamless, transparent, and efficient rental process for renters, property owners, and administrators.

3.ARCHITECTURE:

For React:

```
house-rent/
│
├── src/
│   ├── admin/
│   │   ├── AllBookings.jsx
│   │   ├── AllProperties.jsx
│   │   ├── Dashboard.jsx
│   │   ├── SideNavbar.jsx
│   │   ├── Users.jsx
│   │
│   ├── components/
│   │   ├── MobileNavbar.jsx
│   │   ├── Navbar.jsx
│   │   ├── Slider.jsx
│   │
│   ├── pages/
│   │   ├── GAuth.jsx
│   │   ├── Home.jsx
│   │   ├── Login.jsx
│   │   ├── Logout.jsx
│   │   ├── Profile.jsx
│   │   ├── Search.jsx
│   │   ├── Signup.jsx
│   │
│   │
│   │
│   ├── property/
│   │   ├── BookProperty.jsx
│   │   ├── CreateProperty.jsx
│   │   ├── Properties.jsx
│   │   ├── ViewProperty.jsx
│   │   ├── YourBooking.jsx
│   │   ├── YourProperties.jsx
│   │
│   ├── store/
```

```
│   │   ├── store.js
│   │   └── slices/
│   │       ├── userSlice.js
│   │
│   ├── FirebaseConfig.js
│   ├── .env
│
├── .gitignore
├── eslint.config.js
├── package.json
└── README.md
```

BACKEND:

```
Backend/
│
├── server.js           // Main server file where Express is initialized.
├── .env                // Environment variables file (e.g., database credentials).
│
├── routes/             // Folder for organizing route files
│   ├── adminRoute.js   // Routes specific to admin actions (e.g., user management, viewing all bookings)
│   ├── bookRoute.js    // Routes specific to handling bookings
│   ├── propertyRoute.js // Routes specific to property actions (e.g., CRUD operations for properties)
│   ├── userRoute.js    // Routes for user-related actions (e.g., signup, login, profile)
│
├── middleware/         // Folder for middleware functions
│   ├── authentication.js // Middleware to handle authentication and authorization
│
├── models/             // Folder containing database schemas
│   ├── bookingModel.js  // Mongoose schema for bookings
│   ├── propertyModel.js // Mongoose schema for properties
```

```
│   ├── userModel.js      // Mongoose schema for users
│
└── controllers/          // Folder for route controllers (optional but recommended)
    ├── adminController.js
    ├── bookingController.js
    ├── propertyController.js
    ├── userController.js
```

DESCRIPTION:

Server Initialization (server.js):

- The server.js file initializes the Express server and sets up middleware like express.json() for parsing JSON data.
- Environment variables are configured using the .env file, enabling secure storage of sensitive data like database credentials.
- Routes are imported and linked to specific URL prefixes (/api/admin, /api/property, etc.).

Routing (routes folder):

- Routes are organized by functionality for better maintainability:
- adminRoute.js: Handles admin functionalities like viewing users, bookings, and properties.
- bookRoute.js: Manages endpoints for booking-related actions, such as creating or fetching bookings.

- propertyRoute.js: Contains endpoints for CRUD operations on properties (e.g., add, view, update, delete).
- userRoute.js: Handles user actions like signup, login, and profile management.

Middleware (middleware folder):

- Authentication.js: Ensures routes are accessed only by authenticated users or admins. For example:
- JWT tokens are validated to confirm user sessions.
- Role-based access control can restrict certain actions to admins only.

Models (models folder):

- Mongoose models define the database structure:
- bookingModel.js: Represents the booking schema, including references to the user and property models.
- propertyModel.js: Contains details about rental properties, such as title, description, location, price, and owner.
- userModel.js: Stores user information like name, email, hashed passwords, and roles (e.g., admin, user).
- Controllers (controllers folder):
- Handles the core logic for each route:
- AdminController: Contains methods for admin-specific actions, like approving properties or managing bookings.
- BookingController: Implements booking creation, cancellation, and retrieval logic.
- PropertyController: Manages CRUD operations on properties.
- UserController: Handles user registration, authentication, and profile updates.

Database Connection:

The backend connects to a MongoDB database using Mongoose, ensuring robust and scalable data storage.

4.setup Instructions:

1. Prerequisites

Ensure the following software is installed on your machine:

- Node.js (version 14 or higher)

- MongoDB (if running locally) or a MongoDB Atlas account (for cloud-based database)
- Git (to clone the repository)

2. Installation Steps

- Clone the repository
- Open your terminal and navigate to the directory where you want to store the project.
- Clone the repository using Git:

  git clone repourl

- Navigate to the project directory

  cd <project-directory>
- Install Backend Dependencies
- In the project root directory, navigate to the server folder (if your project has separate folders for backend and frontend).

  cd server
- Install the backend dependencies using npm:

  npm install
- Install Frontend Dependencies
- Go back to the project root directory and navigate to the client (or frontend) folder.

  cd ../client
- Install the frontend dependencies:

  npm install

Environment Variables:

- Create an .env file in both the client and server directories.
- Add the following sample environment variables for each folder.
- Modify these values based on your setup.
- Server .env file (inside server directory):

Env File:

PORT=5000

MONGO_URI=mongodb://localhost:27017/hourent
JWT_SECRET=your_jwt_secret_key
PORT: Port number for the backend server.
MONGO_URI: MongoDB URI (local or cloud-based).
JWT_SECRET: Secret key used for JWT authentication.

