**Social Media App with ChatGPT**

[**https://chatgpt.com/share/678ef9aa-23a4-8008-92b7-c938bf7b6fba**](https://chatgpt.com/share/678ef9aa-23a4-8008-92b7-c938bf7b6fba)

SDLC – Software Development Life Cycle

1. Requirement Gathering - What features to develop
2. Technical Documentation – Designer – All require technical details like what are API, How DB interact with each Schema, Model, What type of response needed etc…
3. Divide the Task to Team members
4. Coding phase – Frontend, Backend, DB
5. Testing the App – Test All features run properly
6. Deployment - Online at any platform
7. Maintenance - For any new feature, any bug etc…

* **Requirement Gathering**
  + Prompt Engineer – line in chatGPT
  + I want to prepare a social media application like twitter using MERN stack, where I want to have features like user can create posts, user can create comments, comments can be on posts or other comments, user can like the posts or like the comments, user can follow other users, user should be having some feed based on other people posts, user should have a user profile as well.

**Response From ChatGPT**

That's a great project idea! It covers various key functionalities that align with common social media applications, showcasing your ability to handle complex features. Here’s a roadmap you can follow to build your project:

**Backend Setup**

1. **Initialize the Server**:
   * Use **Express.js** to create your backend server.
   * Set up a MongoDB database using **Mongoose** for schemas.
2. **Design the Database Models**:
   * **User**: Username, email, password (hashed), bio, avatar, followers, following.
   * **Post**: Content, author, createdAt, likes, comments.
   * **Comment**: Content, author, postId, parentComment (if any), likes.
   * **Feed**: Store user feed-related posts dynamically or generate them on the fly.
3. **Authentication**:
   * Implement JWT for user authentication.
   * Include login, registration, and middleware for protected routes.
4. **APIs**:
   * **User Management**: CRUD operations for users, follow/unfollow functionality.
   * **Posts**: Create, read, update, and delete posts.
   * **Comments**: Support nested comments using parent references.
   * **Likes**: Toggle likes for posts and comments.
   * **Feeds**: Fetch user-specific feed posts based on following.

**Frontend Setup**

1. **Scaffold the App**:
   * Use **React** for the frontend with **Redux** or **Context API** to manage state.
2. **Build Components**:
   * **Authentication**: Login, registration, and token storage.
   * **Home Feed**: Display a list of posts from followed users.
   * **Post Detail**: Show a single post with comments and likes.
   * **Profile**: User’s bio, stats, and posts.
   * **Follow Suggestions**: Component to recommend new users to follow.
3. **Integration**:
   * Use **Axios** to communicate with the backend.
   * Handle authentication tokens for protected requests.
4. **Styling**:
   * Use **CSS frameworks** (like Tailwind CSS or Material-UI) for rapid styling.

**Bonus Features**

* **Search**: Add a search bar to find users or posts.
* **Realtime Updates**: Use **WebSocket** (Socket.IO) for live notifications (e.g., likes, follows).
* **Media Uploads**: Allow users to upload images using **Cloudinary** or a similar service.

**Product Requirements Document (PRD): Social Media Application (MERN Stack)**

**Overview**

The goal of this project is to create a social media application similar to Twitter using the MERN (MongoDB, Express.js, React.js, Node.js) stack. The application will enable users to create posts, interact with other users via comments and likes, build a personalized feed, and manage their profile and relationships with other users.

**Objectives**

* Build an intuitive and user-friendly social media platform.
* Implement scalable architecture for future enhancements.
* Utilize modern web technologies (MERN stack) and ensure secure and efficient performance.

**Features and Functional Requirements**

**1. User Management**

**1.1. User Authentication**

* **Registration**: New users can register with a username, email, password (hashed), and optional profile information.
* **Login**: Users log in with email and password.
* **Authentication**: Use JWT for session management. Tokens will expire after a set duration.
* **Forgot Password**: Allow users to reset their password via email verification.

**1.2. User Profile**

* Editable profile including name, bio, avatar, and location.
* Display user’s posts, followers, and following.
* Followers and following counts visible to the user and others.

**1.3. User Connections**

* Follow/unfollow functionality.
* Ability to view followers and following lists.

**2. Post Management**

**2.1. Create Posts**

* Users can create text-based posts.
* Support for uploading images or media files.

