Project Report: Team Myelin Oligodendrocyte Glycoprotein (G30)

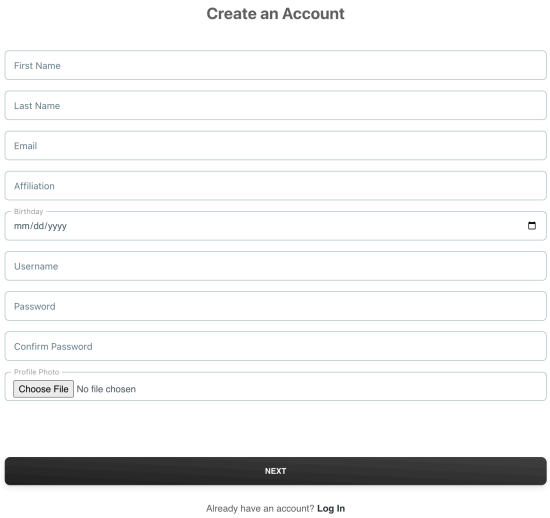
Members: Sam Ngiam, Amish Sethi, Matthew Kuo, Ethan Yu

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

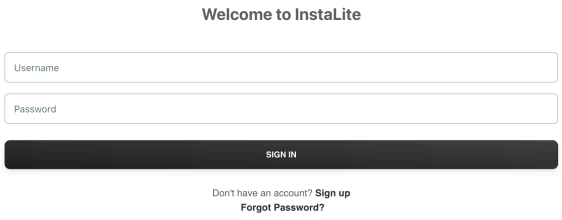


User login and signup functionality centers around a *users* table in RDS. This has fields of user_id, username, hashed_password, linked_nconst, image_link, first_name, last_name, email, affiliation, birthday, interests, logged_in, rank_distribution, and friend_recommendation. Registration occurs in 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 the corresponding 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, req.session.user_id is assigned to be the user's id (this cookie is checked on every API call requiring the user to be logged in).

Posts

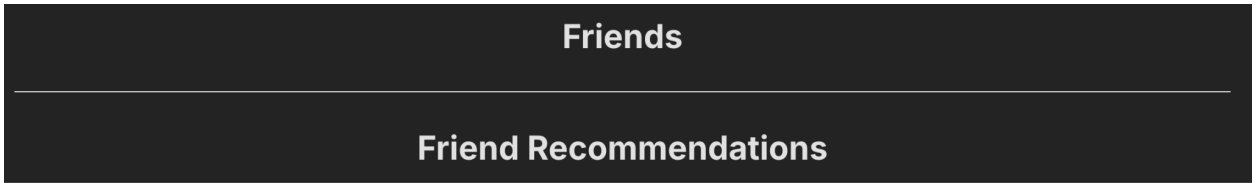
To create posts, users can input text content, an image, and hashtags. The content can also contain HTML tags, which will be displayed properly. Images are processed similarly to in registration, where files uploaded as form data are stored temporarily by Multer and then placed in an S3 bucket. An image HTML tag is then added to the bottom of a post's content containing the S3 link to the image.

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 cannot be processed properly.
For Federated Posts, the post content is also parsed for hashtags so that these can be added to the entry in the post table, if present. 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. Since colons are not allowed in usernames to prevent SQL injections, these should not conflict with existing usernames in our databases. The author_id of these posts is assigned as -1, which links to a dummy user with this user_id in the users table.

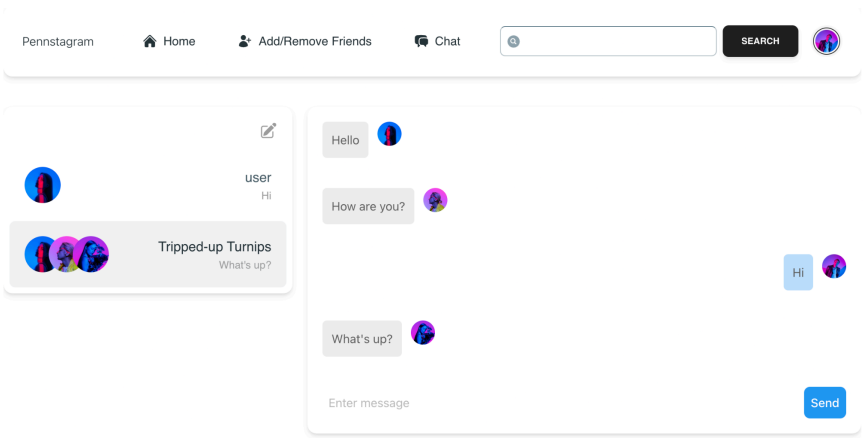
Friends

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 friend_requests table to be removed, and accepting creates two entries in the friends table, which has fields of followed and follower (one for each direction of friendship).



