Pennstagram: Team I Vibe with Ives

Shruti Agrawal, Daniel Rohacs, Zora Mardjoko, Patrick Wu

Overview

We implemented an Instagram-like application, Pennstagram, for our final project. The backend services of the system are implemented in Node.js, and these services are hosted on Amazon EC2. We used Amazon RDS to store our data, Amazon S3 to store images, and image search is powered by ChromaDB. We incorporated search using GPT, and our feed contained news updates using Apache Kafka. The feed ranking algorithm used was adsorption, and we used SocialRank for friend recommendations. The frontend of the application is developed using React and JavaScript. All code is managed on GitHub, facilitating collaborative coding.

Technical Descriptions

Search:

To implement search, we used the OpenAI LangChain to embed relevant posts/users (for posts we embedded captions and for users we concatenated their relevant features and embedded those) and picked the top-5 most relevant posts/users to the query. Then, we have a second version of search (which we call Q&A) that can answer questions using RAG. Q&A takes a question and selects the top-5 most relevant posts and uses those as context for GPT-3.5-turbo which answers the question. We also implemented the EC for Q&A which displays the top-5 posts.

Kafka and Feed:

We used Kafka to subscribe and post to the “Twitter-Kafka” and “FederatedPosts” topic respectively. We implemented a handler in our consumer file that pulled from the Twitter topic every hour and updated a running “tweets” database. When querying for a particular user’s feed, we pull from both the posts and tweets databases. We recognized that tweets had far more fields than our posts did, so we screened out the irrelevant entries (or rather, the entries that didn’t have a place in our post handling) in our getfeed endpoint. We use a dummy user (“twitter_user”) that represents pulling from Kafka. We also posted to the FederatedPosts channel via a KafkaJS producer, allowing us to send some posts hosted in our own RDS to other teams.

Friend Recommendations:

To implement friend recommendations, we used a combination of computing friend of a friend recommendations, along with SocialRank. In computing friend of a friend recommendations daily, we are able to maintain a database (which we called "recommendations"), to store the the mutual friend relationships between pairs of users, and their corresponding strengths. For example, if (A, B) and (B, C) are friends, then our recommendations table would store (A, C, 1) to represent that A and C have 1 mutual friend. The second step in our friend recommendation algorithm is to compute the SocialRank across users, similarly to homework 3 milestone 2. After having computed social rank (again, daily), we store our results in a table called "social_rank". In order to compute the friend recommendations for **a given user X**, we take the HIGHEST social rank users from all the users that X has a mutual friend with. We display this result on a user's profile and home pages. This was done locally in Spark Java, so to deploy this functionality we must have a java server running alongside our other servers.

Face Match:

Starting code was given by Professor Ives for basic face match which made implementing this feature a lot more convenient. We used the given code in app.js from the basic-face-match repo for indexing actors and finding the top-k matches. We then modeled our get-actors route based on the main function. The faces are identified and matched using the Face API. Chroma DB is used to "index" the actor images, in other words, store the embedding for the files. Since the images given in the repo are stored locally, we had to find a way to serve them. We setup a static serving directory through express on the backend server so that images can be accessed. Finally, image uploads on the front end are handled using multer.

Frontend:

Our frontend has several pages, including a user login/registration, home page, notifications page, chat page, profile page, settings page, and logout handler. Each page uses their respective endpoints to manipulate and pull from our database to always show accurate information.

Profile:

The profile page displays a user's preliminary information, including profile picture, name, linked actor, friends, and posts. We have two versions of this page, one being the current logged in user, and the second being any other user whose profile you are visiting. The primary difference between the two is that on your own profile page, you have the option to edit your profile/settings, and on another user's profile, you have the option to send them a friend request. These profile pages also offer an intuitive interface, very similar to that of actual Instagram, and allow users to view all information about a particular post (comments, likes, caption, image content) in a pop up modal when clicked.

Feed/Home:

Our feed page displays a search bar, running list of posts, creating post functionality, and friend recommendations. The running list of posts (i.e. the feed), allows users to view/add comments and like/unlike posts. The create post functionality opens a pop up, where users can input any valid information to create a new post, and update the feed accordingly. Friend recommendations is the last section of the home page, which simply lists any friend recommendations for the user, and links each username to the respective profile.

