HOUSE RENTAL APP USING MERN

Project Description:

A house rent app is typically a mobile or web application designed to help users find rental properties, apartments, or houses for rent. These apps often offer features to make the process of searching for and renting a property more convenient and efficient. Here are some common features you might find in a house rent app:

Property Listings:

The app provides a database of available rental properties, complete with detailed descriptions, photos, location, rent amount, and other relevant information.

Search Filters:

Users can apply various filters to narrow down their search results based on criteria such as location, rent range, property type (apartment, house, room, etc.), number of bedrooms, amenities, and more.

Contact Landlords/Property Managers:

The app might provide a way for users to contact the property owners or managers directly through the app, often through messaging or email.

Table Of Content

Introduction

Project Title: House Rent App Using MERN

The real estate rental market often faces inefficiencies, with outdated listing methods, poor communication, and a lack of transparency in rental agreements. Property owners struggle to find reliable tenants quickly, while renters face difficulties like limited access to detailed property information and cumbersome workflows for inquiries and agreements. Additionally, there is no centralized system that integrates property management, communication, and rental workflows in a seamless, secure way.

RentIt provides a comprehensive solution by modernizing the rental process. The platform enables property owners to easily list and manage properties, while tenants can quickly search for and request rentals. Real-time communication, powered by Socket.IO, allows for instant interactions and live updates, simplifying negotiations and inquiries. RentIt's role-based access ensures that users—whether Admin, Owner, or Tenant—have appropriate access to the platform's features, enhancing security and trust.

With location-based search, digital rental agreements, and an admin moderation panel, RentIt addresses key pain points in the market by creating a secure, transparent, and user-friendly experience. By combining property management, real-time communication, and streamlined workflows, RentIt transforms the way property rentals are handled, bridging the gap between owners and tenants efficiently.

Project Overview

Purpose

A user-friendly real estate rental platform designed to connect property owners with people looking for rental homes. The platform simplifies the entire rental process, starting from listing properties to finalizing rental agreements. It includes features like secure user verification, easy-to-use search filters, and real-time messaging to ensure smooth communication between owners and tenants. Property owners can easily list and manage their properties, while tenants can browse, request rentals, and communicate directly with owners. The platform ensures transparency, security, and convenience for everyone involved, making renting properties faster and stress-free.

Features

User Management

- Multi-role system (Admin, Owner, User) with role-specific functionalities
- Secure authentication with JWT
- Profile management with document verification
- Owner verification system with admin approval

Property Management

- Property listing with multiple images and detailed information
- Advanced property filtering (location-based, amenities, price range)
- Property status tracking (available, occupied, unlisted)
- Legal document verification for properties

Rental Process

- Streamlined rental request system
- Real-time chat between owners and potential tenants
- Digital rental agreement generation and acceptance
- Payment verification through UPI integration

Admin Controls

- Property listing approval system
- Owner verification management
- Platform monitoring and user management
- Content moderation capabilities

Security & Verification

- Document verification for users and properties
- Secure payment confirmation system
- Rate limiting and request validation
- Data encryption and secure storage

Technical Features

- Real-time updates using Socket.IO
- Geolocation-based property search using two different free APIs (Nomatim & Geocode Maps)
- Cloud-based image storage using cloudinary
- Responsive design for all devices
- JWT-based authentication and authorization

This platform aims to modernize the traditional rental process by providing a secure, efficient, and user-friendly environment for both property owners and tenants while maintaining necessary verification and security measures.

Architecture

Frontend Architecture

The frontend is built using React 18, following a modern and modular architecture to ensure scalability and maintainability.

Core Technologies

- **React**: Main UI library (v18.3.1)
- **Zustand**: Lightweight state management
- **React Router**: Client-side routing
- **Tailwind CSS & MUI**: Styling and component libraries
- **Socket.io-client**: Real-time communication
- **Axios**: HTTP requests

Key Features

- Responsive design using Tailwind CSS
- Material UI for advanced components and user-friendly interfaces
- Real-time updates with WebSocket via Socket.io
- Client-side form validation for improved user experience
- Image carousel implemented using react-slick
- Toast notifications for feedback alerts
- Phone number validation integrated into forms
- Location selection using `react-country-state-city`

Backend Architecture

The backend is developed with Node.js and the Express.js framework, following a modular MVC pattern for clean and maintainable code.