Chat

We utilize WebSockets for our chat 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 system uses Socket.IO, a powerful library for real-time web applications that abstracts WebSocket interactions into an easy-to-use API. With Socket.io, we allow real-time messaging as users can send and receive messages instantly without refreshing their browser. Additionally, users' connection states are also tracked to handle join and leave events seamlessly.



The chat system utilizes three main tables within an RDS database to manage and store all chat-related data effectively. The **Chat Sessions Table (chat_sessions)** stores information about each chat session. The **Chat Messages Table (chat_messages)** manages all messages sent within each chat session. The **Session Memberships Table (session_memberships)** tracks which users are part of which chat sessions and whether they have accepted the invite.

Jobs

With the social network graph created by Pennstagram, we have a comprehensive web of users, posts, and hashtags. To make our application refreshing, dynamic, and to encourage user retention, we want to recommend friends and posts to each user. This requires a recommendation pipeline in our backend that periodically uses this network graph to compile a series of recommendations for users.

For our recommendation algorithm selection, we had many options. To point one out, PageRank is a famous algorithm used by Google for its search recommendation. It essentially simulates random walks from each vertex and records the expected number of users at each node, and denotes it as the node/page's "PageRank". However, PageRank doesn't handle specialized graphs, where there are different classes of nodes/vertices.

So we settled on the Adsorption Algorithm from Baluja et. al (https://research.google/pubs/video-suggestion-and-discovery-for-youtube-taking-random-walks-through-the-view-graph/), which was utilized for YouTube in its early days for video recommendation algorithm. Just like how the YouTube graph had both users and videos, we have users, hashtags, and posts. More specifically, we implemented the Adsorption via Averaging

algorithm, which is flexible and scales well on a framework like Apache Spark. Our Spark Job runs until convergence, or 15 iterations as a hard upper bound.
Post recommendations for a user A are then calculated by seeing the value of A at each post, and then normalizing based on the relative values of A at each post. Friend recommendations for a user A are calculated by going through the labels at A at termination, and removing the labels that A is already friends with, and then normalizing the weights.
At the end, results are serialized using JSON and persisted back to a relational database, ensuring that the recommendations are accessible for application use and further analysis.

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 then entered into an OpenAI 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.

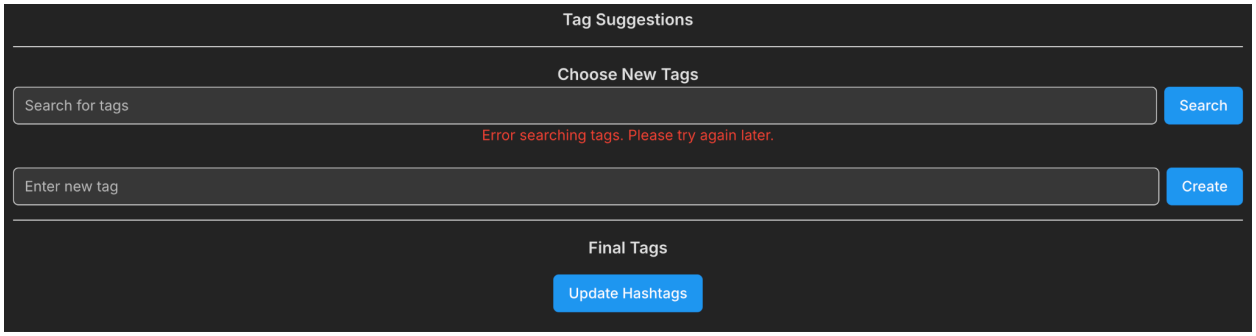
3. Design Decisions

Databases:

As explained above, the main MySQL table used is the user table, which stores information about the users such as their id, name, and other required fields. Then, we also added tables like the post table in which every time a post is created in our backend, it will insert the relevant information into the database so that whenever we need to retrieve this information, we can do it very efficiently.

Frontend:

One of the major decisions we had to make was how we chose to implement the hashtag selection feature. In the end, we designed it with three sections. The first section suggests the current most popular hashtags based on the number of users using each hashtag. The second section allows users to search for existing hashtags. For example, if we have "food1" and "food2" in our database, searching for "food" will return both of these hashtags. This allows users to search for hashtags more efficiently. Finally, the last section of this feature allows users to create new tags to add to the database if they don't already exist.



Chat:
As stated above, the chat functionality is structured around three main tables to support robust, scalable real-time communication: Chat Sessions Table, Chat Messages Table, and Session Memberships Table: These tables are designed to optimize data retrieval and manipulation for chat functionality, as these three tables 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 treat an invitation as adding a user to another chat but setting an active flag equal to false. This makes it so that while a user is not responding to an invitation, other users can continue sending messages, and the user will be blocked from sending in that chat. Additionally, 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

At first, we were just using normal queries for inserting things. However, we became more aware of sql injections and used more parameterized sql queries to protect our routes against these types of attacks.

With this project, we learned a lot more about deploying projects and setting up things 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
-Light mode/dark mode: Light and dark mode for the application can be toggled