

# Best Network

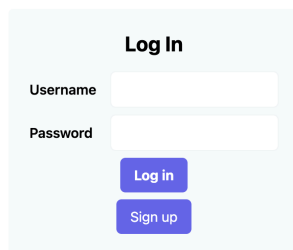
## NETS 2120 Final Project Report

Julia Susser, Vedha Avali, Kimberly Liang

### Feature Overview,

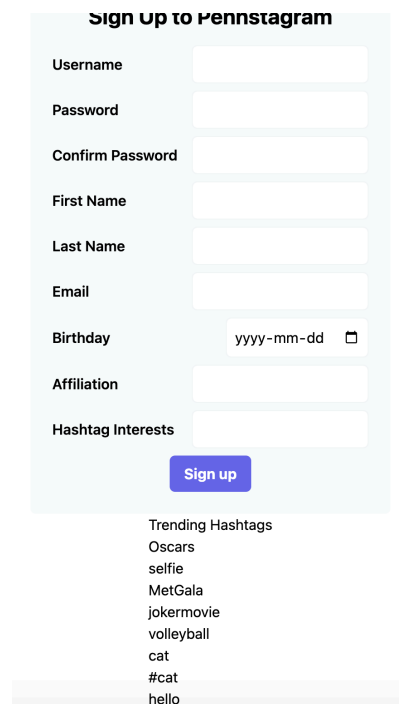
## User Signup and User Accounts

### User registration & User login functionality



A light blue rectangular form titled "Log In" in bold. It contains two input fields: "Username" and "Password". Below the "Password" field are two buttons: a purple "Log in" button and a purple "Sign up" button.

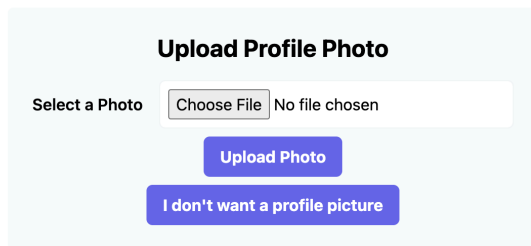
When first logging onto the site, users are taken to a page where they can either log in with their existing account or register a new account. When a user logs in, it compares their username and password to a SQL table “users,” which is stored on an RDS database. Upon clicking the “Log in” button, the values inputted into the text fields are compared to the username and hashed password stored in the database.



A light blue rectangular form titled "Sign up to Pennstagram" in bold. It contains several input fields: "Username", "Password", "Confirm Password", "First Name", "Last Name", "Email", "Birthday" (with a date picker showing "yyyy-mm-dd"), "Affiliation", and "Hashtag Interests". Below the "Hashtag Interests" field is a purple "Sign up" button. At the bottom of the form, there is a section titled "Trending Hashtags" with a list of hashtags: Oscars, selfie, MetGala, jokermovie, volleyball, cat, #cat, and hello.

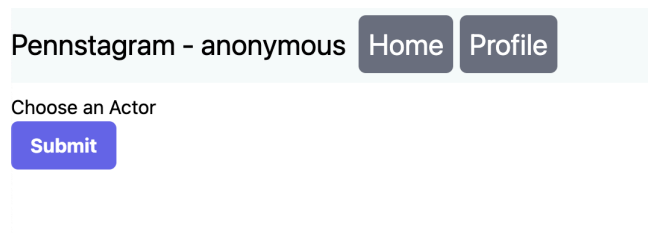
If they instead want to create a new account, they can click the signup button and be taken to this page, where they can input all of the necessary information needed to create a new account. In the backend, we first check if the username they input is already in use, and if it is, they are prompted to reenter a new username. We also check if the two values in “password” and “confirm password match.” In the database, we use password encryption and salting and store only the hashed password in the database to add a layer of security to our user accounts. Aside from the Hashtag Interests, all of the information in the signup page is stored in the users table. A separate “hashtags” table stores an ID, count value, and the hashtag text itself for each hashtag on the server. The count value is incremented everytime a user adds that hashtag as one of their interest or when a hashtag is used in a post. A separate “hashtagInterests” links users to their hashtag interests. Each row stores a user’s ID and the ID of the associated hashtag that they have selected as an interest. The “Trending Hashtags” displayed include a list of the ten most popular hashtags on our application, which are ranked by their “count” values.