Core Technologies

- **Express.js**: Web server framework
- **Socket.io**: Real-time messaging and notifications
- **Mongoose**: MongoDB ODM for database interactions
- **JWT**: Secure token-based authentication
- **Multer & Cloudinary**: File uploads and cloud storage
- **bcryptjs**: Secure password hashing

Key Features

- Real-time messaging and notifications using Socket.io
- Secure file uploads with Cloudinary integration
- JWT-based authentication for user sessions
- Role-based access control for secure endpoints
- Geocoding with caching for efficient location queries
- Centralized error handling middleware
- CORS configuration for cross-origin support
- Input validation middleware for request sanitization

Database Architecture

MongoDB is used as the database, managed with Mongoose ODM for schema modeling and efficient queries.

Database Features

- Indexing on frequently queried fields for faster lookups
- Geospatial indexing for location-based searches
- Document references to establish relationships between models
- Automatic timestamps for all documents
- Cascade deletion & save hooks to maintain data consistency
- Schema-level validation for data integrity

Data Flow

1. Client sends a request through the React frontend.
2. The request is authenticated using JWT middleware.
3. Controllers handle the request, applying business logic.
4. Mongoose interacts with the MongoDB database.
5. Socket.io sends real-time updates when required.
6. The server responds to the client with the requested data.

Core Models

User Model

```
{
  username: string,
  email: string,
  password: string,
  type: enum["user", "owner", "admin"],
  profilePicture: string,
```

```
  contactNumber: string,
  legalVerificationID: string
}
```

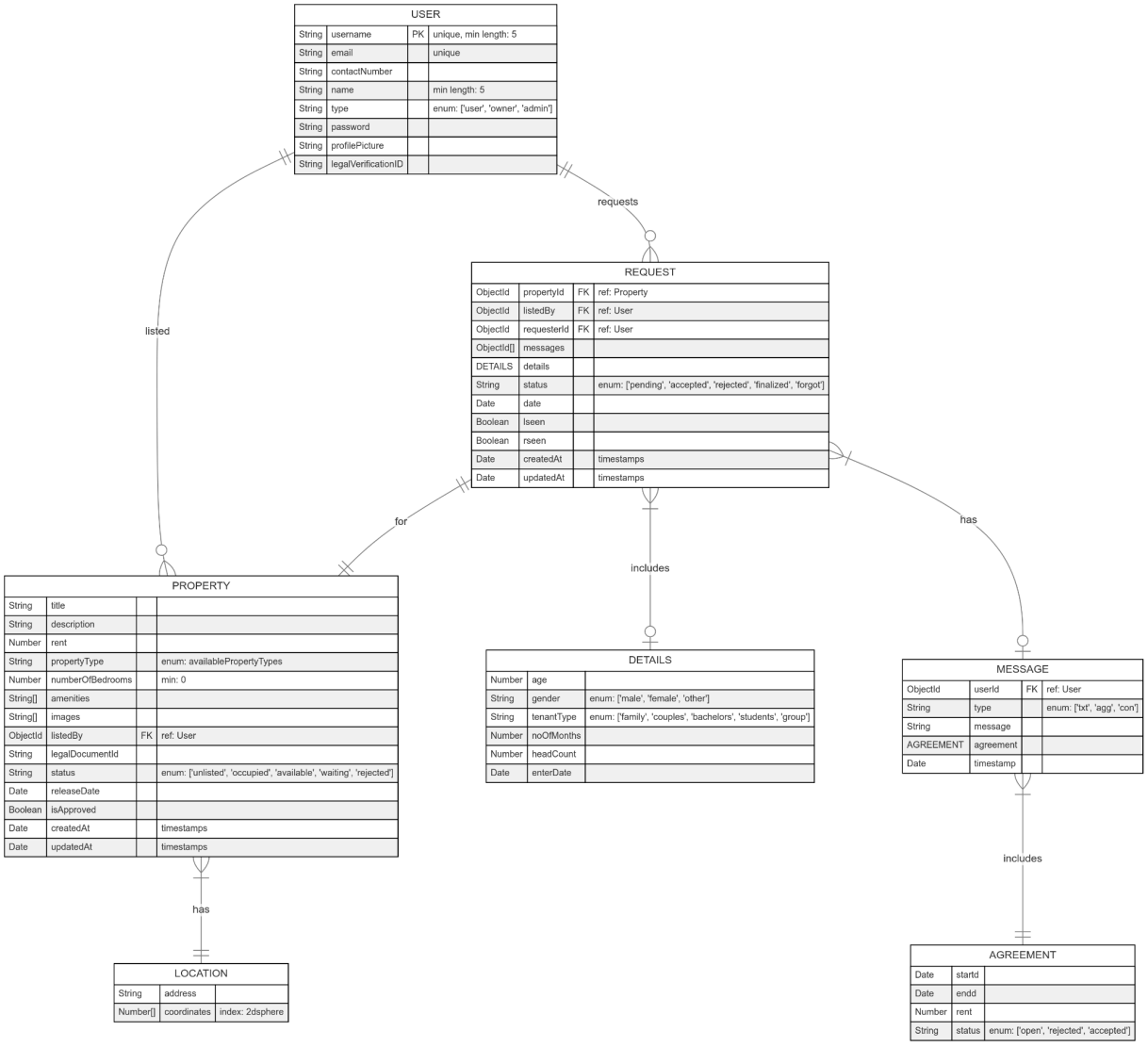
Property Model

```
{
  title: string,
  description: string,
  location: {
    address: string,
    coordinates: [number, number]
  },
  images: string[],
  rent: number,
  amenities: string[],
  propertyType: string,
  status: enum["available", "occupied", "unlisted"],
  listedBy: ref(User),
  isApproved: boolean
}
```

