

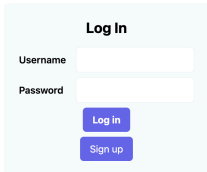
Group 13: Best Network

NETS 2120 Final Project Report

Julia Susser, Vedha Avali, Kimberly Liang, Cameron Bosio-Kim

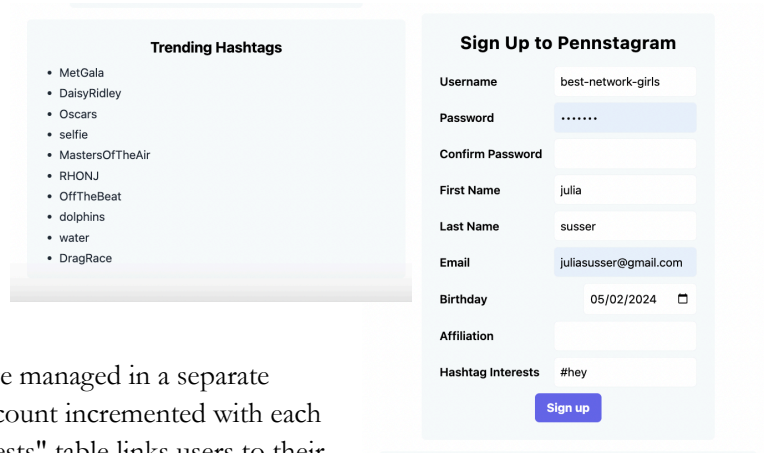
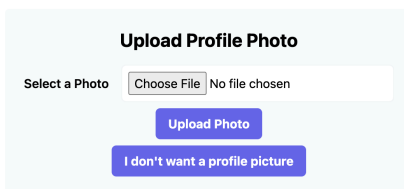
User Signup and User Accounts

User registration & User login functionality

A form titled "Log In" with two input fields: "Username" and "Password". Below the fields are two buttons: "Log in" and "Sign up".

When users access the site, they are directed to a page where they can log in with an existing account or register a new one. Upon logging in, the system compares the entered username and password against those in the "users" SQL table on an RDS database, using the hashed password for security.

When users opt to create a new account, they can click the signup button to access a page for entering necessary information. During registration, the backend checks if the username already exists and prompts for a different one if needed. It also verifies that the entered passwords match. For security, the backend encrypts the password, using salting and hashing, and stores only the hashed password in the database. All user information, except for Hashtag Interests, is stored in the "users" table. Hashtags are managed in a separate "hashtags" table that records each hashtag's ID, text, and a count incremented with each use or when a user adds it as an interest. The "hashtagInterests" table links users to their selected hashtags. The "Trending Hashtags" section displays the ten most popular hashtags, ranked by their count values.

Two side-by-side forms. The left form, titled "Trending Hashtags", lists ten popular hashtags: MetGala, DaisyRidley, Oscars, selfie, MastersOfTheAir, RHONJ, OffTheBeat, dolphins, water, and DragRace. The right form, titled "Sign Up to Pennstagram", contains fields for Username (best-network-girls), Password (masked with dots), Confirm Password, First Name (julia), Last Name (susser), Email (juliasusser@gmail.com), Birthday (05/02/2024), Affiliation, and Hashtag Interests (#hey). A "Sign up" button is at the bottom.A form titled "Upload Profile Photo". It has a "Select a Photo" section with a "Choose File" button and "No file chosen" text. Below this are two buttons: "Upload Photo" and "I don't want a profile picture".

Profile Photo Upload

When registering, users can upload a profile picture via multipart-form data. The backend uses the multer package to extract the file from the request and then uploads it to an S3 bucket. Our S3 bucket has separate folders for each of these image types, and the profilePhotos folder names the image files with the user's userID.

ChromaDB & Actor Account Linking Based on Profile Picture Embeddings

After uploading a profile picture, users are directed to a page where they can link their account to an actor. The actor images are stored on S3, while their embeddings are calculated locally. We also pre-processed the names.csv file to populate an RDS table named "actors," which includes actor names, nconst (unique identifiers), and their 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

Pennstagram - kimilang Home Profile Page

Pennstagram - anonymous Home Profile Settings

Username: anonymous
Email: anon@gmail.com
Profile Photo:

Hashtags: #happy
Actor:

Enter hashtag Add Hashtag Remove Hashtag

No file chosen

Profile Settings

Username

First Name

Last Name

Email

Birthday yyyy-mm-dd

Affiliation

Password

Confirm Password

The user can access their profile page from the home screen. Here, users can update their profile by adding or removing hashtags, changing their profile photo, or selecting a new actor via the “Set New Actor” button. The page also displays recommended hashtags, calculated using Spark. For more comprehensive profile modifications, such as changing usernames, passwords, or names, users can navigate to the Profile Settings page.