## Profile photo upload



Next, the user has the option to upload a profile picture. If they choose to do so, the selected file will be sent in a request to our backend servers using multipart-form data. In the backend, we used the multer package to extract the file from the request, and we then upload it to an S3 bucket. We use an S3 bucket to store all of our images associated with profile photos, posts, and actors. Our S3 bucket has separate folders for each of these image types, and the profilePhotos folder names the image files with the users userID.

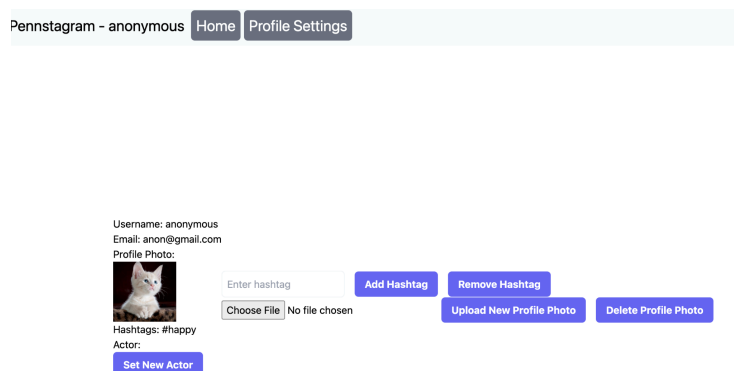
## ChromaDB & Actor Account Linking Based on Profile Picture Embeddings



Upon submitting a profile picture, the user will be redirected to a page where they can select an actor to link their account to. The actor embeddings are calculated locally, but the images are stored on and displayed from S3. Additionally, we parsed the names.csv file and used it to populate a new RDS table titled

“actors” which stores the actors names, nconst, and birth and death years. This allowed us to more easily access an actor’s name given the nconst provided in the image file.

## Profile Page: Actor change, email/password update, hashtag interests update



Next, we have a profile page that the user can access from the home screen. This shows all of the necessary information about their profile, and from here, they can make updates such as adding and removing hashtags, changing their profile photo, or clicking the “Set New Actor” button to be taken back to the page displayed above. Additionally, the recommended hashtags (calculated via Spark) are displayed, and they can visit the Profile Settings page to make further

changes to their profile (changing username, password, first/last name, etc).

Pennstagram - kimiliang [Home](#) [Profile Page](#)

Profile Settings

Username

First Name

Last Name

Email

Birthday

Affiliation

Password

Confirm Password

Change Username

Change First Name

Change Last Name

Change Email

Change Birthday

Change Affiliation

Change Password

## Friends

Pennstagram - anonymous [Home](#) [Feed](#) [Chat](#)

Enter friend's name

Add Friend

Enter friend's name to re

Remove Friend

anonymous's friends

currently online

No online friends

anonymous's friends

currently offline

No offline friends

anonymous's

recommended friends

No recommended friends

From the home page, the user can visit the Friends page where they are able to type in usernames to add and remove friends.

Additionally, three lists are displayed. The first contains all of the user's friends who are currently logged in. The second contains all of the user's friends who are not logged in. The third contains friend recommendations, which are calculated

with Spark. When a user adds friends, it adds a column to our "friends" table on RDS, which has two rows, followed and follower. Since friendships are bidirectional in our application, when an arbitrary user with ID A adds a user with ID B to the table, it adds the columns (A, B) and (B, A). We also ensure that users cannot add existing friends, and removing a friend will remove both relationship columns from the table.

## The User Page / Main Content Feed

### Instagram Style Feed

The feed is designed to show the posts that the user posts, the recommended posts from friends, and the federated posts which are both coming from the "FederatedPosts" topic and the "Twitter-Kafka" topic. We have also implemented an endless scroll functionality for the feed so that users can see all of the posts possible rather than have to click through pages.