Request Model

```
{
  propertyId: ref(Property),
  requesterId: ref(User),
  listedBy: ref(User),
  status: string,
  messages: [{
    user_id: ref(User),
    message: string,
    type: enum["txt", "agg", "con"],
    timestamp: Date
  }]
}
```

Entity Relationship



This architecture provides scalability, maintainability, and real-time functionality while ensuring secure and efficient data processing.

Setup Instruction

Prerequisites

- Node.js (v18 or higher)
- MongoDB (v6 or higher, only when using the local mongodb)
- Git
- npm (Node Package Manager)
- A Cloudinary account
- Maps Geocoding API key

Additional Setup Notes

1. **MongoDB Setup**
    - Install MongoDB locally or use MongoDB Atlas from cloud.mongodb.com
    - Update MONGO_URI in backend .env file
2. **Cloudinary Setup**
    - Create a Cloudinary account
    - Get cloud name, API key, and secret
    - Create an `uploads` folder in Cloudinary
    - Update Cloudinary config in backend .env
3. **Geocoding Setup**
    - Get API key from geocode.maps.co
    - Add key to backend .env file

Installation Steps

1. Clone Repository

```
git clone https://github.com/nonkloq/rentit.git
cd rentit
```

2. Environment Setup

Backend Configuration

```
cd backend
cp .env.example .env   # Copy example environment file
```

Edit `.env` file with your credentials and secrets (refer to the Environment Variables section below).

Frontend Configuration

```
cd frontend
cp .env.example .env   # Copy example environment file
```

Add Backend API URL to the `.env` file.

3. Installation Options

Option 1: Quick Setup (Recommended)

Run from project root directory:

```
cd ~/path/to/rentit
npm run build   # Installs all dependencies and builds the project
```

Option 2: Manual Setup

```
cd backend
npm install
cd frontend
npm install
```

Environment Variables

Backend (.env)

```
PORT=6969                          # Server port
JWT_SECRET=your_jwt_secret         # JWT encryption key
JWT_EXPIRE=7d                      # JWT expiration time
MONGODB_URI=your_mongodb_uri       # MongoDB connection string

# API Keys
GEOCODE_API_KEY=your_geocode_key   # Maps Geocoding API key

# Legal Document Verification (Dummy)
```

```
LEGD_SI=your_secret_key             # Property document start index
LEGD_EI=your_secret_key             # Property document end index
SECRET_PHRASE1=your_secret_phrase   # Property verification phrase

# Legal Verification (Dummy)
LEGV_SI=your_secret_key             # User verification start index
LEGV_EI=your_secret_key             # User verification end index
SECRET_PHRASE2=your_secret_phrase   # User verification phrase

SECRET_VAL=your_secret_value        # General purpose secret

# Registration
REGISTER_PASSPHRASE=your_passphrase # Registration security phrase

# Cloudinary Config
CLOUDINARY_CLOUD_NAME=your_cloud_name
CLOUDINARY_API_KEY=your_api_key
CLOUDINARY_API_SECRET=your_api_secret

# Client URL
CLIENT_URL=http://localhost:3000/   # Frontend URL
```

Frontend (.env)

