Project Report: Team Myelin Oligodendrocyte Glycoprotein (G30)

Members: Sam Ngiam, Amish Sethi, Matthew Kuo, Ethan Yu

1. System Overview

We built an Instagram-like application with support for various features of a typical social media platform. It allows users to create profiles, make posts, form chats with other users, search for posts, and interact with other users through comments and likes. The application is built using React for the frontend, with styling done using Tailwind CSS. The backend utilizes Node.js and Express.js for most functionality, with MySQL in Amazon RDS as the database. Apache Spark is utilized for social network graph analysis, to be used in post and friend recommendations.

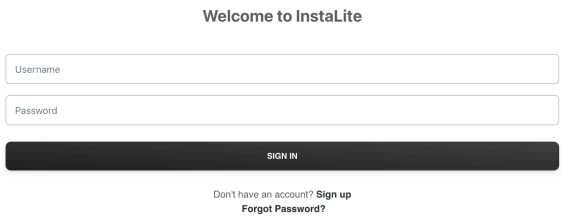
2. Technical Overview

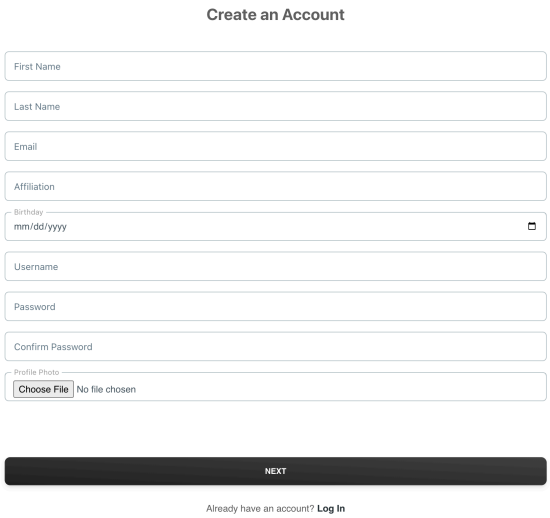
User Login and Signup

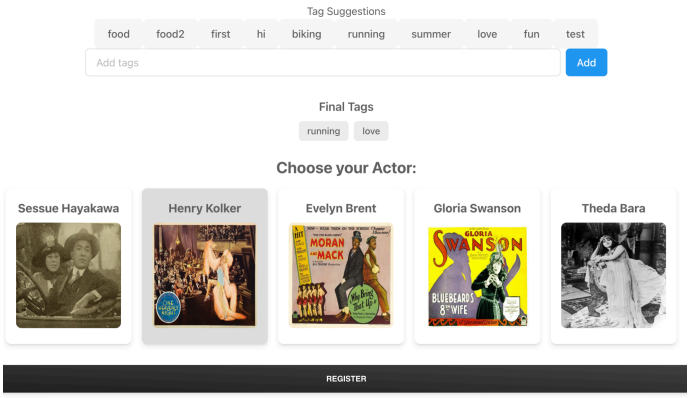
User login and signup functionality centers around a users table in RDS. This stores a user's user id, username, hashed password, linked actor, profile picture link, first and last name, email, affiliation, birthday, interests, post recommendations, friend recommendations, and whether they are logged in. Registration occurs in two steps.

On the first registration page, the user inputs their username, password, first and last name, email, affiliation, and birthday. The user also uploads a profile photo at this stage. Upon progressing to the next page, the profile photo is uploaded to the backend as form data, and Multer is used to temporarily store the file before permanently storing it in an Amazon S3 bucket. Using a ChromaDB collection containing actor images, the 5 actor images which are most similar to the profile picture are found and the corresponding nconsts (ids) are returned. These nconsts are used by the frontend to retrieve corresponding actor images, which are in a folder that is served statically by the backend. On the second registration page, the user selects one of the actors whose images were found to match and inputs interests. In conjunction with the previously inputted information, and the S3 link to the profile picture, this is used to create a profile in the user table. Note that passwords are salted and hashed using bcrypt.

Login involves checking whether an inputted username and password match a username and password (after salting and hashing) in the users table. Upon registration or login, the user's user_id is attached to the session (this is checked on every subsequent API call requiring the user to be logged in).