3. Running the App:

- Run Backend (Server)
- In the server directory, run the backend server:
  npm start
- This will start the backend server on the port specified in the .env file (usually port 5000).
- Run Frontend (Client)
- In the client directory, start the frontend React app:
  npm start
- This will start the React development server, usually at http://localhost:3000.

4. Accessing the App

- Frontend: Visit http://localhost:3000 in your browser to access the web app.
- Backend: The backend will be running on http://localhost:5000.

- Folder Structure
- Client: Describe the structure of the React frontend.

5.Folder Structure
Description:

Root Folder: house-rent
This is the main folder of your project that contains all the files and subdirectories required for the application. The root typically contains configurations, environment variables, and the entry point for the app.

Inside src Folder
This is the source folder where all the main logic and code for the application resides.

1. Admin Folder
This folder contains all components and logic specific to the admin functionalities:

- AllBookings.jsx: Displays all the bookings made by users. Admin can manage and view these bookings.
- AllProperties.jsx: Shows all the properties listed on the platform. Admin can manage these properties.
- Dashboard.jsx: The main dashboard component for admin functionalities, providing an overview of platform activity.
- SideNavbar.jsx: A sidebar navigation component exclusively for admin users, enabling easy navigation between admin-related tasks.
- Users.jsx: Displays a list of all registered users on the platform. Admin can manage user details from here.

2. Components Folder
This folder holds reusable components shared across the app:

- MobileNavbar.jsx: A navigation bar optimized for mobile screens, ensuring a responsive design for smaller devices.
- Navbar.jsx: A navigation bar designed for desktop users, offering seamless navigation for large screens.
- Slider.jsx: A slider component, likely used to display promotional banners or featured properties.

3. Pages Folder
Contains components for pages that are part of the main user flow:

- GAuth.jsx: Handles Google authentication logic, simplifying user login with Google.
- Home.jsx: The root component that acts as the landing page for the application.
- Login.jsx: The page for users to log into their accounts.
- Logout.jsx: Handles user logout functionality.
- Profile.jsx: Displays the user's profile, including their details and preferences.
- Search.jsx: A search page allowing users to query properties based on filters.
- Signup.jsx: The page for user registration.

4. Property Folder

Contains components related to property management:

- BookProperty.jsx: Allows users to book a property by selecting dates and confirming details.
- CreateProperty.jsx: Lets users (landlords) add new properties to the platform
- Properties.jsx: Displays a list of all properties available for rent.
- ViewProperty.jsx: Shows detailed information about a specific property.
- YourBooking.jsx: Displays the booking history of the logged-in user.
- YourProperties.jsx: Allows landlords to view and manage properties they have listed.

5. Store Folder

Holds Redux-related logic for global state management:

- store.js: Configures and initializes the Redux store.
- slices Folder: Contains individual slices of Redux state.
- userSlice.js: Manages state related to user information, such as authentication and profile details.

6. FirebaseConfig.js:

This file holds the Firebase configuration, including API keys and project-specific details, allowing seamless integration with Firebase services like authentication and Firestore.

7. .env File

Stores sensitive credentials such as API keys, database URIs, and Firebase secrets in a secure and environment-specific manner.

Server: Explain the organization of the Node.js backend.

Description:
Server Initialization (server.js):

The server.js file initializes the Express server and sets up middleware like express.json() for parsing JSON data.

Environment variables are configured using the .env file, enabling secure storage of sensitive data like database credentials.

Routes are imported and linked to specific URL prefixes (/api/admin, /api/property, etc.).

Routing (routes folder):
Routes are organized by functionality for better maintainability:

adminRoute.js: Handles admin functionalities like viewing users, bookings, and properties.

bookRoute.js: Manages endpoints for booking-related actions, such as creating or fetching bookings.

propertyRoute.js: Contains endpoints for CRUD operations on properties (e.g., add, view, update, delete).

userRoute.js: Handles user actions like signup, login, and profile management.
Middleware (middleware folder):

Authentication.js: Ensures routes are accessed only by authenticated users or admins. For example:

JWT tokens are validated to confirm user sessions.
Role-based access control can restrict certain actions to admins only.
Models (models folder):

Mongoose models define the database structure:

bookingModel.js: Represents the booking schema, including references to the user and property models.

propertyModel.js: Contains details about rental properties, such as title, description, location, price, and owner.

userModel.js: Stores user information like name, email, hashed passwords, and roles (e.g., admin, user).

Controllers (controllers folder):

Handles the core logic for each route:

AdminController: Contains methods for admin-specific actions, like approving properties or managing bookings.

BookingController: Implements booking creation, cancellation, and retrieval logic.
PropertyController: Manages CRUD operations on properties.

UserController: Handles user registration, authentication, and profile updates.

Database Connection:

The backend connects to a MongoDB database using Mongoose, ensuring robust and scalable data storage.

6.Running the Application:

1. Frontend Server (React)

1. Open another terminal window and navigate to the client directory where your frontend code is located:

cd client [We used houserent(alias) for client]

2. Install frontend dependencies (if you haven't done so already):

npm install

3. Start the frontend server:

npm start

The frontend will start running on the default React development server at http://localhost:5173. [Vite APP]

2. Backend Server (Node.js/Express)

1. Open a terminal and navigate to the server directory where your backend code is located:

cd server [We used Backend(alias) for server]

2. Install backend dependencies (if you haven't done so already):

npm install

3. Start the backend server:

npm start

The backend will start running on the port = http://localhost:3500

7.API Documentation:

Here's a neat and clear API documentation for your routes:

Booking Routes:

POST /createBooking

Description: Create a new booking.

