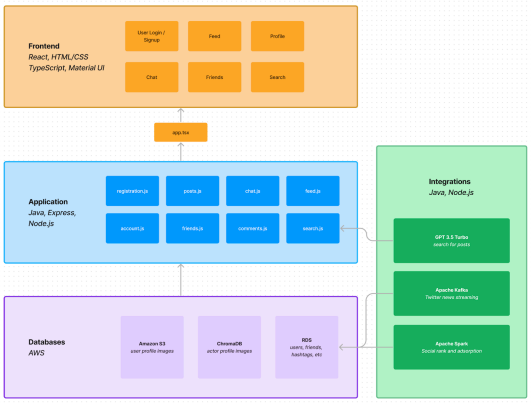
NETS2120 Final Project Report - Java Swingers

Seth Sukboontip, Samuel Wang, Zimo Huang, Sonia Tam



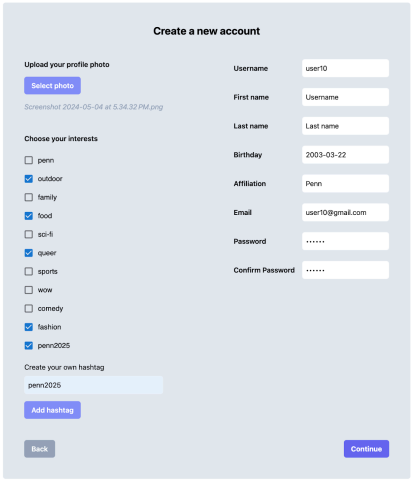
Overview

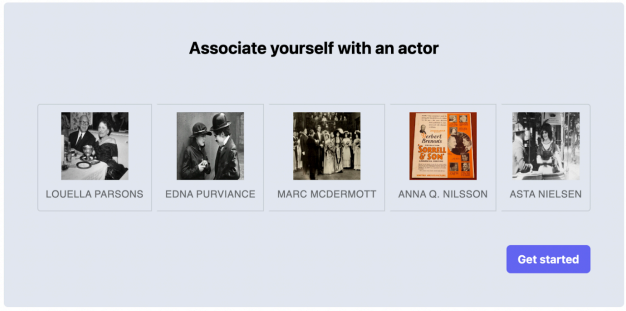
InstaLite is a social media platform built to emulate Instagram. The platform includes a host of features such as creating user profiles, associating users with actors based on matching images, posting and interacting with image content, chatting with friends, and searching for posts and profiles.

The backend was built using Node.js and Java, hosted on Amazon EC2. The databases were hosted in RDS with large objects such as images stored in S3. The frontend was built using React and TypeScript. For our team, we had one member focusing on frontend, one focusing on databases and backend, and two focusing on the backend and frontend integration.

Feature Descriptions and Decisions

Signup





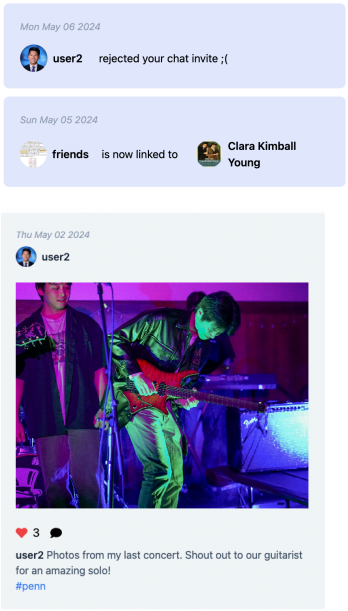
In signup, there are four types of general information that we need to collect about the user: 1) user details, 2) profile photo, 3) interests in the form of hashtags, and 4) the preferred actor to be associated with. The first three types of data did not rely on anything else and could be independently collected; however, the 4th type of data (actor preference) depended on the profile photo submitted. This

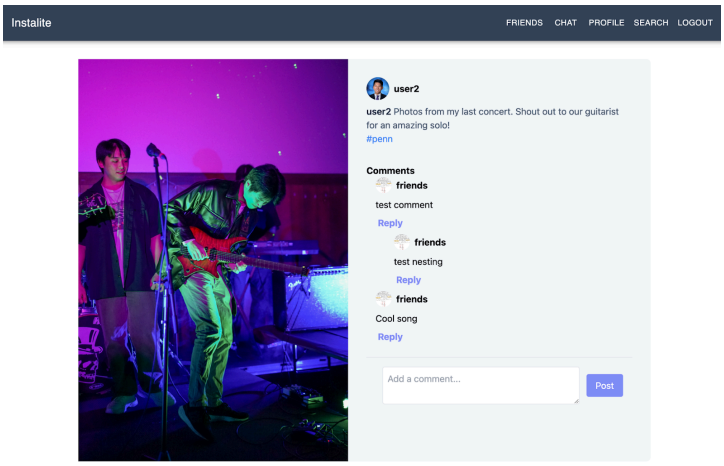
is because the actors that we recommend to the users when signing up are based on similarity to the profile photo that they've submitted.