Notifications:

We have a notifications page that displays all the requests associated with the user. Whenever a user requests someone, the requestee will get a request in their notifications page. They can then choose to either accept or reject the request. Accepting will add the requester to the requestee's friends list, and rejecting will remove the request.

Chat:

We modeled our chatting between users from the chat interface given in homework 4. We implemented at first with axios requests which refreshes the chat each time the user opens it and send a message. Timestamps are kept within the database in order to preserve when messages were sent. Users can chat with just themselves, with one other user, or within a group chat. For extra credit, we redid the chats with websockets using socket.io to keep a persistent connection between the client and the server so that data can be exchanged in both directions at any time.

Adsorption:

We utilized information posted on Ed and also on the writeup to implement this algorithm for getting a user's feed.

Design Decisions

Tables:

We decided to exclusively use RDS. Much of our data is relational, (i.e. lots of attributes for users, users make posts, etc.) and we thought it would overall be easier than having a separate setup for DynamoDB/other no-SQL databases. Perhaps Dynamo would've been better for something like chat if we scaled further, but for simplicity, we stuck with RDS. Most of the backend table setup was pretty straightforward, we typically assigned things a unique ID (users, posts, groups) and used those as foreign keys when necessary.

Additionally, we had originally contemplated the idea of using a combined table named "content" to store both posts and comments since they share many common attributes. However, we encountered several challenges in implementing and maintaining this approach. So, we opted to split posts and comments into two separate tables.

We also used tables for several intermediate setup steps for other more complex algorithms in our project, including friend and post recommendations. For example, while computing friend recommendations, we stored the recommendations and social_rank in intermediate tables (while running the Spark Java server), and then created a Javascript endpoint that would be able to pull from these tables to reach a final conclusion. We found that using tables as an intermediary, a shared resource across Spark Java and Javascript, allowed for a smooth and sensible workflow.

S3 Folder Structure for Organization: We opted for a structured approach to organize photos within the S3 bucket. User profile photos are stored in individual folders named after the respective usernames. Similarly, feed photos are stored in folders labeled with unique post IDs, simplifying the retrieval process.

Friend Recommendations: As computing friend recommendations was quite freeform in this project, we decided to use a combination of friend of a friend recommendations along with SocialRank. We noticed that SocialRank computed the "rank" of each user, irrespective of any GIVEN user. This meant that via computing SocialRank alone, we would not have been able to generate a unique list of recommendations for every user. Therefore, as SocialRank was the same across all users, we incorporated friend of a friend recommendations (mutual friends) to narrow down and personalize the list of recommendations per user.

Reflection and Lessons Learned

Planning/Documentation/Communication:

We found that extremely detailed planning is vital in the process of developing this app. We tried to our best to fulfill the milestone requirements and split work among each other to present to our TAs during check ins, but we definitely feel that additional planning would have been better. We especially ran into merging issues on the frontend, since it was later into the process and we did not plan as carefully when arguably the frontend should have required more. We split different pages of the frontend amongst ourselves, but this results in the styles of pages not being completely uniform and the roughness with the user flow of our app. We had to spend extra time combing to work out these issues and match our different pages with each other. A better way to approach this would have been to establish a common CSS style template that we would agree to use. One thing we did that helped was we drew a detailed diagram of our app on a chalkboard, and then all worked based on our diagram.

Using Git:

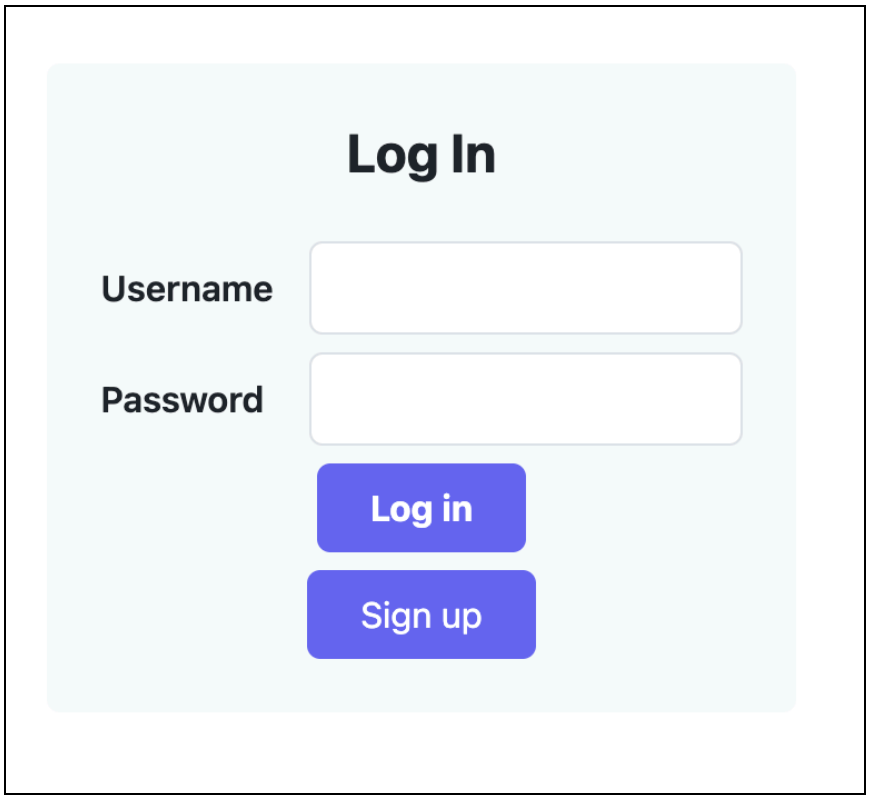
Using Github for this project resulted in several potential sources of confusion. With multiple branches, keeping track of feature branches, and the main branch can become confusing. A few times, it was not managed well, leading to merging the wrong branches or difficulties in tracking which features are ready for deployment. Additionally, merge conflicts were common. Resolving these conflicts were tricky, especially for those of us who were not familiar with Git commands.

Extra Credit

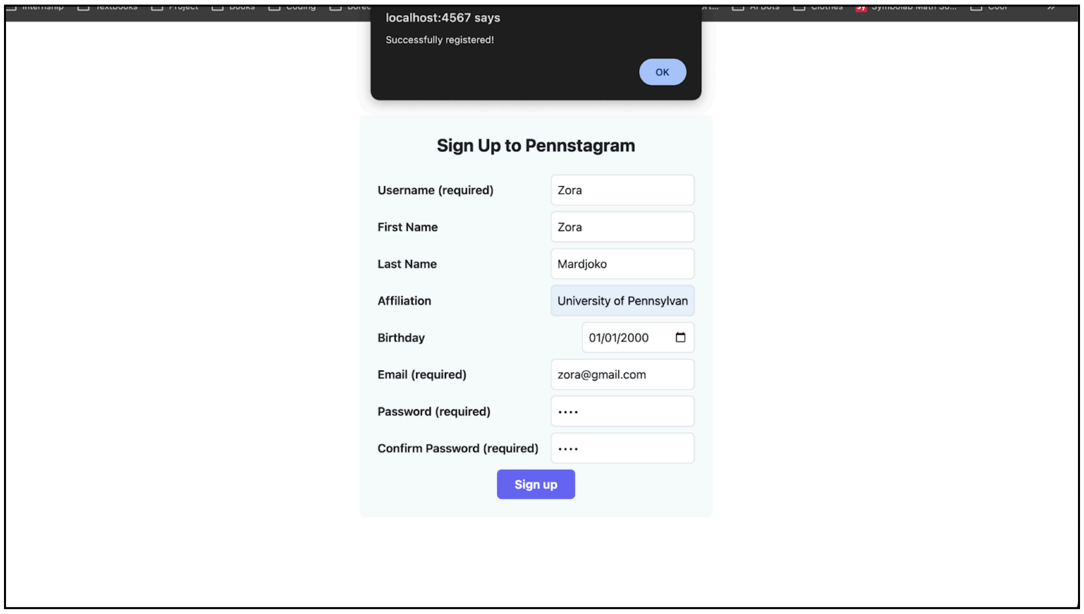
For our project, we implemented the following extra credit functionality: websockets, dark mode, infinite scroll, friend requests, getting posts using search.