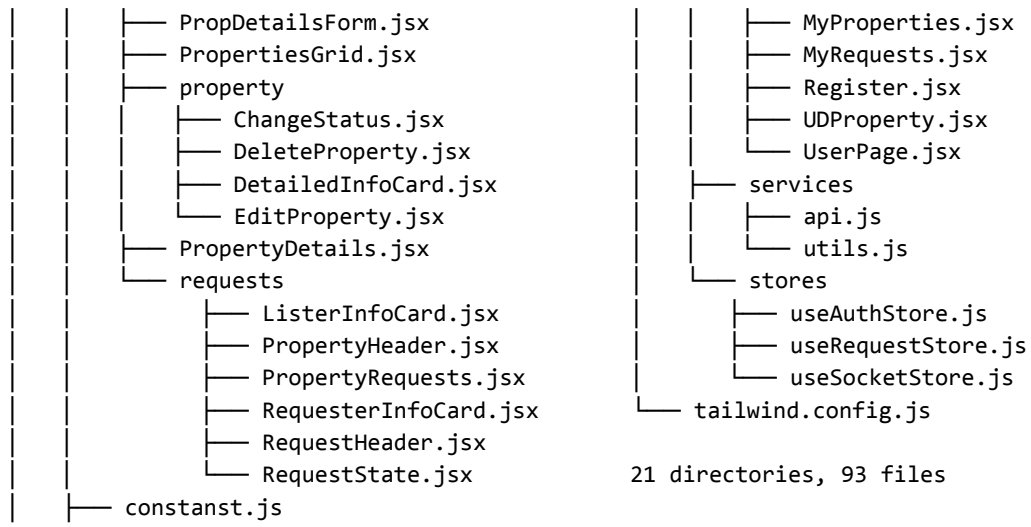
```
REACT_APP_SERVER_URL=http://localhost:6969  # Backend API URL
```

Folder Structure

React.js Frontend (Client)

Folder Structure

```
.
├── .env
├── package.json
├── package-lock.json
├── postcss.config.js
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── manifest.json
│   └── robots.txt
├── README.md
├── src
│   ├── App.jsx
│   ├── components
│   │   ├── AuthForm.jsx
│   │   ├── blocks
│   │   │   ├── loading.jsx
│   │   │   ├── naBlock.jsx
│   │   │   └── noListings.jsx
│   │   ├── chat
│   │   │   ├── AcceptOrDecline.jsx
│   │   │   ├── Agreement.jsx
│   │   │   ├── Contract.jsx
│   │   │   ├── MessageBox.jsx
│   │   │   ├── MessageInput.jsx
│   │   │   └── SendAgreement.jsx
│   │   ├── ChatSkeleton.jsx
│   │   ├── frags
│   │   │   ├── AminitiesSelect.jsx
│   │   │   ├── ImageSlider.jsx
│   │   │   ├── PasswordInput.jsx
│   │   │   └── ProptypeSelect.jsx
│   │   ├── home
│   │   │   └── FilterWidget.jsx
│   │   ├── listing
│   │   │   ├── RequestForm.jsx
│   │   │   └── UserInfoCard.jsx
│   │   ├── navbar.jsx
│   │   ├── profile
│   │   │   ├── ChangePassword.jsx
│   │   │   ├── EditProfile.jsx
│   │   │   └── OwnerRequestButton.jsx
│   │   ├── ProfileBox.jsx
│   │
│   ├── hooks
│   │   ├── admin
│   │   │   ├── useApproveProperty.js
│   │   │   ├── useGetORequests.js
│   │   │   ├── useGetUProperties.js
│   │   │   └── useRespondToORequest.js
│   │   ├── auth
│   │   │   ├── useLogin.js
│   │   │   ├── useLogout.js
│   │   │   └── useRegister.js
│   │   ├── chat
│   │   │   ├── useFetchRequests.js
│   │   │   ├── useGetMessages.js
│   │   │   ├── useRespondToAgreement.js
│   │   │   ├── useRespondToRequest.js
│   │   │   ├── useSendAgreement.js
│   │   │   ├── useSendMessage.js
│   │   │   └── useSendRequest.js
│   │   ├── property
│   │   │   ├── useChangeStatus.js
│   │   │   ├── useCreateProperty.js
│   │   │   ├── useDeleteProperty.js
│   │   │   ├── useEditDetails.js
│   │   │   ├── useFetchProperties.js
│   │   │   ├── useGetMyProperties.js
│   │   │   ├── useGetMyProperty.js
│   │   │   └── useGetProperty.js
│   │   └── user
│   │       ├── useChangePassword.js
│   │       ├── useEditProfile.js
│   │       ├── useGetMyInfo.js
│   │       ├── useGetUserInfo.js
│   │       └── useSendOwnerRequest.js
│   ├── index.css
│   ├── index.js
│   ├── pages
│   │   ├── AddProperty.jsx
│   │   ├── AdminPanel.jsx
│   │   ├── Home.jsx
│   │   ├── Listing.jsx
│   │   ├── Login.jsx
│   │   ├── MyProfile.jsx
```



Core Directories

- ★ **components/**: Reusable UI components organized by features (auth, chat, property, etc.)
  - **blocks/**: Common UI blocks like loading states
  - **frags/**: Reusable fragment components
  - **chat/**: Real-time messaging components
  - **property/**: Property management components
  - **profile/**: User profile related components
- ★ **hooks/**: Custom React hooks to perform api calls organized by feature
  - **admin/**: Admin panel operations
  - **auth/**: Authentication operations
  - **chat/**: Real-time messaging logic
  - **property/**: Property CRUD operations
  - **user/**: User profile management
- ★ **pages/**: Main route components
- ★ **stores/**: Zustand state management stores
- ★ **services/**: API integration and utility functions

