Moggerstram: Project Writeup

Matthew Tsui, Ashley Zhang, Alicia Sun, Kevin He
University of Pennsylvania
matsu@sas.upenn.edu, asun0102@seas.upenn.edu, ashzhang@wharton.upenn.edu, haokunhe@seas.upenn.edu

Overview

The Moggerstram project entails the development of "InstaLite," a social media platform reminiscent of Instagram. Leveraging multiple components, the project encompasses a diverse technology stack, including Node.js, Javascript, RDS, DynamoDB, ChromaDB, S3, Apache Kafka, Apache Spark, React, and JavaScript, managed through GitHub. Users can sign up, log in, and customize their profiles with profile photos and hashtags of interest, linking their accounts to actors on IMDB based on selfie embeddings. The platform features a dynamic feed displaying user posts, comments, and relevant content based on friendships, hashtags, and course project streams. Additionally, it facilitates federated post sharing through Kafka channels. Secondary screens include profile management, friend interactions, chat functionality with persistence and scalability, and natural language search capabilities. Security measures prevent unauthorized access and content manipulation, while scalability ensures system resilience under increased user load.

Table of Contents

Technology Stack and Design Decisions

Database

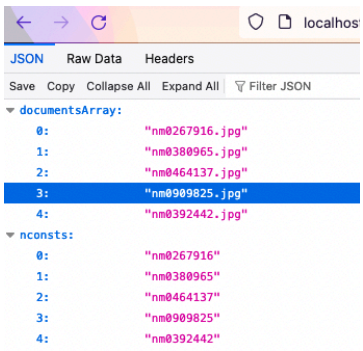
Our database uses several different tables, all stored in a rds as the main storage of information. Outlined below is all of our tables, and their schema/purpose.

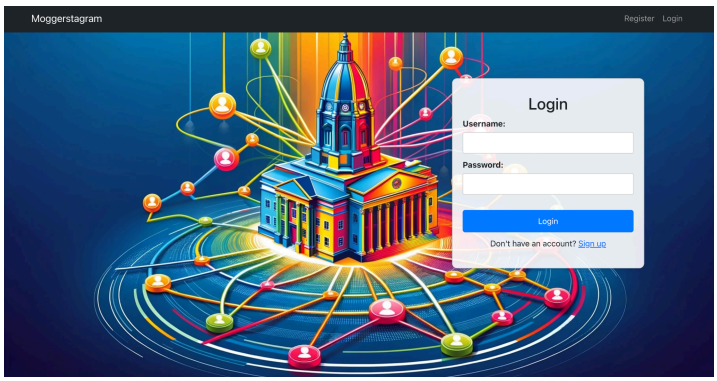
Adsorption: Normalized scores for all nodes (users, hashtags and posts)
ChatRequests: Store chat requests between the sender and the receiver, maintains that only one of each request can be sent, can not send a request if it already exists
Comments: A database of comments (with parent comments for threads), linked to the posts table, and users
FriendRequests: Stores friendrequests;

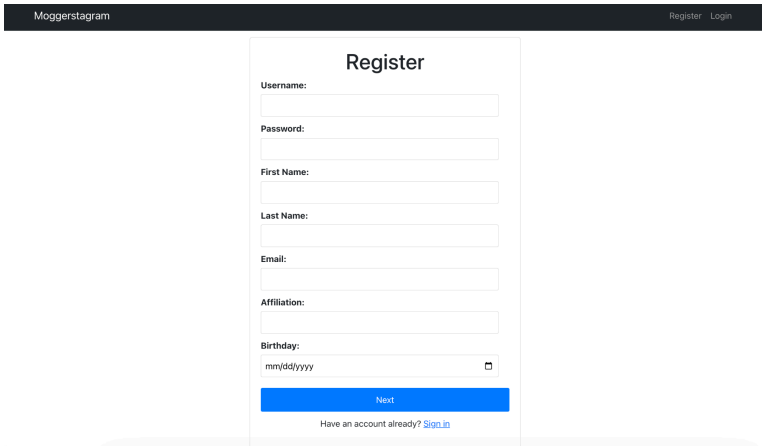
Friends: Stores people who follow/are followed by each other
hashtagPosts: Stores the hashtags that each posts have
Hashtags: database of all hashtags
Interests: Stores each user's selected interests/hashtags
Likes: Stores the people who liked each post, for efficiency don't want to update the main posts table each time
Messages: Stores all the messages, keyed by who sent, content, and which chat its current in
Names: External actor imdb
Posts: Stores all the posts + associated content
Principals: External actor imdb
Ratings : external actor imdb
Recommendations: For recommendations, person strength etc number of second degree follower/followed
Titles: External actor db
User_chats: Stores which users are in which chats
Users: stores all data on each individual user such as id, email, password, etc.

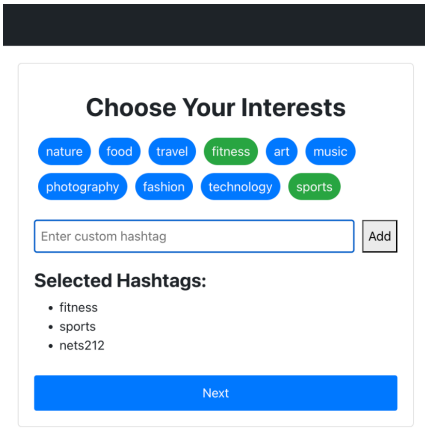
Authentication

We use Bcrypt to salt and encrypt the password upon user registration. When the user signs up, they fill in basic information, including username, password, name, email, affiliation, and birthday. Then, they choose their interests (hashtags) from the next page. We suggest the top 10 popular interests/hashtags and allow them to create custom hashtags. Then, the next page takes them to uploading their profile picture, where it uses ChromaDB to find the 5 most similar actors and allows the user to choose one to link to their profile.