Friends

Pennstagram - anonymous Home Feed Chat

Enter friend's name

Enter friend's name to re

anonymous's friends	anonymous's friends	anonymous's
currently online	currently offline	recommended friends
No online friends	No offline friends	No recommended friends

The image shows the friends page for a user with the username “anonymous.” From the home page, users can navigate to the Friends page to manage their connections. Here, they can add or remove friends by typing usernames. The page displays three lists: friends currently logged in,

friends not logged in, and friend recommendations generated using 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 displays user posts, recommended posts from friends, and federated posts from the “FederatedPosts” and “Twitter-Kafka” topics. An infinite scroll feature is implemented to allow continuous viewing of posts without the need for pagination.

User Post Creation with Optional Image and Text

Pennstagram - anonymous Friends Chat Logout Profile Federated Posts

Create Post

Title

Content

Photo No file chosen

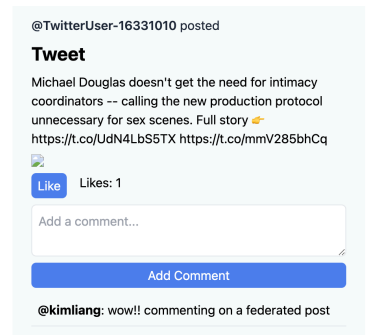
@gt13-kimilang posted
so close
 Likes: 0

Users can create posts with optional images or text, and the frontend uses regex to extract hashtags as an array from content marked with '#'. Submitting a post involves two backend calls: createPost updates the posts table with a new entry, and uploadPost handles image uploads. The latter processes the image through a multipart form using multer to generate an S3 URL that includes the incremented post ID. This post ID and

S3 URL are then recorded in the posts table. The new post features the title, content, username, and options to like and comment, which are all displayed in the refreshed post component.

User Actions: Liking, Commenting, and Hashtag Linking

Users can like posts and tag them with hashtags, managed through database tables linking likes and hashtags to posts and users to their interests. Liking a post updates its like count in the “posts” table and records the user's action in the likesToPosts table. Tagging a post with a hashtag associates it with that post and is recorded in the posts_to_hashtags table. Comments are added to the post table as “sub-posts” with a parent ID pointing to the post that the comment refers to. Each comment includes only the text and the username, with other table values set to null. To ensure comments are up-to-date on the feed, a fetchComments function retrieves comments for each post before it is displayed. Users can comment multiple times on any post, including federated posts.



Feed Ranking & Generation

The feed for each user is decided by the Spark adsorption algorithm, which calculates how likely a user will be to “arrive” at a post in a random walk or how likely they will be to “enjoy” a post. The feed is then generated by selecting posts from the recommendation table, the user's posts, the federated and Twitter posts, and the posts from the user's friends to create a feed.

Kafka & Federated Posts

```
Post created successfully :)
TwitterUser-21217034
Received tweet from author ID 21217034: Kim Kardashian was booted 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 booted 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'
}
```

A response of a successful creation after receiving the message on the Kafka server for a tweet

Reading and sending posts via Kafka

Kafka utilizes a producer and consumer setup for managing federated posts and a separate consumer for tweets. Upon creating an account, users are automatically subscribed to the “FederatedPosts” and “Twitter” topics. The backend is developed using Express.js and KafkaJS, employing the Snappy compression codec to improve the efficiency of message compression, which is particularly beneficial given the high volume of tweets received each hour.

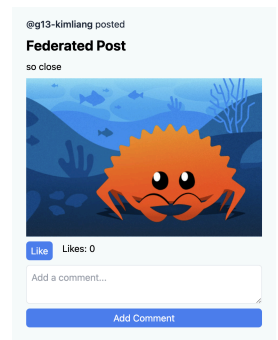
Producer Code:

The producer component of the system is responsible for generating messages and sending them to Kafka topics. The producer component in the backend initializes a Kafka producer and includes a sendFederatedPost function that constructs a JSON object for a federated post, serializes it, and sends it to the 'FederatedPosts' topic. The runProducer function connects the producer to Kafka brokers to start message production. On the frontend, users can create federated posts on a separate page. These posts are broadcast across the topic by calling a backend route registered in kafka_routes.js. This route awaits the sendFederatedPost function from the Kafka file. After executing runProducer, the message is sent to the topic.

