**Full Stack Development with MERN**

**Project Documentation format**

**1. Introduction**

• **Project Title:** Social Media App (MERN)

• **Team Members:** Milan Kumar

•**Skillwallet ID**:SWUID20250199287

**2. Project Overview**

**Purpose:**  
The Social Media App is built to demonstrate how a complete social networking platform can be developed using the **MERN stack**. It aims to combine a modern frontend, RESTful backend, and real-time communication layer in one full-stack project. The goal is to give developers a hands-on example of integrating **React**, **Node**, **Express**, **MongoDB**, and **Socket.io** for a production-style application.

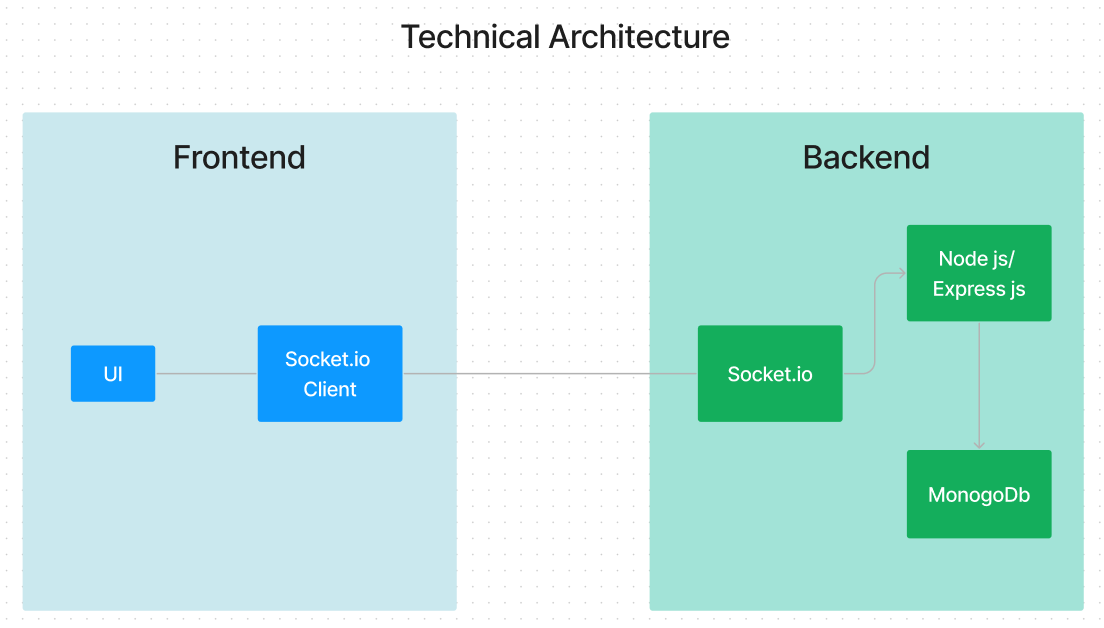
**Features:**

1. **User Authentication:** Register, log in, and manage user sessions securely.
2. **Post System:** Create, view, like, comment, and delete posts.
3. **Stories:** Add and view short-term user stories.
4. **Follow/Unfollow:** Build user connections to personalize the feed.
5. **Real-Time Chat:** Instant one-to-one messaging using Socket.io.
6. **Live Updates:** Posts, comments, and stories update instantly without page reloads.
7. **Profile Management:** Update profile photo, bio, and username.
8. **Responsive UI:** Fully functional on both desktop and mobile screens.

**3. Architecture**

**Frontend:**  
The frontend of the application is built using **React.js**, structured into modular and reusable components. Each page (Feed, Profile, Chat, Stories, etc.) is represented as a separate component with centralized state management using React Hooks. Axios is used for API requests to interact with the backend. Real-time updates are handled via **Socket.io client**, allowing dynamic rendering of posts, messages, and stories without page reloads. The design focuses on responsiveness, ensuring smooth usage on both desktop and mobile devices.