## User Post Creation with Optional Image and Text

[illegible]

Next, you can create a post with an optional image or text. The code does a regex to extract the hashtags as an array from any of the content that is posted with # and the feed will refresh after the post is complete. On submit, the post will do two separate calls to the backend: 1. createPost to actually update the posts table and generate a new row entry with the information given 2. uploadPost to create an s3URL if there is an image that was uploaded (we handle the file change through a multipart form which is sent to the backend and processed with a multer) such that the s3URL will contain the post id (we do a select to find the last post id and increment it by one to ensure it matches). We then add the post id to the posts table through a SQL query and the s3URL will be expressed as “image.” We then display this into the post component when it is reloaded along with the title, content, username, option to like, and comment on the post.

A screenshot of a social media post. At the top, the username '@g13-kipling' is followed by 'posted' and the text 'so close'. Below this is a large, vibrant illustration of a smiling orange crab with large eyes, set against a blue background with coral and other fish. Under the illustration, there is a 'Like' button and the text 'Likes: 0'. At the bottom, there is a text input field with the placeholder 'Add a comment...' and a blue 'Add Comment' button.

## Commenting Functionality

Comments are done as a sub-post of another post (they are still tracked in our posts table but their parent id will connect to the post that the comment refers to). The comments simply only have the text and the username displayed and the rest of the values are set to null in the table. For every post on the feed, there is a fetchComments function in order to keep up to date with the comments on the post before showing it on the feed. Users can comment as many times on the same post as they would like including on federated posts as shown below.



## User Actions: Liking, Commenting, and Hashtag Linking

Users can also like posts and add hashtags to the posts. We are able to implement this by having tables that link a user's like to a post, hashtags to a post, and users to the hashtags that they are interested in. Everytime a user likes a post, it will update the like count of that post element, and also add to the linked post and user table. When a user lists a hashtag as an interest, that hashtag will be linked to that user. When a post contains a hashtag, then that hashtag will be linked to that post. Comments are just posts without pictures and include a parent post id. The parent id links that comment to what post it falls under.

## Feed Updates and Ranking Posts

The feed for each user is decided by the rank of other posts and users, which is calculated by an adsorption algorithm. The algorithm is detailed below. After the ranks are calculated, three tables hold the ranks of posts, users, and hashtags, which have a label and label value that shows how likely that user in the label would like that post, user, or hashtag. The feed then selects from those tables, the users own posts, the federated and twitter posts and the posts from the people that users follows and returns it to the feed.

## Kafka & Federated Posts

### Reading and sending posts via Kafka

Kafka has a producer and consumer for the creation and read in of federated posts and a consumer to handle tweets. When users make an account, they are automatically subscribed to the "FederatedPosts" and the "Twitter" topics. The system is built using Express.js for server-side development and KafkaJS, and uses Snappy compression codec to enhance message compression efficiency, especially since there are so many incoming tweets per hour.

### Producer Code:

The producer component of the system is responsible for generating messages and sending them to Kafka topics. It initializes a Kafka producer instance and provides a function called `sendFederatedPost` for sending federated post messages. This function constructs a JSON object representing a federated post, serializes it, and sends it to the 'FederatedPosts' Kafka topic. A separate function, `runProducer`, is provided to connect the producer to the Kafka brokers and start producing messages.

The code is implemented such that you can make a federated post on the frontend in a separate page than the normal posts, and all of these posts will be broadcasted across the topic. The frontend will then call a backend route which is registered in `kafka_routes.js` which is an await for the function `sendFederatedPost` (where the `async` is in the `kafka` file itself). The `runProducer` will then send the message to the topic. The image upload is more complicated and will be explained below, but we send HTML tags as the image to be processed by groups for the frontend.

### Consumer Code:

On the consumer side, the system subscribes to Kafka topics ('FederatedPosts' and 'Twitter-Kafka') and processes incoming messages through the `handleMessage` function which is responsible for parsing incoming messages, determining their topic (and handling it based on that), and calling appropriate handler functions. For federated posts, the `handleFederatedPost` function processes the message, checks user existence, registers new users if necessary with some filled in dummy variables in items like birthday, etc., and creates posts in the system through calling the `createPost` API. For tweets, the `handleIncomingTweet` function processes tweet messages, checks user existence, and creates posts in the system with tweet data, filtering only for the username, content, and parsing the hashtags if there are any. It will also call the same function to create a post. The `runConsumer` function connects the consumer to Kafka brokers, subscribes to topics, and starts consuming messages. Since federated posts come with no titles, we set the title of the post automatically as "Federated Post" or "Tweet" depending on which one it is and use that to identify whether it should be shown on the feed. To separate between users who may have the same username but are in different sites (Kimberly on site 1 and Kimberly on site 2), we create a new "federated username" for the user where we append their group name before their username as well in order to both identify which site it came from as well as avoid potential issues if the same people with the same usernames were on different NETS2120 sites.