Screenshots

Login



Signup



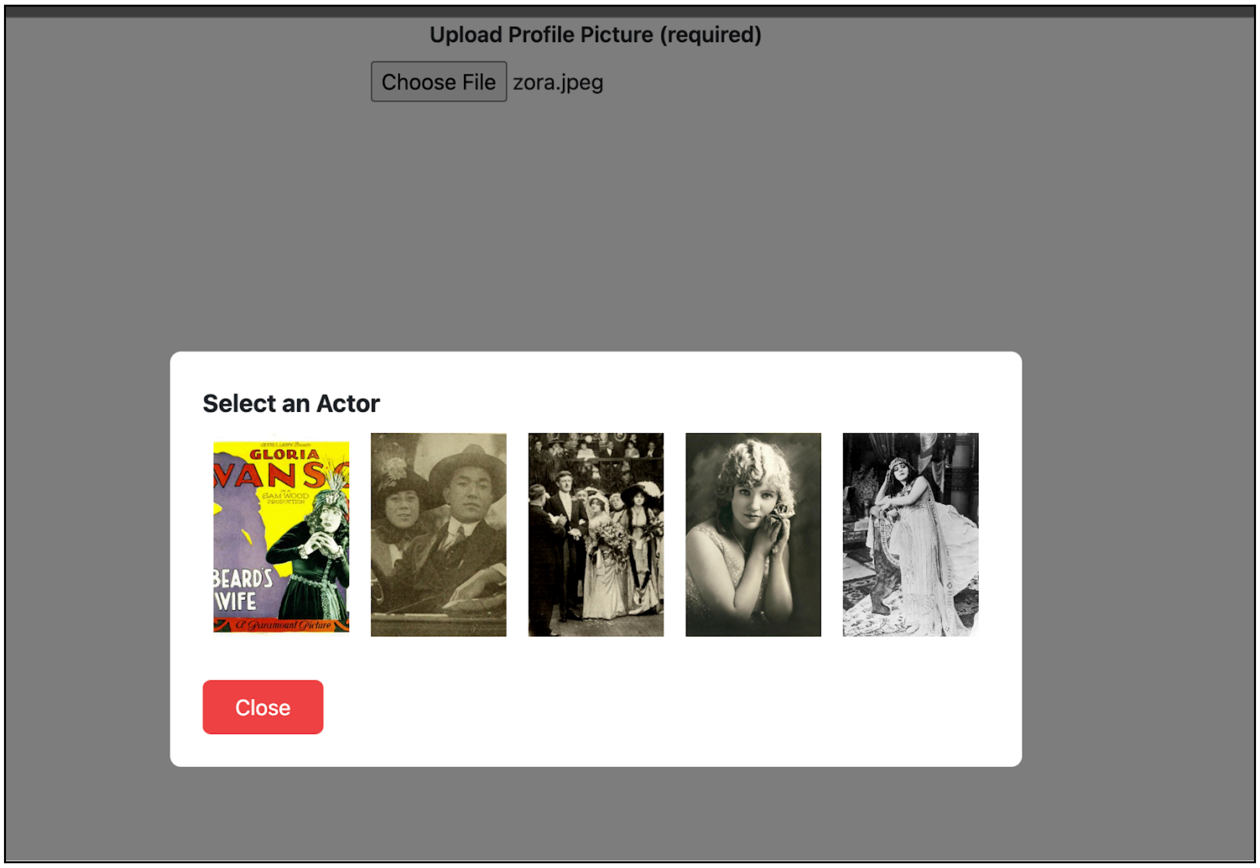
localhost:4567 says

Successfully registered!