**2.2. Post Interactions**

* **Like Posts**: Users can like and unlike posts.
* **Comment on Posts**: Users can write comments on posts (supports nested commenting).

**2.3. Post Feed**

* Personalized feed showing posts from followed users, sorted by recency.
* Ability to paginate and load older posts.
* Trending posts section displaying posts with the highest engagement.

**3. Comment Management**

**3.1. Comment Creation**

* Allow users to add comments on posts.
* Users can also comment on other comments (nested/hierarchical structure).

**3.2. Comment Interactions**

* Users can like and unlike comments.
* View all comments (and sub-comments) for a post in a threaded format.

**4. Feed and Notifications**

**4.1. Personalized Feed**

* The feed is dynamically generated based on posts by followed users.
* Include posts’ metadata, such as total likes and comments.

**4.2. Notifications**

* Real-time notifications for likes, comments, follows, and replies.
* Mark notifications as read.

**5. Search and Discovery**

**5.1. Search Functionality**

* Search for users by username or name.
* Display top results based on relevance.

**5.2. Discovery Section**

* Suggest users to follow based on mutual connections or trending activity.

**6. Admin Panel (Optional for Future Phase)**

* Moderate posts and comments.
* Suspend or delete user accounts for violating policies.

**Technical Requirements**

**1. Technology Stack**

* **Frontend**: React.js (with Context API or Redux for state management).
* **Backend**: Express.js.
* **Database**: MongoDB (using Mongoose for schema definition).
* **Authentication**: JSON Web Tokens (JWT).
* **Storage**: Cloudinary or AWS S3 for media uploads.

**2. API Endpoints**

**User**

* POST /api/auth/register
* POST /api/auth/login
* GET /api/users/:userId
* PUT /api/users/:userId
* GET /api/users/:userId/followers
* GET /api/users/:userId/following

**Posts**

* POST /api/posts
* GET /api/posts/:postId
* PUT /api/posts/:postId
* DELETE /api/posts/:postId
* GET /api/feed

**Comments**

* POST /api/comments/:postId
* GET /api/comments/:postId
* PUT /api/comments/:commentId
* DELETE /api/comments/:commentId

**Likes**

* POST /api/posts/:postId/like
* POST /api/comments/:commentId/like

**Notifications**

* GET /api/notifications
* PUT /api/notifications/mark-read

**3. Performance Considerations**

* Use pagination for feed and comment loading to handle large data sets.
* Implement indexes in MongoDB for faster queries (e.g., on createdAt, likes).
* Optimize image uploads by resizing and compressing.

**4. Security**

* Use bcrypt to hash passwords.
* Secure endpoints with middleware for authorization and rate-limiting.
* Use HTTPS for secure data transmission.

**Non-Functional Requirements**

* **Scalability**: Design the backend to handle increased user traffic.
* **Responsiveness**: Ensure the UI is mobile-friendly and adjusts to different screen sizes.
* **Performance**: Target low response times (<200ms for key APIs).
* **Accessibility**: Follow best practices for accessibility, including ARIA labels and keyboard navigation.

**Timeline and Milestones**

1. **Week 1-2**: Backend setup, authentication, and user model.
2. **Week 3-4**: Post and comment functionality.
3. **Week 5**: Feed and like features.
4. **Week 6**: Search, user profiles, and following system.
5. **Week 7**: Notifications and testing.
6. **Week 8**: UI/UX refinements and deployment.

**Future Enhancements**

* Add a chat/messaging system.
* Implement video uploads.
* Introduce a monetization system for premium users.

**Conclusion**

This application aims to deliver a feature-rich, secure, and scalable social media platform. By following this PRD, development can remain focused and efficient while delivering a high-quality product that meets user expectations.

**Technical Design Document for Social Media Application (MERN Stack)**

**Prompt Engineer -**

I want you to write the technical design document, which should have the feature details, background, backend api specification (api method, url, request body, response body for success and error) then also write the frontend logic that do we need any states or any api calls, where to add the api calls, how to connect backend and frontned, db schema and models, and also write the relevant db queries which will help us in this feature Assume we have to later code it in MERN stack

**Technical Design Document for Social Media Application (MERN Stack)**

**Background**

This document details the technical implementation of a social media application with key features such as user management, posts, comments, feeds, notifications, and search. The design will use the MERN stack, where:

* **MongoDB**: Database for storing data.
* **Express.js**: Backend framework.
* **React.js**: Frontend library for building the user interface.
* **Node.js**: Runtime for backend logic.