We use local storage, which functions like cookies, and express sessions to store user information, such as username and user id upon login/registration. Upon logout, the session storage is deleted. We call the username and user id from the local storage to determine permissions for page access and which actions which user is making, like posting, liking, commenting. The navbar URLs that lead to feed, profile, chat, and post are only available upon authenticated users.

```
const ReactSession = {
  SESSION_OBJECT_NAME: "__moggers__",

  get: function(key) {
    const item = localStorage.getItem(this.SESSION_OBJECT_NAME);
    return item ? JSON.parse(item)[key] : null;
  },

  set: function(key, value) {
    const session = JSON.parse(localStorage.getItem(this.SESSION_OBJECT_NAME)) || {};
    session[key] = value;
    localStorage.setItem(this.SESSION_OBJECT_NAME, JSON.stringify(session));
  },

  remove: function(key) {
    const session = JSON.parse(localStorage.getItem(this.SESSION_OBJECT_NAME)) || {};
    delete session[key];
    localStorage.setItem(this.SESSION_OBJECT_NAME, JSON.stringify(session));
  }
};

export default ReactSession;
```

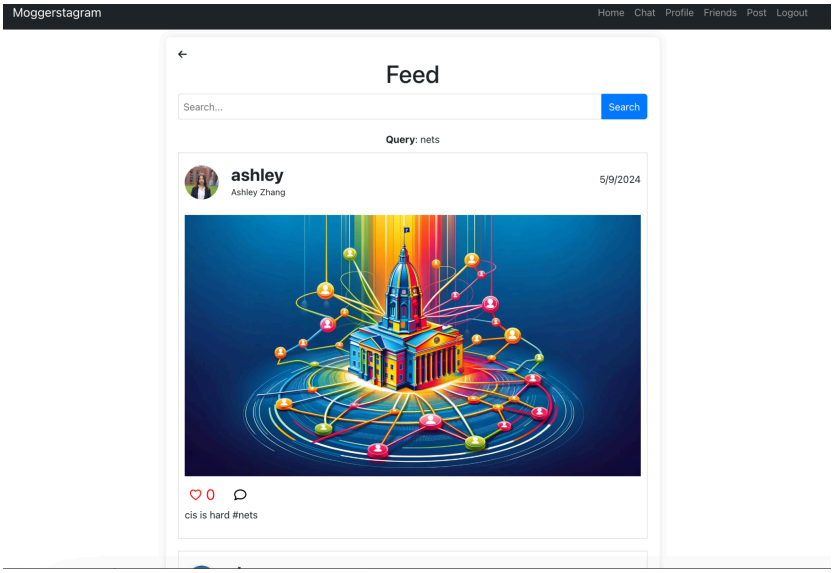
Image Search/Storage and Objects

We decided to store our images in AWS S3, because it's more secure and fault tolerant. This also allows us to avoid CORS errors that come from sending images from the frontend to the backend. From the frontend, we call a backend route that generates a presigned URL, which we then use in the frontend with a PUT fetch request to upload the image to S3. We then send this image URL to the backend. This is used for all image storing/uploading, for posts and profile pictures.

```
const s3Client = new S3Client({region: config.awsRegion, credentials: credentials });
// gets an S3 presigned URL for uploading a file
router.post("/get_presigned_url", async (req, res) => {
    try {
        const fileName = req.body.fileName;
        const fileType = req.body.fileType;

        const uniqueFileName = `${uuidv4()}_${fileName}`;
        const params = {
            Bucket: config.s3BucketName,
            Key: uniqueFileName,
            ContentType: fileType,
        };

        const command = new PutObjectCommand(params);
        const presignedUrl = await getSignedUrl(s3Client, command, {
            expiresIn: 3600,
        });

        res.json({ url: presignedUrl, fileName: uniqueFileName });
    } catch (error) {
```

NLP

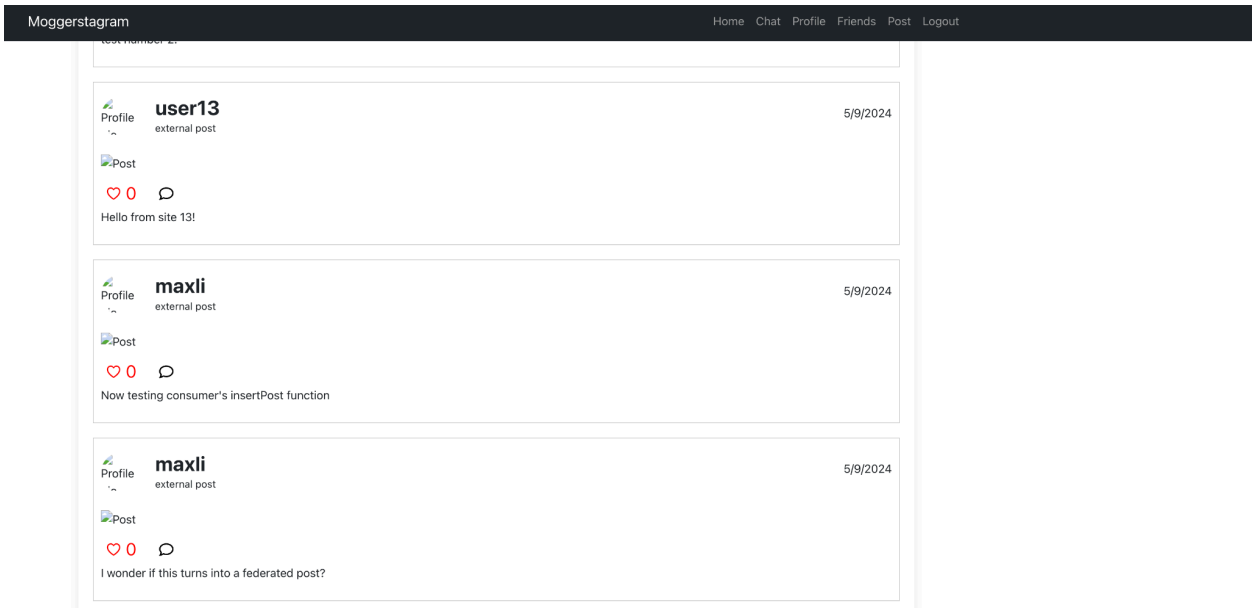
We used the OpenAI API to do a Natural Language search to find posts relevant to the query given. We do this based on content and hashtags, and make sure that all links returned are valid postswith only feeding the API valid posts and by prompt engineering it for its decisions.