Authorization: Required (JWT token in Authorization header)

Body:

```
{
  "propertyId": "string",
  "userId": "string",
  "bookingDate": "YYYY-MM-DD",
  "otherDetails": "string"
}
```

Response:

```
{
  "success": true,
  "message": "Booking created successfully",
  "booking": { /* booking details */ }
}
```

GET /fetchBookings/:id

Description: Fetch all bookings for a user by ID.

Authorization: Required (JWT token in Authorization header)

Response:

```
{ "success": true,
  "message": "Bookings fetched successfully",
  "bookings": [ /* array of booking details */ ]
}
```

Property Routes:

POST /createProperty

Description: Create a new property.

Authorization: Required (JWT token in Authorization header)

Body:

```
{
  "ownerId": "string",
  "title": "string",
  "address": "string",
  "price": "number",
  "details": "string",
  "photos": ["url1", "url2"]
```

}

GET /fetchAllProperty

Description: Fetch all properties.

Response:

{

"success": true,

"properties": [/* array of property details */]

}

GET /findPropertyById/:id

Description: Fetch a property by ID.

Response:

{

"success": true,

"property": { /* property details */ }

}

DELETE /deletePropertyById/:id

Description: Delete a property by ID (Admin only).

Authorization: Required (Admin JWT token in Authorization header)

User Routes

POST /login

Description: User login.

Body:{

"email": "string",

"password": "string"

}

POST /signup

Description: User registration.{

"email": "string",

"password": "string"

}

POST /googleAuth

Description: Google authentication.

Body:{

"email": "string"

}

PUT /updateProfile

Description: Update user profile.

Authorization: Required (JWT token in Authorization header)

Body:

```
{
  "userName": "string",
  "address": "string",
  "pinCode": "string",
  "mobileNo": "string",
  "country": "string",
  "state": "string"
}
```

POST /fetchUser

Description: Fetch user details.

Authorization: Required (JWT token in Authorization header)

Response:

```
{
  "success": true,
  "user": { /* user details */ }
}
```

POST /fetchUserProperty

Description: Fetch user's property listings.

Authorization: Required (JWT token in Authorization header)

Response:

```
{
  "success": true,
  "properties": [ /* user's properties */ ]
}
```

List of API Used:

- /api/admin/getAllUsers
- /api/admin/getUser
- /api/admin/deleteUser
- /api/admin/deleteAllUser
- /api/admin/deleteAllProperty
- /api/admin/fetchBookings
- /api/user/googleAuth
- /api/user/signup
- /api/user/updateProfile
- /api/user/login
- /api/property/createProperty
- /api/property/fetchAllProperty
- /api/property/findPropertyById/:id
- api/property/deletePropertyById/:id
- /api/property/updatePropertyById/:id
- /api/user/fetchUser
- /api/user/fetchUserProperty
- /api/property/searchProperty?searchTerm=india

- /api/booking/createBooking
- /api/booking/fetchBookings/:id

8. Authentication:

Authentication is the process of verifying the identity of a user, typically by using credentials such as a username and password.

How Authentication is Handled:

JWT (JSON Web Tokens) are used for authenticating users in the project.

When a user logs in, a JWT token is generated and sent back to the client. This token contains the user's details, such as their userId and roles (such as whether they are an admin).

The JWT is signed using a secret key to ensure it is tamper-proof.

The client (usually a frontend app like React or a mobile app) stores this token in local storage or cookies and sends it with each request to protected routes using the Authorization header.

Steps of Authentication:

Login:

The user provides credentials (email and password) via the /login route.

The server checks the credentials against the database. If they match, a JWT token is generated and returned to the client.

Token Generation:

After successful login, a JWT token is created on the server using the user's information.

The token is signed using a secret key (e.g., process.env.JWT_SECRET), which ensures that it cannot be tampered with.

The generated token is sent back to the client.

Client-Side:

The client stores the JWT token (usually in local storage or session storage).

This token is then sent in the Authorization header as a Bearer token with every subsequent request to the server that requires authentication.

Token Validation:

Each time a request is made to a protected route, the token is extracted from the Authorization header, and the server verifies it.

The server decodes the JWT using the secret key to extract the user's details (such as userId and isAdmin).

Token Expiry:

JWT tokens have an expiration time, after which the token becomes invalid. The server checks if the token has expired and requests a new one if necessary.

Code Example:

```
const jwt = require("jsonwebtoken");

const authentication = (req, res, next) => {

  if (req.headers.authorization && req.headers.authorization.startsWith("Bearer"))
{

    try {

      const token = req.headers.authorization.split(" ")[1]; // Extract token from
header
```

```
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = decoded;
      next();
    } catch (error) {
      return res.status(401).json({
        success: false,
        message: "Token not authorized",
      });
    }
  } else {
    return res.status(401).json({
      success: false,
      message: "Authentication token is missing",
    });
  }
};
```

2. Authorization:

Authorization is the process of determining whether an authenticated user has the right permissions to access a particular resource or perform a specific action.

How Authorization is Handled:

Role-Based Access Control (RBAC) is implemented for authorization.

In the project, an additional check is added to verify if the authenticated user is an admin before granting access to specific routes (e.g., deleting properties, managing users).

The user object, which contains roles (like isAdmin), is decoded from the JWT token and checked for authorization.