**Backend:**  
The backend is developed using **Node.js** with **Express.js** as the server framework. The backend exposes RESTful APIs for authentication, user management, posts, comments, and stories. It also integrates **Socket.io** to enable real-time communication for chat and live updates (likes, comments, and new posts). Middleware like body-parser and cors are used for handling requests and cross-origin communication. The backend structure follows a clean separation of routes, controllers, and models, ensuring scalability and maintainability.



**Database:**  
The application uses **MongoDB** as the database, accessed through **Mongoose** for object data modeling (ODM). The database includes multiple collections:

* **Users:** Stores user details such as username, email, password (hashed), profile picture, followers, and following.
* **Posts:** Contains post data including text, image, likes, and comments.
* **Chats:** Stores real-time conversation data between users.
* **Stories:** Holds user-uploaded short-term stories with viewer tracking.

Each model is interconnected through unique user IDs, allowing efficient querying and relationship mapping. MongoDB’s flexible schema design makes it ideal for managing unstructured social data while ensuring high performance during real-time updates.

**4. Setup Instructions**

**Prerequisites:**  
Before running the project, make sure you have the following installed:

* **Node.js** (v16 or higher) – for running the backend server and React build tools
* **npm** – installed with Node.js for dependency management
* **MongoDB** (local or cloud instance) – for database operations
* **Git** – to clone the project repository
* **Code Editor** (e.g., VS Code) – for editing and debugging
* **Browser** (e.g., Chrome) – for running the web app

**Installation Steps:**

1. **Clone the Repository:**
2. git clone https://github.com/milan/social\_media\_app.git
3. cd social\_media\_app
4. **Navigate to the Server Folder:**
5. cd server
6. npm install

This installs all backend dependencies like Express, Mongoose, Cors, Body-Parser, and Socket.io.

1. **Set Up the Frontend:**
2. cd ../client
3. npm install

This installs frontend dependencies including React, Axios, and Socket.io-client.

1. **Configure Environment Variables:**  
   Create a .env file inside the server folder and add:
2. PORT=6001
3. MONGO\_URI=mongodb://localhost:27017/socialeX
4. JWT\_SECRET=your\_secret\_key

(You can modify the MongoDB connection string if using a remote or cloud database.)

1. **Start MongoDB:**  
   Open a terminal and run:
2. mongod

Ensure MongoDB is running before starting the server.

1. **Run the Backend Server:**
2. cd server
3. npm start

The backend runs on [**http://localhost:6001**](http://localhost:6001)

1. **Run the Frontend App:**
2. cd ../client
3. npm start

The React frontend runs on [**http://localhost:3000**](http://localhost:3000)

1. **Access the App:**  
   Open your browser and go to [**http://localhost:3000**](http://localhost:3000) to view the application.

### ****5. Folder Structure****

The project is organized into two main parts — **client (frontend)** and **server (backend)** — following the standard MERN architecture pattern. This modular separation makes the codebase easier to maintain, scale, and debug.

#### ****Client (Frontend)****

The **client/** folder contains all React.js code responsible for the user interface and frontend logic.

client/

│

├── public/ # Static files like index.html and favicon

├── src/

│ ├── components/ # Reusable React components (Navbar, PostCard, Story, etc.)

│ ├── pages/ # Page-level components (Login, Register, Feed, Profile, Chat)

│ ├── api/ # Axios API configuration and HTTP request functions

│ ├── context/ # Context API files for global state management

│ ├── socket/ # Socket.io client setup for real-time events

│ ├── assets/ # Images, icons, and static resources

│ ├── App.js # Main React component defining routes and structure

│ ├── index.js # Entry point rendering the React app

│ └── styles/ # CSS or styled-component files for UI design

│

└── package.json # Frontend dependencies and scripts

**Explanation:**  
The React app uses functional components with Hooks for state management. App.js manages routing between pages, while socket/ integrates Socket.io for real-time updates such as new posts, messages, and stories.