Because of this dependency, we chose to put the first three types of data collection on the first page of signup. Then once they click the 'Continue' button on the bottom right, all the collected data will be sent to the backend, where the image will be stored in the S3 database. With the given set of actor photos in a ChromaDB database, we then run a face-matching algorithm to match the user's provided profile photo to a set of actor embeddings. We then take the top 5 most similar actors and suggest it to the user in the next step of the signup process. When the user clicks on the 'Get Started' button, then a new entry is created in the user RDS database with the associated S3 image link.

For the frontend, we chose to use Google's Material UI library. This is because we wanted the user to be able to toggle between different options and be able to see which one they selected, and MUI's Toggle Button Group component served this functionality for the second page of signup.

Feed



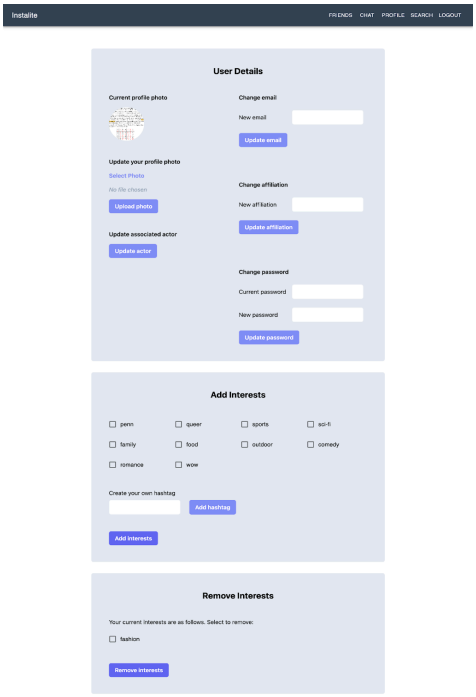


For feed, we had two types of data inputs: 1) posts and 2) notifications. The Posts are denoted by the light gray background, and the notifications are denoted by the light blue background. There are two types of posts: user posts and Twitter posts, which are parsed from the Twitter/X feed fetched from Apache Kafka and added to the posts table under the user 'TwitterKafka'. The posts are ranked using the adsorption algorithm in Spark, where each post for every user is assigned a weight that is calculated periodically per hour. When a user clicks on the comment icon on a post, they will see the post in an expanded form where they can see the comment threads and add a comment. The user can also choose to reply under each threaded comment, and to make this functionality very clear we decided to include the reply text box under each comment (like LinkedIn, as opposed to Instagram which streamlines all commenting in one comment box).

At the top of the feed, users are able to select the photo they want to include and input the caption. Once they click on ‘create post’, our backend would be able to parse through the caption and associate hashtags in the post, so that users do not have to manually input their hashtags.

There are three types of notifications: 1) chat invitation acceptance or reject, 2) friend invitation acceptance or reject, and 3) when a friend is linked with another actor. Each notification has the date as well as the username and profile photo of the relevant profiles. Since these are notifications, we chose to place them near the top of the feed, where they are sorted by date.