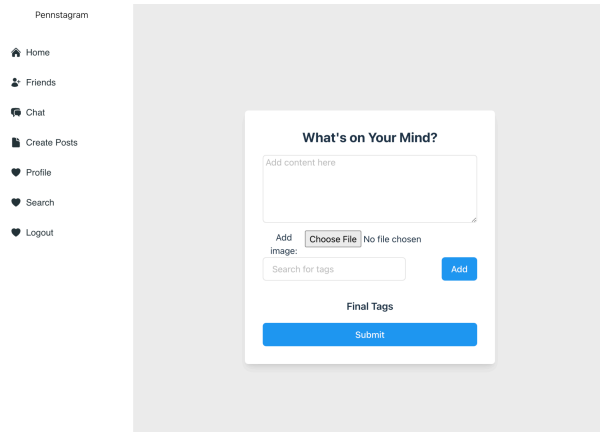






Posts

To create posts, users can input text content, an image, and hashtags. The content can also contain HTML tags. Images are processed similarly to how they are handled in registration, where files uploaded as form data are stored temporarily by Multer and then placed in an S3 bucket, with the S3 linked stored along with the text and hashtags in a posts table.



Feed

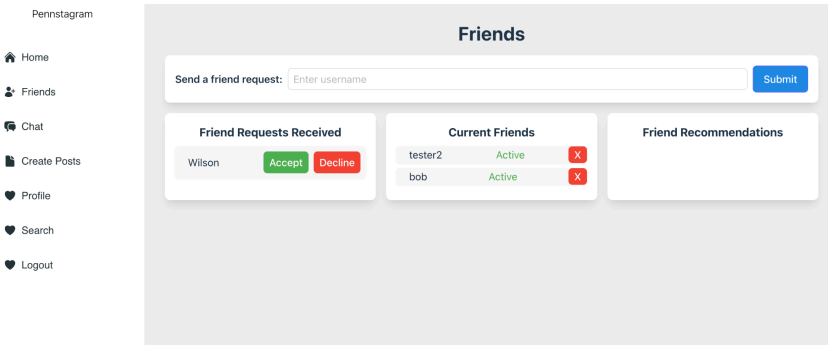
Kafka is used to retrieve Twitter posts and Federated Posts from other applications. These are then added to a posts table. The backend attempts to parse consumed posts as JSON and ignores any posts which are formatted improperly.

For Federated Posts, the post content is also parsed for hashtags so that these can be added to the corresponding entry in the post table, if present. HTML image tags found in text content will automatically be shown properly by the frontend when the posts are displayed. Since these posts may have usernames which conflict with usernames in our application, they are given a username of the form groupNumber:username, where groupNumber is the project group that posted and username is the username provided.

Twitter posts similarly have usernames of the form twitter:userId. The author_id of these posts is assigned as -1, which links to a dummy user with this user_id in the users table.

Friends

Bi-directional friendships can be established through sending friend requests. This creates an entry in a friend requests table, which consists of fields of sender and recipient. Users can see a list

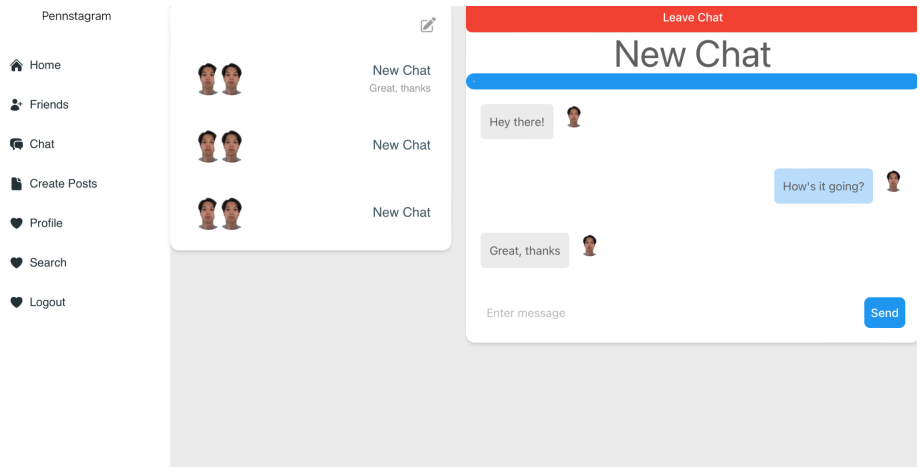


of incoming friend requests, and have the option to decline or accept. Both options cause the corresponding entry in the table to be removed, and accepting creates two entries in a friends table, which has fields of followed and follower (one for each direction of friendship).

