Full Stack Development with MERN

1. Introduction

- **Project Title:** Rent Ease [A house hunt web application]
- **Team Members:**

  1. A Mani Ratna
  2. Korru Kiranmayee
  3. A Bhavana
  4. E Gayathri

  Everyone is involved in all the phases of development.

2. Project Overview

- **Purpose:**

  Rent Ease aims to revolutionize the rental and home-buying market by providing a comprehensive, user-friendly platform that connects renters, buyers, and house owners. The primary goal is to simplify and streamline the process of finding and securing rental houses or houses for purchase. By leveraging modern web technologies, Rent Ease offers a seamless and efficient experience for both users looking for their next home and house owners seeking to fill vacancies or sell their houses.

- **Features:**

  1. **User Authentication:** Secure login and registration for users, ensuring that only authorized users can access certain features.
  2. **House Listings:** A comprehensive and up-to-date list of available rental and for-sale houses, with detailed descriptions and photos.
  3. **Search and Filter:** Advanced search and filtering options allow users to narrow down their choices based on specific criteria such as price and if they want to buy or rent a house.
  4. **Favourites:** Users can save their favourite houses for quick access and comparison.
  5. **House Details:** Detailed views of each house, including images, descriptions, and owner contact information.
  6. **Chat:** Direct messaging functionality enables users to communicate with house owners in real-time.
  7. **Responsive Design:** The application is optimized for both desktop and mobile devices, providing a seamless user experience across all platforms.
  8. **Listings:** A comprehensive and up-to-date list of available rental and for-sale houses,

3. Architecture

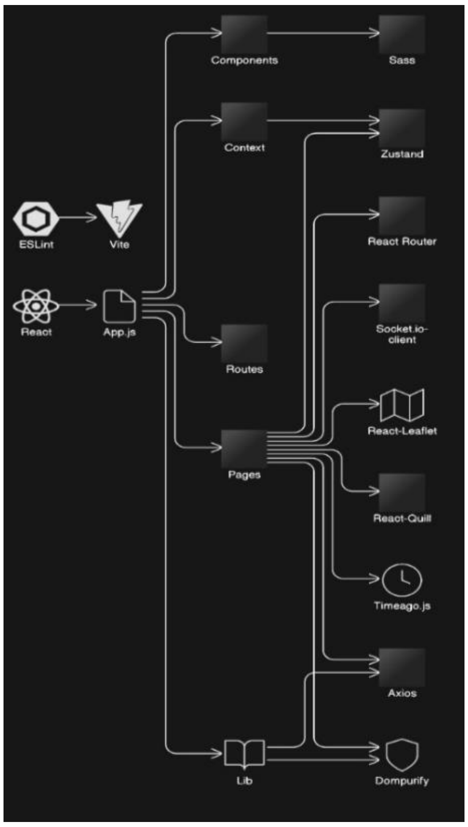
Frontend: Our frontend architecture for the House Hunt project is designed to be modular, scalable, and maintainable. It uses React, Zustand, and React Router to build a responsive and dynamic user interface.

Overview:

1. React: For building user interfaces.
2. Zustand: Lightweight state management.
3. React Router: Handles routing.
4. Sass: For styling.
5. Axios: For HTTP requests.
6. Socket.io-client:Real-time communication.
7. React-Leaflet: Interactive maps.
8. React-Quill: Rich text editing.
9. Dompurify: HTML input sanitization.
10. Timeago.js: Relative timestamps.

Key Components :

1. App.js: Main application layout and routing.
2. components: Reusable UI components.
3. pages: Major views like Home, Profile, and List.
4. routes: Routing configurations.
5. context: Global state management (authentication, notifications).
6. lib: Utility functions, loaders, and API request