#### ****Server (Backend)****

The **server/** folder manages all backend logic, including APIs, database connections, and real-time communication.

server/

│

├── controllers/ # Contains logic for handling requests (Auth, Posts, Chat, etc.)

├── models/ # Mongoose schemas for Users, Posts, Chats, Stories

├── routes/ # Express route definitions for APIs

├── SocketHandler.js # Handles all real-time Socket.io events

├── server.js # Entry point – sets up Express, Socket.io, and MongoDB

├── config/ # (Optional) Database and environment configurations

└── package.json # Backend dependencies and scripts

**Explanation:**  
The backend uses Express.js to define RESTful routes, Mongoose for database operations, and Socket.io for real-time features. Each module is separated by function (routes, models, controllers) to maintain a clean MVC-style structure.

**6. Running the Application**

Commands to start the frontend and backend servers locally.

•**Frontend:** npm start in the client directory.

•**Backend:** npm start in the server directory.

The React app will typically run on <http://localhost:3000>.

### ****Notes****

* Ensure MongoDB is running locally or your database URI is correctly set in the backend .env.
* The frontend will communicate with the backend API and the Socket.io server automatically if URLs are configured correctly.

## 7. API Documentation

The backend server exposes several endpoints for user authentication, posts, messaging, and more. Below is a detailed description.

### ****A. User Authentication****

#### ****Register User****

* **Endpoint:** /api/auth/register
* **Method:** POST
* **Request Body:**

{

"username": "johndoe",

"email": "johndoe@example.com",

"password": "password123"

}

* **Response (Success):**

{

"message": "User registered successfully",

"userId": "64f8b2c4a1e7f01234567890"

}

* **Response (Error):**

{

"error": "Email already exists"

}

#### ****Login User****

* **Endpoint:** /api/auth/login
* **Method:** POST
* **Request Body:**

{

"email": "johndoe@example.com",

"password": "password123"

}

* **Response (Success):**

{

"message": "Login successful",

"token": "jwt-token-here",

"user": {

"id": "64f8b2c4a1e7f01234567890",

"username": "johndoe",

"email": "johndoe@example.com"

}

}

* **Response (Error):**

{

"error": "Invalid credentials"

}

### ****B. Posts****

#### ****Create Post****

* **Endpoint:** /api/posts
* **Method:** POST
* **Headers:** Authorization: Bearer <token>
* **Request Body:**

{

"content": "Hello world!",

"imageUrl": "https://example.com/image.jpg"

}

* **Response (Success):**

{

"message": "Post created successfully",

"postId": "64f8b2d5a1e7f01234567891"

}

#### ****Get All Posts****

* **Endpoint:** /api/posts
* **Method:** GET
* **Response:**

[

{

"id": "64f8b2d5a1e7f01234567891",

"content": "Hello world!",

"imageUrl": "https://example.com/image.jpg",

"author": "johndoe",

"createdAt": "2025-10-14T09:00:00Z"

}

]

#### ****Like a Post****

* **Endpoint:** /api/posts/:postId/like
* **Method:** PUT
* **Headers:** Authorization: Bearer <token>
* **Response (Success):**

{

"message": "Post liked successfully",

"likes": 5

}

### ****C. Messaging****

#### ****Send Message****

* **Endpoint:** /api/messages
* **Method:** POST
* **Headers:** Authorization: Bearer <token>
* **Request Body:**

{

"receiverId": "64f8b2c4a1e7f01234567890",

"text": "Hello, how are you?"

