Full Stack Development with MERN

API Development and Integration Report

Date	11-July-2024
Team ID	SWTID1720075141
Project Name	Project – House Hunt
Maximum Marks	6 Marks

Project Title: Rent Ease - House Hunt
Date: 11-July-2024
Prepared by: E Gayathri & A Mani Ratna

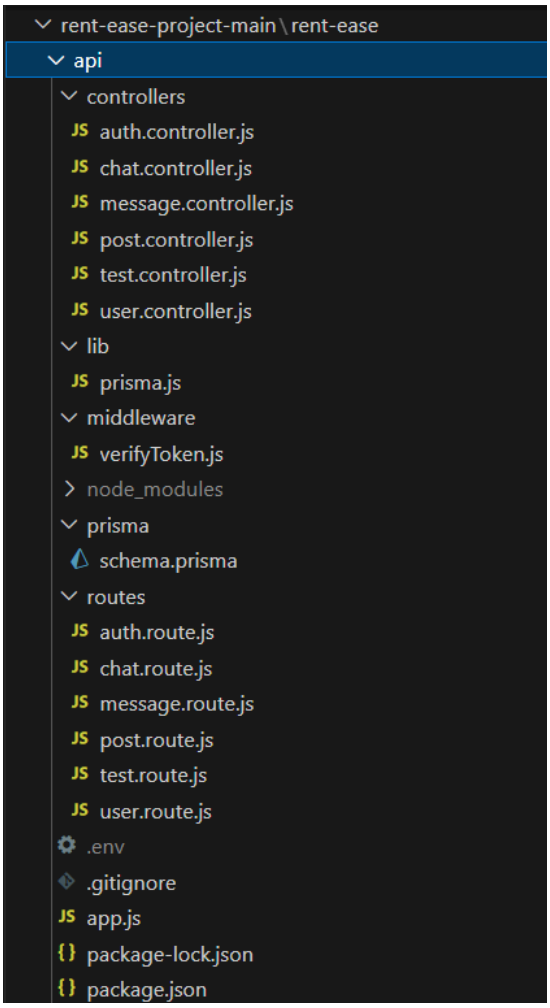
Objective

The objective of this report is to document the API development progress and key aspects of the backend services implementation for the “House Hunt - Rent Ease” project.

Technologies Used

- **Backend Framework:** Node.js with Express.js
- **Database:** MongoDB (Prisma ORM)
- **Authentication:** JWT

Project Structure :



Key Directories and Files

1. /controllers

Contains functions to handle requests and responses.

- **auth.controller.js**: Manages user authentication tasks such as registering, logging in, and logging out.
- **chat.controller.js**: Handles chat-related functionalities including retrieving chats, getting a specific chat, adding a chat, and marking chats as read.
- **message.controller.js**: Manages messages within chats, including adding new messages to a specific chat.

- **post.controller.js**: Manages CRUD operations for posts, such as adding, updating, retrieving, and deleting posts.
- **user.controller.js**: Handles user-related tasks such as updating user information, deleting users, saving posts, retrieving profile posts, and getting notification numbers.

2. /models

Includes Mongoose schemas and models for MongoDB collections.

- **User.js**: Schema and model for user data, including fields for username, email, password, etc.
- **Post.js**: Schema and model for posts, detailing attributes like title, description, location, and more.
- **Chat.js**: Schema and model for chat sessions, including participants and messages.
- **Message.js**: Schema and model for individual messages within chats.

3. /routes

Defines the API endpoints and links them to controller functions.

- **auth.route.js**: Endpoints for user authentication (/register, /login, /logout).
- **chat.route.js**: Endpoints for chat operations (/, /:id, /read/:id).
- **message.route.js**: Endpoints for managing messages (/:chatId).
- **post.route.js**: Endpoints for post operations (/, /:id).
- **user.route.js**: Endpoints for user management (/, /profilePosts, /notification).

4. /middlewares

Custom middleware functions for request processing.

- **verifyToken.js**: Middleware to verify the JWT token for secure access to API endpoints.

5. env

Configuration file for database connection URL, frontend server and JWT secret key.

API Endpoints

A summary of the main API endpoints and their purposes:

User Authentication

- **POST /api/auth/register** - Registers a new user.
- **POST /api/auth/login** - Authenticates a user and returns a token.
- **POST /api/auth/logout** - Logs out the authenticated user.

