Team G43 Leftovers
Irene Kim kirene@sas.upenn.edu
John Majernik johnmaj@seas.upenn.edu
Kevin Lee kleekev@sas.upenn.edu
Mantas Viazmitinas mantasv@seas.upenn.edu

Overview of the system

The system is a social media platform that allows users to register, login, manage their profiles, interact with friends, post content, engage with hashtags, participate in chats, and more. It incorporates features like user registration, login authentication, profile management, hashtag management, infinite scrolling for posts, friend requests, real-time chat using WebSockets, Kafka streaming for Twitter and federated posts, Apache Spark for post and user ranking, S3 storage for large object storage, image search using face-matching algorithms, and semantic search using ChromaDB.

Technical description of the components

User Registration

User registration is handled through SignupForm.jsx, which creates a user registration form using React. The form captures essential user information like username, password, name, email, school, birthday, and allows users to upload a profile photo and include hashtags of interest as well. The uploaded photo is then compared to images in the actor database and the top five most similar actor images are displayed for the user to choose from. It also includes functionality for adding and removing hashtags of interest, with a list of top ten hashtags (by post occurrence) for reference. Once submitted, the component sends a POST request to a local server endpoint /api/users/signup to register the user. Successful registration redirects the user to the home page and additionally, there is a link to log in for existing users.

Login

Login is handled through LoginForm.jsx, which is responsible for rendering a user login form using React. It manages form state for email and password using the useState hook. useUserContext accesses the user context to dispatch login actions upon successful authentication. This component employs useNavigate to handle redirection upon successful login, directing users to the home page. When the form is submitted, it sends a POST request to the backend API endpoint /api/users/login for user authentication. If the login attempt succeeds, it stores user information in local storage and updates the user context, followed by redirecting the user to the home page. Additionally, it provides a link to sign up for new users.

Profile

Profile changes are handled through Profile.jsx, which allows for a user profile management interface using React. It employs React's useState hook to manage form state, including fields for updating the user's profile photo, first name, last name, email, password, and hashtags of interest. Users must edit at least 1 field, but can choose which fields they would like to update. Utilizing useParams from react-router-dom, it extracts the userId from route parameters, enabling personalized profile updates. Upon form submission, it sends a PATCH request to the backend API endpoint /api/users/:userId, updating modified profile fields.

Hashtags

This backend route /top10 retrieves the top 10 hashtags based on their occurrence frequency in the posts. It executes a SQL query that joins the post_hashtags table with the hashtags table, grouping the results by hashtagId and ordering them by count in descending order. The frontend code in the SignupForm and Profile components fetch these top hashtags upon component mounting using the useEffect hook. It then displays these hashtags in a list format. Users can select hashtags of interest by clicking on them, and these selected hashtags are added to the formData state. Users can also manually add and remove hashtags from their interests. When the user submits the signup form, the selected hashtags are included in the form data sent to the backend for user registration. This mechanism allows users to indicate their interests by selecting from the top hashtags or adding custom ones during signup.

Infinite Scrolling

Home.jsx implements an infinite scrolling mechanism to load posts dynamically as the user scrolls down. It uses the react-infinite-scroll-component library to achieve this functionality. Initially, it fetches a batch of posts from the server using the fetchFeed function when the component mounts. As the user scrolls down, the next prop of InfiniteScroll triggers the fetchFeed function again to load more posts. The hasMore prop controls whether there are more posts to load, and if so, a loader is displayed while new content is being fetched. Once all the posts are loaded, the endMessage is displayed, indicating that there are no more posts to fetch. Overall, this setup ensures a smooth browsing experience by dynamically loading posts as the user explores the feed.

Friends

There are multiple ways to become friends with one another. One is to go to a user profile and send a friend request. Another is to recommend a user by the algorithm. The user will then get a friend request and will have to wait until the user accepts in order to be friends. This is all done through API calls to the backend. The notifications of receiving requests are done through socket.io.

