**HOUSE RENT APP USING MERN**

Team Members

* Mejila A
* Manikandan V
* Navanith V
* Neha Sugumar

Purpose:

The purpose of a MERN-based house rental app is to simplify property rentals by connecting renters and owners through features like secure listings, advanced search, direct messaging, and easy applications, all in a responsive, mobile-friendly design.

Features:

* Search and Filters – Renters can search by location, price, rooms, and amenities.
* Property Details Page – Detailed info with maps, amenities, and rules.
* User Authentication – Secure login for renters and owners.
* Application Process – Renters can apply directly, upload documents.
* Favorites/Wishlist – Save preferred properties.

Architecture:

Front End :

* React Component Structure – Use reusable components (e.g., Header, PropertyCard, SearchFilters, UserProfile) organized by feature or page (like Home, Listings, Details, Dashboard).
* Routing with React Router – Define routes for main pages (/home, /listings, /details/:id, /profile, /login) to handle navigation across different views.
* State Management – Use Context API or Redux for global state (e.g., user authentication, search filters, and favorites), while keeping local component states for individual fields.
* API Integration – Create services (like authService, propertyService) to connect with the backend API (Express) for data fetching, user login, and listing CRUD operations.
* Responsive Design – Use CSS frameworks (like Tailwind or Bootstrap) or styled-components to ensure the app is mobile-friendly and adjusts to different screen sizes.
* Error Handling and Loading States – Implement error and loading state management for smooth UX, especially during data fetches or actions.
* Reusable Utility Functions – Include utilities for tasks like data formatting, validation, and error handling to simplify code across components.

Backend:

* MongoDB: A NoSQL database to store data like user profiles, property listings, bookings, payments, etc.
* Express.js: A web framework to handle API routes, middlewares, and manage HTTP requests.
* Node.js: A runtime environment to build the backend, using JavaScript to handle server-side logic and communication with MongoDB.
* Authentication: JWT (JSON Web Tokens) or OAuth for secure user authentication and role-based authorization (landlord, tenant).
* API Endpoints: RESTful API for managing properties, bookings, user profiles, reviews, and payments.
* Payment Integration: Integration with third-party services like Stripe or PayPal for payment processing.
* File Storage: Cloud storage like AWS S3 for storing property images or documents.
* Notifications: Use libraries like Nodemailer or third-party services (Twilio) for email or SMS notifications

Database:

1. MongoDB: A NoSQL database to store user data, property listings, rental transactions, and reviews. Collections could include:
   * Users (name, email, password, contact info, role)
   * Properties (address, owner ID, price, amenities, images)
   * Bookings (user ID, property ID, rental dates, payment status)
   * Reviews (user ID, property ID, rating, comments)
2. Express.js: The server framework to handle API routes, such as:
   * POST /properties (create listing)
   * GET /properties (list available properties)
   * POST /bookings (book a property)
   * GET /reviews (fetch reviews for properties)
3. React: The frontend user interface to display listings, manage bookings, and user profiles. React would interact with the backend API to perform CRUD operations.
4. Node.js: The backend runtime to execute the server (Express.js), handle business logic, and manage user authentication and authorization (using JWT or Passport.js).

Prerequisite:

1. **Backend (Node.js/Express)**:
   * express (Web framework)
   * mongoose (MongoDB ODM)
   * dotenv (Environment variable management)
   * jsonwebtoken (JWT authentication)
   * bcryptjs (Password hashing)
   * cors (Cross-origin resource sharing)
   * body-parser (Parsing incoming request bodies)
2. **Frontend (React)**:
   * react (UI library)
   * react-dom (Rendering React components)
   * react-router-dom (Routing in React)
   * axios (HTTP requests)
   * redux (State management, if used)
   * react-redux (React bindings for Redux)
3. **Development Tools**:
   * nodemon (Automatic server restarts)
   * concurrently (Running both client and server simultaneously)
   * webpack (Bundling assets, if not using Create React App)
   * eslint (Code linting)
4. **Database**:
   * **MongoDB** (Database for storing listings, user data, etc.)

Top of Form

Bottom of Form

Installation

### 1. Clone the Repository:

### git clone <repository-url>

cd <project-directory>

### 2. Install Backend Dependencies:

Navigate to the backend folder and install dependencies:

cd backend

npm install

### 3. Install Frontend Dependencies:

Navigate to the frontend folder and install dependencies:

cd ../frontend

npm install

### 4. Set Up Environment Variables:

Create a .env file in both the backend and frontend directories with necessary variables.