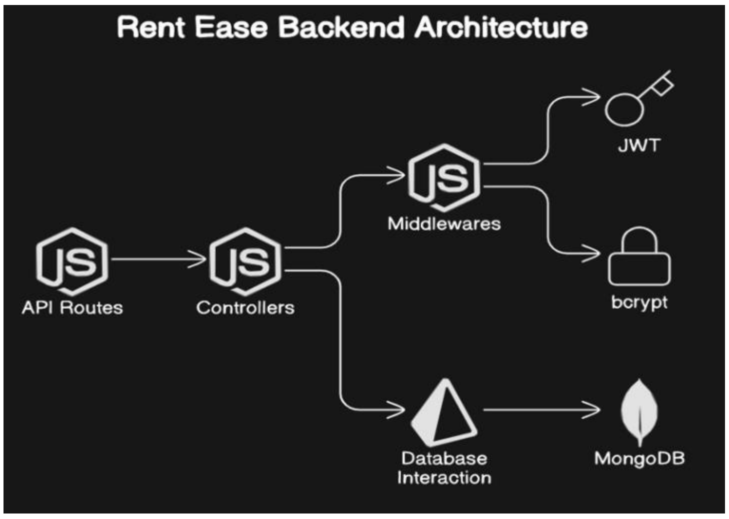
Backend: Our backend architecture for the Rent Ease project is designed to be modular, scalable, and maintainable. It uses Node.js, Express.js, and Prisma to interact with a MongoDB database.

Overview :

1. Node.js: JavaScript runtime for server-side operations.
2. Express.js: Web framework for creating the RESTful API.
3. MongoDB: NoSQL database for storing application data.
4. Prisma: ORM for interacting with MongoDB, ensuring type-safe database operations.
5. JWT: For secure token-based authentication.
   bcrypt: For password hashing.

Key Components :

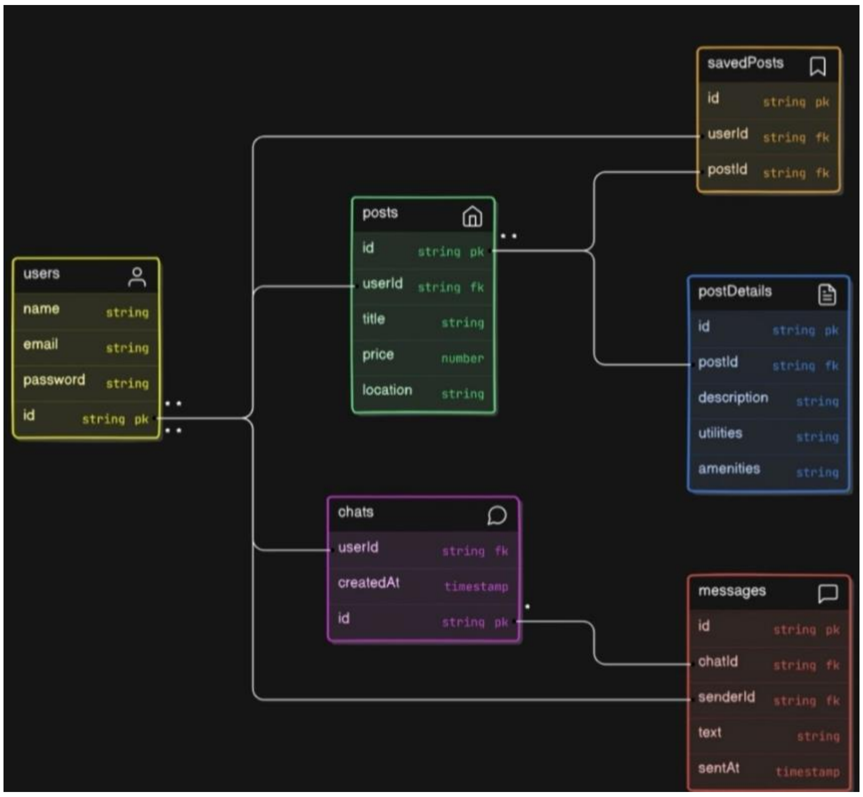
1. API Routes: Define endpoints and link to controllers.
2. Controllers: Handle request processing and response generation.
3. Middlewares: Custom middleware functions for authentication, error handling, and validation.
4. Database Interaction: Prisma for type-safe and efficient database operations.



Database: The Rent Ease project's database architecture utilizes MongoDB and Prisma ORM for efficient and type-safe data management. The architecture includes the following key collections:

1. Users Collection: Stores user information and manages authentication and relationships.
2. Posts Collection: Contains details about user-listed properties, including property specifics and user associations.
3. PostDetails Collection: Holds additional property information such as descriptions, utilities, and proximity to amenities.

4. SavedPosts Collection: Tracks posts saved by users, linking users to their saved listings.
5. Chats Collection: Manages user communications, storing chat participants, messages, and metadata.
6. Messages Collection: Stores individual messages within chats, including text, sender, and associated chat.