Social News Streaming

We stream news from two topics, specifically Twitter-Kafka, and FederatedPosts. We have two consumers for this purpose, and then we also send our own posts to FederatedPosts every single

time one of our users create a post (use one producer here). Attached below is an example of other platforms/instagrams posts on our own site.



Ranking Posts

Below is a snippet of our overall graph with initial edge weights before we perform adsorption, one can see that the weights of edges have been adjusted such that for each user, the weights from users to edges are averaged and sum to 0.3, the weights of users to posts sum to 0.4, and weights between users sum to 0.3. Additionally, outgoing edges from post nodes have weights of 1.0.

```
user:5 user:8 0.3
hashtag:joy user:2 0.3
user:8 user:7 0.09999999999999999
user:25 user:26 0.3
user:6 user:7 0.15
user:6 user:11 0.15
user:11 user:6 0.075
user:11 user:7 0.075
user:11 user:8 0.075
user:11 user:10 0.075
user:12 user:10 0.3
user:7 user:6 0.075
user:7 user:8 0.075
user:7 user:10 0.075
user:7 user:11 0.075
user:4 user:5 0.3
user:7 post:23 0.4
post:36 hashtag:testhashtags 1.0
post:37 hashtag:testhashtags 1.0
post:38 hashtag:testhashtags 1.0
post:38 hashtag:cat 1.0
post:39 hashtag:testhashtags 1.0
post:39 hashtag:cat 1.0
user:4 hashtag:hiking 0.3
user:3 user:2 0.09999999999999999
user:3 user:4 0.09999999999999999
user:3 user:5 0.09999999999999999
post:21 hashtag:nets 1.0
post:21 hashtag:penn 1.0
post:22 hashtag:nets 1.0
post:23 hashtag:nets 1.0
post:24 hashtag:aws 1.0
user:8 user:10 0.09999999999999999
user:10 user:7 0.075
user:2 user:3 0.09999999999999999
user:10 user:8 0.075
```

After the initial graphing, we perform the logic of adsorption for 15 iterations, or before convergence (when the maxDiff < 0.05) as seen with the code below:

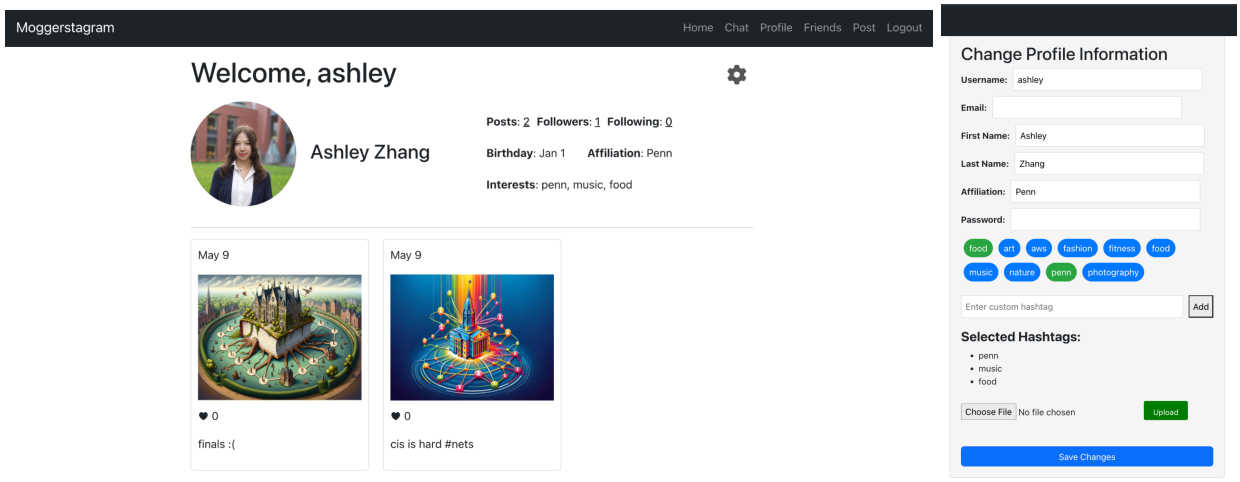
```
// Combine all edges
JavaPairRDD<String, Tuple2<String, Double>> edges = friendEdgeWeights
                                                    .union(likesEdgeWeights)
                                                    .union(userhashtagsEdgeWeights)
                                                    .union(hashtaguserEdgeWeights)
                                                    .union(posthashtagsEdgeWeights);

edges.foreach(x -> System.out.println(x._1() + " " + x._2()._1() + " " + x._2()._2()));

// users
JavaPairRDD<String, Tuple2<String, Double>> nodes = userhashtagsCounts.mapToPair(x -> new Tuple2<>(x._1(), new Tuple2<>(x._1(), 1.0)));
// this one we will not change
JavaPairRDD<Tuple2<String, String>, Double> usersConstant = nodes.mapToPair(x -> new Tuple2<>(new Tuple2<>(x._1(), x._1()), 1.0));
// this one we will change weights
JavaPairRDD<Tuple2<String, String>, Double> usersChanging = nodes.mapToPair(x -> new Tuple2<>(new Tuple2<>(x._1(), x._1()), 1.0));

// perform adsorption
for (int i = 0; i < 15; i++) {
    //
    JavaPairRDD<String, Tuple2<String, Double>> newNodes = nodes.join(edges).mapToPair(x -> new Tuple2<>
        (x._2._2._1, new Tuple2<>(x._2._1._1,x._2._1._2 * x._2._2._2)));

    JavaPairRDD<String, Double> wL = newNodes.mapToPair(x -> new Tuple2<>(x._1(), x._2()._2()));
    JavaPairRDD<String, Double> Lv = wL.reduceByKey((x, y) -> x + y);
    JavaPairRDD<String, Tuple2<Tuple2<String, Double>, Double>> newNodesJoinLv = newNodes.join(Lv);

    // want to normalize the Lv to L1 norm
    newNodes = newNodesJoinLv.mapToPair(x -> new Tuple2<>(x._1(), new Tuple2<>(x._2()._1()._1(), x._2()._1()._2() / x._2()._2())));
    JavaPairRDD<Tuple2<String, String>, Double> newNodesChanging = newNodes.mapToPair(x -> new Tuple2<>(new Tuple2<>(x._1(), x._2()._1()), x._2()._2()));

    newNodesChanging = newNodesChanging.reduceByKey((x, y) -> x + y);
    newNodesChanging = newNodesChanging.subtractByKey(usersConstant);
    newNodesChanging = newNodesChanging.union(usersChanging);

    Double maxDiff = newNodesChanging.join(usersChanging).mapToDouble(x -> Math.abs(x._2()._1() - x._2()._2())).max(Comparator.naturalOrder());
    System.out.println("MAXDIFFFFFF: " + maxDiff);

    if (maxDiff < 0.05) {
        break;
    }
```

Interests/Hashtags

We wanted to utilize these for natural language search and also wanted to make it easy for users. When users create posts, we parse the content/caption and add hashtags/interests found. Then, we suggest these to users upon registration and upon updating their profile (this is where they can view the most recent top 10 hashtags/interests).

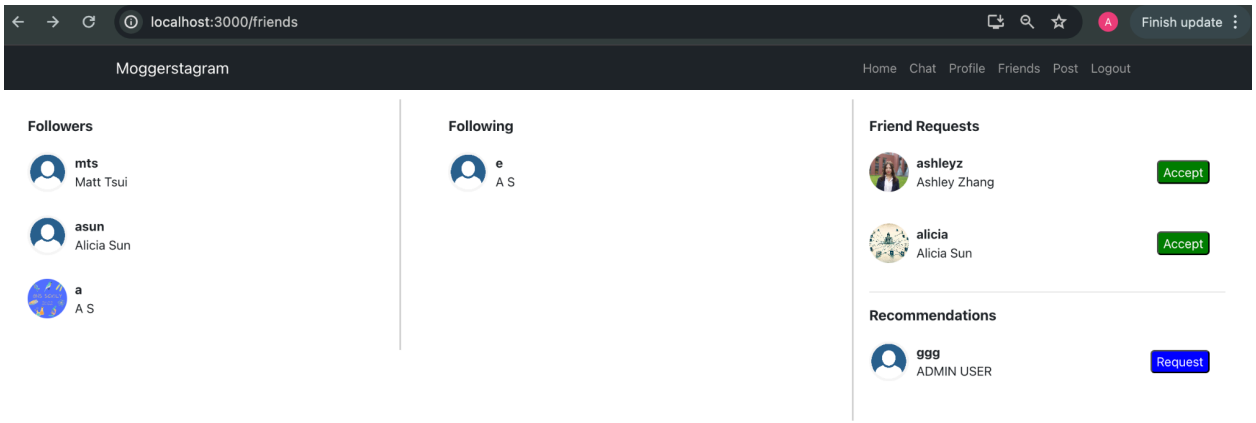
Secondary Screens

Profile Page



We wanted to display the profile similarly to Instagram. We have the user posts displayed below, from most recent to least. The posts links to feed, and followers and following links to friends. The settings gear leads to an edit profile page, where you can select different interests, change your profile picture, and edit other information.