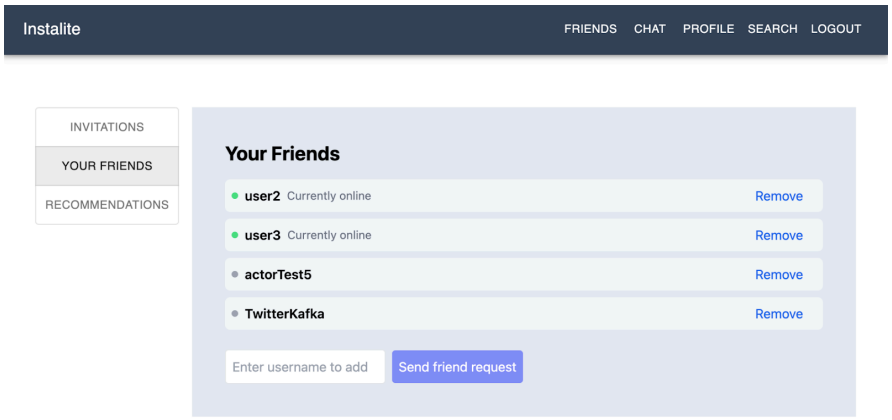
User Profile



The profile section manages the 3 types of data that we initially collected in the signup flow. However, here we chose to structure the frontend such that it is sorted by 1) user details and 2) managing interests. This is because the action of updating a user profile photo impacts the associated actor, hence we put both action items together. We chose to separate adding interests and removing interests because for adding interests, we need to show the suggested interests, while for removing interests we need to show the user’s current interests. These are two different API calls, and hence, we chose to display them separately on the frontend. For each of these action buttons on the user profile section (‘remove interests’, ‘add interests’, ‘update email’ etc), we wrote backend calls to update the relevant database.

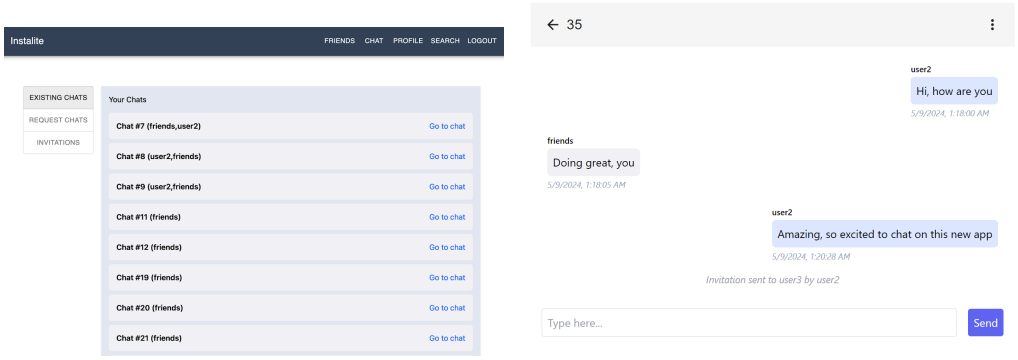
Similar to signup, we created a separate page for updating the associated actor. We included an ‘upload photo’ button in the main profile page so that when that happens, the backend can recalculate and choose which recommended actors to show.

Friends



Friends has three features: invitations, your friends, and recommendations. Invitations are where you can see incoming invitations from other friends, and you can accept or reject them like LinkedIn. Your Friends is where you can see your current friends, and add or remove friends. Recommendations are where you can see which friends are recommended to you and send invite requests. These recommendations are based on the social ranking generated by the adsorption algorithm. This is implemented by creating edges from hashtags, posts, and users and running PageRank on them to determine which items are the most influential.

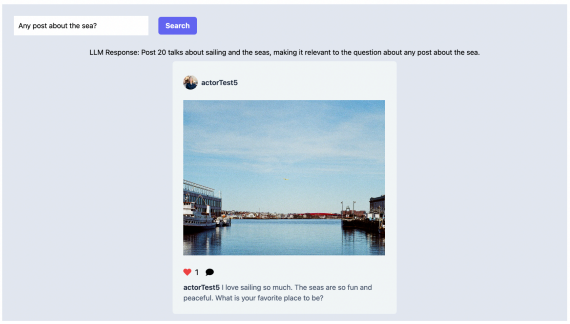
Chat



The chat feature has 3 functionalities: showing the list of existing chats, sending chat invitations to friends, and receiving chat invitations from friends. We decided to have a separate page for each chat room itself instead of having it navigable within the chat tab so that it would be easier in the backend to create individual rooms for each chat. Each time a user sends a chat invite to their friend, the friend receives a notification on their feed page about the invitation.

For each chatroom, we used socket.io to enable users to chat with each other. Within each chatroom, a user has the optionality (on the top right bar) to leave the chat or to add new friends. To efficiently exchange messages across the chats, there are different rooms set up for each of the sockets that represent a given active channel. Additionally, to make chats persistent, each of the messages is inserted into a table storing all messages. To improve the user experience, a special username that can send messages in the socket represents an announcer. The announcer's texts are formatted differently and alert users when someone has joined a chat or has been invited to it. To ensure that all messages are received in the same order, when a user sends a chat the message does not get added to their frontend. Instead, there is a listener attached to the socket that adds the message to the list of messages in the React application.