4. Setup Instructions

- **Prerequisites:**

1. Node.js
2. MongoDB
3. Git
4. Cloudinary Account to store posts, user avatars

- **Installation:**

1. Clone the repository:

```
git clone https://github.com/bhavanaareddy/House-hunt-using-MERN.git
```

2. Install dependencies for client , server and socket

```
cd client
npm install
```

```
cd api
npm install
```

```
cd socket
npm install
```

3. Set up environment variables:

Create a .env file in the server directory (api) with following environment variables

```
DATABASE_URL= "your database url"

JWT_SECRET_KEY= "your jwt_secret_key"

CLIENT_URL=http://localhost:5173
```

5. Folder Structure

- **Client:**

```
client/
│
├── node_modules/
│   └── …
│
├── public/
│   └── … (Assets)
│
├── src/
│   ├── components/
│   │   └── card
│   │   │   └── card.jsx
│   │   │   └── card.scss
│   │   └── chat
│   │   │   └── chat.jsx
│   │   │   └── chat.scss
│   │   └── filter
│   │   │   └── filter.jsx
│   │   │    └──filter.scss
│   │   └── list
│   │   │    └──list.jsx
│   │   │    └──list.scss
│   │   └── map
│   │   │    └── map.jsx
│   │   │    └──map.scss
│   │   └── navbar
│   │   │    └── navbar.jsx
│   │   │    └── navbar.scss
│   │   └── pin
│   │   │    └── pin.jsx
│   │   │    └── pin.scss
│   │   └── searchBar
│   │   │    └── searchBar.jsx
│   │   │    └── searchBar.scss
│   │   └── slider
│   │   │    └── slider.jsx
│   │   │    └── slider.scss
│   │   └── uploadWidget
│   │   │    └── uploadWidget.jsx
│   │   │    └── uploadWidget.scss
│   │
│   ├── context/
│   │   └── AuthContext.jsx
│   │   └── SocketContext.jsx
│   │
│   ├── context/
│   │   └── apiRequest.js
│   │   └── dummydata.js
│   │   └── loaders.js
│   │   └── notificationStore.js
│   │
│   ├── routes/
│   │   └── homePage
│   │   │    └── homePage.jsx
│   │   │    └── homePage.scss
│   │   └── layout
│   │   │    └── layout.jsx
│   │   │    └── layout.scss
│   │   └── listPage
│   │   │    └── listPage.jsx
│   │   │    └── listPage.scss
│   │   └── login
│   │   │    └──login.jsx
│   │   │    └──login.scss
│   │   └── newPostPage
│   │   │    └── newPostPage.jsx
│   │   │    └── newPostPage.scss
│   │   └── profilePage
│   │   │   └── profilePage.jsx
│   │   │   └── profilePage.scss
│   │   └── profileUpdatePage
│   │   │   └── profileUpdatePage.jsx
│   │   │   └── profileUpdatePage.scss
│   │   └── register
│   │   │   └── register.jsx
│   │   │   └── register.scss
│   │   └── singlePage
│   │   │   └── singlePage.jsx
│   │   │   └── singlePage.scss
│
│   │── App.jsx
│   │── index.css
│   │── index.scss
│   │── main.jsx
│   │──  responsive.scss
│
│── index.html
└── package.json
```

- **Server:**

```
api/
│
├── controllers/
│   ├── auth.controller.js
│   ├── chat.controller.js
│   ├── message.controller.js
│   ├── post.controller.js
│   ├── test.controller.js
│   └── user.controller.js
│
├── lib/
│   └── prisma.js
│
├── middleware/
│   └── verifyToken.js
│
├── node_modules/
│   └── ...
│
├── prisma/
│   └── schema.prisma
│
├── routes/
│   ├── auth.route.js
│   ├── chat.route.js
│   ├── message.route.js
│   ├── post.route.js
│   ├── test.route.js
│   └── user.route.js
│
├── .env
├── .gitignore
├── app.js
├── package-lock.json
└── package.json
```

- **Socket:**

```
socket/
│
├── node_modules/
│   └── …
│
├── public/
│   └── … (Assets)
├── app.js
└── package.json
```

6. Running the Application

- Commands to start the frontend and backend servers locally.
    - **Frontend:** `npm run dev` in the client directory.
    - **Backend:** `nodemon –env-file .env app.js` in the api directory.
    - **Socket:** `nodemon app.js` in the socket directory.