Steps of Authorization:

Admin Authorization:

The adminAuthorization middleware checks whether the user has admin privileges. If the user is not an admin, they are denied access.

The check is performed on the decoded JWT (req.user.isAdmin).

Code Example:

```
const adminAuthorization = (req, res, next) => {
  if (req.user && req.user.isAdmin) {
    return next(); // User is an admin, proceed to next middleware or route handler
  } else {
    return res.status(403).json({
      success: false,
      message: "Access denied: Admins only",
    });
  }
};
```

3. Token Storage and Management:

Client-Side Storage:

The JWT token is stored client-side (usually in localStorage or sessionStorage).

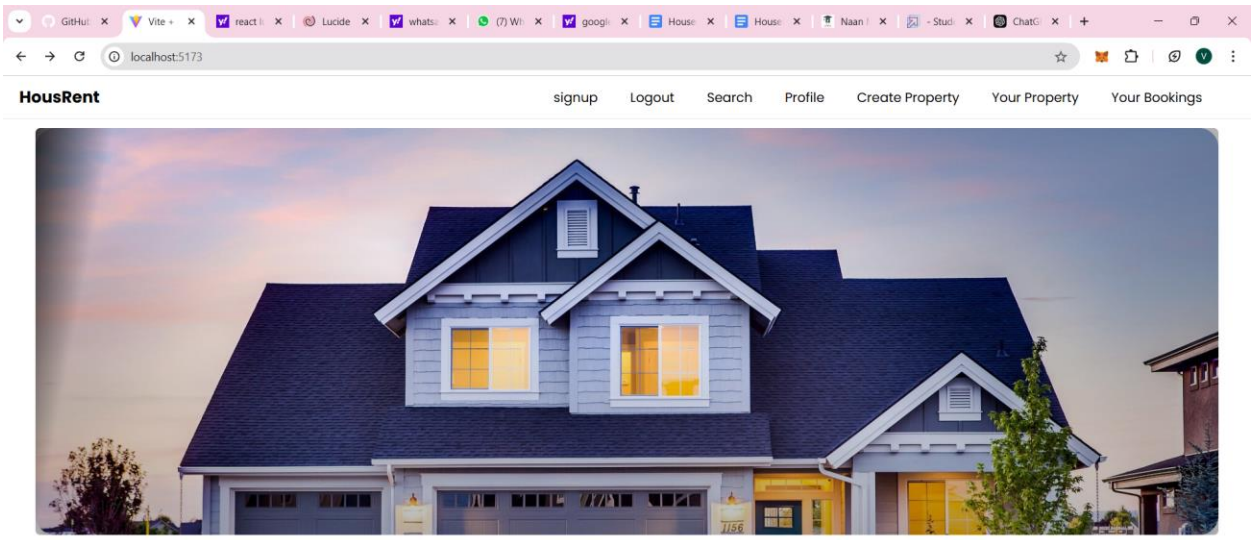
The token is sent with each request that requires authentication, typically using the Authorization header as Bearer <token>.

Server-Side:

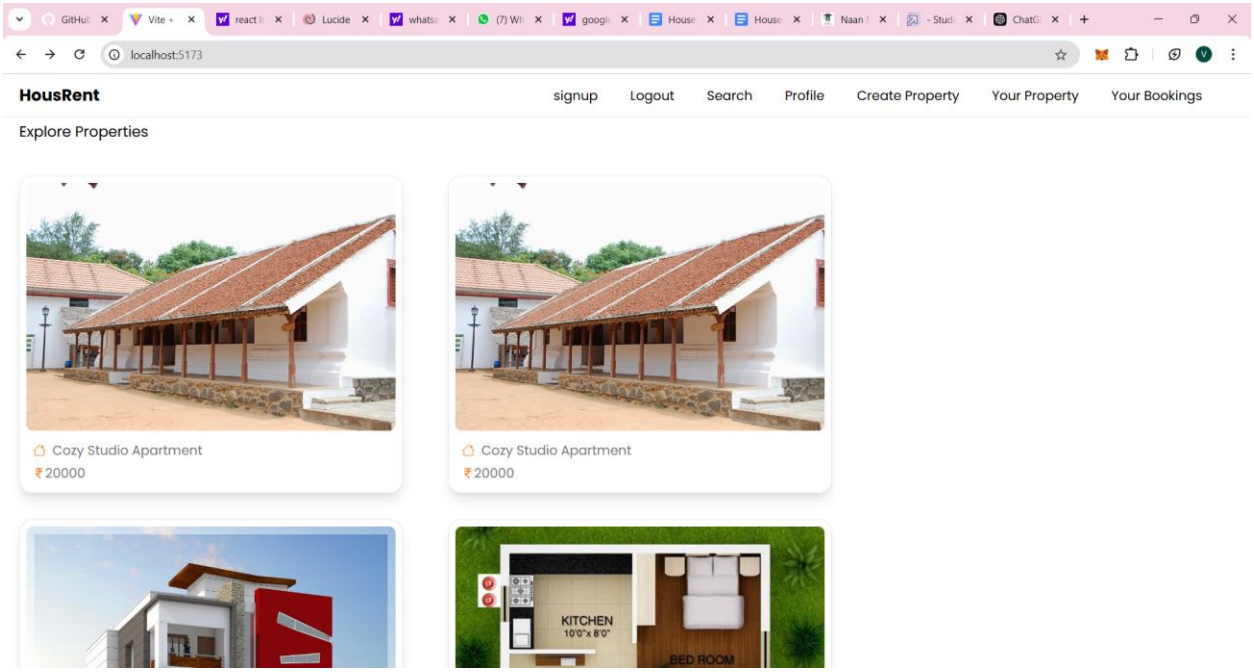
The server does not store any session data. All session data (like the user identity) is embedded within the JWT itself. This stateless approach improves scalability since there is no need to store sessions on the server.