Chat

The chat was implemented using socket.io. On the server side, it listens for a connection where it does certain actions based on what the client emits. To chat with a user, a user must be friends with one another and online. Afterwards, they can send a request to the requested user to chat with and they will have to wait until they accept their request. Afterwards they can chat with one another and see messages instantaneously through web sockets. If they want to create a group chat, they can invite another user to a new group chat and if they accept then the group chat will start otherwise it will be deleted.

Kafka Feed

In order to get Twitter and federated posts, we implemented Kafka streaming. There is a cron-job scheduler that allows the function to run once a day. The function initializes a consumer and reads in the posts to add into the database. In order to produce federated post, every time that a user creates a post from the client, there is a function that produces a post and sends it to the FederatedPost topic.

Who to Follow

To implement who to follow, find friends of friends of the user and users that have common hashtags. Afterwards the strength is based on how often a certain user occurs between these two occurrences. This will then be added to the database where the client will use an API call that will query the database. This algorithm is runned once a day and will remove any recommendations where the user has sent a friend request to.

Adsorption

Adsorption ranking of posts and individuals is performed through Apache Spark and the creation of JavaRDD and JavaPairRDDs through parallelization of the persistent data stored in our RDS to represent the user, hashtag, and post nodes. Edges are then created through another JavaPairRDD, as well as label weights, initialized on each of the users with a weight of 1.0. At the beginning of each cycle, operations such as mapToPair and reduceByKey are performed on the RDDs to combine necessary relationships and propagate label weights in the forward from the users to the other nodes. The label weights are then normalized to sum to 1.0, and the same process is performed to propagate the label weights backwards to the users. The weights are then again normalized, and in order to reduce "wash out" on the label weights, we hard code the weight of the origin vertex to always be 1.0. We repeat this process for 15 iterations or until convergence through a for-loop, and then output the label weights for each hashtag and post. Finally, we use this information to sort through the posts and hashtags for each user in terms of descending user label weight in order to determine the posts to suggest to that user. Posts that have already been suggested are not recommended, and this process is repeated every hour.

S3 Storage

We created an S3 bucket for large object storage. This includes the images that are used for profile pictures. In order to upload from the front-end directly to the S3 bucket, we use Multer as middleware to handle form-data, which includes profile pictures, using a connection through the .env variables to the S3 bucket. Once uploaded, the images can be retrieved through the URL assigned to the images. In order to do so, we wrote a custom bucket policy in S3 to allow access to the objects in the bucket.

Image Search

Image search is implemented through a face-matching algorithm provided by the starter files. We use those files to call on the functions to store and access embeddings in ChromaDB. After the uploading of a profile picture, an embedding is created and compared to those of famous actors. The top 5 are chosen by the algorithm and outputted back on the frontend for the user to choose from.

Search

ChromaDB is used to generate and manage embeddings. These embeddings are created from textual content (user details, post contents) to capture their semantic meanings. The system generates embeddings leveraging OpenAI's embedding API. These embeddings are stored and managed within ChromaDB, allowing for quick semantic searches and similarity checks. The search functionality is exposed through a RESTful API, enabling the frontend to perform search queries and display results to users.

Nontrivial design decisions

Database Schema Design:

- Use of normalized tables to reduce data redundancy and ensure data integrity.
- Employing foreign key constraints to enforce referential integrity between tables.
- Separation of concerns by having dedicated tables for posts, comments, friendships, likes, hashtags, chats, groups, etc.

API Specifications:

- RESTful API design to ensure scalability, flexibility, and interoperability.
- Clear endpoints for user management, posts management, comments management, friendships and interactions, chat and communication, search and recommendations

User Management:

- Token-based authentication for user login to enhance security.
- Ability to register, login, update profile, and delete user accounts.
- Top 5 similar actors endpoint to provide personalized recommendations based on user preferences.

Posts Management:

- Endpoint to create, retrieve, and delete posts.
- Like a post functionality to allow users to express their preferences.
- Support for hashtag management with endpoints to fetch top 10 hashtags.

Comments Management:

- Ability to add comments to posts and retrieve comments for a specific post.

Friendships and Interactions:

- Endpoints for managing friendships, including creating, updating, and removing friendships.
- Implementation of LinkedIn-style friend requests with confirmation.