Consumer Code:

On the consumer side, the system listens to Kafka topics ('FederatedPosts' and 'Twitter-Kafka') and processes messages using the handleMessage function, which parses messages, identifies their topic, and invokes the appropriate handling functions. For federated posts, the handleFederatedPost function verifies user existence, registers new users if necessary with pre-filled dummy data (like birthdays), and creates posts by calling the createPost function in the backend server. For tweets, the handleIncomingTweet function processes messages, checks for user existence, and creates posts by calling the backend server with Twitter data, which contains usernames content, and hashtags from the tweet data.

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 on 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 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.

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 of 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 receiving 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

The Spark Java code runs the adsorption algorithm covered in lecture in maven package project.best network. The functionality of the Spark algorithm can be broken down into three parts: fetching the social network from the database & computing edge weights, running the adsorption algorithm propagation, and then sending the results to the recommendation database.

Fetching Social Network from Database & Computing Edge Weights

The algorithm first fetches data from RDS regarding social network relationships and stores the pairs of (user, user), (user, hashtag), and (user, post) in a bidirectional list. After fetching from the RDS database, the list is then transformed into a JavaRDD and subsequently into a JavaPairRDD, where each pair denotes an edge where the key is the first vertex and the value is 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”) symbolizing user, post, or hashtag. The relationships fetched from the database include users' likes of posts users' interest in specific hashtags, friendships between users, and hashtags associated with posts. In the subsequent step, the algorithm computes edge weights within the social network JavaPairRDD. It begins by segmenting the RDD into five distinct RDDs based on the type of outgoing vertex: posts, hashtags, user-to-posts, users-to-users, and user-to-hashtags. For each RDD, the algorithm aggregates the total number of edges leaving each vertex to derive the total count of outgoing connections and then calculates the normalized weight for each outgoing edge. This normalization involves taking the reciprocal of the total count of outgoing edges for each node (i.e., $1/\text{total_edges}$) or using predefined weights (.3 for user-to-user/hashtags and .4 for user-to-post edges). Once the weights are normalized, they are integrated back into the original dataset. The edgeRDD is returned as `JavaPairRDD<Tuple2<Integer, String>, Tuple2<Tuple2<Integer, String>, Double>>` so that the edge is the key and the weight is the value. An example edge would be: `<Tuple2<user_id1, “u”>, Tuple2<Tuple2<hashtag_id, “h”>, Double>>`.

Adsorption Propagation:

To run the adsorption algorithm, the algorithm starts by initializing a graph vertexLabelsRDD. This involves fetching all user IDs and assigning each a label (their user ID) with a weight of 1.0. These labels from the vertexLabels RDD, structured as `JavaPairRDD<Tuple2<Integer, String>, Tuple2<Integer, Double>>` where the key is the vertex and the value is the user_id for the label and corresponding weight.

Then, the algorithm runs the adsorption propagation loop, which executes up to 15 times or until convergence. In each iteration, labels propagate across edges by joining vertexLabels with edgeRDD. After the join, label weights are multiplied by edge weights to compute the propagated weight. Results are then aggregated by vertex and labeled using a reduceByKey operation to sum weights for each label at a vertex. After aggregation, the total weight sum for each vertex is computed, and each user_label weight is normalized by dividing it by this sum to ensure that the total weight of all user_labels at a vertex is equal to 1. Finally, the algorithm ensures that user vertices maintain their initial labels by resetting self-label weights to 1.0 and reintroducing any missing self-labels by merging them back into the propagatedVertexLabels RDD. Finally, in each iteration, the algorithm runs a convergence check to see if the difference in label weights is below a threshold (d_max).

Output Handling

```

project-best-network > spark-network > vertexLabelRankings.csv
1 vertex_id,type,user_id,weight
2 4,u,3,0.047438760198828554
3 4,u,8,0.1188421024167598
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,3,0.05404207740298465
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

```

Example CSV output from adsorption algorithm with columns for vertex id, vertex type(user=u, hashtag=h, post_id=p), user_label, and weight (probability that user_label arrives at a vertex in random walk).

The algorithm concludes by creating three tables to send results to database: **socialNetworkFriendRecommendations**, which holds potential friend recommendations with columns for user_id, userlabel_id, and weight;

socialNetworkHashtagRecommendations, for hashtag suggestions to users, featuring hashtag_id, userlabel_id, and weight; and

socialNetworkPostRecommendations, listing recommendations for posts with post_id, userlabel_id, and weight. These tables represent the probability that a user label reaches a specific vertex in a random walk, reflecting tailored recommendations.