Node.js Backend (Server)

Folder Structure

```
.
├── .env
├── package.json
├── package-lock.json
├── README.md
├── src
│   ├── config
│   │   ├── cloudinaryConf.js
│   │   ├── constants.js
│   │   ├── database.js
│   │   ├── multerConf.js
│   │   └── socket.js
│   ├── controllers
│   │   ├── admin.js
│   │   ├── auth.js
│   │   ├── listing.js
│   │   ├── owner.js
│   │   └── user.js
│   ├── lib
│   │   ├── errors.js
│   │   ├── geocode.js
│   │   ├── utils.js
│   │   └── validators.js
│   ├── middlewares
│   │   ├── auth.js
│   │   └── error.js
│   ├── models
│   │   ├── ownerRequest.js
│   │   ├── property.js
│   │   ├── propertyRequest.js
│   │   └── user.js
│   ├── routes
│   │   ├── admin.js
│   │   ├── auth.js
│   │   ├── listing.js
│   │   ├── owner.js
│   │   └── user.js
│   └── server.js
└── uploads

9 directories, 30 files
```

Core Directories

- ★ **config/**: Configuration files
  - Database connection
  - Cloudinary setup
  - Socket.io configuration
  - Multer setup
- ★ **controllers/**: Business logic handlers
  - User management
  - Property operations
  - Authentication
  - Admin operations
  - Listing management
- ★ **models/**: Mongoose schemas
  - User model
  - Property model

    - Request models
      - Property Request Model
      - Owner Request Model
- ★ **routes/**: API route definitions
- ★ **middlewares/**: Custom middleware (auth, error handling)
- ★ **lib/**: Utility functions, validators, and error handlers

This structure ensures scalability, maintainability, and clear separation of concerns in both frontend and backend codebases.

Running The Application

Development Mode

Run frontend and backend separately for development with hot-reloading:

Frontend

```
cd frontend
npm install
npm start     # Runs on http://localhost:3000
```

Backend

```
cd backend
npm install
npm run dev   # Runs on http://localhost:6969 with nodemon
```

Production Mode

Run entire application from project root:

```
npm run build   # Installs dependencies and builds frontend
npm run start   # Serves both frontend and backend
```

You can skip the installation commands once you have successfully set up both the frontend and backend.

Access Application

When running from project root (production mode), both frontend and backend are served at http://localhost:6969. For development mode,

- Frontend runs at : http://localhost:3000
- Backend API on: http://localhost:6969

Note

The project includes sampled property listings from the Houses Dataset with details generated using Gemini-1.5-Flash via the Gemini API.

For sample data and generation process, refer to `datafiller.ipynb` in the project root.

Deployment

The application is configured to be deployed on Render. The Deployment script is the same as the commands from Production Mode. Visit the live site for a demo. Please note that it may take some time to start the server on the initial load, as it is running on a free tier and experiences a cold start after periods of inactivity.

Troubleshooting

- Ensure MongoDB is running locally or connection string is correct
- Check if all environment variables are properly set
- Verify port 6969 and 3000 are not in use
- Clear browser cache if frontend changes don't reflect
- Check Node.js version compatibility

API Documentation

REST API for House Rental Application (RentiT) that handles user authentication, property management, rental requests, and real-time messaging. Serves static frontend in production and provides API endpoints in development.

Authentication

- All routes except `/auth/login`, `/auth/register`, and `/list/*` require JWT token
- Token Format: `authorization: Bearer <token>`

Response Formats

Success Response

```
{
    "message": "Success message",
    "data": {} // response data
}
```

Error Response Format

All errors follow a consistent format with appropriate HTTP status codes (400, 401, 403, 404, 429, 500):

```
{
    "success": false,
    "message": "Unable to process your request",
    "stack": "Error: ....", // only in development
    "code": "STATUS_CODE"
}
```

Root Route (/)

- **GET /** - Returns API version in development, serves React frontend in production
- **Returns Dev**: `"HRA-private-API-v0.1.0b1"` (`String`)
- **Returns Prod**: `Serves static frontend from /frontend/build/index.html` (`StaticFiles`)

Authentication Routes (`/auth`)

POST /auth/register

- **Description**: Register a new user.
- **Input**: `{ name: string, email: string, password: string, passphrase: string }`
- **Returns**: `{ message: string, token: string, userData: { uid: string, type: "user" | "owner" | "admin" } }`

POST /auth/login

- **Description**: Login using email and password.
- **Input**: `{ email: string, password: string }`
- **Returns**: `{ message: string, token: string, userData: { uid: string, type: "user" | "owner" | "admin" } }`

