Implementation Document

for

Roomble

Version <1.0>

Prepared by

Group #: 7 **Group Name: Marauders**

Aarsh Jain	**230015**	aarshjain23@iitk.ac.in
Aritra Ambudh Dutta	**230191**	aritraad23@iitk.ac.in
Aritra Ray	**230193**	aritrar23@iitk.ac.in
Bhukya Vaishnavi	**230295**	bhukyav23@iitk.ac.in
Bikramjeet Singh	**230298**	bsingh23@iitk.ac.in
Hitarth Makawana	**230479**	hitarthkm23@iitk.ac.in
Shlok Jain	**230493**	jainshlok23@iitk.ac.in
Ronav Puri	**230815**	ronavg23@iitk.ac.in
Rathod Ayushi	**230844**	rathoday23@iitk.ac.in
Saksham Verma	**230899**	sakshamv23@iitk.ac.in
Surepally Pranaysriharsha	**231057**	surepally23@iitk.ac.in

Course: **CS253**

Mentor TA: **Nij Bharatkumar Padariya**

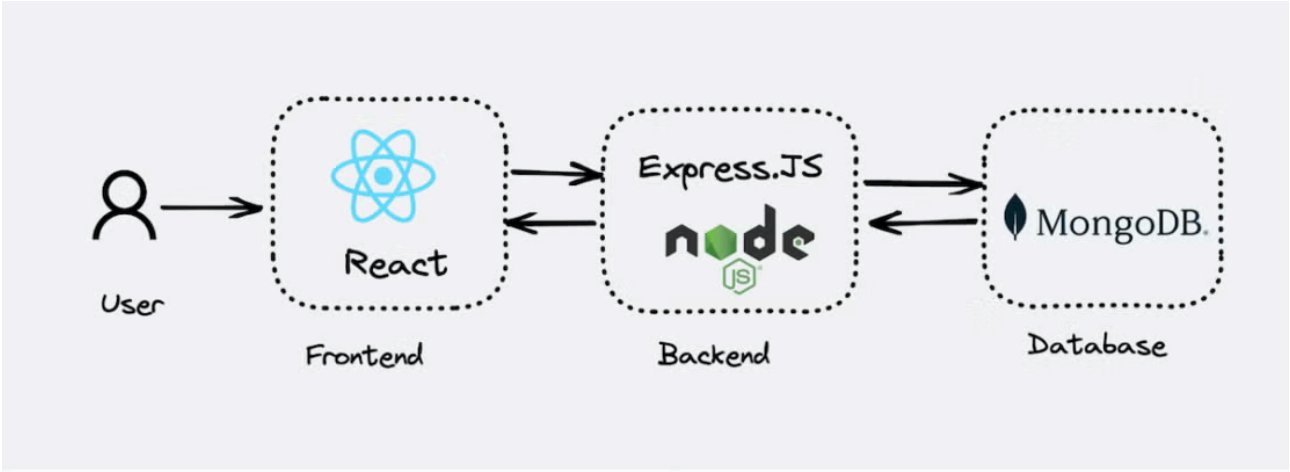
Date: **28/03/2025**

Contents

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
v1.0	Aarsh Jain Aritra Ambudh Dutta Aritra Ray Bhukya Vaishnavi Bikramjeet Singh Hitarth Makawana Shlok Jain Ronav Puri Rathod Ayushi Saksham Verma Surepally Pranaysriharsha	First Version of the Implementation Document	28/03/25

1 Implementation Details



Roomble is an end-to-end property rental and flatmate discovery platform that leverages the **MERN** stack—**MongoDB**, **Express.js**, **React.js**, and **Node.js**—to provide swift, scalable, and user-friendly solutions. The system seamlessly connects tenants searching for accommodations with landlords listing properties, all while simplifying the user journey from registration to closing deals.

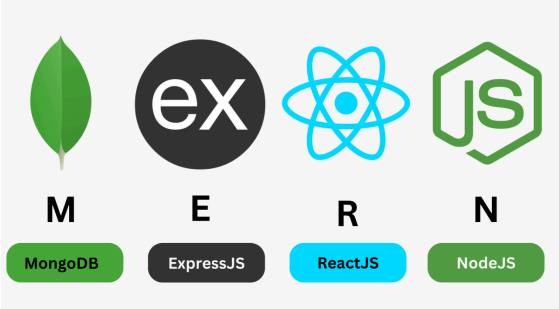
MongoDB serves as the platform's NoSQL database, allowing the application to store user profiles, property data, bookmarks, and reviews in a flexible JSON-based schema. This flexible data model is essential for supporting a variety of user needs—tenants, landlords, and property information—without the rigidity of traditional relational databases.

Express.js and **Node.js** serve as the backbone of the server-side implementation. Express.js handles all routing and middleware logic, including secure JWT-based authentication endpoints for both tenants and landlords, while Node.js empowers the application to efficiently manage asynchronous operations. This combination creates a streamlined backend that accepts requests from any connected frontend while interacting cleanly with the MongoDB database.

React.js composes a dynamic and intuitive user interface, allowing seamless navigation across pages where tenants can search for flatmates or properties based on personal preferences, bookmark results, and message potential matches. React's component-based architecture makes organising business logic easier, promotes reusable UI elements, and ensures users a positive, responsive experience.

Overall, Roomble's implementation showcases how the MERN stack can be used to create a robust solution for community-driven marketplaces. By pairing MongoDB's flexible schemas with Node.js and Express.js for blazing-fast server operations and React.js for an appealing user interface, Roomble delivers a modern web application that's intuitive, scalable, and built to handle the complexities of matching people with the perfect place to live.

z



Programming Languages, Framework, and Libraries

Programming Language

Roomble has been primarily developed using **Javascript** language. It follows the **Model-View-Controller** architecture pattern.

For the **Backend**, we have used:

1. **Javascript:** The benefit of using Javascript (**Node.js**) for the backend is that, as compared to C++/Java, it offers much more flexibility by providing a wide range of frameworks enabling efficient asynchronous operations, code reusability with React.js, and scalability with its robust ecosystem of tools and resources.