Sign Up to Pennstagram

Username (required)	Zora
First Name	Zora
Last Name	Mardjoko
Affiliation	University of Pennsylvan
Birthday	01/01/2000
Email (required)	zora@gmail.com
Password (required)	••••
Confirm Password (required)	••••

Upload Picture and Choose Actor:

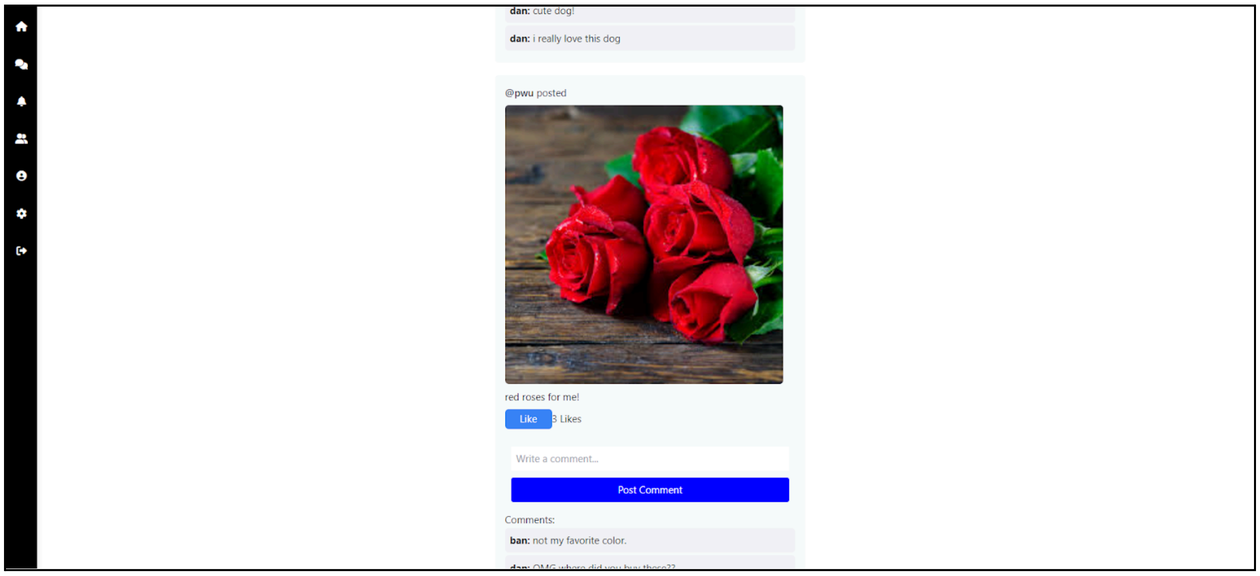

Upload Profile Picture (required)

Choose File zora.jpeg

Select an Actor

Close

Home Page:



dan: cute dog!

dan: i really love this dog

@pwu posted

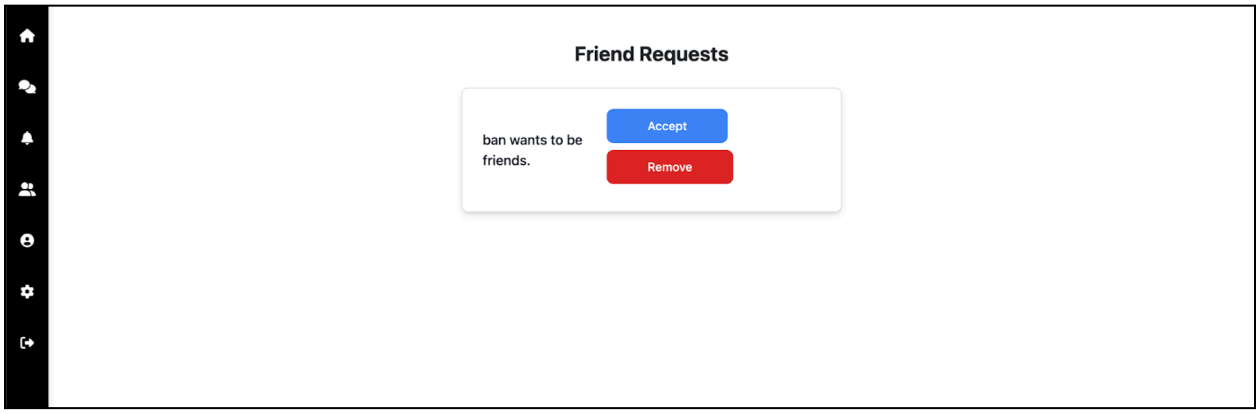
red roses for me!

Comments:

ban: not my favorite color.

dan: OMG where did you buy these??