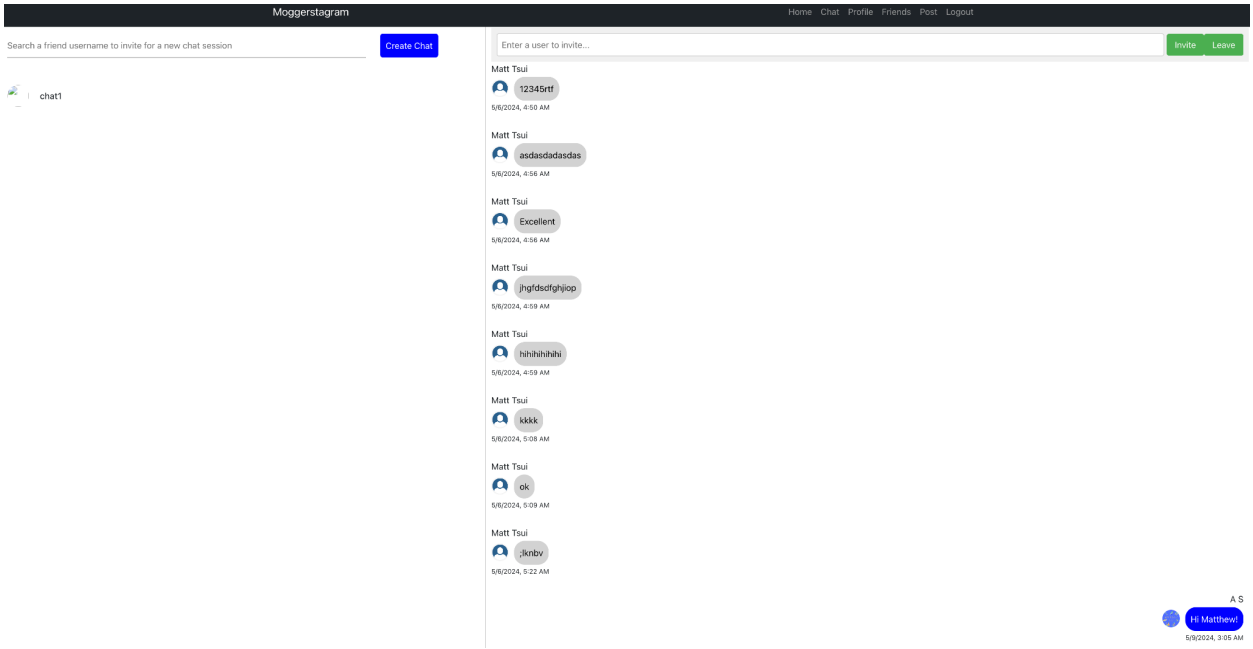
Adding/Removing Friends



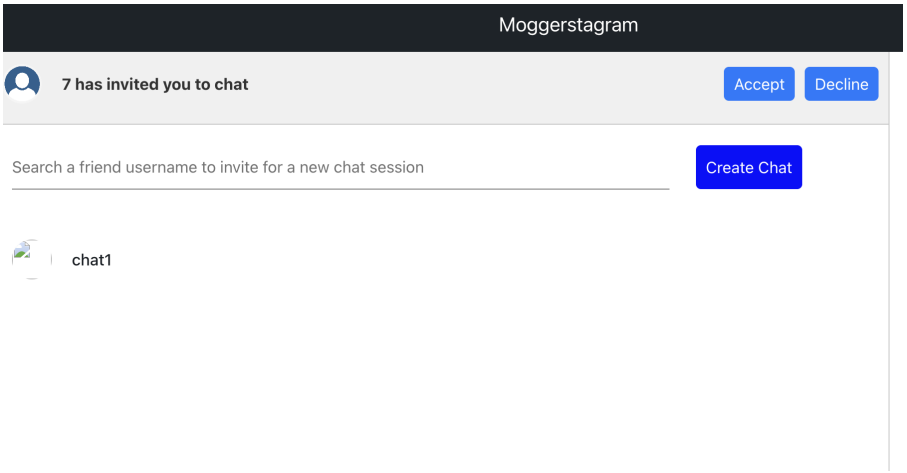
Upon acceptance of a friend or request of a recommendation, the user accepted/requested disappears from their respective lists. Accepting a friend request automatically moves the friend into the followers list. You can click on any profile in this page, and you'll see the profile of that person.

Chatting

In this page of our application, users can select a conversation from a chat they are already in. You can send messages at the bottom and press send. There are two types of invites here. If you want to start a new chat, you must invite only one person and you can do so on the left side. Once you have a chat, you can click on it and invite people to the chat, making it a group chat. These buttons are on the right panel as shown.

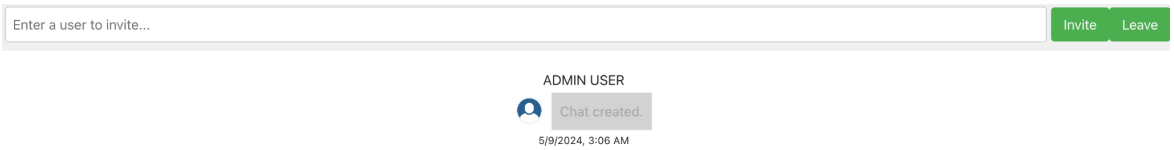


Once they click it, the application will load all the messages sent within the chat. They will be joined into a websocket room corresponding to that chat, so that they can access incoming messages in real time with others in the same room/chat. Users can also invite others to chat by typing in their username first. The invite will appear on the left bar if they receive any.