For the **Frontend**, we have used:

1. **HTML(Hypertext Markup Language):** *HTML* is used to structure web pages. Every browser supports HTML, and it is very easy to use.

2. **CSS(Cascading Style Sheets):** *CSS* is employed for styling HTML elements, providing a consistent and visually appealing layout across multiple web pages. It is also easy to maintain. Making a global change is simple: Just update the style, and all elements across all web pages will be automatically updated.

3. **JS(Javascript):** *JavaScript* plays a dual role here. While Node.js handles the backend, JavaScript on the front end adds interactivity to the user interface and makes the application dynamic for the user. JavaScript is very fast because it can be run immediately within the client-side browser. Unless outside resources are required, JavaScript is unhindered by network calls to a backend server. Also, it is easy to implement.

For the **DBMS** (**Database Management System**), We have used:

1. **MongoDB:** The reason behind this is that *MongoDB* is faster than *MySQL* due to its ability to handle large amounts of unstructured data when it comes to speed. This makes it perfect for storing the diverse user details and property information required to run *Roomble.*

Frameworks

Build System: We have used **NPM** along with **Node.js** due to the following benefits:

1. **Largest Ecosystem of Open-Source Libraries:** Node.js comes with a built-in package manager called **NPM (Node Package Manager)**. *NPM* is the world's largest package registry, allowing developers to access and integrate a vast collection of open-source libraries effortlessly.

2. **Efficient Dependency Management:** *NPM* allows developers to easily install, manage, and share reusable code packages, making adding functionality to *Node.js* applications quick and convenient.

3. **Code Reusability & Modularity** – With *NPM*, developers can easily reuse existing packages and modules, reducing redundancy and speeding up the development process.

Node.js: We have used *Node.js* as the backend framework of our web application. This is due to its numerous advantages:

1. **High Performance & Speed** – Node.js is built on Google Chrome's V8 engine, a high-performance C++-based JavaScript engine, ensuring fast execution and runtime efficiency.

2. **Asynchronous & Non-Blocking I/O** – Its event-driven, non-blocking architecture allows for efficient handling of multiple operations simultaneously, making it ideal for real-time, high-throughput applications like web servers and chat applications.

3. **Cross-Platform & Scalable** – Node.js runs seamlessly on various operating systems (Windows, macOS, Linux) and integrates effortlessly with cloud platforms, enabling easy deployment and scaling of applications.

4. **Industry Adoption & Community Support** – Leading companies like Netflix, PayPal, and LinkedIn rely on Node.js for its efficiency, and its open-source nature ensures strong community support, making development easier and more resourceful.

Vite.js: We chose *Vite.js* with the *React.js* template for our project due to its exceptional speed, efficiency, and developer-friendly features:

1. **Lightning-Fast Development** – Vite.js offers instant hot module replacement (HMR) and optimized builds, significantly improving development speed and productivity.

2. **Optimized Performance** – Its modern build system ensures faster page loads and efficient bundling, making the application highly responsive.

3. **Seamless Integration with Modern Tooling** – Vite.js is designed to work smoothly with modern JavaScript frameworks, ensuring a streamlined development experience.

4. **Simplified Development Workflow** – It provides an intuitive setup, minimal configuration, and built-in support for ES modules, reducing complexity in the development process.

5. **Cross-Device Consistency** – Ensures uniform development experience across different devices, reducing compatibility issues and enhancing collaboration.

Libraries

We have used the **Express.js** and **React** libraries for our project due to the following benefits:

Express.js: To route our APIs, we used the *Express.js* library. The advantages of using Express.js are:

1. **Minimalistic & Fast** – Express.js is a lightweight framework built on Node.js, enabling high-performance web applications and APIs with minimal overhead.

2. **Powerful Routing Mechanism** – It provides an intuitive way to efficiently define and manage routes for handling various HTTP methods (GET, POST, PUT, DELETE).

3. **Middleware Support** – Express allows modular request handling using middleware, enabling tasks like authentication, logging, and request parsing in a structured manner.

4. **Asynchronous & Scalable** – Leveraging Node.js's non-blocking architecture, Express efficiently manages concurrent requests, making it ideal for scalable applications.

5. **Robust Error Handling** – Built-in middleware support simplifies error management, making debugging and maintenance more efficient.

React.js: React.js is a popular JavaScript library developed by Facebook for building user interfaces, particularly for web applications. We used React.js for building the frontend of our application due to its flexibility, modularity, and ease of use:

1. **Component-Based Architecture** –.React.js allows for building reusable UI components, making development more organized and maintainable.

2. **Declarative Syntax** – React's declarative approach simplifies interactive UI design, making it easier to read, debug, and update.

3. **Unidirectional Data Flow** – React ensures a clear and predictable data flow throughout the application by passing data from parent components to child components through props, improving maintainability.

4. **Virtual DOM:** React.js utilizes a virtual DOM to improve performance instead of directly manipulating the browser's DOM. React.js works with a lightweight representation of the DOM called the virtual DOM. React then compares the virtual DOM with the actual DOM and only updates the parts that have changed, resulting in faster rendering.

Authentication

Roomble implements a secure authentication system with these key components:

1. **Password Protection:** Uses `bcrypt` for hashing and salting passwords before storing them in MongoDB, preventing exposure in data breaches.

2. **JWT Authentication:** Generates JSON Web Tokens upon successful login that contain encrypted user information and expire after a set period (5 hours).

3. **Email Verification**: Employs `nodemailer` to send one-time passwords (OTPs) to verify user email addresses during registration and password reset.

4. **Middleware Security**: Implements the `checkUser` middleware that validates tokens in request headers before allowing access to protected routes.

5. **Dual User Types**: Maintains separate authentication processes for tenants and landlords while using a unified middleware system.

This multi-layered approach ensures that user credentials remain secure while providing convenient account management features.