**Feature Details**

**1. User Management**

**1.1 User Authentication**

**Backend API Specification**

* **Registration**
  + **Method**: POST
  + **URL**: /api/auth/register
  + **Request Body**:
  + {
  + "username": "string",
  + "email": "string",
  + "password": "string"
  + }
  + **Response Body (Success)**:
  + {
  + "message": "User registered successfully",
  + "userId": "string"
  + }
  + **Response Body (Error)**:
  + {
  + "error": "string"
  + }
* **Login**
  + **Method**: POST
  + **URL**: /api/auth/login
  + **Request Body**:
  + {
  + "email": "string",
  + "password": "string"
  + }
  + **Response Body (Success)**:
  + {
  + "token": "string",
  + "user": {
  + "id": "string",
  + "username": "string",
  + "avatar": "string"
  + }
  + }
  + **Response Body (Error)**:
  + {
  + "error": "Invalid email or password"
  + }
* **Forgot Password**
  + **Method**: POST
  + **URL**: /api/auth/forgot-password
  + **Request Body**:
  + {
  + "email": "string"
  + }
  + **Response Body (Success)**:
  + {
  + "message": "Password reset email sent"
  + }
  + **Response Body (Error)**:
  + {
  + "error": "Email not found"
  + }

**Database Schema**

* **User Schema**:
* const UserSchema = new mongoose.Schema({
* username: { type: String, required: true, unique: true },
* email: { type: String, required: true, unique: true },
* password: { type: String, required: true },
* avatar: { type: String },
* bio: { type: String },
* location: { type: String },
* followers: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }],
* following: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }]
* });

**Relevant DB Queries**

* Register User:
* const user = new User({ username, email, password: hashedPassword });
* await user.save();
* Login User:
* const user = await User.findOne({ email });

**Frontend Logic**

* **States**:
  + isAuthenticated: Boolean state for authentication.
  + user: Object containing user details.
* **API Calls**:
  + Use axios for API calls.
  + On registration or login success, store the JWT token in local storage and update the isAuthenticated and user states.
* **Implementation**:
  + Add api/auth/register and api/auth/login calls in AuthContext or Redux store.

**2. Post Management**

**2.1 Create Posts**

**Backend API Specification**

* **Create Post**
  + **Method**: POST
  + **URL**: /api/posts
  + **Request Body**:
  + {
  + "content": "string",
  + "media": "string (optional)",
  + "author": "userId"
  + }
  + **Response Body (Success)**:
  + {
  + "message": "Post created successfully",
  + "post": {
  + "id": "string",
  + "content": "string",
  + "media": "string",
  + "author": "userId",
  + "createdAt": "string"
  + }
  + }

**Database Schema**

* **Post Schema**:
* const PostSchema = new mongoose.Schema({
* content: { type: String, required: true },
* media: { type: String },
* author: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
* likes: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }],
* comments: [{ type: mongoose.Schema.Types.ObjectId, ref: "Comment" }],
* createdAt: { type: Date, default: Date.now }
* });

**Relevant DB Queries**

* Create Post:
* const post = new Post({ content, media, author });
* await post.save();

**Frontend Logic**

* **States**:
  + posts: Array of posts displayed in the feed.
* **API Calls**:
  + Add a POST request to /api/posts in the PostContext or Redux slice.
* **Implementation**:
  + Create a CreatePost component with form inputs for content and optional media upload.
  + On submission, call the backend API, then update the posts state.

**3. Comment Management**

**3.1 Add Comments**

**Backend API Specification**

* **Add Comment**
  + **Method**: POST
  + **URL**: /api/comments
  + **Request Body**:
  + {
  + "content": "string",
  + "postId": "string",
  + "author": "userId",
  + "parentComment": "string (optional)"
  + }

**Database Schema**

* **Comment Schema**:
* const CommentSchema = new mongoose.Schema({
* content: { type: String, required: true },
* postId: { type: mongoose.Schema.Types.ObjectId, ref: "Post", required: true },
* author: { type: mongoose.Schema.Types.ObjectId, ref: "User", required: true },
* parentComment: { type: mongoose.Schema.Types.ObjectId, ref: "Comment" },
* likes: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }],
* createdAt: { type: Date, default: Date.now }
* });

**Relevant DB Queries**

* Add Comment:
* const comment = new Comment({ content, postId, author, parentComment });
* await comment.save();

**Frontend Logic**

* **States**:
  + comments: Array of comments for a post.
* **API Calls**:
  + Add a POST request to /api/comments in the CommentContext or Redux slice.
* **Implementation**:
  + Create a CommentSection component.
  + On adding a comment, update the comments state and call the backend API.