Here is a response of a successful creation after receiving the message on the kafka server for a tweet:

```
Post created successfully :)
TwitterUser-21217034
Received tweet from author ID 21217034: Kim Kardashian was booed during a live taping of Netflix's "The Greatest Roast of All Time: Tom Brady," and despite the moment going viral on the internet, you won't see it on the streaming platform, as it was edited out https://t.co/TKg29AawJy
Request body: {
  username: 'TwitterUser-21217034',
  parent_id: null,
  hashtags: [],
  title: 'Tweet',
  content: 'Kim Kardashian was booed during a live taping of Netflix's "The Greatest Roast of All Time: Tom Brady," and despite the moment going viral on the internet, you won't see it on the streaming platform, as it was edited out https://t.co/TKg29AawJy'
}
```

### JSON format for posts

The posts are sent through JSON objects and use the `"JSON.stringify"` function before sending and we also parse the JSON messages using `JSON.parse` upon receipt. We recognize that not all groups may send JSON formats for posts so we have error handling with a `console.error` when that happens so that we are aware why a post was not successfully created.

## Image attachments

The image attachments are handled using an optional variable called “attach” as specified in the instructions in which the images are handled with tags being sent. We first handle the file change similarly to how image uploads are handled for normal posts, create an S3url, parse that into an HTML tag with src being the S3URL, and send the tag through the attach variable. On the receival end, we check if the attach variable is not null, undefined, or an empty string, and parse the S3URL, attaching it to the corresponding post if it is correct. We then display it on our end through the feed.

# Spark Java Recommendation System

## Feed updates and ranking posts

```
project-best-network > spark-network > vertexLabelRankings.csv
1 vertex_id,type,user_id,weight
2 4,u,3,0.047438760198828554
3 4,u,8,0.1108421024167598
4 4,u,13,0.054654354231849306
5 4,u,1,0.7678116042272521
6 4,u,6,0.009626589462655138
7 4,u,30,0.009626589462655138
8 5,h,1,0.6295999027253254
9 5,h,30,0.007578446397146066
10 5,h,13,0.04425959884199382
11 5,h,8,0.2724335626306827
12 5,h,6,0.007578446397146066
13 5,h,3,0.03855004300770608
14 26,h,6,0.05985930140955042
15 26,h,2,0.8195598003470556
16 26,h,30,0.05985930140955042
17 26,h,4,0.048012050240774276
18 26,h,8,0.012709546593069445
19 27,p,18,0.5456561344753339
20 27,p,9,0.4543438655246662
21 6,u,8,0.0020384294762565156
22 6,u,13,0.06296946831588367
23 6,u,3,0.05404207740290465
24 6,u,1,0.032394407603687475
25 6,u,6,1.0
26 6,u,30,0.4242778086006338
27 11,h,6,0.001664211106336964
28 11,h,3,0.012882907841690841
```

In Spark Java, we ran the adsorption algorithm. The core functionality is in package project.best.network.local in file runSocialNetwork.java.

## Creating Social Network

### Fetching User, post, hashtag relationships from RDS Database

The algorithm first calls `getSocialNetworkFromJDBC()`, which fetches data from RDS regarding social network relationships and stores them in bidirectional format in a list. This list is then transformed into a `JavaRDD` and subsequently into a `JavaPairRDD` to create a distributed dataset suitable for processing with Apache Spark. The Java Pair RDD returned will have the key be the first vertex and the value be the second vertex. A vertex is a `Tuple2<Integer, String>` where the Integer is an id (user\_id, post\_id, hashtag\_id) and the String is a type (“u”, “p”, “h”) denoting user, post or hashtag. The relationships include likes from users to posts where the posts have no parent (original posts), user interest in specific hashtags, and friendships between users. Each tuple in the network graph represents a link in the social network, illustrating interactions such as likes, interests, and friendships, which are essential for further analysis and algorithmic processing in Spark.