In conclusion, Roomble leverages a well-orchestrated blend of technologies in the MERN stack, each meticulously chosen to optimize performance and streamline the property and flatmate matching process. JavaScript, acting as both foundation (Node.js) and interactive layer (React), provides flexibility across the entire application stack, while MongoDB's schemaless architecture perfectly handles diverse information from user profiles and property listings to reviews and conversations, with the embedded review system demonstrating how its document model optimizes query performance while maintaining data integrity. The power of Express.js for robust routing and middleware implementation, combined with Socket.io for real-time messaging and JWT authentication for secure user sessions, manages the complexities of tenant-landlord interactions, ultimately empowering Roomble to deliver a robust, scalable, and user-centric platform where users can find their ideal living place.

2 Codebase

GitHub Repository: Roomble

Code Structure:

The above link takes you to the GitHub repository of Roomble, which contains all the source code for our web application. The project repository is mainly composed of 3 main parts:

- Frontend
- Backend
- Documents

Overall Structure of the repository for Roomble

|-- Roomble/
|-- backend/ // Backend Directory
|-- frontend/ // Frontend Directory
|-- documents/ // Project Documents
|-- .gitignore // Git ignore configuration
|-- package-lock.json // Locked npm dependencies
|-- package.json // Project metadata and dependencies

Frontend

After opening the frontend folder, you can see the subsequent directory structure. It has mainly the following major parts:

- **public**: Contains static assets like images, logos, and icons used throughout the application.
- **index.html**: The root HTML document that serves as the entry point for the React application
- **src**: Contains all source code for the application, organized into:
  - **components**: React components (`.jsx` files) are organized by feature (Authentication, property management, messaging etc.).
  - **css**: Styling files organized to match their corresponding components.
  - **context**: State management using React Context API (`Basecontext.js`, `Basestate.jax`)
  - **Configuration files**: Socket setup, config.json for environment variables.
  - **main.jsx**: The application's entry point that renders the Web component

The project follows a component-based architecture with careful separation of concerns. Components are logically grouped by functionality (e.g., TenantProfilePage, LandlordDashboard) with matching CSS files. The application uses React Router for navigation, with routes defined in App.jsx. A consistent colour scheme is maintained throughout the UI for a cohesive look and feel. The frontend communicates with the backend through RESTful API calls and real-time updates via Socket.io. Build and development workflows are managed using Vite, providing fast Hot Module Replacement during development and optimized production builds.

Frontend Structure

```
|-- Frontend/
   |-- public/            // Static assets accessible directly
      |-- delete-icon.png // UI element for deletion actions
      |-- home.jpg        // Homepage background
      |-- key.png         // Icon for authentication screens
      |-- house.jpg       // Icon for house for authentication screens
      |-- key_white.png   // Alternate key icon for dark themes
      |-- logo.png        // Main Roomble logo
      |-- logo_nav.png    // Navigation bar logo variant
      |-- property-img.png // Default property placeholder image
      |-- sampleUser_Img.png // Default user avatar
      |-- 1111111.jpg     // Sample property
   |-- src/               // Application source code
      |-- components/     // UI components organized by feature
         |-- AddPropertyComponents/     // Property listing creation
            |-- DragAndDrop.jsx         // Image upload component
         |-- animations/                // Reusable animation components
            |-- FadeInAnimation.jsx     // Fade-in effect wrapper
         |-- FindFlatmateComponents/    // Flatmate search functionality
            |-- FindFlatmate.jsx        // Main flatmate search page
            |-- SearchFlatmatesFilter.jsx // Filter controls
         |-- FindPropertyComponents/    // Property search functionality
            |-- PropertyCardTenant.jsx // Property card for tenants
            |-- SeachArea.jsx          // Location search component
         |-- ForgotPassword/            // Password recovery flow
            |-- ForgotPassword.jsx      // Email entry screen
            |-- OTPPageForgot.jsx       // OTP verification screen
            |-- SetNewPassword.jsx      // New password creation
         |-- LandlordDashboard/         // Landlord user dashboard
            |-- HomePage.jsx           // Main dashboard view
            |-- PropertyCard.jsx       // Property display card
         |-- LandlordProfile/           // Landlord profile management
            |-- LandlordEditProfile.jsx // Profile editing
            |-- LandlordProfile.jsx     // Profile display
         |-- MessageComponents/         // Chat functionality
            |-- ChatBox.jsx            // Main chat container
            |-- ChatBoxEmpty.jsx       // Empty state for chat
            |-- MessageBox.jsx         // Message input box
            |-- MessageCard.jsx        // User conversation card
            |-- OwnMessage.jsx         // User's sent messages
            |-- RecievedMessage.jsx    // Incoming messages
            |-- SampleMessages.jsx     // Placeholder messages
         |-- OTPPage/                   // OTP verification screens
            |-- OTPDeletePage.jsx       // Deletion confirmation
            |-- OTPPageLandlord.jsx     // Landlord registration verification
            |-- OTPPageTenant.jsx       // Tenant registration verification
         |-- TenantProfilePage/         // Tenant profile management
            |-- TenantEditPage.jsx      // Profile editing
```

```
    |-- TenantProfilePage.jsx  // Profile display
  |-- AddProperty.jsx          // Property listing creation
  |-- BookmarkedFlatmates.jsx  // Saved flatmate profiles
  |-- BookmarkedProperties.jsx // Saved property listings
  |-- EditProperty.jsx         // Property listing editing
  |-- FindProperty.jsx         // Property search main page
  |-- FlatmateCard.jsx         // Tenant profile card
  |-- FlatmateCardExpand.jsx   // Detailed tenant profile view
  |-- Home.jsx                 // Main landing page
  |-- Login.jsx                // User authentication
  |-- MessageStart.jsx         // Initial messaging screen
  |-- Messages.jsx             // Message center
  |-- Navbar.jsx               // Site navigation
  |-- OtherLandlord.jsx        // View other landlord profiles
  |-- PropertyDisplay.jsx      // Detailed property view
  |-- Review.jsx               // User/property review component
  |-- SignUpForm.jsx           // Generic signup form
  |-- SignUpTenant.jsx         // Tenant registration
  |-- SignupformLandlord.jsx   // Landlord registration form
  |-- SignupLandlord.jsx       // Landlord registration page
|-- context/                   // React Context API state management
  |-- base/                    // Application-wide state
    |-- Basecontext.js         // Context definition
    |-- Basestate.jsx          // State provider
|-- css/                       // Styling organized by component
  |-- AddPropertyStyles/       // Property creation styles
    |-- AddProperty.css
    |-- DragAndDrop.css
  |-- FindPropertyStyles/      // Property search styles
    |-- FindProperty.css
    |-- SearchArea.css
  |-- LandlordProfileStyles/   // Landlord profile styles
    |-- LandlordProfile.css
  |-- MessageBoxStyle/         // Messaging UI styles
    |-- ChatBox.css
    |-- MessageBox.css
    |-- Messages.css
  |-- OTPPage/                 // OTP screen styles
    |-- OTPDeletePage.css
    |-- OTPPageLandlord.css
    |-- OTPPageTenant.css
  |-- TenantProfilePageStyles/ // Tenant profile styles
    |-- TenantEditPage.css
    |-- TenantProfilePage.css
  |-- BookmarkedFlatmates.css  // Saved flatmates list styling
  |-- BookmarkedProperties.css // Saved properties list styling
  |-- FindFlatmate.css         // Flatmate search styling
  |-- FlatmateCard.css         // Tenant card styling
  |-- FlatmateCardExpand.css   // Expanded tenant card styling
  |-- Home.css                 // Homepage styling
```

```
        |-- LandlordDashboard.css      // Landlord dashboard styling
        |-- Login.css                  // Authentication screen styling
        |-- Navbar.css                 // Navigation styling
        |-- PropertyCard.css           // Property card styling
        |-- PropertyDisplay.css        // Detailed property view styling
        |-- SignUpTenant.css           // Tenant registration styling
        |-- SignupLandlord.css         // Landlord registration styling
        |-- temp.css                   // Temporary/experimental styles
    |-- App.css                        // Global application styles
    |-- App.jsx                        // Main application component/router
    |-- config.json                    // Frontend configuration variables
    |-- index.css                      // Root CSS styles
    |-- main.jsx                       // Application entry point
    |-- socket.js                      // Socket.io client setup
    |-- useDidMountEffect.jsx          // Custom React hook
|-- .gitignore                         // Git ignore configuration
|-- eslint.config.js                   // ESLint configuration
|-- index.html                         // HTML entry point
|-- package-lock.json                  // Locked npm dependencies
|-- package.json                       // Project metadata and dependencies
|-- README.md                          // Project documentation
|-- vite.config.js                     // Vite build configuration
```