POST /auth/logout

- **Description**: Logout the current user.
- **Input**: `none`
- **Returns**: `{ message: string }`

User Routes (`/user`)

GET /user/profile

- **Description**: Retrieve current user profile details.
- **Input**: `none`
- **Returns**: `{ message: string, user: { name: string, email: string, username: string, type: string, contactNumber: string, profilePicture: string, legalVerificationID: string } }`

PUT /user/profile

- **Description**: Update user profile.
- **Input**: `{ username?: string, contactNumber?: string, name?: string, profilePicture?: File, legalVerificationID?: string }`
- **Returns**: `{ message: string, user: UserType }`

PUT /user/profile/password

- **Description**: Change user password.
- **Input**: `{ currentPassword: string, newPassword: string }`
- **Returns**: `{ message: string }`

POST /user/orequest

- **Description**: Request to become an owner.
- **Input**: none
- **Returns**: `{ message: string, userData?: { type: "owner" } }`

POST /user/prequest/:propertyId

- **Description**: Send property rental request.
- **Input**: `{ age: number, gender: string, tenantType: string, noOfMonths: number, headCount: number, enterDate: Date, message?: string }`
- **Returns**: `{ message: string, newRequest: RequestType }`

GET /user/prequest

- **Description**: Get all rental requests made by the user.
- **Input**: none
- **Returns**: `{ requests: Array<{ propertyId: string, status: string, details: RequestDetailsType, messages: MessageType[], date: Date }> }`

GET /user/prequest/:requestId

- **Description**: Get messages for a specific request.
- **Input**: none
- **Returns**: `{ messages: MessageType[], rent: number }`

POST /user/message/:requestId

- **Description**: Send a message in a property request.
- **Input**: `{ message: string }`
- **Returns**: `{ requestId: string, newMessage: MessageType, rseen: boolean, lseen: boolean }`

POST /user/respond/:requestId

- **Description**: Respond to a rental agreement.
- **Input**: `{ aggId: string, upiId: string, pin: string, click: "accept" | "reject" }`
- **Returns**: `{ message: string, request: RequestType }`

Admin Routes (`/admin`)

PUT /admin/preq/:propertyId

- **Description**: Approve or reject property listings.
- **Input**: `{ flag: "accept" | "reject" }`
- **Returns**: `{ message: string }`

PUT /admin/oreq/:requestId

- **Description**: Approve or reject owner requests.
- **Input**: `{ flag: "accept" | "reject" }`
- **Returns**: `{ message: string }`

GET /admin/preq

- **Description**: Retrieve all unapproved properties.
- **Input**: `none`
- **Returns**: `Array<PropertyType>`

GET /admin/oreq

- **Description**: Retrieve all pending owner requests.
- **Input**: `none`
- **Returns**: `Array<{ requesterId: string, status: string, timestamp: Date }>`

Owner Routes (`/owner`)

POST /owner/property

- **Description**: Add a new property listing.
- **Input**: `{ title: string, description: string, rent: number, propertyType: string, numberOfBedrooms: number, amenities:`

`string[], address: string, legalDocumentId: string, images: File[] }`

- **Returns**: `{ message: string }`

GET /owner/property

- **Description**: Retrieve all properties listed by the owner.
- **Input**: `none`
- **Returns**: `{ message: string, properties: PropertyType[] }`

PUT /owner/property/:propertyId

- **Description**: Update property details.
- **Input**: `{ title?: string, description?: string, rent?: number, propertyType?: string, numberOfBedrooms?: number, amenities?: string[], address?: string, legalDocumentId?: string, images?: File[], removeImages?: string[] }`
- **Returns**: `{ message: string }`

DELETE /owner/property/:propertyId

- **Description**: Delete a property listing.
- **Input**: `none`
- **Returns**: `{ message: string }`

PUT /owner/property/status/:propertyId

- **Description**: Change the status of a property.
- **Input**: `{ status: "unlisted" | "available" }`
- **Returns**: `{ message: string }`

POST /owner/agreement/:requestId

- **Description**: Send a rental agreement.
- **Input**: `{ startd: Date, endd: Date, rent: number, optMessage?: string }`
- **Returns**: `{ requestId: string, newMessage: MessageType, rseen: boolean, lseen: boolean }`

POST /owner/respond/:requestId

- **Description**: Respond to a property request.