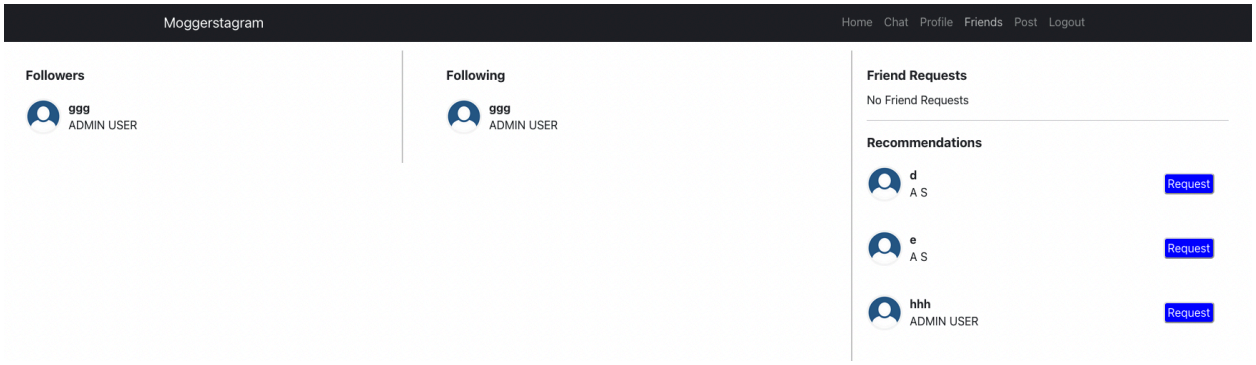


The input must be exactly matching their username. Users can leave a chat at any time and also can be invited into a chat room by others.

Additionally, whenever the chat is created, or people leave or join, a different kind of message is sent which is from the admin system. This is aligned in the middle to distinguish this from the rest as shown below.



Recommendations



We integrated FriendsofFriendsSpark.java that computes the recommendation list every hour (we use Thread.sleep to accomplish this) - here, we can see that the user has an id of 12, and is only friends with one person (user with id of 10), and so recommendations will give recommendations of the users with ids 7, 8 and 11, since 10 is friends with them. We also implemented the recommendations so that the highest recommendation also has the greatest strength (number of mutual connections).

follower	followed
3	2
2	3
2	4
3	4
2	5
3	5
4	5
7	6
11	6
6	7
8	7
10	7
11	7
5	8
7	8
10	8
11	8
7	10
8	10
11	10
12	10
6	11
7	11
8	11
10	11
10	12
25	26

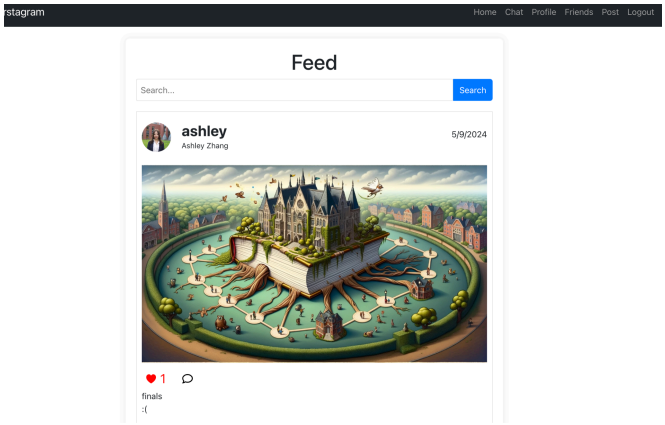
27 rows in set (0.00 sec)

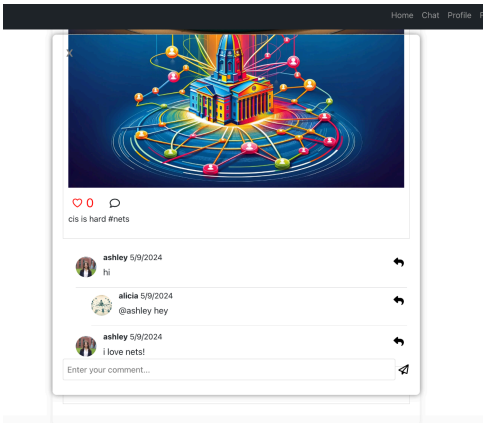
```
mysql> SELECT * FROM recommendations;
```

person	recommendation	strength
2	8	1
3	8	1
4	8	1
5	10	1
6	8	2
6	10	2
7	12	1
8	6	2
8	12	1
10	6	2
11	12	1
12	7	1
12	8	1
12	11	1

14 rows in set (0.00 sec)

Feed





The feed displays the ranked posts. This shows federated and twitter posts, along with posts by friends of the user. The user can like a post by pressing the heart file, which toggles the heart button and increments the like count. They can unlike the post by pressing the heart again, which decrements the like count. The user can expand a post to view its details and comment on it by clicking the comment button. In the expanded view, the user can post comments, along with also replying to comments by clicking the reply arrow next to each comment. These comments are updated immediately and displayed in a threaded comment view. We did this by first finding the comments with no parent comments, then finding comment threads from there.