Backend

The backend folder structure reveals the following major components:

- **index.js**: The main server entry point that initializes Express, connects routes, and sets up middleware
- **mongodb.js**: Handles database connection to MongoDB using Mongoose ODM
- **models**: Contains schema definitions for all data entities (Tenant, Landlord, Property, Review, etc.)
- **routes**: API endpoints organized by functionality (authentication, property management, messaging)
- **middlewares**: Authentication and request processing utilities (e.g., `checkuser.js` for JWT verification)
- **controllers**: Business logic separated from routes for cleaner code organization
- **Helper Functions**: Utility services like email sending (`mailSender.js`) and file handling.

The backend follows the **MVC** (**Model-View-Controller**) architectural pattern with RESTful API design principles. Authentication is implemented using **JWT** (**JSON Web Tokens**) with `bcrypt` for password hashing. Real-time communication for the messaging feature is handled through Socket.io integration.

Environment variables in the `.env` file configure sensitive information like database connections and API keys. The server provides comprehensive error handling with consistent response formats, success flags and meaningful error messages. File upload functionality is managed through `express-fileupload` middleware with static file serving for property images and user avatars.

The application clearly separates concerns with models defining data structure, routes handling API endpoints, middlewares for cross-cutting concerns, and controllers implementing business logic.

Backend Structure

```
|-- Backend/
   |-- .env            // Environment variables (MongoDB URI, JWT secret, etc.)
   |-- index.js        // Main server entry point
   |-- mongodb.js      // Database connection setup
   |-- package.json    // Dependencies and scripts
   |-- seed.js         // Database seeding script
   |-- controllers/
      |-- reviewcontroller.js
   |-- helper_funcs/
      |-- mailSender.js      // Email functionality
      |-- Saveimage.js       // Image upload handling
   |-- middlewares/
      |-- checkuser.js       // JWT authentication middleware
   |-- models/
      |-- Conversation.js    // Chat conversations schema
      |-- Landlord.js        // Landlord user schema
      |-- OTP_models.js      // OTP verification schemas
      |-- Property.js        // Property listing schema
      |-- Review.js          // User reviews schema
      |-- Tenant.js          // Tenant user schema
      |-- Towns.js           // Location data schema
   |-- Pictures/
      |-- Default.png        // Default profile image
      |-- ... (Uploaded images)
   |-- routes/
      |-- addProperty.js     // Property creation routes
      |-- Bookmark.js        // Bookmark management
      |-- changePassword.js  // Password update routes
      |-- deleteProfile.js   // Account deletion routes
      |-- deleteProperty.js  // Property deletion routes
      |-- ForgotPassword.js  // Password recovery
      |-- Landlord_auth.js   // Landlord authentication
      |-- list_delist_prop.js // Property visibility toggling
      |-- message.js         // Messaging functionality
      |-- reviewProperty.js  // Property review routes
      |-- reviewroutes.js    // User review routes
      |-- Searching_Routes.js // Search functionality
      |-- Tenant_auth.js     // Tenant authentication
      |-- update.js          // Profile update routes
      |-- view_profiles.js   // Profile viewing routes
      |-- viewProperty.js    // Property viewing routes
```

Database

The databases section contains various databases and collections of the form:

Database Structure

|-- Database/
|-- admin/
|-- local/
|-- test/
|-- conversations/ // Conversations between users in the form of messages
|-- landlord_otps/
|-- landlords/ // Landlord Users
|-- properties/ // Properties owned by Landlords
|-- tenant_otps/
|-- tenants/ // Tenant Users
|-- towns/ // Available localities and precomputed distances between the localities

Architecture

Our Roomble application implements the **Model-View-Controller (MVC)** architectural pattern, providing a clean separation of concerns across the codebase.