Hourly updates on who and what to follow

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

Natural Language Search

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. Afterward, 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 on 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 can get a result, which is an English answer based on the posts.

Chat Mode

Image: Chat Page Screen for user Leo who has three pending chat invitations as well as a notification from Nathan in room 29.

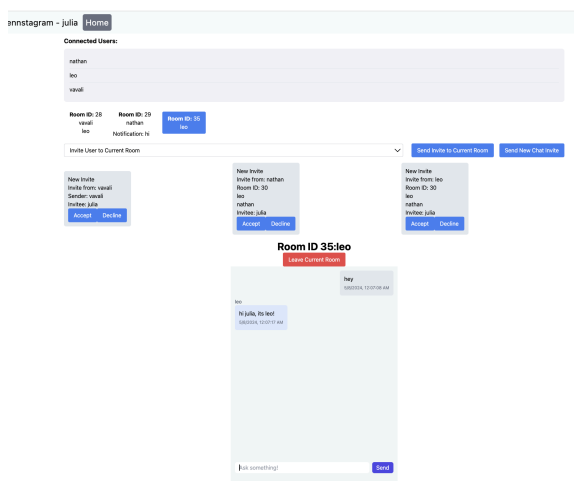
The ChatPage uses Socket.IO for real-time communications between the backend server (socketHandler.js) and the frontend (chatPage.tsx)

State Management

The React frontend of the chat application manages the state with several useState hooks, including user authentication (isLoggedIn), connected users (connectedUsers), chat rooms (rooms), the active room (currentRoom), current room messages (messages), the message being composed (currentMessage), and chat invites (incomingInvites). The Room object, integral to both frontend and backend, holds the roomId, user list, and, on the frontend, notifications for new messages. The backend maintains a list of connected users by socket ID and username, along with tracking pending chat invites.

Initialization & Socket Connection:

Upon mounting, the ChatPage component checks user login with an Axios request to the backend. If authenticated, it sets isLoggedIn true and establishes a WebSocket connection. Upon socket connection, the client emits a send_username event via socket with the client's username to register with the backend. The server then notifies all other connected users of the new



login as well as sends clients their room details, pending invites (which are stored in a list in the backend) & connected users. The frontend then sets their current room to the default first room and requests messages for that room from the backend via socket. If a user reconnects (e.g., after a page reload), the backend updates their socket_id and sends specific data to the reconnected client without alerting other users.

Invitations

Invitations are initiated by the client, checked for validity (whether the invitee is connected and not already in a room together for individual or not already in room for group chat), and sent via socket. The server verifies the invite for duplicates, locates the invitee's socket ID, and notifies them. On the frontend, new invitations are added to the incomingInvites state and checked for uniqueness to avoid displaying duplicates.

Creating New Rooms:

When a chat invitation is accepted, the server creates a new room in the database and adds users to the chatRoomUsers table. The server then instructs the inviter to join the new socket channel and notifies both the inviter and the invitee of the successful join via socket. For group chats, it simply adds new users to the existing room and notifies all members in the room of the new user's addition via socket.

Leaving a Room:

When a user leaves a room, the frontend updates by clearing the current room state and removing the room from the user's list, while emitting a leave_room request via socket to the server. The server then checks if the room is empty and, if so, deletes it from the database, notifying remaining room users of the user's departure.

Sending Messages:

When a user submits a message, the sendMessage function triggers a send_room_message event via socket, including room details, message content, and the sender's username. The server saves the message in the database and broadcasts it to other users in the room via socket. If a user receives a message for a room but is not currently in the room, then it pops up as a notification on their screen, as the notification can be stored in the room's state on the frontend.

Persistent Messages:

Messages are persistently stored in a database with columns for messageID, roomID, message, userID, and timestamp. This setup allows for retrieving historical messages when a user joins a room or logs back in. A get_room_messages event is emitted via socket to fetch messages for a room, and the server responds with a receive_room_messages event, delivering the messages to the client and updating the messages state.

This overview encapsulates the essential functions related to user interactions, room management, and messaging within the ChatPage component.

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 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 a separate frontend page 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:

Web Sockets

The ChatPage component employs Socket.IO to facilitate real-time communication between the backend server, managed by socketHandler.js, and the frontend, represented by chatPage.tsx. This setup enables seamless interaction and messaging functionalities, ensuring a dynamic and engaging user experience.