Decision Changes

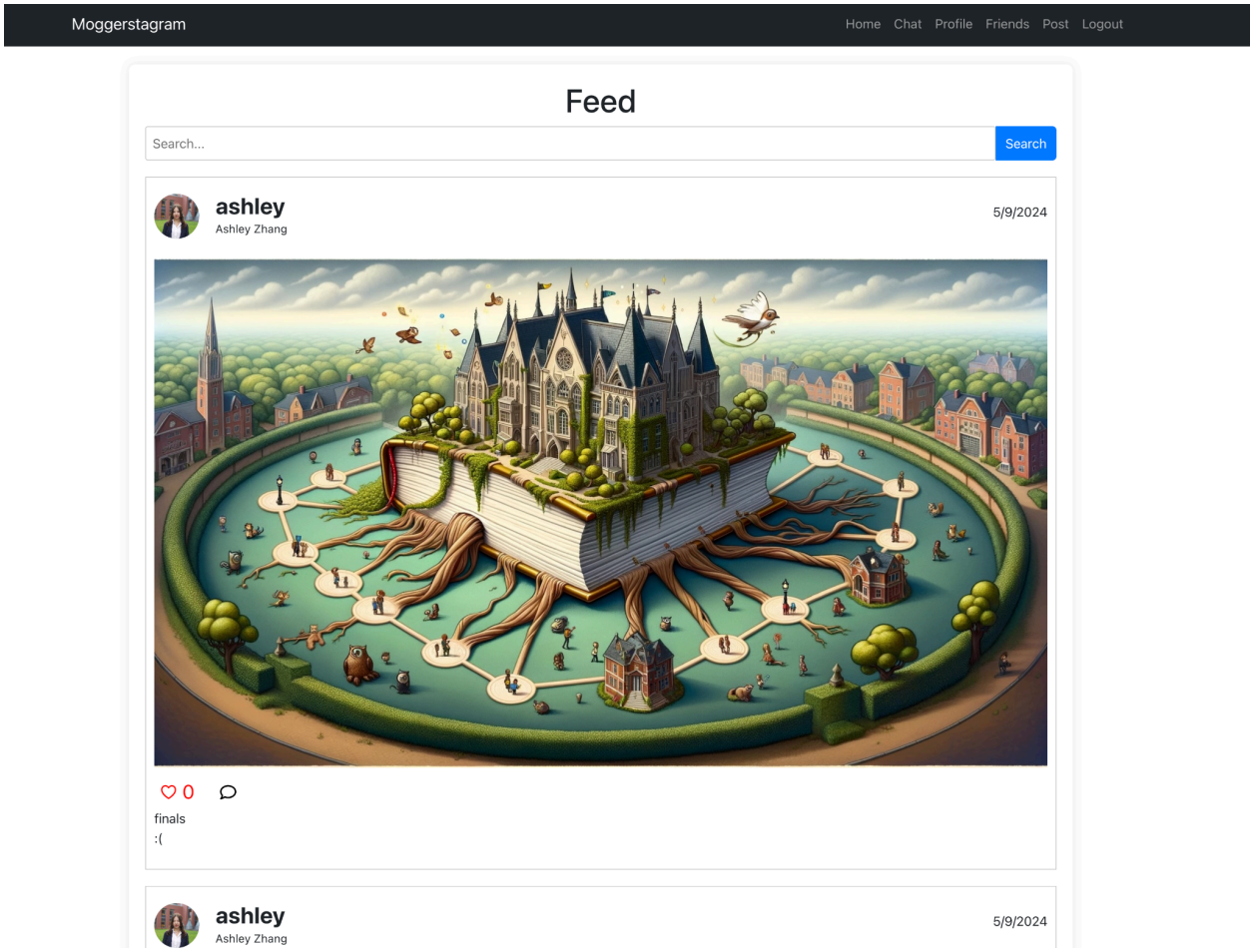
Realized that chat needed several tables. Initially, we stored it as a comment-separated string, but realized the more efficient way would be through another table, called user_chats, to enforce that we didn't need to read the string, edit the string(append to the end), every time we wanted to add/remove a user from the chat. Similarly, we did the same for likes, comments, and so on. Not only would this speed up our code (fewer reads into the table when a number of actors go up), but we're protecting our data through less rds calls.

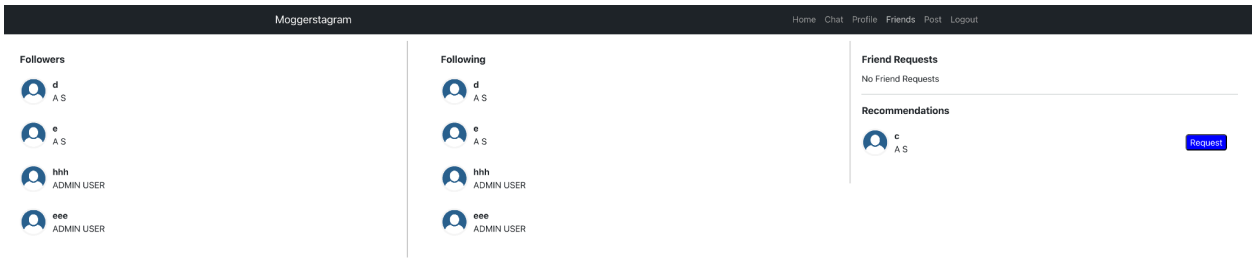
We stored the images through the s3 frontend for profile photos/posts instead of sending the backend, so that everyone could access the image through s3, this is better than storing it otherwise through the backend as that's a lot slower.

ChatRequestsSuper was main as a superset of the previous data table so we didn't need to delete it. It holds an extra field which is the origin. This holds the origin of the chat request, which represents the id of the source. If the chat request is for a new conversation one on one, then the chat id doesn't exist yet so we set it to -1. This was important for making group chats.

Extra Credit

Infinite scrolling,
Sockets used for chat conversations where users join a room and wait on messages
Friend requests
Natural language always returns valid links.