Architectural Components

Model: MongoDB schemas in the 'backend/models' directory define our data structure:
- User models (Tenant, Landlord) with preferences and authentication information
- Property model with comprehensive listing details
- Support models for Reviews, Conversations, and location data

View: Frontend React components that render the user interface:
- Feature-based organization with dedicated component folders
- Responsive layouts adapting to different screen sizes
- Form components for data entry and validation

Controller: Logic that processes user actions and updates models:
- Routes that map API endpoints to specific functionality
- Authentication handling through JWT verification
- Data transformation and business rule implementation

Key Technical Implementations of the Architecture

- **Data Optimization**: Embedded reviews in user documents for faster retrieval.
- **Security Layer**: Middleware that verifies user identity before allowing protected operations.
- **State Management**: Context API maintains the application state across components.
- **Real-time Features**: Socket.io implementation for instant messaging.

This architecture enables the core functionality of matching tenants with suitable properties or flatmates while maintaining a modular and maintainable codebase.

Other Important Files

- `index.js` (**Backend**): Located in the '`backend`' directory, this is the main server entry point. It initializes Express, sets up middleware, configures Socket.io for real-time messaging, and connects all route handlers. It also establishes the MongoDB connection and starts the server listening on the configured port.

- `main.jsx` and `App.jsx`: Located in the '`src`' folder within the '`frontend`' directory. '`main.jsx`' serves as the application entry point, rendering the App component wrapped in React Router. '`App.jsx`' defines all routes, implements conditional navbar rendering, and wraps the entire application in the BaseState context provider for global state management.

- `Basestate.jsx` and `Basecontext.js`: Found in the '`src/context/base`' directory. These files implement the Context API for global state management, providing user authentication state, profile data, and helper functions that are accessible throughout the application.

- `middlewares`: The '`checkuser.js`' file in the '`backend/middlewares`' directory verifies JWT tokens in request headers, identifies the user type (tenant or landlord), and attaches the user object to the request for use in protected routes.

- `routes`: This folder in the '`backend`' directory contains files that define API endpoints for different features. Each route file (e.g., '`Tenant_auth.js`', '`message.js`', '`Searching_Routes.js`') handles specific functionality and often uses the middleware for authentication. Routes connect client requests to the appropriate controller logic.

- `models`: Located in the '`backend/models`' directory, these files define MongoDB schemas using Mongoose. Key models include `Tenant.js`, `Landlord.js`, `Property.js`, and `Conversation.js`, which define data structure, validation rules, and relationships.

- `Socket.js`: In the '`frontend/src`' directory, this file establishes the Socket.io client connection for real-time messaging, enabling instant chat updates between users.

API Endpoints

The following is a summary of all the possible API calls in the application:

1. Authentication

a. Tenant Authentication

i. **Register Tenant**

URL: `/api/Tenant/auth/Tenant_register`

Method: `POST`

Data: `{ name, email, password, locality, city, gender, smoke, veg, pets, flatmate }`

Status: {

```
If successful: {
          200 Success (returns user ID for OTP verification)
             }
else: {
         400 User already exists,
         500 Server error
    }
}
```

ii. **Verify Tenant OTP**

URL: `/api/Tenant/auth/verifyTenant/:id`

Method: `POST`

Data: `{ Entered_OTP }`

Status: {

```
If successful: {
          201 User successfully registered
             }
else: {
         400 Invalid OTP,
```

```
            404 Session Expired,
            500 Server error
        }
    }
```

iii. **Tenant Login**

URL: `/api/Tenant/auth/Tenant_login`

Method: `POST`

Data: `{ email, password }`

Status: `{`

```
    If successful: {
             200 Login successful (returns JWT token)
                }
    else: {
            401 Wrong password,
            404 User not found,
            500 Server error
        }
    }
```

iv. **Get Tenant by ID**

URL: `/api/Tenant/auth/getuser`

Method: `POST`

Data: `{ id }`

Status: `{`

```
If successful: {
          200 Success (returns tenant data)
             }
else: {
         200 with success: false
    }
}
```

b. Landlord Authentication

i. **Register Landlord**

URL: `/api/Landlord/auth/Landlord_register`

Method: `POST`

Data: `{ name, email, password }`

Status: {

```
If successful: {
          400 Success (returns user ID for OTP verification)
             }
else: {
         400 User already exists,
         500 Server error
    }
}
```

ii. **Verify Landlord OTP**

URL: `/api/Landlord/auth/verifyLandlord/:id`

Method: POST

Data: { Entered_OTP }

Status: {

```
If successful: {
          201 User successfully registered
             }
else: {
          400 Invalid OTP,
          404 Session Expired,
          500 Server error
     }
}
```

iii. **Landlord Login**

URL: /api/Landlord/auth/Landlord_login

Method: POST

Data: { email, password }

Status: {

```
If successful: {
          200 Login successful (returns JWT token)
             }
else: {
          401 Wrong password,
          404 User not found,
```

```
            500 Server error
        }
    }
```