Chat

WebSockets is utilized for chat functionality as it provides a full-duplex communication channel over a single, long-lived connection, allowing the server and client to send messages back and forth without needing to reestablish connections. Our application uses Socket.IO, a library that abstracts WebSocket interactions into an easy-to-use API. In the app, users can send and receive messages in real time without refreshing their browser. Additionally, users' connection states are tracked to handle join and leave events.

The chat system utilizes three main tables to manage and store all chat-related data. A chat sessions table (chat_sessions) stores information about each chat session, a chat messages table (chat_messages) manages all messages sent within each chat session, and a session memberships table (session_memberships) tracks which users are part of which chat sessions and whether they have accepted the invite.



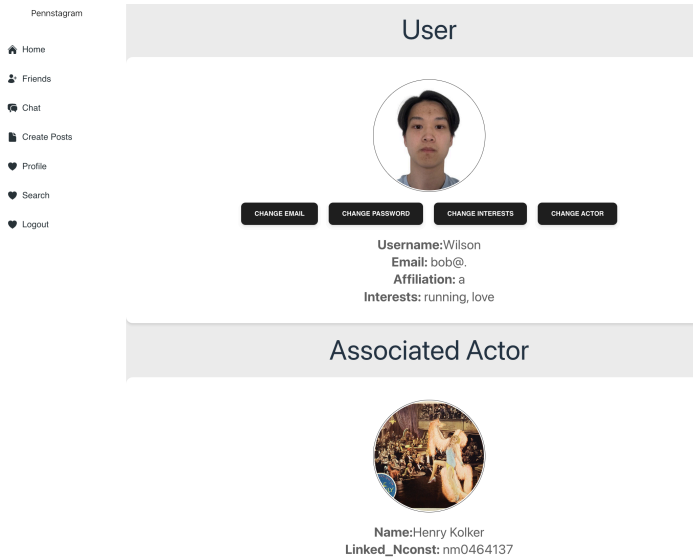
Post and Friend Recommendations

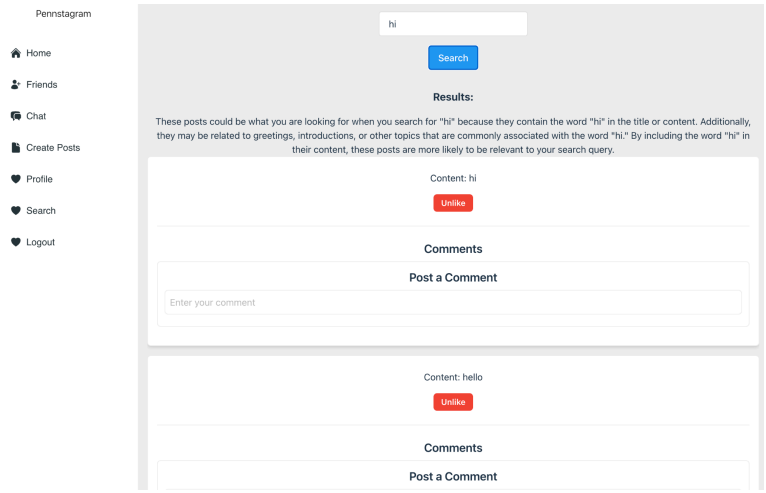
In order to create a recommendation pipeline in our backend that periodically uses user, post, and hashtag information to generate post and friend recommendations for users, we implemented the adsorption algorithm from Baluja et. al. In particular, we implemented the Adsorption via Averaging algorithm, which is flexible and scales well on a framework like Apache Spark. Using a graph of users, posts, and hashtags, the Spark Job runs until convergence, or 15 iterations as a hard upper bound.

