**Social Media App (with Context API, Promises, Redux, MongoDB)**

In this assignment, you will build a **social media platform** where users can interact with each other by posting content, commenting, liking posts, following or unfollowing users, and engaging in real-time chat. The app should use **React** on the frontend, **Node.js** and **Express** for the backend, and **MongoDB** as the database. This project also involves **Redux** for state management, **Context API** for authentication, **Promises** for asynchronous operations, and **WebSockets** for real-time messaging.

**Core Concepts:**

* **Context API**: Used for handling user authentication and managing user sessions.
* **Redux**: Manages global application state for posts, comments, followers, likes, and chat messages.
* **Promises**: Used to handle asynchronous API calls for loading posts, comments, likes, and other data.
* **MongoDB**: Serves as the database for storing users, posts, comments, likes, follow relations, and chat history.

**Key Features and Detailed Breakdown:**

**1. User Authentication (with JWT and Context API)**

* **User Signup & Login**: Users can register and log in to the platform. Store password securely using **bcrypt** and handle user authentication with **JWT** (JSON Web Tokens).
* **Context API for Authentication**: Use the Context API to manage user sessions across the app, keeping track of the current authenticated user.
* **Protected Routes**: Ensure that only authenticated users can access certain features like creating posts or following others.

**2. Post Creation, Commenting, and Likes (with Redux and MongoDB)**

* **Post Creation**: Users can create a new post by submitting text, images, or videos. Posts should be stored in **MongoDB**, and their data fetched via a **REST API**.
* **Commenting on Posts**: Users can comment on posts. Use **Promises** to fetch, create, and display comments asynchronously.
* **Like Functionality**: Users can like posts, and the total number of likes should be updated dynamically. Use Promises to handle asynchronous updates when a user likes a post.
* **Redux for State Management**: Manage the state of posts, likes, and comments globally using **Redux**. This ensures that any change (e.g., a new comment or like) reflects across all relevant components.

**3. Follow/Unfollow System**

* **Follow/Unfollow Users**: Users can follow or unfollow other users. This should be stored in **MongoDB** in a "Followers" collection.
* **Notification for Follow/Unfollow**: When a user follows or unfollows someone, send an **email notification** to the respective user. You can use **NodeMailer** or similar services for sending email notifications.

**4. Asynchronous Data Handling with Promises**

* **Loading Posts and Comments**: Use **Promises** to fetch posts and their associated comments asynchronously. Ensure the UI shows a loading state while waiting for data to arrive.
* **Error Handling**: Implement proper error handling using Promises. For example, if fetching data from the API fails, display an appropriate error message to the user.

**5. REST API for Backend Logic**

* **CRUD Operations for Posts and Comments**: Build a **RESTful API** using **Express.js** that supports CRUD (Create, Read, Update, Delete) operations for posts and comments.
* **Authentication Middleware**: Use JWT authentication middleware in Express to protect routes (e.g., only authenticated users can create posts or like content).
* **Endpoints for Likes and Follow Relations**: Include dedicated endpoints for liking posts and following/unfollowing users, making sure to update the MongoDB collections accordingly.

**6. Email Notifications for Likes, Comments, and Follows**

* **Email Notifications**: When a user likes a post, comments, or follows/unfollows someone, an email notification should be sent to the relevant user. For example:
  + If a post receives a new comment, notify the post owner via email.
  + If a user follows another user, send a follow notification to the user being followed.
  + Use a library like **NodeMailer** or an email service like **SendGrid** to send email notifications.

**7. Real-time Chat (with WebSockets and Redux)**

* **Real-time Messaging**: Implement a real-time chat feature where users can send messages to each other. Use **WebSockets** for real-time bi-directional communication.
* **WebSockets (Socket.io)**: Use **Socket.io** to enable real-time messaging. Each user can send and receive messages in real-time without needing to refresh the page.
* **Redux for Chat State**: Manage chat state using **Redux** to ensure the chat history is available and updated across the app.
* **Chat History Storage**: Store chat history in **MongoDB**, allowing users to see previous conversations when they return to the chat.

**Additional Features:**

**8. Notifications for Real-Time Interactions**

* **In-App Notifications**: Along with email notifications, implement in-app notifications using **WebSockets** to notify users when someone likes their post, comments, or follows them.
* **Push Notifications (Optional)**: Integrate push notifications (using a service like **Firebase Cloud Messaging**) for notifying users of real-time activities like new messages, likes, or comments.

**9. Profile Page and Post Feed**

* **User Profile Page**: Each user should have a profile page displaying their posts, followers, and following lists.
* **Global Feed**: Implement a global feed showing posts from all users or from followed users. The feed should be fetched from the backend using **Promises**.

**Database Structure (MongoDB):**

* **Users Collection**:
  + Fields: username, email, password, followers, following, posts, etc.
* **Posts Collection**:
  + Fields: postContent, userId, likes, comments, createdAt, etc.
* **Comments Collection**:
  + Fields: commentText, userId, postId, createdAt, etc.
* **Chat Collection**:
  + Fields: userId, message, createdAt, etc.

**Tech Stack:**

* **Frontend**:
  + **React.js**
  + **Redux** for global state management
  + **Context API** for user authentication
  + **Socket.io** for real-time messaging
* **Backend**:
  + **Node.js** and **Express.js** for REST API
  + **MongoDB** for data storage (MongoDB Atlas for cloud storage)
  + **JWT** for authentication
  + **Socket.io** for WebSocket-based real-time communication
* **Email Service**:
  + **NodeMailer** or **SendGrid** for sending email notifications

This comprehensive social media app will give you practical experience with advanced MERN stack concepts, handling both front-end and back-end challenges, while also involving real-time features and asynchronous programming with Promises.

**Please host the assignment on any free platform ( Render, Netlify, Vercel).**