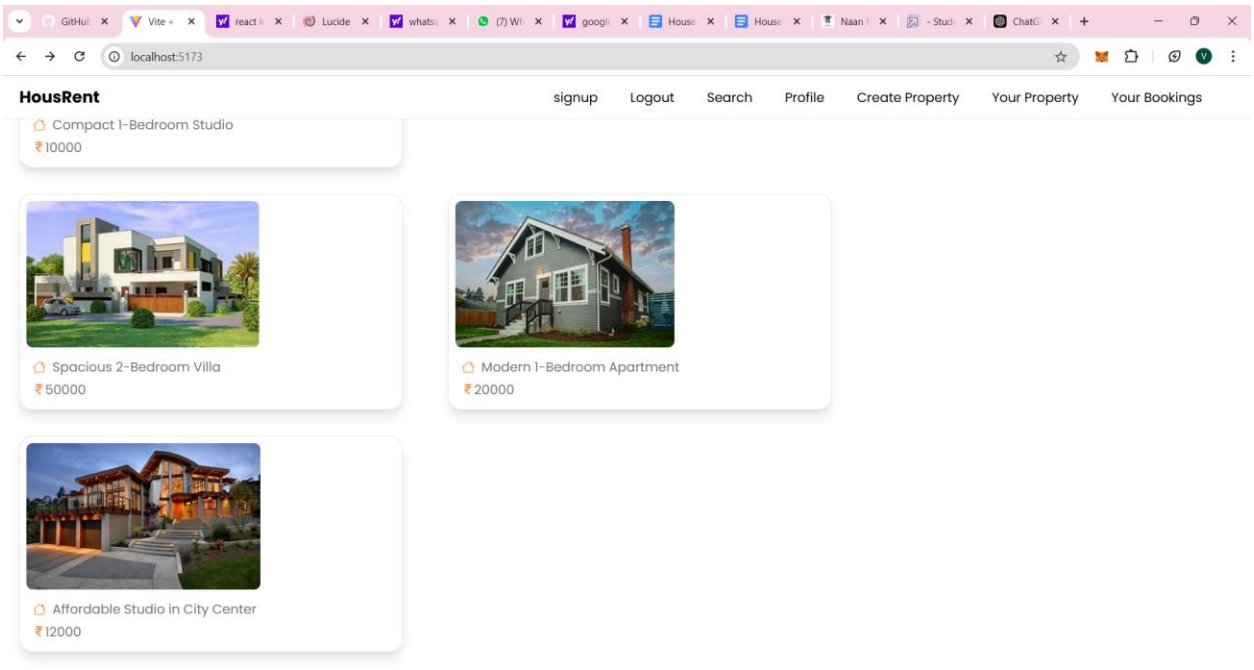
The server only validates the JWT and decodes the user information when a request comes in.

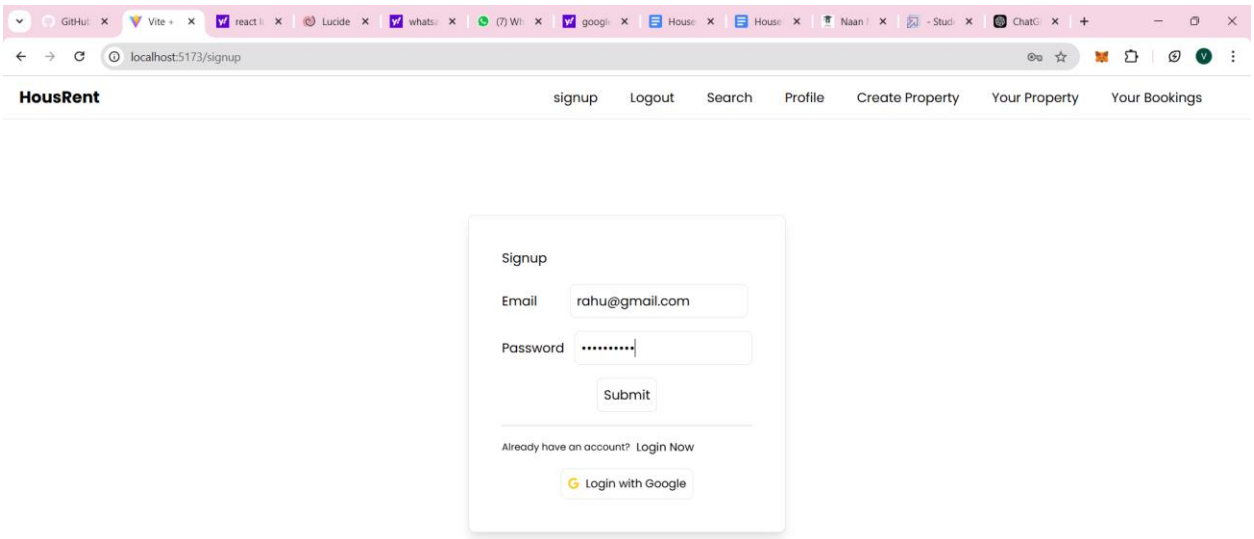
9.UserInterface:

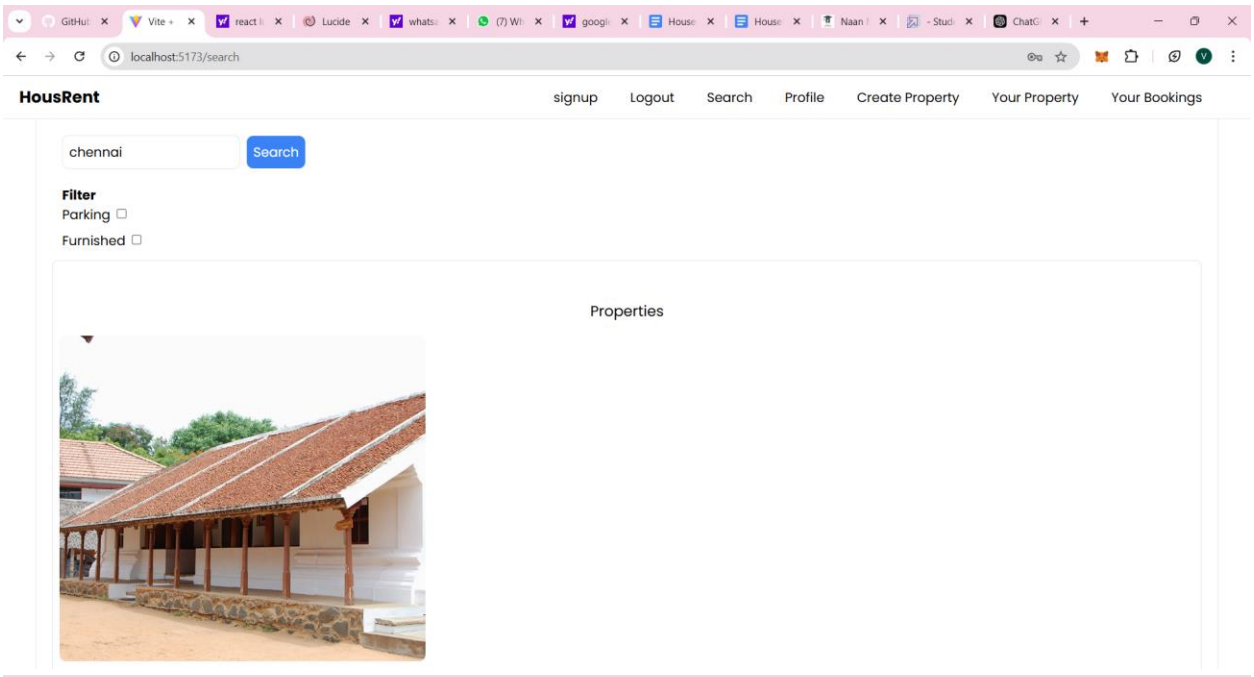


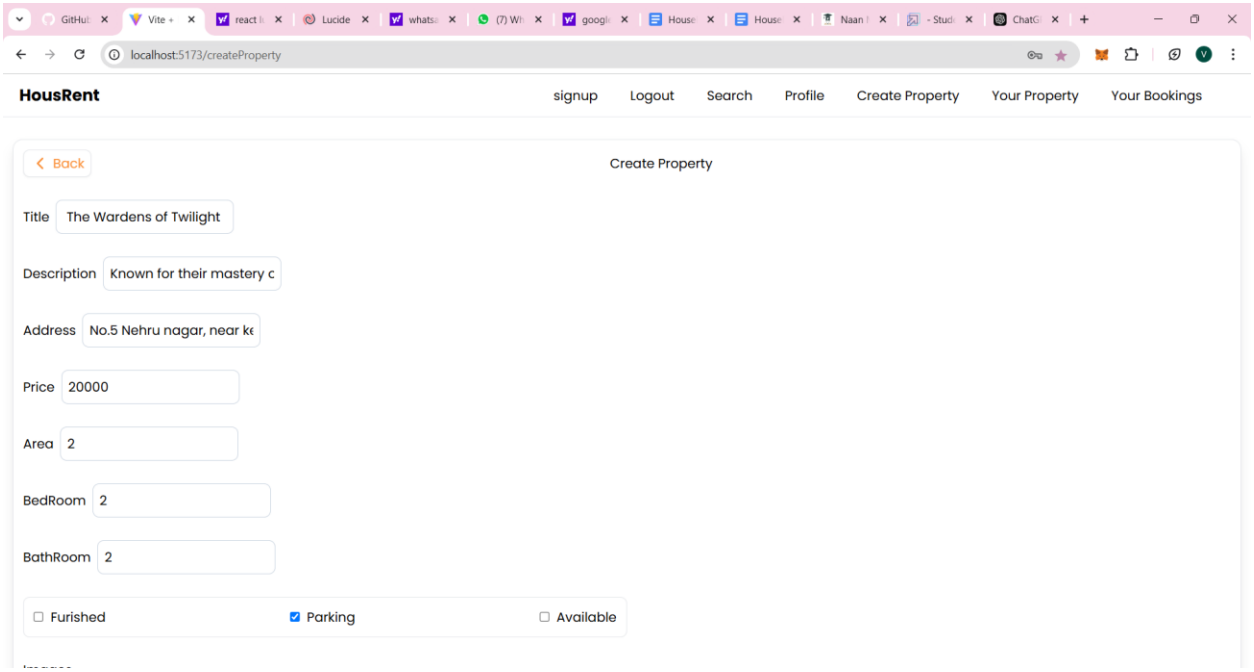
Explore Properties




localhost:5173
HousRent
signup
Logout
Search
Profile
Create Property
Your Property
Your Bookings
Compact 1-Bedroom Studio
₹10000
Spacious 2-Bedroom Villa
₹50000
Modern 1-Bedroom Apartment
₹20000
Affordable Studio in City Center
₹12000