}

* **Response:**

{

"message": "Message sent successfully",

"messageId": "64f8b2f6a1e7f01234567892"

}

#### ****Get Messages with a User****

* **Endpoint:** /api/messages/:userId
* **Method:** GET
* **Headers:** Authorization: Bearer <token>
* **Response:**

[

{

"senderId": "64f8b2c4a1e7f01234567890",

"receiverId": "64f8b2c4a1e7f01234567891",

"text": "Hello!",

"createdAt": "2025-10-14T09:05:00Z"

}

]

### ****D. User Profile****

#### ****Get User Profile****

* **Endpoint:** /api/users/:userId
* **Method:** GET
* **Response:**

{

"id": "64f8b2c4a1e7f01234567890",

"username": "johndoe",

"email": "johndoe@example.com",

"bio": "Loves coding",

"profilePicture": "https://example.com/profile.jpg"

}

#### ****Update User Profile****

* **Endpoint:** /api/users/:userId
* **Method:** PUT
* **Headers:** Authorization: Bearer <token>
* **Request Body:**

{

"bio": "I love coding and coffee",

"profilePicture": "https://example.com/new-profile.jpg"

}

* **Response:**

{

"message": "Profile updated successfully"

}

## 8. Authentication

The social media app implements **authentication** and **authorization** to ensure that only legitimate users can access protected resources and perform actions. The project uses **JWT (JSON Web Tokens)** for secure authentication.

### ****A. User Authentication Flow****

1. **Registration**
   * Users create an account by providing a username, email, and password.
   * Passwords are **hashed** using bcrypt before being stored in the database to ensure security.
   * On successful registration, the user receives a success message; no token is issued at this stage.
2. **Login**
   * Users log in by providing their email and password.
   * The backend verifies the credentials by comparing the hashed password stored in the database.
   * If valid, the server generates a **JWT token** containing the user ID and optionally roles or permissions.
3. **Token Delivery**
   * The JWT token is sent back to the client in the login response.
   * Example:

{

"token": "jwt-token-here",

"user": {

"id": "64f8b2c4a1e7f01234567890",

"username": "johndoe",

"email": "johndoe@example.com"

}

}

### ****B. Authorization with JWT****

* Protected routes (e.g., creating posts, sending messages) require the **Authorization header** with the token:

Authorization: Bearer <jwt-token-here>

* The backend middleware **verifies the token** using a secret key stored in .env.
* If the token is valid, the request proceeds; otherwise, a **401 Unauthorized** error is returned.

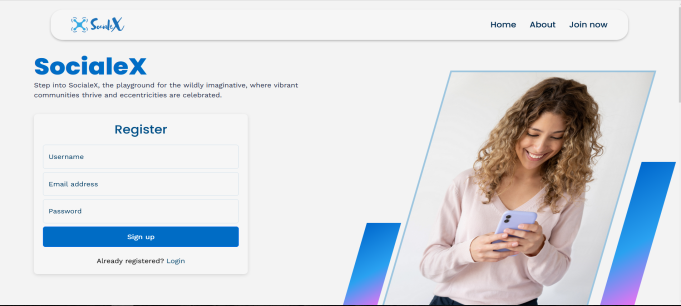
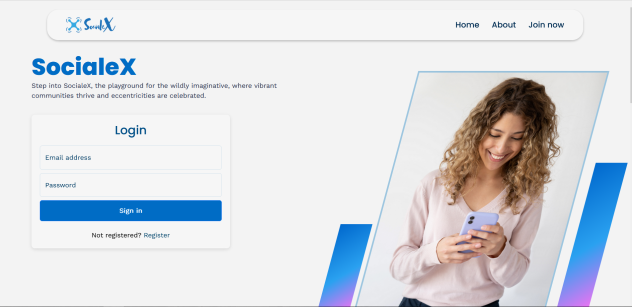
### ****C. Token Expiry & Security****