- **Input**: `{ flag: "accepted" | "rejected" }`
- **Returns**: `{ message: string, request: RequestType }`

Public Listing Routes (`/list`)

GET /list/filter

- **Description**: Retrieve filtered properties based on various parameters.
- **Input Query**: `{ address?: string, maxDistance?: number, priceRange?: string, numberOfBedrooms?: number, amenities?: string, propertyType?: string }`
- **Returns**: `{ message: string, properties: PropertyType[] }`

GET /list/:propertyId

- **Description**: Retrieve public property details by ID.
- **Input**: `none`
- **Returns**: `{ message: string, property: { details: PropertyType, listedBy: PublicUserType } }`

GET /list/u/:username

- **Description**: Retrieve public user profile and their listed properties.
- **Input**: `none`
- **Returns**: `{ message: string, properties: PropertyType[], user: PublicUserType }`

Authentication

Authentication Flow

Registration

1. User registers with email, password, and registration passphrase
2. Password is hashed using bcryptjs
3. User document created with default role "user"
4. JWT token generated and returned

Login

```
type LoginResponse = {
    token: string, // JWT Token that will be saved in localStorage
    userData: {
        uid: string,
        type: "user" | "owner" | "admin"
    }
}
```

Token Management

- JWT (JSON Web Token) based authentication
- Token stored in client's local storage
- Included in Authorization header: `Bearer <token>`
- Token expiration: 7 days

Authorization Levels

User Roles

1. **User**
    - Basic property browsing
    - Send rental requests
    - Chat with owners
    - Profile management
2. **Owner** (Requires admin approval)
    - All user permissions
    - Property management
    - Handle rental requests

    - Send rental agreements
3. **Admin**
    - Approve/reject properties
    - Manage owner requests
    - Full platform access

Middleware Protection

```
authMiddleware("owner", "admin")  // Requires owner or admin role
authMiddleware("admin")           // Requires admin role
authMiddleware()                  // Requires any authenticated user
```

Security Features

- Password hashing with bcryptjs
- JWT secret key encryption
- Role-based route protection
- Registration passphrase requirement
- Document verification system
- Request validation
- Rate limiting in Geocode API

Logout

- Client removes JWT from local storage
- Socket connection terminated
- Server-side token invalidation

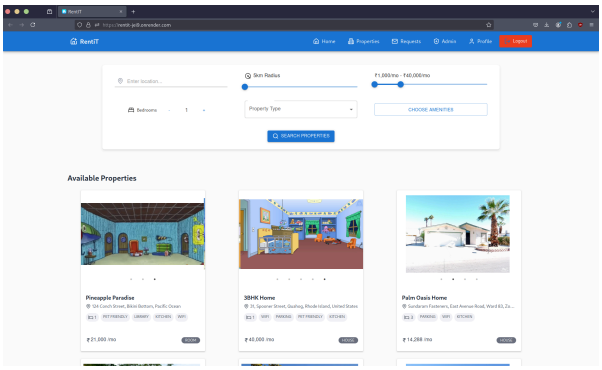
This system ensures secure access control while maintaining a smooth user experience across different role levels.

User Interface & Demo

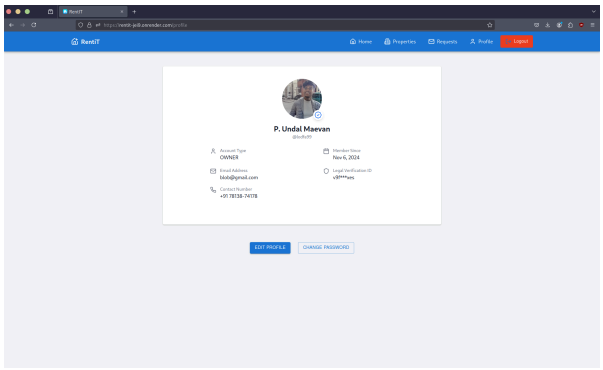
The application has a modern, responsive React interface with Material-UI and Tailwind CSS for styling, and Lucid-react for icons. For a comprehensive view of all features and interactions, Watch the demo video here (https://youtu.be/SlHoExFpXx8) or visit the GitHub repository for detailed screenshots.