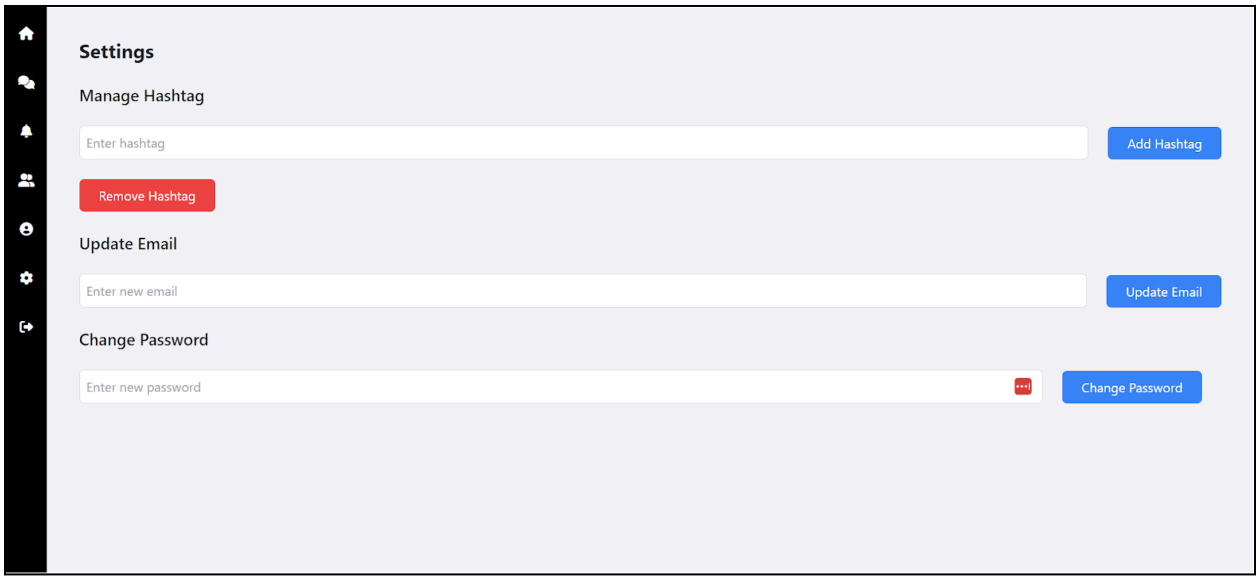
Notifications



Friend Requests

ban wants to be friends.

Settings:


Settings

Manage Hashtag

Enter hashtag

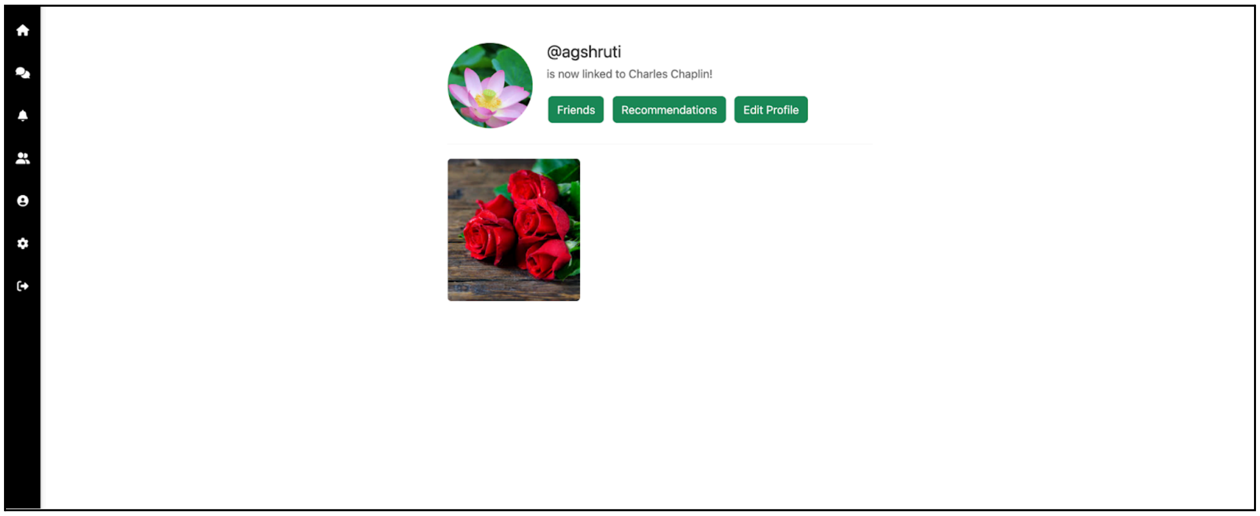
Update Email

Enter new email

Change Password

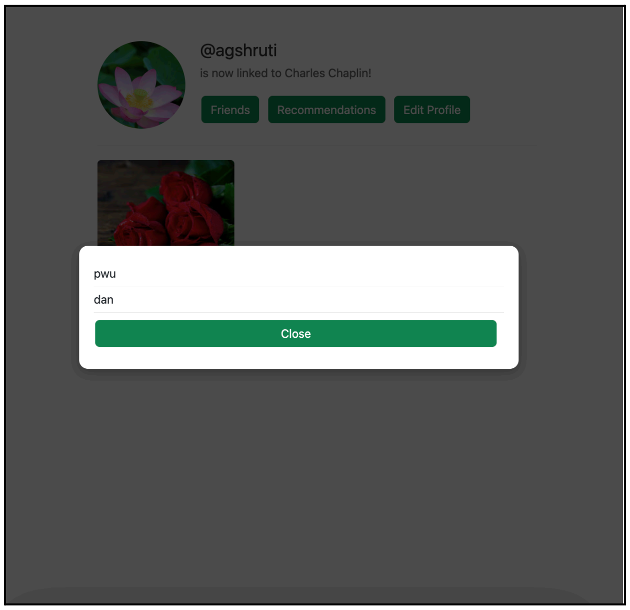
Enter new password

Logged In User Profile:


@agshruti

is now linked to Charles Chaplin!

Friends + Recommendations Popups:

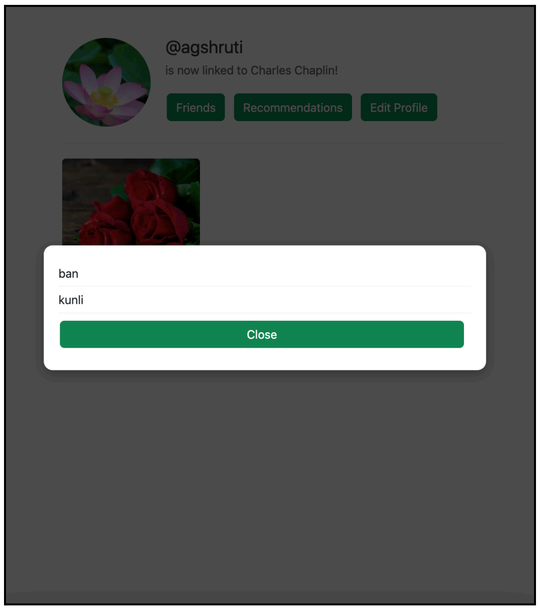


@agshruti

is now linked to Charles Chaplin!