### Graph Edge Weight Initialization

Then, the algorithm is called `computeEdgeRDD()`, which performs edge computation of the social network RDD. This converts it into `JavaPairRDD<Tuple2<Integer, String>,`

`Tuple2<Tuple2<Integer, String>, Double>>` , so that the edge is the key and the weight (double) is the value. An example edge would be: `<Tuple2<user_id1, "u">, Tuple2<Tuple2<hashtag_id, "h">, Double>>`. To normalize the edges, the algorithm splits into five different rdds depending on edge type (posts, hashtags, user->posts, users->users, user->hashtags). Then for each rdd, first aggregates the total weights for all outgoing edges a `reduceByKey` operation. This operation sums up all the initial weights (1s) for edges originating from each post node, resulting in the total count of outgoing connections for each post. Then, the method calculates the normalized weight for each outgoing edge from a post by taking the reciprocal of the total outgoing edge count ( $1 / \text{total\_edges}$ ) or (.3 in the case users->users/hashtags or .4 in the case of users->posts). Then, the weights are merged into the original dataset and the five different RDDs are unioned to create one `JavaPairRDD` social network edgeRDD.

Next, the algorithm is called using adsorption propagation.

Graph Initialization: First, I initialize a `JavaPair RDD` of the `userLabelsMapped`. To do this, I first call a function to get all user IDs in the graph. Then for each user, I give them a label with their own user ID and a weight of 1.0. The labels are then used to initialize the `vertexLabels RDD`, which will store the current state of labels for each vertex as the algorithm progresses. `VertexLabels` is a where `JavaPairRDD<Tuple2<Integer,String>, Tuple2<Integer, Double>>` wear `<vertex1, <label, weight>>` and vertex is `<user_id,"u">` or `<hashtag_id,"h">` or `<post_id,"p">`.

## 2. Label Propagation Loop

The algorithm enters a loop that will execute up to 15 times or until convergence is achieved (whichever comes first).

In each iteration, labels are propagated across edges:

In one iteration of the propagation, the `vertexLabels RDD` is joined with the `edgeRDD`, linking a `<vertex1, <label, weight>>` pair from `vertexLabelsRDD` and a `<vertex1,vertex2>` pair from `edgeRDD`. After the join operation, the algorithm then performs weight calculation where the label weight is multiplied by the edge weight to calculate the propagated weight and returns `<<vertex2,label>, propagated_weight>`. Then, the algorithm aggregates by `<vertex,label>` (key) the weights for labels propagated to the same vertex are aggregated using a `reduceByKey` operation, to find the sum of the weights for a given label at vertex.

## 3. Normalization of Weights

The sum of all propagated label weights for each vertex is computed.

Normalization: Each propagated label weight is then normalized by dividing it by the total weight sum for the corresponding vertex, ensuring that the sum of outgoing weights for each vertex equals

## 4. Maintenance of Original User Labels

Self-Label Weight Reset: The code iterates through the `propagatedVertexLabels RDD`, identifying and resetting the label weight to 1.0 for entries where the vertex represents a user ("u") and the label ID matches the vertex ID.

Reintroduction of Missing Self-Labels: After the weight reset, the code identifies any user vertices that have lost their self-labels during the propagation phase by performing a subtraction operation (`subtractByKey`) between the initial user labels set (`userLabelsMapped`) and the modified `propagatedVertexLabels`. It then merges (union) these missing labels back into the



propagatedVertexLabels RDD to ensure all user vertices have their self-labels, preserving the completeness of the dataset for subsequent iterations or analyses.

### **5. Convergence Check**

Difference Calculation: After updating the labels, the algorithm calculates the maximum absolute difference in label weights between the current iteration and the previous iteration for each vertex.

Termination Condition: If the maximum difference is less than or equal to a predefined threshold ( $d_{\max} = 0.1$ ), the algorithm concludes that it has converged and exits the loop.