Main Pages

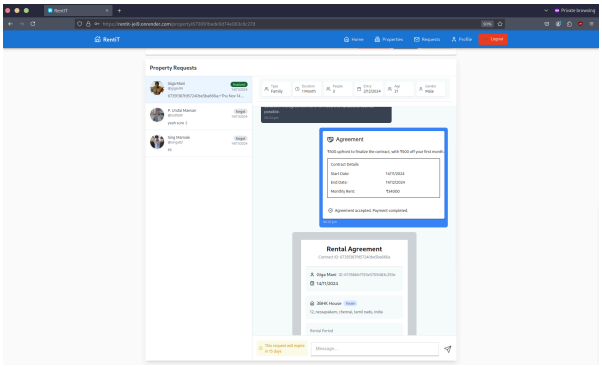
Home



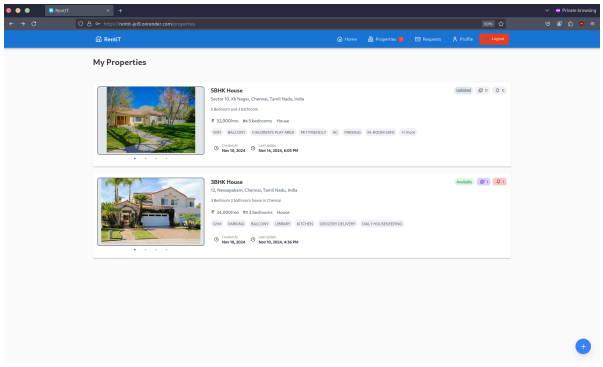
Profile



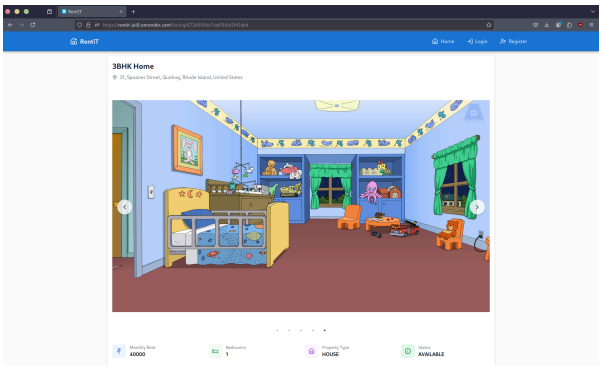
Realtime Chat



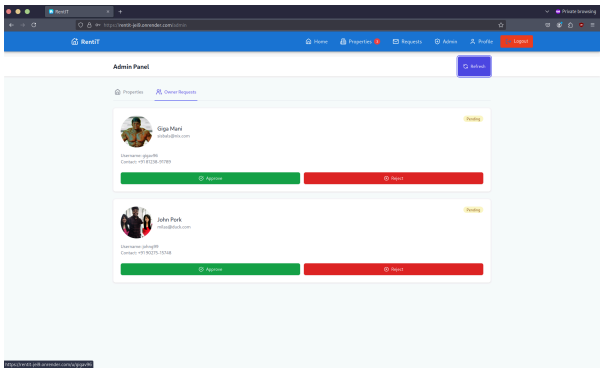
Property Management



Property Listing



Admin Panel



For more screenshots go to the ss folder from the repository or check out the screenshots in readme.

Testing

While no automated testing tools were implemented, the application underwent rigorous manual testing:

Testing Strategy

1. **User Flow Testing**
    - Multiple test accounts across different roles
    - Cross-browser compatibility verification
    - Mobile responsiveness testing
2. **Real-time Feature Testing**
    - Concurrent user sessions
    - Socket connection stability
    - Message delivery verification
3. **Edge Case Testing**
    - Network interruption handling
    - Large file upload scenarios
    - Concurrent request management
4. **Security Testing**
    - Role-based access control
    - Token validation
    - Input sanitization

Known Issues

1. **Verification System**
    - Currently implements dummy verification for both users and properties
    - Legal document verification is simulated
2. **Payment Integration**
    - Payment system is simulated
    - UPI verification is not connected to actual payment gateways
3. **Address Search**
    - Basic text input without address suggestions
    - No address validation or standardization
    - Limited geocoding accuracy

Future Enhancements

Payment Integration

- Integrate Stripe payment gateway
- Real-time payment status tracking
- Payment history and analytics
- Multiple payment method support

Verification Systems

- Implement proper KYC verification
- Digital signature integration
- Blockchain-based document verification
- Real-time ID verification

Location Services

- Google Places API integration for address autocomplete
- Real-time location validation
- Interactive map for property location
- Proximity-based property suggestions

Additional Features

- In-app video tour scheduling
- AI-powered property recommendations
- Property analytics dashboard
- Mobile application development

These enhancements would significantly improve the platform's functionality and user experience while adding necessary security and verification features.

Conclusion

RentIt is a comprehensive MERN stack rental platform that demonstrates modern web development practices and real-time features. The application successfully implements:

- Multi-role user system with secure authentication
- Real-time messaging using Socket.IO
- Property management with image handling
- Location-based property search
- Rental agreement workflow

While certain features like payment processing and verification systems are currently simulated, the architecture is designed to easily integrate production-ready solutions. The combination of React, Node.js, MongoDB, and Socket.IO provides a robust foundation for scaling and future enhancements.

The project serves as both a practical solution for property rental management and a showcase of full-stack development capabilities using the MERN stack. Visit the live demo or watch the demo video to explore the full functionality.