Post recommendations for a user A are calculated by observing the value of A at each post, and then normalizing to produce a probability distribution. Friend recommendations for a user A are calculated by going through the labels at A at termination, removing the labels that A is already friends with, and then normalizing the weights. At the end, results for each user are serialized using JSON and persisted back to entries in the user database, ensuring that the recommendations are usable by the application.

Profile Page:

The profile page displays a user's username, email, affiliation, interests, associated actor, profile picture, and picture of the associated actor. There are options to change the user's email, password, interests, and actor, which modify the corresponding details in the user table.





Search

Searching for posts is done with retrieval-augmented-generation. Using Langchain, every post in the database is added as a document to a ChromaDB collection utilizing an OpenAI embedding model. This collection is then used to perform cosine similarity search on an inputted query and find the 5 most similar posts. These posts and the query are entered into an OpenAI GPT LLM to receive a written explanation of why the posts might be what the user is looking for. The posts and this explanation are displayed to the user.

3. Design Decisions

Database

As explained above, the main MySQL table used is the users table, which stores information about the users such as their id, name, and other fields. Then, most tables to implement other features such as posts, chat, and friends utilize the users table's user id field as a foreign key in order to link the various tables. This table structure allowed information in the database to be queried quite easily, especially since most app features, and thus routes, are centered around a single, logged in user or between users. Other tables are typically structured with an id field as well as other fields relevant to an individual instance of what the table is representing. For instance, each posts table entry consists of a post id, author id, content, hashtag ids, ids of users who liked, and foreign username (for posts coming from Kafka).

General Backend

Most backend functions are grouped by the aspect of the app they pertain to, such as user, posts, comments, tags, and chat functionality. This made it simpler to manage our codebase and reuse functional components. More complex features such as RAG, ChromaDB, and Kafka operations are handled in their own files and then exported to the relevant routes for ease of testing and development.

Chat

As stated above, chat functionality is structured around three main tables to support scalable, real-time communication: a chat sessions table, chat messages table, and session memberships table. These tables are designed to optimize data retrieval and manipulation as they represent the functional separations in the different aspects of chat. Another decision we had to make with chat is how to handle invitations. We chose to handle invitations by adding a recipient to a chat but setting an active flag equal to false. While a user has not responded to an invitation, other users can continue sending messages and the non-responsive user will be blocked from sending in that chat. If a user accepts the invitation, switching their status to active is a simple update that instantly integrates them into the chat. This method avoids the overhead and potential delays of re-adding a user to the session database and allows for a smoother transition into active participation. From a technical perspective, managing user states within a chat session using active flags simplifies the system architecture as it reduces the complexity of managing multiple user states and conditions.

4. Changes Made/Lessons Learned

From a project coordination perspective, we at first faced significant difficulties with accidentally producing bugs and errors when merging our code. This also became problematic when we tried working on multiple Git branches, which led to individuals working on very inconsistent codebases. Over time, we became more adept at handling merge conflicts and distributing work to avoid situations in which multiple team members were editing the same file separately.

One interesting lesson we learned was regarding our SQL queries. At first, we mostly used non-parameterized queries when dealing with our tables. However, after noticing a consumed Kafka post containing what appeared to be SQL code, we became more aware of the dangers of SQL injections and began using parameterized queries when handling data inputted by outside users to ensure greater security.

We also encountered many issues while attempting deployment and so learned a lot more about the challenges associated with deploying projects and setting up resources like EC2 and Spark.

5. Extra Credit

-Friend requests: As explained above, users can send friend requests, and accept or decline incoming friend requests
-"Forgot password": Users can submit their username on a forgot password page. They will then be emailed a link containing a generated token sends them to a page where they can reset their password
-Infinite scrolling: Users can continuously scroll on their home feed to see more posts
-Site-wite "what's trending": Top posts are displayed site wide
-WebSockets for chat: WebSockets is used for chat, as explained above
-LLM search results: Rather than returning links to posts, the search feature directly returns the found posts (which can be interacted with by the user)
-Dynamic search: When selecting interests, options for interests that have previously been entered by users show up dynamically based on what is being typed