This document can be extended to other features such as notifications, feeds, and admin panels with similar details for APIs, frontend states, backend schemas, and queries. Let me know if you’d like the complete document or focus on any specific feature!

**Ans From Blackbox**

Here’s an updated technical design document based on the provided feedback, covering the features and functional requirements for the social media application.

**Technical Design Document: Social Media Application**

**Feature Details:**

1. **User Management**
   * **User Authentication**
     + Registration, login, JWT authentication, and password reset.
   * **User Profile**
     + Editable profile with posts, followers, and following counts.
   * **User Connections**
     + Follow/unfollow functionality and viewing connections.
2. **Post Management**
   * **Create Posts**
     + Text-based posts with image/media upload support.
   * **Post Interactions**
     + Like/unlike posts and comment on posts (nested comments).
   * **Post Feed**
     + Personalized feed with pagination and trending posts.
3. **Comment Management**
   * **Comment Creation**
     + Add comments on posts and other comments.
   * **Comment Interactions**
     + Like/unlike comments and view threaded comments.
4. **Feed and Notifications**
   * **Personalized Feed**
     + Dynamically generated feed with metadata.
   * **Notifications**
     + Real-time notifications for interactions.
5. **Search and Discovery**
   * **Search Functionality**
     + Search users by username or name.
   * **Discovery Section**
     + Suggest users to follow based on activity.
6. **Admin Panel (Optional for Future Phase)**
   * Moderate content and manage user accounts.

**Background:**

The social media application aims to provide a platform for users to connect, share content, and engage with each other. The application will support user authentication, post creation, commenting, and notifications, fostering a vibrant online community.

**Backend API Specification:**

1. **User Registration**
   * **API Method:** POST
   * **URL:** /api/users/register
   * **Request Body:**
   * {
   * "username": "string",
   * "email": "string",
   * "password": "string",
   * "profile": {
   * "bio": "string",
   * "avatar": "string",
   * "location": "string"
   * }
   * }
   * **Response Body (Success):**
   * {
   * "message": "User registered successfully",
   * "user": { "id": "string", "username": "string", "email": "string" }
   * }
   * **Response Body (Error):**
   * {
   * "error": "string"
   * }
2. **User Login**
   * **API Method:** POST
   * **URL:** /api/users/login
   * **Request Body:**
   * {
   * "email": "string",
   * "password": "string"
   * }
   * **Response Body (Success):**
   * {
   * "token": "string",
   * "user": { "id": "string", "username": "string", "email": "string" }
   * }
   * **Response Body (Error):**
   * {
   * "error": "string"
   * }
3. **Create Post**
   * **API Method:** POST
   * **URL:** /api/posts
   * **Request Body:**
   * {
   * "userId": "string",
   * "content": "string",
   * "image": "string"
   * }
   * **Response Body (Success):**
   * {
   * "post": { "id": "string", "userId": "string", "content": "string", "image": "string" }
   * }
   * **Response Body (Error):**
   * {
   * "error": "string"
   * }
4. **Like Post**
   * **API Method:** POST
   * **URL:** /api/posts/:postId/like
   * **Request Body:**
   * {
   * "userId": "string"
   * }
   * **Response Body (Success):**
   * {
   * "message": "Post liked successfully"
   * }
   * **Response Body (Error):**
   * {
   * "error": "string"
   * }
5. **Comment on Post**
   * **API Method:** POST
   * **URL:** /api/posts/:postId/comments
   * **Request Body:**
   * {
   * "userId": "string",
   * "content": "string"
   * }
   * **Response Body (Success):**
   * {
   * "comment": { "id": "string", "userId": "string", "content": "string" }
   * }
   * **Response Body (Error):**
   * {
   * "error": "string"
   * }