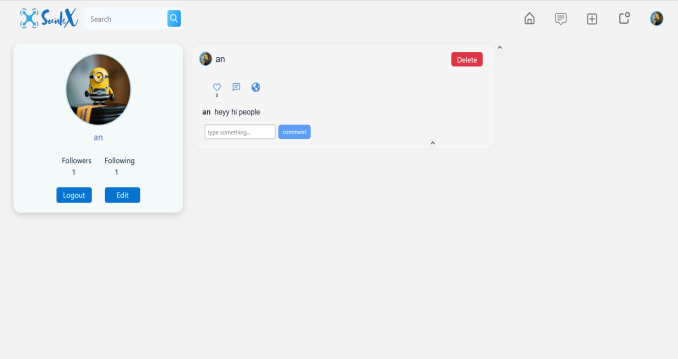
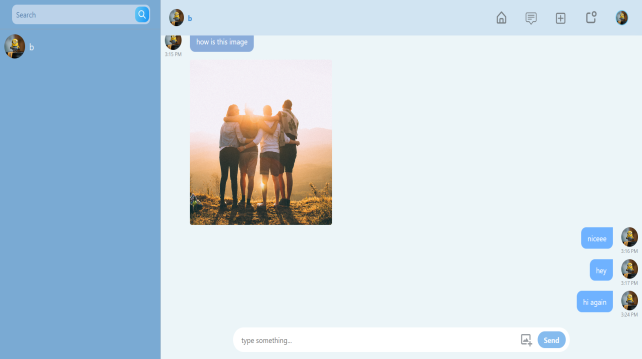
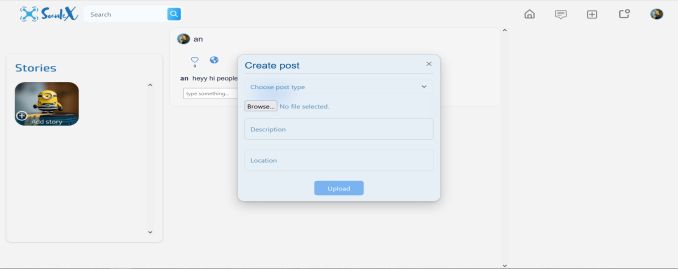
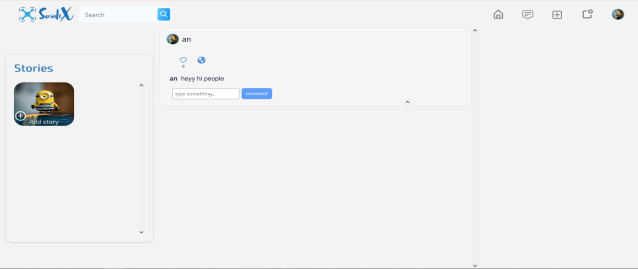
* Tokens are usually set with an **expiration time** (e.g., 1 hour) to reduce the risk if stolen.
* Once expired, the user must log in again to receive a new token.
* Optionally, **refresh tokens** can be implemented for longer sessions, but the current project uses **stateless JWTs** for simplicity.

### ****D. Session Management****

* The app does **not use server-side sessions**; instead, authentication is **stateless** via JWTs.
* This allows scalability, as the backend does not need to maintain session storage.
* All necessary user information is encoded in the token and verified on each request.

**9. User Interface**





### 10. Testing

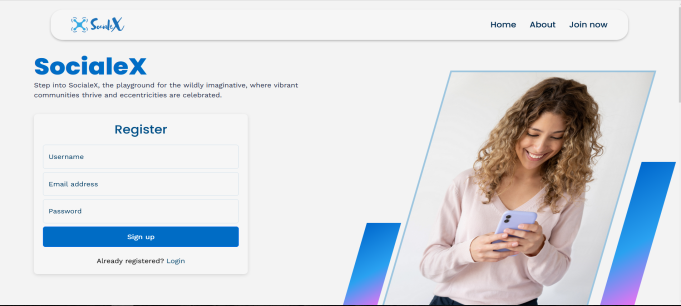
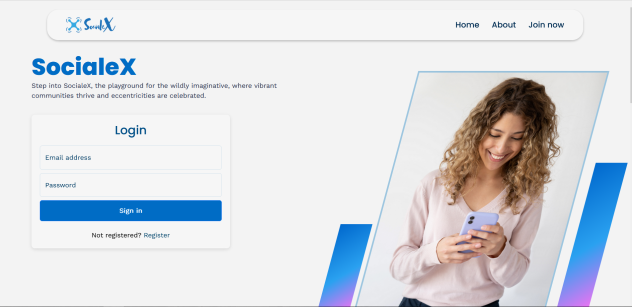
**Testing Strategy:**  
The testing strategy for the social media application follows a multi-level approach to ensure functionality, performance, and reliability:

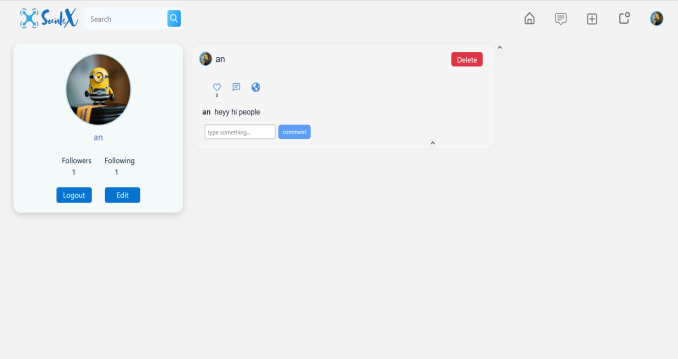
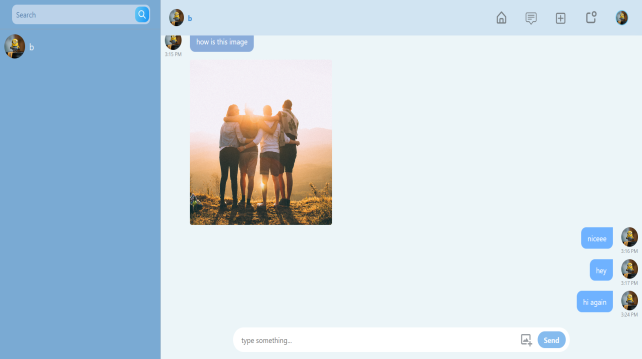
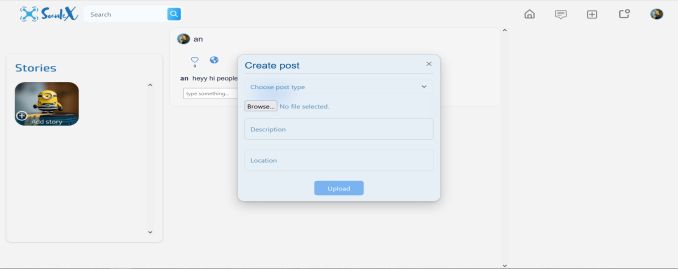
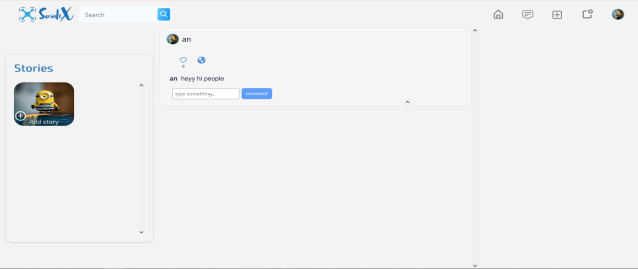
1. **Unit Testing:**
   * Focuses on testing individual components and functions in isolation.
   * Ensures that each module performs its intended function correctly.
   * Example: Testing React components in the frontend and API endpoints in the backend.
2. **Integration Testing:**
   * Validates the interaction between different modules and services.
   * Ensures that the frontend communicates correctly with the backend APIs.
   * Example: Testing the registration/login flow, post creation, and comment functionalities.
3. **End-to-End (E2E) Testing:**
   * Simulates real user scenarios from start to finish.
   * Ensures that the complete system works as expected.
   * Example: A user logging in, creating a post, liking a post, and logging out successfully.
4. **Performance Testing:**
   * Evaluates the system’s response time and stability under various loads.
   * Helps identify bottlenecks in APIs, database queries, and frontend rendering.
5. **Security Testing:**
   * Focuses on identifying vulnerabilities such as XSS, SQL Injection, and authentication/authorization issues.
   * Ensures that user data and communications are secure.