localhost:5173/signup
HousRent
signup
Logout
Search
Profile
Create Property
Your Property
Your Bookings
Signup
Email
rahu@gmail.com
Password
Submit
Already have an account? Login Now
Login with Google


localhost:5173/search
HousRent
signup
Logout
Search
Profile
Create Property
Your Property
Your Bookings
chennai
Search
Filter
Parking
Furnished
Properties


localhost:5173/createProperty
HousRent
signup
Logout
Search
Profile
Create Property
Your Property
Your Bookings
Back
Create Property
Title
The Wardens of Twilight
Description
Known for their mastery c
Address
No.5 Nehru nagar, near ke
Price
20000
Area
2
BedRoom
2
BathRoom
2
Furished
Parking
Available

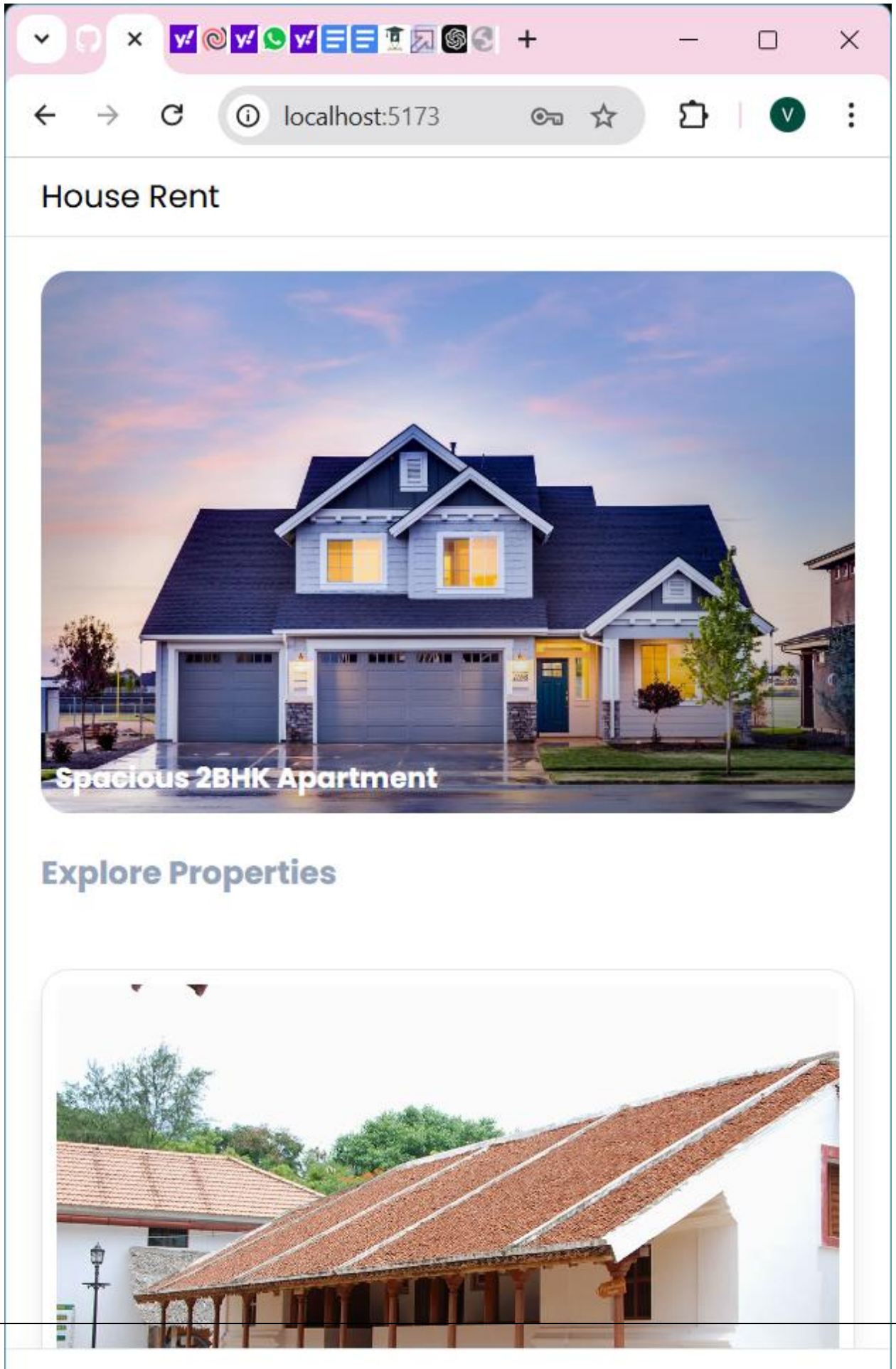
10.Testing Strategy and Tools Used:

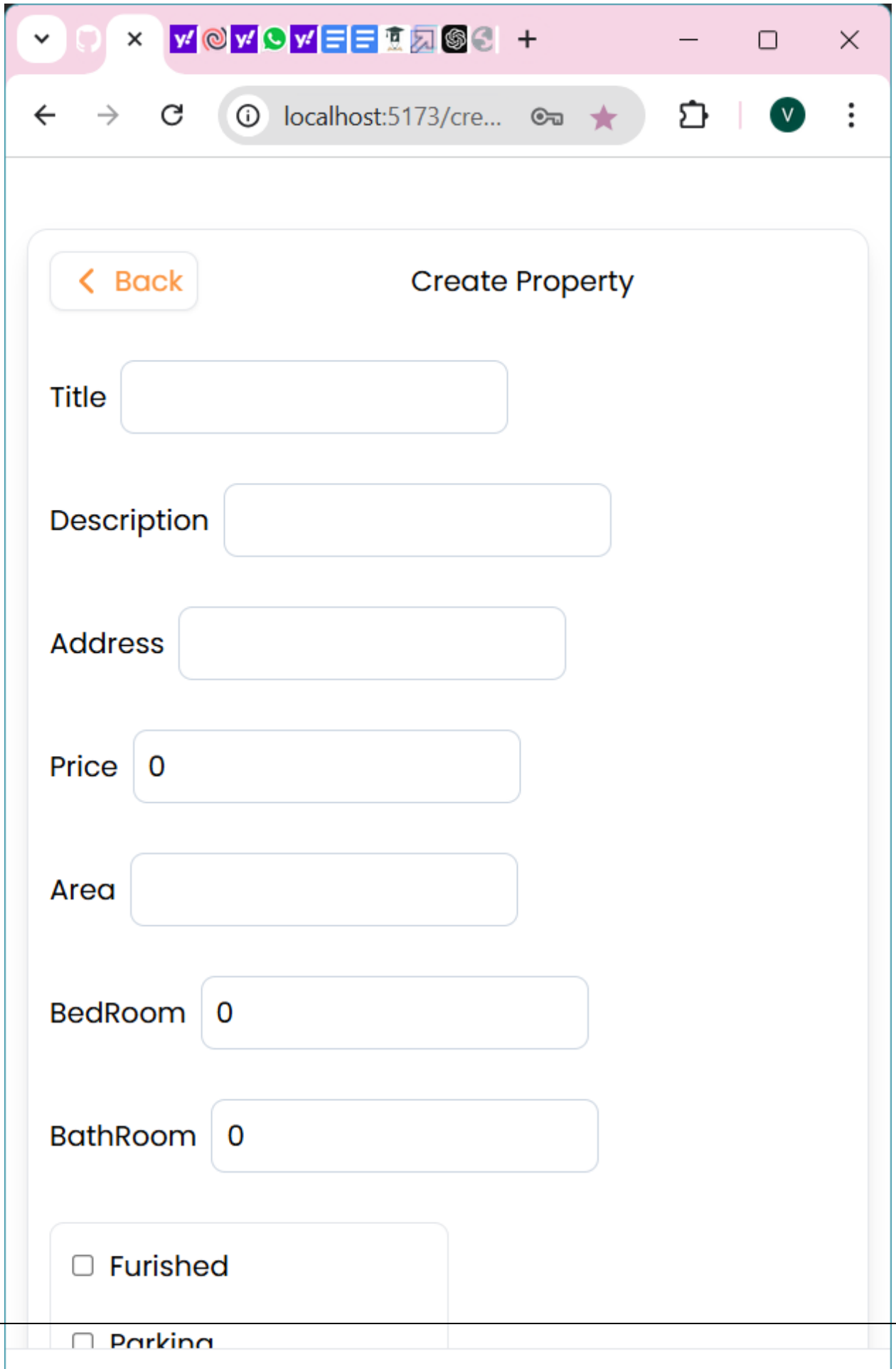
For testing the APIs, I used Postman. It helped me ensure the APIs work correctly by sending different types of requests (GET, POST, PUT, DELETE) and checking the responses. Here's how I used it:

- Testing Endpoints: I tested each API endpoint to make sure they returned the correct status codes and data.
- Checking Data: For requests that send data, I verified that the data was correctly saved or updated in the database.
- Authentication: I tested login and ensured the system properly handled user authorization using JWT tokens.
- Error Handling: I tested APIs with invalid data to ensure they returned the right error messages.

Postman helped automate testing and made sure everything worked as expected.

11.SCREENSHOTS:


localhost:5173
House Rent
Spacious 2BHK Apartment
Explore Properties


localhost:5173/cre...
Back
Create Property
Title
Description
Address
Price
0
Area
BedRoom
0
BathRoom
0
Furished
Parking

localhost:5173/you...

Back Your Properties

Cozy Studio Apartment

₹20000

Demo:

https://drive.google.com/file/d/1twX202zMFSS9IOQybsHBDQiT-eC4c7RM/view?usp=sharing

12.Known Issues:

UI Responsiveness on Mobile: Some elements may not scale properly on smaller screens. This might cause the user interface to appear misaligned on certain mobile devices.

Search Filter Performance: The search filter sometimes takes longer to load when there are many listings. This can be improved for better performance.

13.Future Enhancements:

Advanced Search Filters: Add more detailed filters such as price range, property size, or neighborhood to refine the search results further.

User Ratings & Reviews: Implement a feature where users can rate and review properties they've rented, providing more insights to other users.

Admin Dashboard Improvements: Enhance the admin dashboard with features like property management, user activity tracking, and advanced reporting tools.

Payment Integration: Integrate a secure payment system to allow users to pay for rent or booking through the platform.