7. API Documentation

1. User Authentication

- **POST /api/auth/register**
    - Description: Registers a new user.
    - Parameters:
        - username (string, required)
        - email (string, required)
        - password (string, required)
    - Example Response:

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

- **POST /api/auth/login**
    - Description: Authenticates a user and returns a token.
    - Parameters:
        - email (string, required)
        - password (string, required)
    - Example Response:

```
{
  "token": "JWT_TOKEN_HERE"
}
```

- **POST /api/auth/logout**
    - Description: Logs out the user.
    - Example Response:

```
{
  "message": "User logged out successfully"
}
```

2. User Management

- **GET /api/users**
    - Description: Retrieves all users.
    - Example Response:

```
[
  {
    "id": "USER_ID",
    "username": "USERNAME",
    "email": "EMAIL"
  }
]
```

- **PUT /api/users/**
    - Description: Updates user information by ID.
    - Parameters:
        - username (string, optional)
        - email (string, optional)
    - Example Response:

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

- **DELETE /api/users/**
    - Description: Deletes a user by ID.
    - Example Response:

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

- **POST /api/users/save**
  - Description: Saves a post for the user.
  - Parameters:
    - postId (string, required)
  - Example Response:

```
{
  "message": "Post saved successfully"
}
```

- **GET /api/users/profilePosts**
  - Description: Retrieves posts saved by the user.
  - Example Response:

```
[
  {
    "id": "POST_ID",
    "title": "Post Title",
    "description": "Post Description"
  }
]
```

- **GET /api/users/notification**
  - Description: Retrieves the number of notifications for the user.
  - Example Response:

```
{
  "notifications": 3
}
```

3. Post Management

- **GET /api/posts**
  - Description: Retrieves all posts.
  - Example Response:

```
[
  {
    "id": "POST_ID",
    "title": "Post Title",
    "description": "Post Description"
  }
]
```

- **GET /api/posts/**
  - Description: Retrieves a post by ID.
  - Example Response:

```
{
  "id": "POST_ID",
  "title": "Post Title",
  "description": "Post Description"
}
```

- **POST /api/posts**
  - Description: Creates a new post.
  - Parameters:
    - title (string, required)
    - description (string, required)
  - Example Response:

```
{
  "message": "Post created successfully"
}
```

- **PUT /api/posts/**
  - Description: Updates a post by ID.
  - Parameters:
    - title (string, optional)
    - description (string, optional)
  - Example Response:

```
{
  "message": "Post updated successfully"
}
```

- **DELETE /api/posts/**
  - Description: Deletes a post by ID.
  - Example Response:

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

4.Chat Management

- **GET /api/chats**
  - Description: Retrieves all chats for the logged-in user.
  - Example Response:

```
[
  {
    "id": "CHAT_ID",
    "participants": ["USER_ID1", "USER_ID2"],
    "messages": ["MESSAGE_ID1", "MESSAGE_ID2"]
  }
]
```

- **GET /api/chats/**
    - Description: Retrieves a chat by ID.
    - Example Response:

```
{
  "id": "CHAT_ID",
  "participants": ["USER_ID1", "USER_ID2"],
  "messages": ["MESSAGE_ID1", "MESSAGE_ID2"]
}
```

- **POST /api/chats**
    - Description: Creates a new chat.
    - Parameters:
        - participants (array of user IDs, required)
    - Example Response:

```
{
  "message": "Chat created successfully"
}
```

- **PUT /api/chats/read/**
    - Description: Marks a chat as read by ID.
    - Example Response:

```
{
  "message": "Chat marked as read"
}
```

5.Message Management

- **POST /api/messages/**
    - Description: Adds a new message to a chat.
    - Parameters:
        - content (string, required)
    - Example Response:

```
{
  "message": "Message sent successfully"
}
```

8. Authentication

In our Rent Ease project, authentication and authorization are handled through a secure token-based system using JSON Web Tokens (JWT). Below are the key aspects of how authentication and authorization are managed:

1.Authentication Flow

1. **User Registration:**

    - Endpoint: POST /api/auth/register
    - Users can register by providing their username, email, and password.
    - The password is hashed before being stored in the database for security.
2. **User Login:**
    - Endpoint: POST /api/auth/login
    - Users authenticate by providing their email and password.
    - If the credentials are correct, a JWT token is generated and sent back to the client.
3. **User Logout:**
    - Endpoint: POST /api/auth/logout
    - Users can log out, which invalidates the token on the client side.