**Frontend Logic:**

1. **State Management:**
   * Use React's useState to manage user authentication, posts, and comments.
   * Example state:
   * const [user, setUser] = useState(null);
   * const [posts, setPosts] = useState([]);
   * const [newPost, setNewPost] = useState({ content: '', image: '' });
2. **API Calls:**
   * Use axios or fetch to make API calls to the backend.
   * Example API calls:
     + **User Registration:**
     + const registerUser = async () => {
     + const response = await axios.post('/api/users/register', userData);
     + setUser(response.data.user);
     + };
     + **Create Post:**
     + const createPost = async () => {
     + const response = await axios.post('/api/posts', newPost);
     + setPosts([...posts, response.data.post]);
     + };
3. **Connecting Backend and Frontend:**
   * Ensure that the frontend makes API calls to the correct endpoints defined in the backend.
   * Handle loading states and error messages appropriately.

**Database Schema and Models:**

1. **User Model:**
2. const mongoose = require('mongoose');
3. const userSchema = new mongoose.Schema({
4. username: { type: String, required: true, unique: true },
5. email: { type: String, required: true, unique: true },
6. password: { type: String, required: true },
7. profile: {
8. bio: { type: String, default: '' },
9. avatar: { type: String, default: '' },
10. location: { type: String, default: '' }
11. },
12. followers: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
13. following: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }]
14. });
15. module.exports = mongoose.model('User', userSchema);
16. **Post Model:**
17. const postSchema = new mongoose.Schema({
18. userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
19. content: { type: String, required: true },
20. image: { type: String, default: '' },
21. likes: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
22. comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment' }],
23. createdAt: { type: Date, default: Date.now },
24. updatedAt: { type: Date, default: Date.now }
25. });
26. module.exports = mongoose.model('Post', postSchema);
27. **Comment Model:**
28. const commentSchema = new mongoose.Schema({
29. postId: { type: mongoose.Schema.Types.ObjectId, ref: 'Post', required: true },
30. userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
31. content: { type: String, required: true },
32. likes: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
33. createdAt: { type: Date, default: Date.now }
34. });
35. module.exports = mongoose.model('Comment', commentSchema);

**Relevant Database Queries:**

1. **Create User:**
2. const newUser = await User.create({ username, email, password: hashedPassword });
3. **Create Post:**
4. const newPost = await Post.create({ userId, content, image });
5. **Like Post:**
6. const post = await Post.findById(postId);
7. if (!post.likes.includes(userId)) {
8. post.likes.push(userId);
9. await post.save();
10. }
11. **Comment on Post:**
12. const newComment = await Comment.create({ postId, userId, content });

This document outlines the necessary components for implementing the social media application based on the provided features and functional requirements. If you need any modifications or additional details, please let me know.

**implement the following/followers feature**

This time I want to implement the following/followers feature . I want you to write the technical design document which should have the feature details: background ,backend Api Specification consisting of api method , url request body,response body for success and error. Then, also write the frontend logic that do we need any states or any API calls,where to add the API calls etc. and also how to connect back-end and front-end ,db schema and models and also write the relevant db queries which will help us in this feature.

**Prompt Engineer**

**Technical Design Document for Followers/Following Feature Background**

The followers/following feature allows users to connect with others in the application. A user can follow another user, and this relationship should be reflected in the database, enabling functionalities such as personalized feeds and social connectivity.

**Feature Details**

**Key Functionalities**

1. **Follow User**: A user can follow another user by clicking a button.
2. **Unfollow User**: A user can unfollow another user.
3. **Followers/Following Lists**: Users can view the list of their followers and the users they are following.
4. **Counts**: Display follower and following counts on user profiles.

**Backend API Specification**

**1. Follow a User**

* **Method**: POST
* **URL**: /api/follow
* **Request Body**:
* {
* "userId": "string",
* "followUserId": "string"
* }
  + userId: ID of the user initiating the follow action.
  + followUserId: ID of the user being followed.
* **Response Body (Success)**:
* {
* "message": "User followed successfully"
* }
* **Response Body (Error)**:
* {
* "error": "User is already followed" // or any relevant error message.
* }

**2. Unfollow a User**

* **Method**: POST
* **URL**: /api/unfollow
* **Request Body**:
* {
* "userId": "string",
* "unfollowUserId": "string"
* }
* **Response Body (Success)**:
* {
* "message": "User unfollowed successfully"
* }
* **Response Body (Error)**:
* {
* "error": "User is not being followed"
* }