2. Password Management

a. Forgot Password

i. **Request Password Reset**

URL: `/api/forgotPassword/enteremail`

Method: `POST`

Data: `{ email, accounttype }`

Status: `{`

```
    If successful: {
             200 OTP sent (returns JWT token)
                }
    else: {
            400 Bad account type,
            401 user not found,
            500 Server error
        }
    }
```

ii. **Verify OTP for Password Reset**

URL: `/api/forgotPassword/enterOTP`

Method: `POST`

Headers: `{ authtoken }`

Data: { Entered_OTP, accounttype }

Status: {

```
If successful: {
          200 OTP verified
             }
else: {
         401 Wrong OTP,
         404 OTP expired,
         500 Server error
    }
}
```

iii. **Set new Password**

URL: /api/forgotPassword/ForgotPassword

Method: POST

Headers: { authtoken }

Data: { newPassword, accounttype }

Status: {

```
If successful: {
          200 Password updated
             }
else: {
         401 Not authorized,
         404 OTP expired,
```

```
            500 Server error
        }
    }
```

b. Change Password

i. Request Password Change

URL: `/api/changePassword/enteremail`

Method: `POST`

Data: `{ email, accounttype }`

Status: `{`

```
    If successful: {
             200 OTP sent (returns JWT token)
                }
    else: {
            400 Bad account type,
            401 user not found,
            500 Server error
        }
    }
```

ii. Verify OTP for Password Change

URL: `/api/changePassword/enterOTP`

Method: `POST`

Headers: `{ authtoken }`

Data: `{ Entered_OTP, accounttype }`

Status: {

```
If successful: {
          200 OTP verified
             }
else: {
         401 Wrong OTP,
         404 OTP expired,
         500 Server error
    }
}
```

iii. **Change Password**

URL: /api/changePassword/ChangePassword

Method: POST

Headers: { authtoken }

Data: { oldPassword, newPassword, accounttype }

Status: {

```
If successful: {
          200 Password updated
             }
else: {
         400 Incorrect old password,
         401 Not authorized,
         404 OTP expired,
```

```
            500 Server error
        }
    }
```

3. User Profile Management

a. View Profiles

i. **Get Own Profile**

URL: `/api/view_profiles/Self_profile`

Method: `POST`

Header: `{ authtoken, accounttype }`

Status: `{`

```
    If successful: {
            200 Success (returns user data)
            }
    else: {
            404 User not found,
            500 Server error
        }
    }
```

ii. **View Other User Profile**

URL: `/api/view_profiles/other_users`

Method: `POST`

Data: `{requested_id, accounttype }`

Status: `{`

```
    If successful: {
              200 Success (returns user data)
                 }
    else: {
             400 Invalid account type,
             404 User not found,
             500 Server error
        }
}
```

iii. **Get User by ID**

URL: `/api/view_profiles/user`

Method: `POST`

Data: `{ id }`

Status: `{`

```
    If successful: {
              200 Success (returns user data)
                 }
    else: {
             400 Invalid account type,
             404 User not found,
             500 Server error
        }
    }
```

iv. **Get Current User**

URL: `/api/auth/user`

Method: `POST`

Headers: `{ authtoken }`

Status: `{`

```
If successful: {
          200 Success (returns user data)
             }
else: {
         400 Invalid token,
         500 Server error
    }
}
```

b. Update Profile

i. **Update User Profile**

URL: `/api/updates/updateProfile`

Method: `POST`

Header: `{ authtoken, accounttype }`

Data: `FormData with updated fields and optional image`

Status: `{`

```
If successful: {
          200 Profile updated
             }
```

```
else: {
          400 Invalid data/image,
          404 User not found,
          500 Server error
      }
}
```

c. Delete Acoount

i. **Initiate Account Deletion**

URL: `/api/Deleting_routes/deleteInitiate`

Method: `POST`

Data: `{ email, accounttype }`

Status: {

```
If successful: {
           200 OTP sent (returns JWT token)
              }
else: {
          400 Bad account type,
          404 User not found,
          500 Server error
      }
}
```

ii. **Confirm Account Deletion**

URL: `/api/Deleting_routes/enterOTPtoDelete`

Method: POST

Headers: { authtoken }

Data: { Entered_OTP, accounttype }

Status: {

```
If successful: {
          200 Account deleted
             }
else: {
         401 Wrong OTP,
         404 User not found/OTP expired,
         500 Server error
    }
}
```

4. Property Management

a. Property CRUD

i. **Add Property**

URL: /api/listproperty/listProperty

Method: POST

Header: { authtoken, accounttype }

Data: FormData with property details and images

Status: {

```
If successful: {
          201 Property added
```

```
                }
    else: {
            400 Invalid data/missing fields,
            404 Landlord not found,
            500 Server error
        }
    }
```

ii. **Get Property**

URL: `/api/property/get_property`

Method: `POST`

Data: `{ id }`

Status: `{`

```
    If successful: {
             200 Success (returns property data)
                }
    else: {
            404 Property not found,
            500 Server error
        }
    }
```

iii. **Update Property**

URL: `/api/updates/updateProperty`

Method: `POST`

Headers: { authtoken }

Data: FormData with updated property fields and id

Status: {

```
If successful: {
          200 Property updated
             }
else: {
         400 Invalid data/missing fields,
         404 Property not found,
         500 Server error
    }
}
```

iv. **Delete Property**

URL: /api/deleteproperty/deleteProperty/:propertyId

Method: DELETE

Headers: { authtoken }

Status: {

```
If successful: {
          200 Property deleted
             }
else: {
         400 Invalid ID,
         403 Not Authorized,
```

```
            404 Property not found,
            500 Server error
        }
    }
```

v. **List/Delist Property**

URL: `/api/Listing_Delisting/List_Delist_Prop`

Method: `POST`

Headers: `{ authtoken }`

Data: `{ property_id, action } (action: "enlist" or "delist")`

Status: `{`

```
    If successful: {
            200 Property status updated
            }
    else: {
            400 Invalid action/ID,
            401 Not Authorized,
            500 Server error
        }
    }
```