### **Output Handling**

Then calls `createAndClearTables()` and `sendResultsToDatabase()` method to take final vertex label rankings from the adsorption propagation algorithm and inserts them into the database.

This creates three tables `socialNetworkFriendRecommendations`, which stores potential friend recommendations (columns: `user_id`, `userlabel_id`, and `weight`), `socialNetworkHashtagRecommendations`, which stores hashtag recommendations for users (columns: `hashtag_id`, `userlabel_id` and `weight`) and `socialNetworkPostRecommendations`, which stores post recommendations to users (columns: `post_id`, `userlabel_id` and a `weight`). The `userlabel_id` and `weight`, is the probability that `userlabel_id` arrives at vertex (`hashtag_id`, `user_id`, `post_id`) in random walk so it is `user_label_ids` recommendations

### **Hourly updates on who and what to follow**

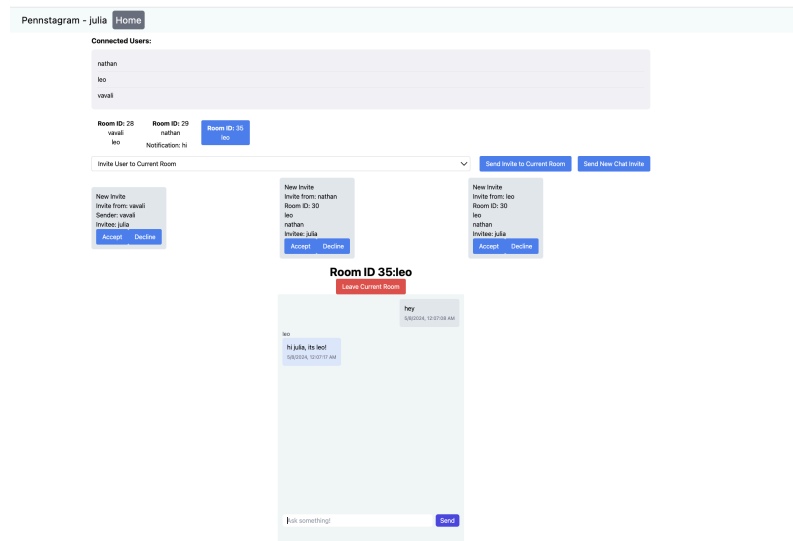
On the EC2 cloud instance, we ran an hourly cronjob which ran the entire spark algorithm and rewrote the recommendations table.

## **Natural Language Search**

### **Search functionality for people and posts, retrieval-augmented generation, Large Language Model usage**

The natural language search uses ChromaDB, langchain, and OpenAI tools to create a search interface that uses the data from the posts as context to answer the user's questions. The steps to implement this were first indexing the data and loading it into a ChromaDB collection. This starts with making a collection that has a text embedder. Then whenever someone posts, the text content of that post is turned into embeddings and put into the chroma collection. Then we got the vector stores from the collection and made a retriever. Afterwards, the asynchronous function used for the search will take in an input. We then take in a prompt telling how the LLM should use the context and question. The context is derived from the retriever which is based off of the embeddings in the collection, therefore, the LLM knows to answer the question based on the content of the posts. This entire process of creating a context and question is the process of making a rag chain, or retrieval and generation. Given the user's input, we invoke the rag chain and are able to get a result, which is an english answer based on the posts.

# Chat Mode



## Group and individual chat creation:

The ChatPage uses Socket.IO for real-time communications between the backend server (socketHandler.js) and the frontend (chatPage.tsx)

## Initialization and State Management

The react frontend stores the current state of the chat using many useState objects including checking if user is logged in (isLoggedIn), a list of usernames of connected users (connectedUsers), a list of chat rooms that user belongs to (rooms), the current room user is looking at on the screen (currentRoom), the messages in the current room (messages), the current message that the user is typing (currentMessage), and a list of invites (inviteUsername, incomingInvites).

A key feature of the frontend and backend is the Room object which stores the roomId, a list of users in the room, and (just in frontend) a notification if a new message appears in the room.

The backend stores a list of connected users (socket id associated with username) and a list of pending invites.