**3. Get Followers List**

* **Method**: GET
* **URL**: /api/users/:userId/followers
* **Response Body (Success)**:
* {
* "followers": [
* {
* "id": "string",
* "username": "string",
* "avatar": "string"
* }
* ]
* }
* **Response Body (Error)**:
* {
* "error": "User not found"
* }

**4. Get Following List**

* **Method**: GET
* **URL**: /api/users/:userId/following
* **Response Body (Success)**:
* {
* "following": [
* {
* "id": "string",
* "username": "string",
* "avatar": "string"
* }
* ]
* }
* **Response Body (Error)**:
* {
* "error": "User not found"
* }

**Database Schema and Models**

**User Schema**

Update the User schema to include followers and following references:

const UserSchema = new mongoose.Schema({

username: { type: String, required: true, unique: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

avatar: { type: String },

bio: { type: String },

followers: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }],

following: [{ type: mongoose.Schema.Types.ObjectId, ref: "User" }]

});

**Relevant Database Queries**

1. **Follow a User**:
2. const user = await User.findByIdAndUpdate(userId, {
3. $addToSet: { following: followUserId }
4. });
5. const followedUser = await User.findByIdAndUpdate(followUserId, {
6. $addToSet: { followers: userId }
7. });
8. **Unfollow a User**:
9. const user = await User.findByIdAndUpdate(userId, {
10. $pull: { following: unfollowUserId }
11. });
12. const unfollowedUser = await User.findByIdAndUpdate(unfollowUserId, {
13. $pull: { followers: userId }
14. });
15. **Get Followers List**:
16. const followers = await User.findById(userId).populate("followers", "username avatar");
17. **Get Following List**:
18. const following = await User.findById(userId).populate("following", "username avatar");

**Frontend Logic**

**States**

1. followers: Array of follower data.
2. following: Array of following data.
3. isFollowing: Boolean indicating if the current user follows the profile user.

**API Calls**

1. **Follow/Unfollow**:
   * **Add API Call**:
   * const followUser = async (userId, followUserId) => {
   * try {
   * await axios.post("/api/follow", { userId, followUserId });
   * // Update state (e.g., isFollowing)
   * } catch (error) {
   * console.error(error);
   * }
   * };
2. **Fetch Followers/Following**:
   * **Get Followers**:
   * const fetchFollowers = async (userId) => {
   * const { data } = await axios.get(`/api/users/${userId}/followers`);
   * setFollowers(data.followers);
   * };
   * **Get Following**:
   * const fetchFollowing = async (userId) => {
   * const { data } = await axios.get(`/api/users/${userId}/following`);
   * setFollowing(data.following);
   * };

**Implementation**

1. **Add Follow/Unfollow Button**:
   * Use the isFollowing state to conditionally render the follow/unfollow button.
2. **Integrate API Calls**:
   * Call the followUser function on the button click for follow/unfollow actions.
   * Use the fetchFollowers and fetchFollowing functions when rendering user profile pages.
3. **Profile Page Component**:
   * Render followers/following counts dynamically from the API responses.

**Connecting Backend and Frontend**

1. **API Endpoints**:
   * Use axios or fetch to call the endpoints.
   * Ensure the backend is accessible (CORS or proxy setup for local development).
2. **Global State**:
   * Use Context API or Redux for global user data (e.g., auth, profile, isFollowing).
3. **Error Handling**:
   * Display toast notifications or alerts for errors like network failures or invalid actions.

This design ensures robust implementation of the followers/following functionality while maintaining modularity and scalability. Let me know if you'd like to refine or extend this further!

**Create Supportive Schema for the Project**

**Prompt Engineer**

I already have an auth project that has following user model made from mongoose, now for this social media app I want to build things on top of it, please write all the required models for the application like Post, comment, like etc all the relevant, add the required amount of validations as well keeping in mind the origin project requirements

Below are the **additional models** required for your social media application built on top of the existing User model, adhering to the original project requirements and maintaining robust validations.