5. Search and Recommendations

a. Search Flatmates

URL: `/api/Search_Routes/SearchFlatmates`

Method: `GET`

Header: { authtoken }

Query Params: locality, gender, smoke, veg, pets

Status: {

If successful: {

200 Success (returns matching tenants)

}

else: {

400 Invalid parameters,

500 Server error

}

}

b. Search Properties

URL: /api/Search_Routes/SearchProperties

Method: GET

Query Params: town, min_price, max_price, min_area, max_area, bhk, ...filters

Status: {

If successful: {

200 Success (returns properties)

}

else: {

400 Town is required/Invalid parameters,

404 Town not found,

```
            500 Server error
        }
    }
```

6. Bookmarks

a. Get Bookmarks

URL: `/api/BookMarking_Routes/get_bookmarks`

Method: `GET`

Header: `{ authtoken }`

Status: `{`

```
    If successful: {
             200 Success (returns bookmarked flatmates and
properties)
                }
    else: {
            404 User not found,
            500 Server error
        }
    }
```

b. Edit Bookmarks

URL: `/api/BookMarking_Routes/edit_bookmarks`

Method: `POST`

Header: `{ authtoken }`

Data: { action, thing, id } (action: "bookmark" or "unmark", thing: "flatmate" or "property")

Status: {

If successful: {

200 Bookmark added/removed

}

else: {

400 Invalid parameters,

404 Item not found,

500 Server error

}

}

c. Check Bookmark Status

URL: /api/BookMarking_Routes/check_bookmark

Method: POST

Header: { authtoken }

Data: { thing, id }

Status: {

If successful: {

200 Success (returns bookmark status)

}

else: {

404 User not found,

```
            500 Server error
        }
    }
```

7. Reviews

a. User Reviews

i. **Create Review**

URL: `/api/reviews`

Method: `POST`

Header: `{ authtoken }`

Data: `{ reviewee, rating, comment }`

Status: {

```
    If successful: {
            200 Review added
            }
    else: {
            400 Invalid data or already Reviewed,
            404 User not found,
            500 Server error
        }
    }
```

ii. **Get Reviews for User**

URL: `/api/reviews/reviewee`

Method: `POST`

Header: { authtoken }

Data: { reviewee }

Status: {

```
If successful: {
          200 Success (returns reviews)
             }
else: {
         404 User not found,
         500 Server error
    }
}
```

b. Property Reviews

i. **Create Property Review**

URL: /api/reviewProperty/addreview

Method: POST

Header: { authtoken }

Data: { propertyId, rating, review }

Status: {

```
If successful: {
          200 Review added
             }
else: {
         400 Already Reviewed,
```

```
            404 Property not found,
            500 Server error
        }
    }
```

ii. **Get Property Reviews**

URL: `/api/reviewProperty/getreviews`

Method: `POST`

Data: `{ propertyId }`

Status: `{`

```
    If successful: {
             200 Success (returns reviews)
                }
    else: {
            404 Property not found,
            500 Server error
        }
    }
```

8. Messaging

a. User Status

URL: `/messages/getUserNameStatus`

Method: `POST`

Data: `{ userID }`

Status: {

```
If successful: {
          200 Success (returns name and status)
             }
else: {
         200 with success: false,
         500 Server error
    }
}
```

b. Conversations

i. **Get All Conversations**

URL: /messages/getConversations

Method: POST

Headers: { authtoken }

Status: {

```
If successful: {
          200 Success (returns conversations)
             }
else: {
         200 with success: false,
         500 Server error
    }
}
```

ii. **Get Specific Conversations**

URL: `/messages/getConversation`

Method: `POST`

Data: `{ conversation_id }`

Status: {

```
If successful: {
          200 Success (returns conversation)
             }
else: {
         200 with success: false,
         500 Server error
    }
}
```

iii. **Send Message**

URL: `/messages/sendMessage`

Method: `POST`

Headers: `{ authtoken }`

Data: `{ conversation_id, message }`

Status: {

```
If successful: {
          200 Success (returns updated conversation)
             }
else: {
```

```
            200 with success: false,
            500 Server error
        }
    }
```

iv. **Create Conversation**

URL: `/messages/createConversation`

Method: `POST`

Headers: `{ authtoken }`

Data: `{ user2 }`

Status: `{`

```
    If successful: {
             200 Success (returns conversation_id)
                }
    else: {
            200 with success: false,
            500 Server error
        }
    }
```

3 Completeness

Implemented Features

The project includes a **fully functional rental and flatmate-finding platform** with authentication, property management, messaging, and real-time interactions. Below is the list of all implemented features:

1. User Authentication & Middleware

- JWT-based authentication for secure login and session management.
- Middleware (`checkuser.js`) to verify JWT tokens and authenticate API requests (`Searching_Routes.js`).
- Separate authentication for Landlords (`Landlord_auth.js`) and Tenants (`Tenant_auth.js`).
- Role-based access control:
  - Tenants and landlords have different permissions.
  - Unauthorized actions (e.g., a tenant trying to delete a property) are blocked.
- OTP verification via email for user registration and password recovery.
- Login/logout functionality for both landlords and tenants.

2. Forgot Password & Change Password Features

- Users can reset their passwords using an OTP-based email verification (`ForgotPassword.js`)and can securely change their passwords after OTP verification (`ChangePassword.js`).
- Old password verification is required before updating the new password.
- Bcrypt hashing is used to store passwords securely in all cases.

3. Profile Management

- Users can create, update (`updateProfile.js`), and delete (`deleteProfile.js`) their profiles, excluding email and password .
- Profile information for landlords includes:
  - Name
  - List of properties
- Profile information for tenants includes:
  - Name
  - Locality
  - Gender
  - Smoking habits
  - Vegetarian preferences
  - Pets
  - Flatmate preference
- View profiles of other users, including landlords and tenants (`view_profiles.js`).
- Users can retrieve their own profile details using JWT authentication (`getUser.js`).

4. Property Listings and Management

- Landlords can list properties with details like name, town, address, BHK, amenities, and pricing (`addProperty.js`).
- Landlords can delete properties they no longer want to list (`deleteProperty.js`).
- Landlords can enable or disable property availability (`lisst_delist_prop.js`).