## Initialization & Socket Connection:

When the ChatPage component mounts, it makes an HTTP GET request using Axios to check if the client is logged in (which sends cookies with credentials). If the response 202 OK confirms that the user is logged in, the state isLoggedIn is set to true. If the check fails or an error occurs, it might redirect the user or display a login error. The socket connection with the server is initialized (via port 8080). Upon successful login check, the client emits a send\_username event via Socket.IO, which helps the server associate the current WebSocket connection with the user's username. Then, the server emits notification to all other connected users that a new user connected via socket.

Additionally, the server emits a response to the client with the current rooms they belong to and a list of current connected users. The frontend then sets their current room to default first room and

requests messages for that room from the backend via socket. The server also sends all pending room invites specific to the user, which are stored in a list on the backend (when the server stops running, the pending invites are reset).

If a user is reconnecting via new socket (ie. after page reload), the backend will appropriately update the socket\_id of the user in the list of connected users, send user specific information to the reconnected client, but will not notify other connected users of new connection.

## **2. Invitation Handling**

Storing and Managing Invitations:

First, invitations are sent from the client. For an invite to be sent, the invite handlers check if the invited user is a connected user. For individual invites, the handler checks if users are already in a room together, For group chat invites, the handler checks if the invited user already belongs to the current room. The server, upon receiving a notification of new invite, then finds the socket id of the username being invited and emits notification to the invited user. On the frontend, new invitations are received and added to the incomingInvites state. Invites are checked to ensure no duplicates exist in the list, so that client does not see two of the exact same invite from the same user.

## **Creating New Rooms:**

If a new chat invitation is accepted, the server sends a query to the database to create a new room with auto increment id and also adds users to the table with chatRoom Users. Tell the sender of the invitation to join the socket channel. Notify both the invited user and the sender that the user has joined.

If group chat, then just add new users to chatRoom users. Have them join a socket channel with roomID so that message notifications can be sent to the room. Notify other current users in the socket channel that a new user has joined.

## **3. Leaving a Room**

When a user decides to leave a room, the frontend clears the current room state, removes the room from the user's list, and emits a leave\_room request. The server then updates the database, if there are no more users in the room, then it deletes the room, and notifies other users in the room of the departure.

## **4. Messaging and Persistence**

### **Sending Messages:**

The sendMessage function is called when a user submits a message. This function emits a send\_room\_message event with the room details, message content, and sender's username. The server then saves the message to the database and broadcasts it to other users in the room.

### **Persistent Messages:**

Messages are stored persistently in the database using a table structure that likely includes columns for messageID, roomID, message, userID, and timestamp. This structure enables the application to retrieve historical messages when a user enters a room or when they re-login.

Upon entering a room, a get\_room\_messages event is emitted to request all messages for that room. The server responds with a receive\_room\_messages event, providing the messages to the client, which then updates the messages state.

This high-level overview addresses the core functionality related to user login and connection, handling invitations, room management, and message handling within the ChatPage component of your application.

## **Design Decisions**

Kafka was designed to be processed in one file rather than two separate ones (although testing was originally done in two separate ones with one being for twitter and another being for federated posts) to reduce the need for unneeded files and because the user needs to be subscribed to both topics automatically. Users also cannot make tweets so there is no producer code for that, but we did test producer code to see if it would work and it did through the backend if that functionality were to be added in the future.

Images are processed using a public S3URL link in an HTML tag for ease of processing for each group and because most groups are deciding to display posts in that way. We also check for user registration before automatically making a user to not have duplicates as well as have robust error handling to see if there are any problems. The await-asyncs allow for the posts to be processed in a way that can interact through the frontend and backend. We also chose to make federated posts its own separate frontend so that users are not confused about having to remove the title for the federated post as well as to prevent accidental sends to the whole topic.

## **Extra Credit:**

### **Infinite Scroll:**

The feed has an infinite scroll functionality, where all of the posts in the feed are loaded up into the home feed and the feed shown on the federated posts page rather than having a set number of results displayed per page. We had the feed be stored as an array of JSON bodies representing posts and the frontend parses these JSON bodies into the post component to be displayed so that it gives a more authentic Instagram experience.