2.Token Handling

- **JWT Generation:**
    - Upon successful login, a JWT is generated using a secret key.
    - The token contains the user's ID and expiration information.
- **Token Storage:**
    - The token is stored on the client side, typically in localStorage or cookies.
    - For API requests requiring authentication, the token is included in the request headers.
- **Token Verification:**
    - Each protected route on the backend uses middleware to verify the JWT.
    - Middleware: verifyToken checks the validity of the token. If the token is valid, the request is allowed to proceed; otherwise, it is rejected with an error.

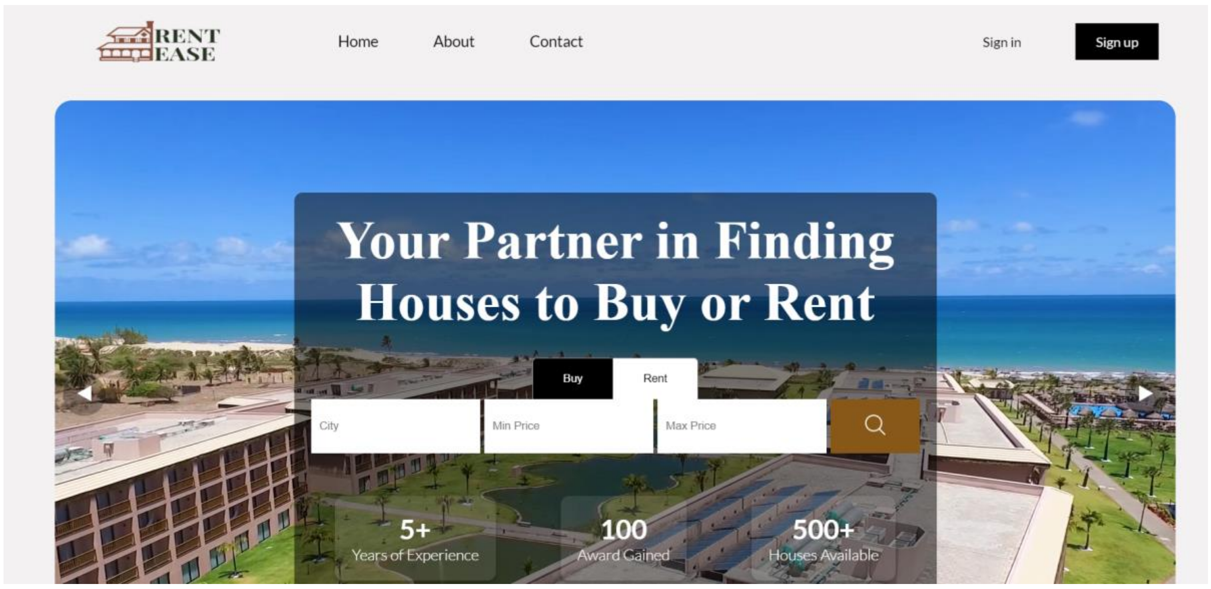
3.Authorization

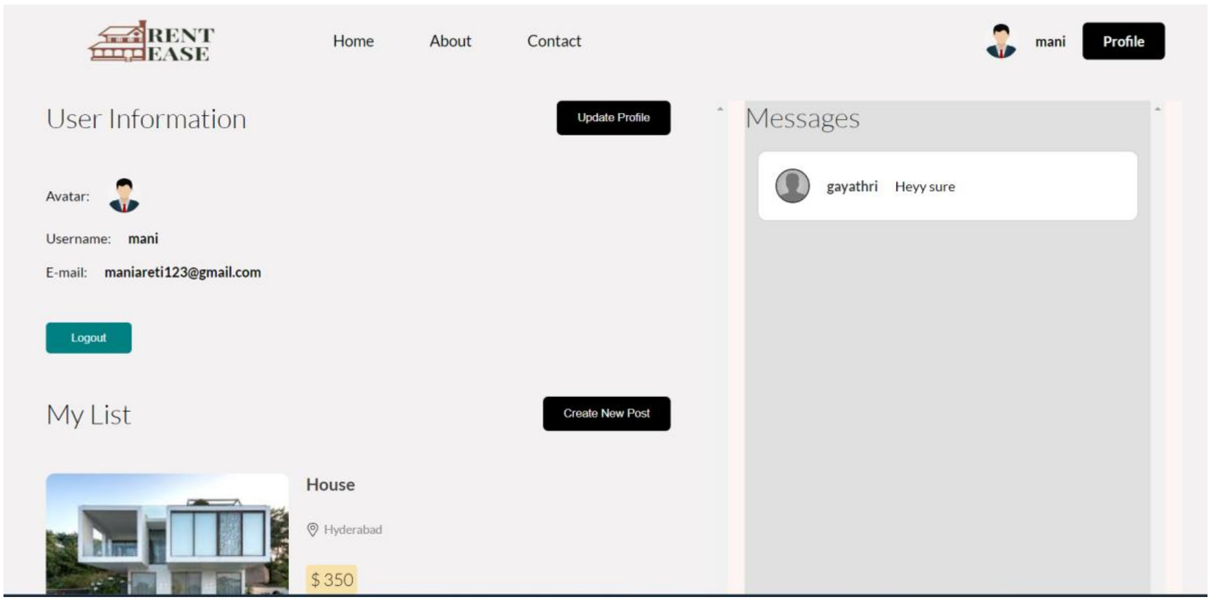
- **Role-Based Access Control (RBAC):**
    - Certain actions are restricted based on user roles (e.g., admin, regular user).
    - Middleware can be implemented to check user roles before allowing access to specific routes.
- **Protected Routes:**
    - Routes that require authentication use the verifyToken middleware to ensure only authenticated users can access them.
    - Example of protected routes:
        - Creating, updating, or deleting posts.
        - Accessing chat functionalities.

4.Security Measures

- **Password Hashing:**
    - User passwords are hashed using bcrypt before being stored in the database, ensuring that plain text passwords are never stored.
- **Token Expiration:**
    - JWT tokens have an expiration time set to limit their validity, reducing the risk of token misuse.
- **Secure Token Storage:**
    - Storing tokens in secure, http-only cookies to protect against cross-site scripting (XSS) attacks.
- **HTTPS:**
    - Ensure that all communications between the client and server are encrypted using HTTPS to protect sensitive data in transit.

9. User Interface:





User Information

Avatar:

Username: mani

E-mail: maniareti123@gmail.com

My List

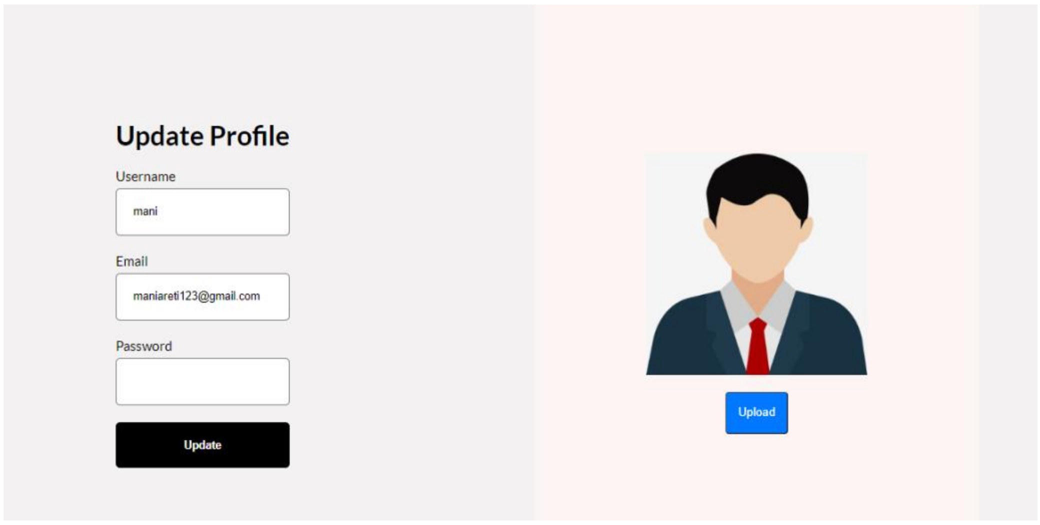
House

Hyderabad

$ 350

Messages

gayathri Heyy sure



Update Profile

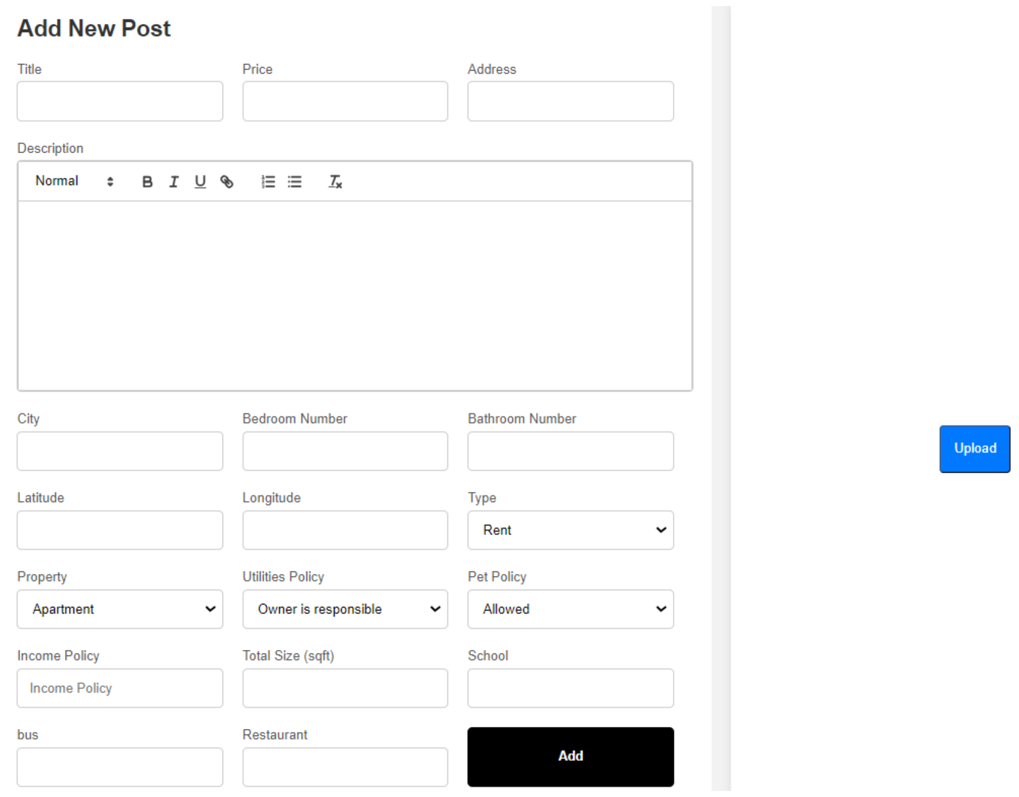
Username

mani

Email

maniareti123@gmail.com

Password


Add New Post

Title	Price	Address
Description		
City	Bedroom Number	Bathroom Number
Latitude	Longitude	Type: Rent
Property: Apartment	Utilities Policy: Owner is responsible	Pet Policy: Allowed
Income Policy: Income Policy	Total Size (sqft)	School
bus	Restaurant	

10. Testing

Testing Strategy and Tools Used:

Our testing strategy for the house hunt website primarily focused on ensuring functionality, performance, and usability across various components. We employed a combination of manual testing by team members and automated testing using Postman. The goal was to validate both frontend and backend functionalities, ensuring seamless integration and user experience.

- **Manual Testing:** Team members conducted thorough manual testing across different browsers and device sizes to ensure responsiveness and visual consistency. Key aspects tested include user registration, login/logout, house search, listing details, and user interactions.
- **Automated Testing with Postman:** We utilized Postman for API testing to validate backend functionalities such as CRUD operations for house listings, user authentication, and data validation. This approach helped us ensure API endpoints returned expected responses and handled edge cases effectively.

11. Screenshots or Demo

Demo Link :https://drive.google.com/file/d/1gXwq0y4SlJHzEO_aN2In-FbtWAmNFO7X/view?usp=sharing

12. Known Issues

Currently, there are no known issues with the developed features of the Rent Ease application. The team has thoroughly tested the existing functionalities, and everything is working as expected.

However, as with any software project, new issues may arise as more features are added and more users interact with the application.

13. Future Enhancements

Outline potential future features or improvements that could be made to the project :

Enhanced Search Functionality

- Advanced filters (e.g., neighborhood, amenities).
- Map-based search feature.

User Reviews and Ratings

- Allow users to leave reviews and ratings for houses and owners.
- Display ratings and review summaries.

Booking and Scheduling

- Schedule house viewings through the platform.
- Calendar integration for tracking appointments.

Admin Dashboard

- Manage listings, users, and platform activity.
- Analytics and reporting tools.

Multi-language Support

- Implement multi-language options for a broader audience.