Moggerstagram
Home Chat Profile Friends Post Logout
Feed
Search...
Search
ashley
Ashley Zhang
5/9/2024
0
finals
:(
ashley
Ashley Zhang
5/9/2024


Moggerstagram
Home Chat Profile Friends Post Logout
Followers
d
A S
e
A S
hhh
ADMIN USER
eee
ADMIN USER
Following
d
A S
e
A S
hhh
ADMIN USER
eee
ADMIN USER
Friend Requests
No Friend Requests
Recommendations
c
A S
Request

Moggerstagram

Home Chat Profile Friends Post Logout

Welcome, b

A S

Posts: 9 **Followers**: 3 **Following**: 1

Birthday: Jan 1 **Affiliation**: Upenn

Interests:

May 8

♥ 0

#aws

May 8

♥ 1

#nets #penn

May 8

♥ 0

paragraph 1 paragraph2

May 8

♥ 0

'#nets #penn'

May 8

Post

♥ 0

'#nets #insta'

May 7

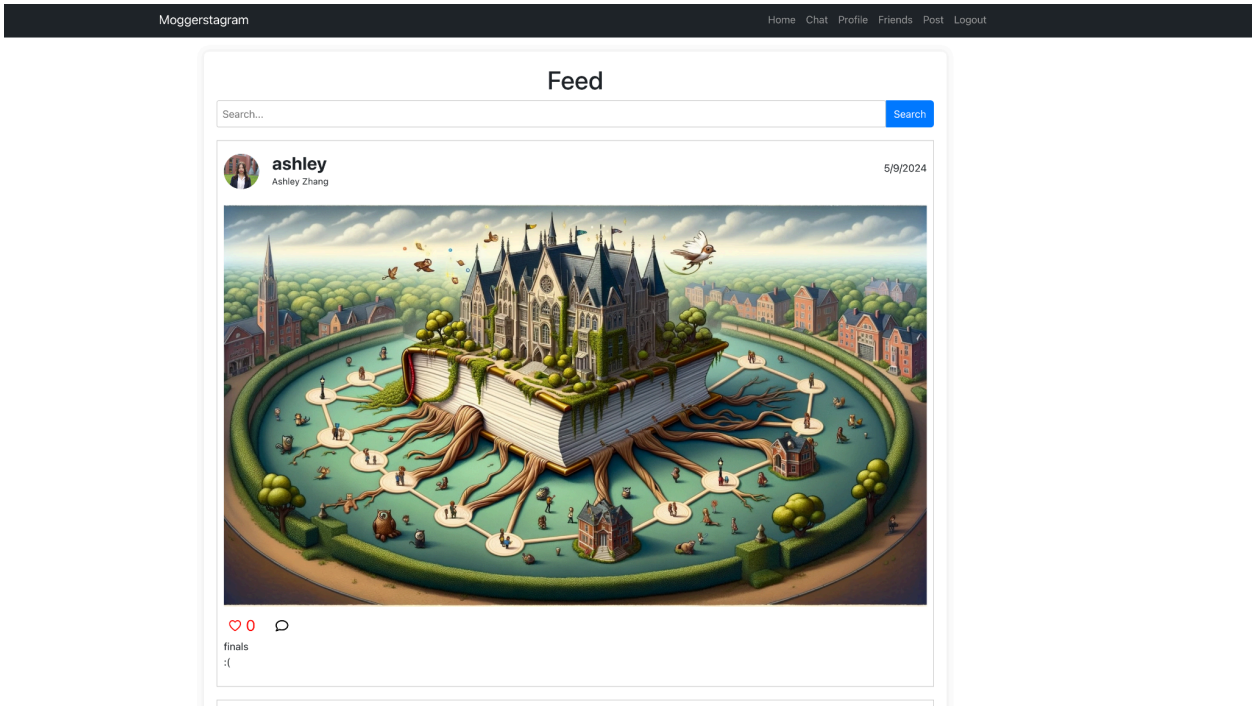
♥ 1

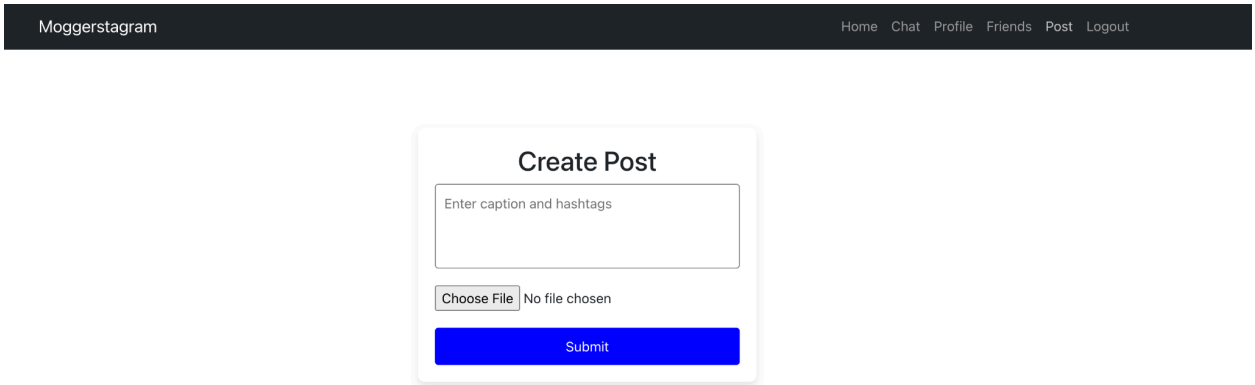
'hihi jsonify :(('

May 7

May 7

May 7


Moggerstagram
Home Chat Profile Friends Post Logout
Feed
Search...
Search
ashley
Ashley Zhang
5/9/2024
0
finals
:(


Moggerstagram
Home Chat Profile Friends Post Logout
Create Post
Enter caption and hashtags
Choose File
No file chosen
Submit