**Testing Tools Used:**

* **Frontend:**
  + **Jest:** For unit testing React components.
  + **React Testing Library:** For testing component behavior and DOM interactions.
* **Backend:**
  + **Mocha & Chai:** For unit and integration testing of Node.js API endpoints.
  + **Supertest:** For testing HTTP requests and responses.
* **End-to-End Testing:**
  + **Cypress:** For simulating real user interactions and verifying end-to-end workflows.
* **Performance Testing:**
  + **Postman / Apache JMeter:** For testing API response times and load handling.
* **Security Testing:**
  + **OWASP ZAP:** For detecting common web vulnerabilities.
  + **Manual Penetration Testing:** For validating authentication, session management, and input validation.

**11. Screenshots or Demo**





### 12. Known Issues

1. **Slow Navigation on Large Libraries:**
   * Users may experience lag when browsing or searching through a very large collection of posts, media, or files.
   * Optimization is ongoing for smoother scrolling and faster search indexing.
2. **Occasional Image/Media Loading Delays:**
   * High-resolution images or videos may take longer to load on slower internet connections.
   * A caching strategy is being considered to improve load times.
3. **Browser Compatibility:**
   * Some older browsers may not fully support all frontend features (e.g., modern React hooks or CSS effects).
   * Recommended browsers: Chrome, Edge, Firefox (latest versions).
4. **Real-Time Updates Delay:**
   * Socket-based notifications or real-time updates (likes, comments) may experience minor delays under high server load.
   * Scaling solutions and message queue optimizations are planned.
5. **Limited Offline Functionality:**
   * Currently, certain actions (like posting or liking) require an active internet connection. Offline support is not fully implemented.
6. **Minor UI/UX Inconsistencies:**
   * Some layout or styling inconsistencies may appear on specific screen sizes or mobile devices.
   * Responsive design improvements are ongoing.
7. **Authentication Edge Cases:**
   * Rare cases where token expiration or session handling may require a manual refresh or re-login.
   * Backend improvements are being considered to handle all token edge cases smoothly.

### 13. Future Enhancements

1. **Advanced Search and Filtering:**
   * Implement more sophisticated search functionality, allowing users to filter posts by hashtags, date, popularity, media type, or user.
2. **Offline Mode & Caching:**
   * Enable offline access to previously loaded posts and media using local storage or service workers.
   * Improve media caching for faster load times.
3. **Enhanced Notifications:**
   * Introduce customizable push notifications for likes, comments, follows, and direct messages.
   * Support in-app, browser, and mobile notifications.
4. **AI-Powered Features:**
   * Content recommendation engine based on user behavior and interests.
   * Automatic tagging and categorization of posts using AI.
   * Image moderation and spam detection using machine learning.
5. **Dark Mode and Theme Customization:**
   * Allow users to toggle between light/dark mode and choose custom themes for the UI.
6. **Improved Media Support:**
   * Support additional file types, higher-resolution uploads, and live streaming capabilities.
7. **Multi-Language Support:**
   * Implement internationalization (i18n) to support multiple languages and regional settings.
8. **Enhanced Security Measures:**
   * Two-factor authentication (2FA) and advanced account recovery options.
   * End-to-end encryption for private messages.
9. **Performance and Scalability Improvements:**
   * Optimize backend APIs and database queries for faster response times.
   * Implement horizontal scaling and load balancing to handle higher traffic.
10. **Analytics Dashboard for Users:**
    * Provide insights like post reach, engagement rates, and follower growth trends.