5. Property Search

- Tenants can search for desired rental properties (`Searching_Routes.js`) by specifying city, locality, and other optional filters like price range, area (in sq. feet) range, and BHK.
- Properties that satisfy all the filters in the specified locality will be displayed first in the search results, followed by suitable properties in the nearest two other localities.

6. Flatmate Search

- Tenants can easily find compatible flatmates, thanks to our in-house novel recommendation algorithm (`Searching_Routes.js`)
- The algorithm predicts a compatibility score between the searching tenant and potential flatmates. We use a similarity function that combines -

  (i) **Locality Proximity**: More similar if desired localities are closer. Let d(l1,l2) represent the distances between localities. We use a function to map distances to similarity scores:

  sim(locality) = 1/(1 + d(l1,l2)),

  where a smaller distance gives a higher score.

  (ii) **Boolean Attribute Matching**: More similarity if preferences (gender, smoke, veg, pets) match. For each Boolean feature (gender, smoke, veg, pets):

  sim(bool) = matches / total boolean attributes

  The final recommendation score is given by -

  S = α · sim(locality) + (1−α) · sim(bool),

  where α is a weight factor (we have set 0.7 for locality and 0.3 for boolean preferences).

- The recommended flatmates are shown to the tenant in descending order with respect to their recommendation scores. Tenants can further filter this list based on locality and other preferences to suit their convenience.

7. Reviews and Ratings

- Users can leave reviews and ratings for landlords, properties, and flatmates (`reviewroutes.js`).
- Features include:
  - Creating a new review

- Fetching reviews for a specific user

8. Messaging System (Real-Time Communication)

- Users can send and receive messages in a chat system (`message.js`).
- Features include:
  - Viewing conversation history
  - Check user online/offline status
  - WebSockets for real-time messaging
  - Conversations are stored, retrieved, and updated in the database.

9. Email Notifications

- Automated email alerts for:
  - Password reset requests (`ForgotPassword.js`).
  - OTP verification for new user registrations (`Landlord_auth.js`, `Tenant_auth.js`).

10. Bookmark System

- Users can bookmark rental properties or potential flatmates (`Bookmark.js`).
- Features include:
  - Adding and removing property bookmarks.
  - Adding and removing flatmate bookmarks.
  - Retrieving all bookmarks for a user.
- Bookmarked properties and flatmates are stored in the user's profile for easy access.

11. Web UI

- The front-end user interface is designed to be both visually appealing and highly intuitive, with a primary focus on desktop users to ensure an optimal user experience.
- The application leverages **ReactJS** for dynamic and efficient component-based development, **HTML** and **CSS** for structured and responsive layouts, and **Material UI (MUI)** to deliver a sleek, modern design with a professional aesthetic.

12. Strong Data Validation & Error Handling

- Error messages and validation checks prevent invalid data from being stored.
- Example:
  - Invalid property IDs return a 400 Bad Request (`deleteProperty.js`).
  - Users trying to delete properties they don't own are blocked (`lisst_delist_prop.js`).
  - Invalid OTP attempts return error messages instead of allowing login (`Landlord_auth.js`).

13. Environment Variables for Security

- Sensitive data like JWT secret keys and database credentials are stored in `.env` files (`Landlord_auth.js`, `Tenant_auth.js`).
- Prevents security leaks by keeping API keys and database URLs private.

Future Development Plans

1. Expansion to multiple cities

Presently the software caters to properties and flatmates located in Mumbai. We plan to extend this to several metro cities across India where finding property and flatmates is a major issue.

2. Dynamic Distances from Map

Currently, the user can search for flatmates and properties in a limited number of localities, as we are storing pre-computed distances between the localities in the database (test/towns). In the future, we wish to make locality choice more flexible by reading the latitude and longitude directly off the map and computing the distances on the fly.

3. Secure Review System

We wish to make our two-way review system more secure by ensuring that only authorised tenants can review properties (that they have actually lived in), and only legitimate flatmates can review each other.

4. Dark Theme Support

We plan to add dark mode support to our website in the future to give it a sleek and professional look and enhance the browsing experience.

5. Interactive 3D Property Tours

Currently the software shows photos of listed properties, in the future we plan to include interactive 3D models of rental properties, allowing users to virtually explore spaces with realistic layouts and interiors. Users can rotate, zoom, and navigate through the model for a detailed view. The models will be mobile-friendly and may include AR/VR compatibility for immersive virtual tours.

Appendix A - Group Log

Date	Timings	Duration	Discussion & Work done
10 Feb	23:00 - 01:00	2 hrs	**Project Kickoff:** Discussed project scope, core features, and tech stack. Assigned initial tasks and divided work among team members.
17 Feb	22:00 - 01:00	3 hrs	**Project Setup:** Initialized Git repository, set up frontend (React + Vite) and backend (Node.js + Express + MongoDB). Defined folder structures and planned database schema.
24 Feb	20:00 - 23:00	3 hrs 30 min	**Backend Development Begins:** Configured MongoDB, created user authentication models, and implemented initial Express routes.
3 Mar	22:00 - 02:30	4 hrs 30 min	**Backend API Implementation:** Developed authentication APIs (JWT-based login/signup), implemented middleware for security, and tested endpoints using manual API calls in frontend.
10 Mar	20:00 - 01:30	5 hrs 30 min	**Frontend Development:** Designed UI wireframes, set up React components for login/signup, and implemented routing with React Router.
17 Mar	20:00 - 02:30	6 hrs 30 min	**Integration & Testing:** Connected frontend to backend APIs, built Axios API calls, tested authentication flow, and debugged CORS issues. using JWT.
26 Mar	22:00 - 01:00	3 hrs	**Debugging & Refinements:** Fixed UI inconsistencies, improved property listing management, and conducted preliminary functionality testing to identify and resolve issues.

28 Mar	10:00 - 12:00	2 hrs	**Completed Implementation Document:** Documented all development phases, including architecture, API routes, and integration details.