User Management

- **GET /api/users** - Retrieves all users.
- **PUT /api/users/**- Updates user information by ID.
- **DELETE /api/users/**- Deletes a user by ID.
- **POST /api/users/save** - Saves a post for the user.
- **GET /api/users/profilePosts** - Retrieves posts made by the user.
- **GET /api/users/notification** - Retrieves the number of notifications for the user.

Chat Management

- **GET /api/chats** - Retrieves all chats for the authenticated user.
- **GET /api/chats/**- Retrieves a specific chat by ID.
- **POST /api/chats** - Adds a new chat.
- **PUT /api/chats/read/**- Marks a chat as read by ID.

Message Management

- **POST /api/messages/**- Adds a new message to a specific chat.

Post Management

- **GET /api/posts** - Retrieves all posts.
- **GET /api/posts/**- Retrieves a specific post by ID.
- **POST /api/posts** - Adds a new post.

- **PUT /api/posts/-** Updates a post by ID.
- **DELETE /api/posts/-** Deletes a post by ID.

Integration with Frontend

The backend communicates with the frontend via RESTful APIs. Key points of integration include:

- **User Authentication**: Tokens are passed between the frontend and backend to handle authentication. Upon successful login, the backend issues a token, which the frontend stores and includes in the headers of subsequent requests to access protected routes.
- **Data Fetching**: Frontend components make API calls to fetch necessary data for display and interaction. For example:
    - To display user information, the frontend sends a GET request to /api/users/:id.
    - To list all available posts, the frontend sends a GET request to /api/posts.
    - For messaging functionality, the frontend interacts with /api/chats and /api/messages to retrieve and send chat messages.
- **Posting Data**: Users can submit forms on the frontend, which send POST requests to the backend to add new data. Examples include:
    - Creating a new post by sending a POST request to /api/posts.
    - Adding a new chat message by sending a POST request to /api/messages/:chatId.
- **Updating Data**: The frontend can send PUT requests to update existing data. Examples include:
    - Updating user information by sending a PUT request to /api/users/:id.
    - Marking a chat as read by sending a PUT request to /api/chats/read/:id.
- **Deleting Data**: The frontend can send DELETE requests to remove data. An example includes:
    - Deleting a post by sending a DELETE request to /api/posts/:id.

- **Real-Time Updates**: While not explicitly mentioned in the provided code, the integration can be enhanced by using WebSockets (e.g., Socket.IO) for real-time updates, such as receiving new chat messages instantly.
- **File Uploads**: For uploading images and other files, the frontend can send POST requests to appropriate endpoints, possibly integrating with a service like AWS S3 for storage.

Error Handling and Validation

Error Handling:

- **Centralized Error Handling:** The project implements centralized error handling using middleware functions. These middleware functions are designed to intercept errors and provide consistent error responses across the application. For example, when an error occurs during request processing or database operations, the middleware catches the error and formats it into a standardized response format before sending it back to the client. This approach ensures that error messages are clear, uniform, and helpful for debugging purposes.

Validation:

- **Input Validation:** Input validation is crucial for ensuring data integrity and security. In this project, input validation is implemented using the express-validator library. This library allows defining validation rules and middleware functions that automatically validate incoming request data against these rules. For instance, before processing a request to create or update user data (POST /api/user/register, PUT /api/user/:id), the backend validates input fields such as email format, password strength, and required fields. If any validation rule fails, appropriate error responses are sent back to the client, preventing invalid data from entering the system.

Security Considerations

Authentication:

- **Secure Token-Based Authentication:** The project employs secure token-based authentication using JSON Web Tokens (JWT). When a user successfully logs in (POST /api/user/login), the backend issues a JWT containing user credentials. Subsequent requests from the authenticated user include this token in the request headers (Authorization: Bearer <token>) to access protected routes (GET /api/user, PUT /api/user/:id, etc.). This mechanism ensures that only authenticated users can access authorized resources, enhancing application security.

Data Encryption:

- **Encryption of Sensitive Data:** Sensitive data is encrypted both at rest and in transit to maintain confidentiality and prevent unauthorized access. At rest, data stored in the MongoDB database is encrypted using MongoDB's encryption-at-rest features or similar mechanisms to protect against data breaches or unauthorized database access. In transit, data transmitted between the frontend and backend is encrypted using HTTPS/TLS protocols. This encryption ensures that data exchanged over the network remains secure and cannot be intercepted or modified by malicious actors.