pwu

dan



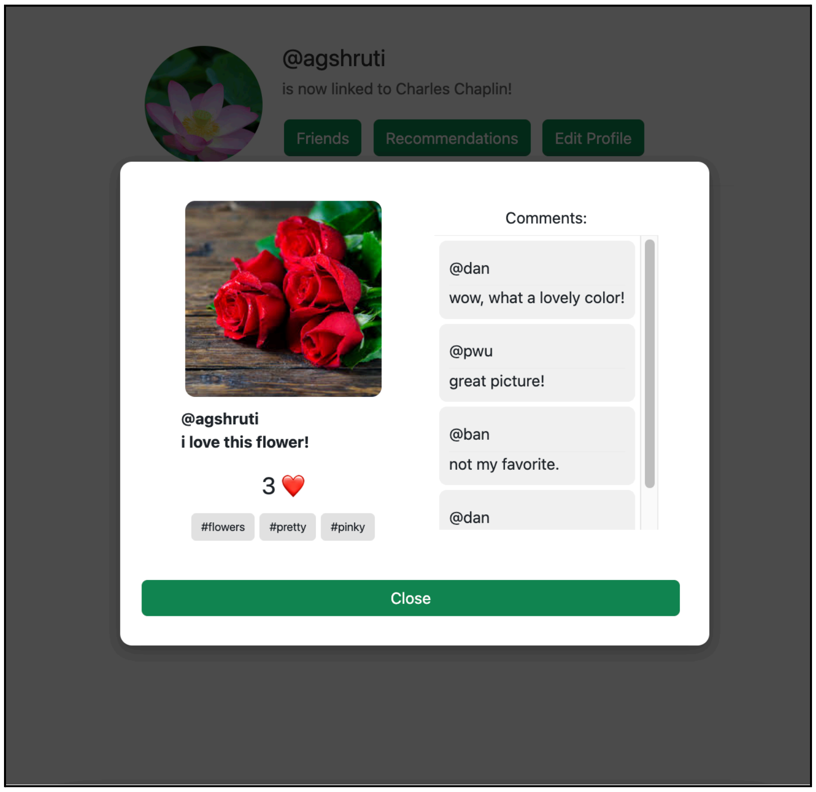
@agshruti

is now linked to Charles Chaplin!

ban

kunli

Post Popup:



@agshruti

is now linked to Charles Chaplin!

@agshruti

i love this flower!

3 ❤️

#flowers #pretty #pinky

Comments:

@dan

wow, what a lovely color!

@pwu

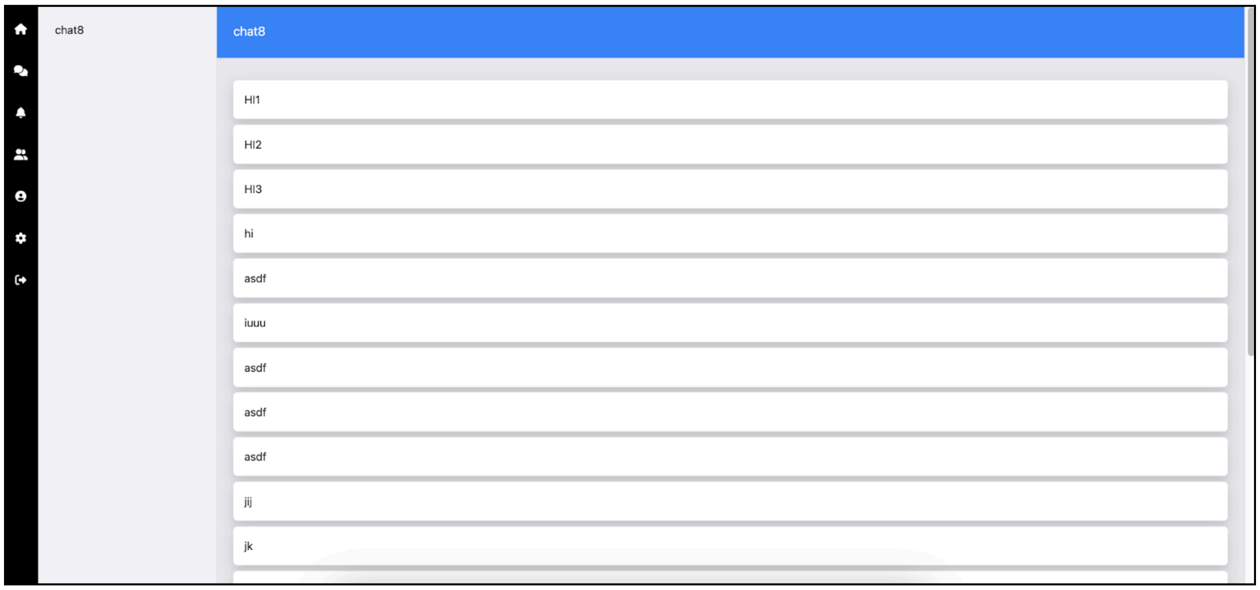
great picture!

@ban

not my favorite.

@dan

Chat Page:


chat8

chat8

HI1

HI2

HI3

hi

asdf

iuuu

asdf

asdf

asdf

jij

jk