**Post Model**

Represents user-created posts, supporting text and optional media.

const mongoose = require('mongoose');

const postSchema = new mongoose.Schema(

{

content: {

type: String,

required: [true, 'Content is required'],

maxlength: [500, 'Content cannot exceed 500 characters'],

trim: true,

},

media: {

type: String, // URL of media file (optional)

validate: {

validator: function (value) {

return /^(http(s)?:\/\/.\*\.(?:png|jpg|jpeg|gif|webp|mp4|avi|mkv))$/.test(

value

);

},

message: 'Invalid media URL',

},

},

author: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

likes: [

{

type: mongoose.Schema.Types.ObjectId,

ref: 'User', // Tracks users who liked the post

},

],

comments: [

{

type: mongoose.Schema.Types.ObjectId,

ref: 'Comment', // References associated comments

},

],

},

{

timestamps: true, // Automatically manage `createdAt` and `updatedAt`

}

);

const Post = mongoose.model('Post', postSchema);

module.exports = Post;

**Comment Model**

Supports hierarchical/nested commenting with parent-child relationships.

const mongoose = require('mongoose');

const commentSchema = new mongoose.Schema(

{

content: {

type: String,

required: [true, 'Comment content is required'],

maxlength: [250, 'Comment cannot exceed 250 characters'],

trim: true,

},

author: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

post: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Post',

required: true,

},

parentComment: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Comment', // Enables nested comments

default: null,

},

likes: [

{

type: mongoose.Schema.Types.ObjectId,

ref: 'User', // Tracks users who liked the comment

},

],

},

{

timestamps: true,

}

);

const Comment = mongoose.model('Comment', commentSchema);

module.exports = Comment;

**Follow Model**

Tracks relationships between users for the followers/following functionality. (Optional as this relationship can also be handled in the User model.)

const mongoose = require('mongoose');

const followSchema = new mongoose.Schema(

{

follower: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

following: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

},

{

timestamps: true,

indexes: [

{ unique: true, fields: ['follower', 'following'] }, // Prevent duplicate follow entries

],

}

);

const Follow = mongoose.model('Follow', followSchema);

module.exports = Follow;

**Like Model (Optional)**

If likes need to be maintained in a separate collection for scalability and analytics.

const mongoose = require('mongoose');

const likeSchema = new mongoose.Schema(

{

user: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

post: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Post',

},

comment: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Comment',

},

},

{

timestamps: true,

}

);

const Like = mongoose.model('Like', likeSchema);

module.exports = Like;

**Notification Model**

Tracks user notifications for events like likes, comments, and follows.

const mongoose = require('mongoose');

const notificationSchema = new mongoose.Schema(

{

user: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User',

required: true,

},

event: {

type: String,

enum: ['FOLLOW', 'LIKE', 'COMMENT', 'REPLY'],

required: true,

},

initiator: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User', // User who performed the action

},

post: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Post', // Optional reference to the associated post

},

comment: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Comment', // Optional reference to the associated comment

},

isRead: {

type: Boolean,

default: false,

},

},

{

timestamps: true,

}

);

const Notification = mongoose.model('Notification', notificationSchema);

module.exports = Notification;

**Database Relationships**

1. **User and Post**:
   * One-to-Many: A user can create multiple posts.
   * Defined via the author field in the Post schema.
2. **Post and Comment**:
   * One-to-Many: A post can have multiple comments.
   * Defined via the post field in the Comment schema and the comments array in the Post schema.
3. **Follow System**:
   * Many-to-Many: Users follow other users. Managed via an array (followers and following) in the User schema or the separate Follow schema.
4. **Likes**:
   * One-to-Many: A post or comment can have multiple likes. Managed via arrays in the Post and Comment models or a separate Like schema.

**Validations Recap**

* Emails must follow a proper format.
* Passwords must be alphanumeric and >= 6 characters.
* Post content must not exceed 500 characters.
* Comments content must not exceed 250 characters.
* URLs must match valid media file formats.
* Relationships should be consistent (e.g., ensure referenced IDs exist).

**Modifying the Auth User Model**

If you'd like, we can enhance the existing user schema to also track followers and following using:

followers: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],

following: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],

Let me know if you’d like a specific adjustment or further optimization!

**Create All POST API and Routes, Controllers**

**Prompt Engineer**

Now I want to prepare all post API such as Create Post, Get Post, Get All Posts, Update Post, Delete Post, search Post write relevant controller method, code, repository method that can help to achieve all this API routes for POST start from /api/posts

Below are the implementation details for **Post APIs**, starting with /api/posts for operations such as create, retrieve, update, delete, and search posts.

We'll follow a layered architecture with **controllers**, **routes**, and **repository methods** to ensure modularity and maintainability.