Search



The search feature currently supports searching for posts. For example, if there is a post where the caption says “Photos from my last concert” and the user inputs “concert” into the search bar, then they will be able to retrieve the entire concert post. The user will be able to interact with the post within the search results, including liking it, going to the comments (navigating to the expanded page of that post), and so on. The search feature works by using retrieval-augmented generation over an embedding of the captions from the posts database to provide context and uses GPT-3.5-turbo to find the best matches.

Database Design Decisions

The main tables store fundamental information, and the auxiliary tables help us relate the data on the main databases. For example, we have the hashtags table as one of our main RDS databases. This database relies on hashtag_rank, user_hashtags, hashtags_to_posts, and hashtags_to_chat to actually utilize the various tags in our hashtags table. Throughout development, our team implemented more tables as needed. During the beginning phases of our project, we only needed the fundamental functionalities such as logging in and logging out, and adding tables like chat_rooms in later stages.

Moreover, we employed the notion of primary and foreign keys to impose various invariants on our database schema. The users table underscores this design choice, acting as a reference table to the majority of other tables. Intuitively, we would want our tables to rely on our users table after all due to the app being the social media type. We can see these schema invariants in play when adding a new post. For example, if User1 decides to create a post, the database would relate the post itself to User1’s user_id, which is a primary key in the users table. Furthermore, we would need to relate hashtags and the social rank to the post; we can utilize the hashtags_to_posts and posts_rank tables to do so, respectively.

By nature of the profile photos and post content, we needed the AWS S3 buckets to store files that our RDS tables could not support. As such, we created a public S3 bucket to store our images. Note that the bucket only allows for public reads not writes for security concerns. Within our bucket, we created two sub-buckets: posts and profile_pictures. The posts bucket is keyed on the post_id, and the profile_pictures bucket is keyed on the user_id. Again, these IDs relate to the RDS tables that also store these S3 links in their respective tables, maintaining our intended invariants.

Since we implemented the face recognition feature, we needed a vector database in order to retrieve the actors whom the user looked like. Clearly, ChromaDB provided the infrastructure that matched our requirements. As for the design decisions, we mirrored the design choices made on the homework, allowing us to efficiently implement the feature.

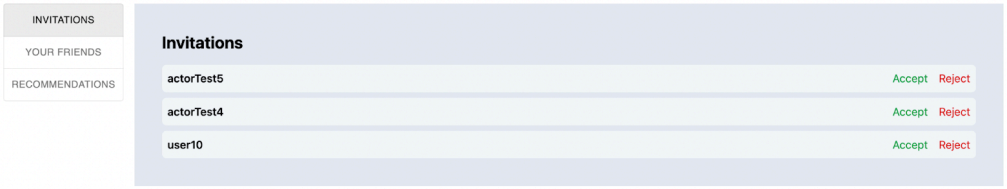
Bugs Faced & Lessons Learned

GitHub merging and concurrent development posed itself as one of our major setbacks during the project. Initially, each team member merged directly into the main branch from their local machines, altering or even completely overwriting features that another team member already implemented. The collision in the registrations.js route file highlighted the importance of good practices when it comes to updating the GitHub repository. Before standardizing merging procedures, we unintentionally overwrote working existing endpoints due to suboptimal word assignments for two of our team members. After the incident, we standardized development and merging procedures such that no team members would

unintentionally overwrite existing code. Each team member will work from unique separate branches and merge when a major feature is completed. After merging, every team member will pull from the main branch to update their local code base as needed. Essentially, we learned that a standardized repository procedure acts as a fundamental pillar to development progress, allowing for clean feature additions.

Extra-Credit Features

- LinkedIn-style friend requests, with accept and reject
    - In the Friends tab, users can receive friend request invitations and choose to either accept or reject them. If accepted, the friends table will be updated, as well as the status in the friend_requests table.



- Infinite scrolling on the Feed
    - We use the React infinite scroll component to support infinite scrolling so that the server fetches more posts on the user's demand.
- WebSockets for chat
    - We use socket io to implement chat. This makes the speed of chatting very fast and efficient and also makes sure that there is temporal consistency in the message display for different chat end users
- Returning valid links to posts for Search
    - We directly display posts that the user is looking for (like in Feed) along with GPT's response in search results. We also enable users to directly like and view comments by clicking on the post.