* **Backend**: Example .env file:

DB\_URI=mongodb://localhost:27017/houserent

JWT\_SECRET=your\_jwt\_secret

PORT=5000

* **Frontend**: Example .env file:

REACT\_APP\_API\_URL=http://localhost:5000

### 5. Run the Application:

* **Start Backend**:

cd backend

npm run dev

* **Start Frontend**:

cd frontend

npm start

Top of Form

Folder structure:

Client:

* Components:
  + HomePage: Displays featured listings, search bar, filters.
  + ListingPage: Shows individual house details (images, price, description).
  + SearchResults: Displays filtered search results.
  + Login/Signup: User authentication.
  + Profile: User's account details and rental history.
  + Navbar: Navigation bar for routing.
  + Footer: Common footer with links and info.
* State Management: React's useState or Context API for managing global state (e.g., user authentication, search filters).
* Routing: React Router for handling different pages/views.
* API Calls: Axios or Fetch for interacting with the backend (Node.js/Express), e.g., fetching house listings or posting new rental requests.
* Styling: CSS modules, styled-components, or a UI library like Material-UI for responsive design

Server:

* Server Setup (Express.js): Express handles routing and HTTP requests.
* Database (MongoDB): Stores user data, property listings, bookings, etc.
* Models (Mongoose): Define the schema for users, properties, reviews, and bookings.
* Controllers: Business logic for processing requests like adding a property, user authentication, and booking.
* Routes: Handle incoming API requests and link them to respective controller functions.
* Middleware: Functions like authentication (JWT) and error handling
* Security: Implement measures like CORS, data validation, and password hashing

Running the Application :

Frontend: npm start in the client directory

Backend:npm start in the server directory

API Documentation

* User Authentication
  + Login: POST /api/auth/login – { email, password } → { token, user }
  + Register: POST /api/auth/register – { name, email, password } → { message, user }
* Properties
  + Get All Properties: GET /api/properties – Optional filters → [ {property data} ]
  + Get Property by ID: GET /api/properties/:id – id → { property data }
  + Create Property: POST /api/properties – { property details } → { message, property }
  + Update Property: PUT /api/properties/:id – { updates } → { message, property }
  + Delete Property: DELETE /api/properties/:id → { message }
* Bookings:
  + Get User Bookings: GET /api/bookings/user/:userId – userId → [ {booking data} ]
  + Create Booking: POST /api/bookings – { propertyId, userId, dates, totalCost } → { message, booking }

Cancel Booking: DELETE /api/bookings/:id – id → { message}

Authentication :

1.**Authentication**:

* + **User Login**: The user submits their credentials (email/password). The server verifies these against hashed credentials stored in MongoDB.
  + **Token Generation**: Upon successful login, a **JWT (JSON Web Token)** is generated. This token contains the user's ID and other relevant information, signed with a secret key.
  + **Token Storage**: The token is sent to the client, usually stored in **HTTP-only cookies** (for better security) or **localStorage** (less secure).

1. **Authorization**:
   * **Protected Routes**: For API routes requiring access control (e.g., viewing/editing house listings), the client includes the JWT in the request headers (e.g., Authorization: Bearer <token>).
   * **Token Verification**: The server verifies the token using the secret key. If valid, the user's role and permissions are checked to ensure they are authorized for the requested action.
   * **Role-Based Access Control (RBAC)**: Different user roles (e.g., admin, owner, tenant) determine the level of access to various features.
2. **Session Management**:
   * JWTs are stateless, meaning the server doesn't maintain sessions. However, sessions can be managed using cookies with an expiration time, enabling token invalidation by clearing cookies.
   * Refresh tokens can also be implemented for prolonged sessions. These tokens are stored securely and used to generate new access tokens when they expire.

Top of Form

User interface:

* Homepage: Search bar, featured listings.
* Listings Page: Grid/list view with filters (price, type, amenities).
* Details Page: Property images, description, map view, and contact button.
* User Dashboard: Saved properties, manage listings/inquiries.
* Authentication: Login/Signup with email/social options.
* Admin Panel: Manage users, listings, analytics.
* Interactive Map: Property pins with quick previews.
* Responsive Design: Mobile-optimized views.

Test:

* Unit Testing: Test individual components with Jest and Mocha.
* Integration Testing: Use Supertest for backend API testing.
* E2E Testing: Simulate user flows with Cypress or Playwright.
* Performance Testing: Test scalability with Postman and JMeter.
* Manual Testing: Explore edge cases and UI issues

Bottom of Form

Known issue:

* Authentication Issues: Token expiry, unauthorized access, weak password hashing.
* Database Performance: Slow queries, missing indexes.
* API Error Handling: Unhandled errors, lack of rate limiting.
* State Management in React: Sync issues, memory leaks.
* Responsive Design: Layout inconsistencies, poor image scaling.
* Payment Gateway Integration: Transaction failures, security vulnerabilities.
* Real-time Features: WebSocket disconnections, slow event handling.
* SEO Optimization: Missing meta tags, SSR issues.
* Cross-Browser Compatibility: Inconsistent behavior, missing CSS prefixes.
* Security Concerns: Injection attacks, XSS vulnerabilities.

Future enhancement:

User Authentication & Authorization

* Role-based access, social media logins, and two-factor authentication.

Property Search & Filtering

* Advanced filters, map integration, and search history.

Landlord Dashboard

* Property management, lease tracking, and maintenance requests.

Tenant Management

* Messaging system, document upload, and reviews/ratings.

Payment Integration

* Online payments, invoice generation, and payment reminders.

AI Features