**Route Definitions**

Define routes in /routes/postRoutes.js:

const express = require('express');

const postController = require('../controllers/postController');

const router = express.Router();

// Post routes

router.post('/', postController.createPost);

router.get('/:postId', postController.getPost);

router.get('/', postController.getAllPosts); // For fetching with pagination

router.patch('/:postId', postController.updatePost);

router.delete('/:postId', postController.deletePost);

router.get('/search', postController.searchPosts);

module.exports = router;

**Controller Methods**

Controller methods are responsible for processing the request, invoking the appropriate repository logic, and sending responses.

**1. Create Post**

const Post = require('../models/Post');

exports.createPost = async (req, res) => {

try {

const { content, media } = req.body;

const author = req.user.id; // Assuming `req.user` is populated via middleware

if (!content) {

return res.status(400).json({ error: 'Content is required' });

}

const post = new Post({ content, media, author });

await post.save();

res.status(201).json({ message: 'Post created successfully', post });

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**2. Get a Single Post**

exports.getPost = async (req, res) => {

try {

const { postId } = req.params;

const post = await Post.findById(postId).populate('author', 'username avatar').populate('comments');

if (!post) {

return res.status(404).json({ error: 'Post not found' });

}

res.status(200).json(post);

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**3. Get All Posts (with Pagination)**

exports.getAllPosts = async (req, res) => {

try {

const { page = 1, limit = 10 } = req.query; // Pagination query params

const posts = await Post.find()

.populate('author', 'username avatar')

.sort({ createdAt: -1 })

.skip((page - 1) \* limit)

.limit(Number(limit));

res.status(200).json(posts);

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**4. Update a Post**

exports.updatePost = async (req, res) => {

try {

const { postId } = req.params;

const { content, media } = req.body;

const post = await Post.findOneAndUpdate(

{ \_id: postId, author: req.user.id },

{ content, media },

{ new: true, runValidators: true }

);

if (!post) {

return res.status(404).json({ error: 'Post not found or unauthorized' });

}

res.status(200).json({ message: 'Post updated successfully', post });

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**5. Delete a Post**

exports.deletePost = async (req, res) => {

try {

const { postId } = req.params;

const post = await Post.findOneAndDelete({ \_id: postId, author: req.user.id });

if (!post) {

return res.status(404).json({ error: 'Post not found or unauthorized' });

}

res.status(200).json({ message: 'Post deleted successfully' });

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**6. Search Posts**

exports.searchPosts = async (req, res) => {

try {

const { query } = req.query;

if (!query) {

return res.status(400).json({ error: 'Search query is required' });

}

const posts = await Post.find({ content: { $regex: query, $options: 'i' } })

.populate('author', 'username avatar')

.sort({ createdAt: -1 });

res.status(200).json(posts);

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Internal Server Error' });

}

};

**Repository Layer (Optional)**

Use a repository layer to centralize database logic in /repositories/postRepository.js:

const Post = require('../models/Post');

// Create a post

exports.createPost = (postData) => new Post(postData).save();

// Get post by ID with populates

exports.getPostById = (postId) => Post.findById(postId).populate('author comments');

// Get all posts with pagination

exports.getAllPosts = (page, limit) =>

Post.find()

.sort({ createdAt: -1 })

.skip((page - 1) \* limit)

.limit(Number(limit))

.populate('author comments');

// Update a post

exports.updatePost = (postId, authorId, updateData) =>

Post.findOneAndUpdate(

{ \_id: postId, author: authorId },

updateData,

{ new: true, runValidators: true }

);

// Delete a post

exports.deletePost = (postId, authorId) =>

Post.findOneAndDelete({ \_id: postId, author: authorId });

// Search posts

exports.searchPosts = (query) =>

Post.find({ content: { $regex: query, $options: 'i' } }).populate('author comments');

**Connecting Layers**

1. **Import Repositories in Controllers**: Call repository methods from the controller to abstract database logic.
2. **Routing**: Add the postRoutes.js to your app.js:
3. const postRoutes = require('./routes/postRoutes');
4. app.use('/api/posts', postRoutes);
5. **Middleware**: Ensure req.user is populated using authentication middleware before accessing routes that require user identification.

This structured implementation ensures scalability and maintains clean separation between the routes, controllers, and repository logic. Let me know if you need additional customizations!