Chat and Communication:

- Creation of chat sessions, sending messages, and fetching chat messages.
- Use of WebSockets for real-time communication to enhance user experience.

Changes made and lessons learned along the way

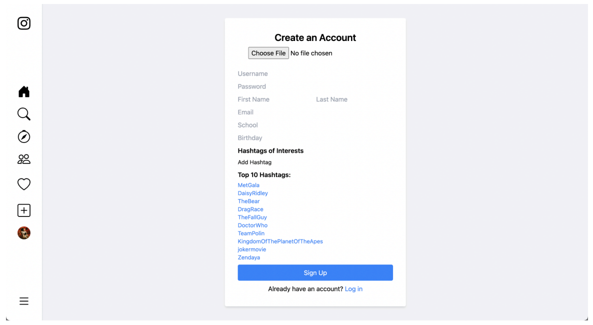
1. New API calls
    a. We added new API calls as the project progressed, such as `/api/users/actors`, `/api/posts/{post_id}`, `/api/posts/{user_id}`, `api/hashtags/top10` to allow for actor face matching, get single posts, get all the posts for a given user's feed, and to get the top 10 post hashtags

Extra-credit features

1. Infinite Scrolling
2. Web sockets using Socket.io
3. Friend requests

Screenshots

User Registration



Create an Account

Choose File No file chosen

Username

Password

First Name Last Name

Email

School

Birthday

Hashtags of Interests

Add Hashtag

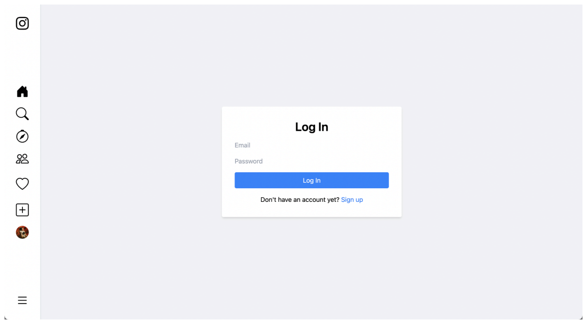
Top 10 Hashtags:

MetGala
DaisyRidley
TheBear
DragRace
TheFallGuy
DoctorWho
TeamPolin
KingdomOfThePlanetOfTheApes
jokermovie
Zendaya

Sign Up

Already have an account? Log in

Login Page



Log In

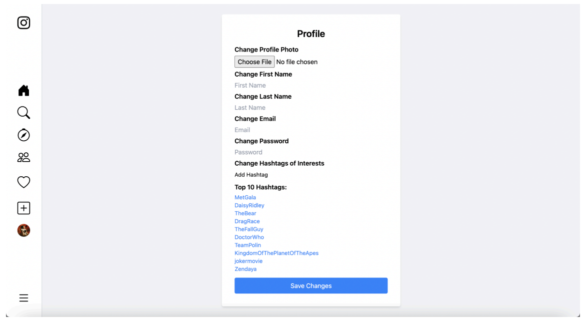
Email

Password

Log In

Don't have an account yet? Sign up

Profile Changes



Profile

Change Profile Photo

Choose File No file chosen

Change First Name

First Name

Change Last Name

Last Name

Change Email

Email

Change Password

Password

Change Hashtags of Interests

Add Hashtag

Top 10 Hashtags:

MetGala
DaisyRidley
TheBear
DragRace
TheFallGuy
DoctorWho
TeamPolin
KingdomOfThePlanetOfTheApes
jokermovie
Zendaya

Save Changes

MySql Database Schema:

1. users Table
   - `user_id` (Primary Key)
   - `username`
   - `password_hash`
   - `email`
   - `first_name`
   - `last_name`
   - `birthday`
   - `profile_photo_url`
   - `created_at`
   - `updated_at`
2. posts Table
   - `post_id` (Primary Key)
   - `user_id` (Foreign Key linked to users)
   - `image_url` (optional)
   - `video_url` (optional)
   - `text`
   - `created_at`
   - `updated_at`
3. comments Table
   - `comment_id` (Primary Key)
   - `post_id` (Foreign Key linked to posts)
   - `user_id` (Foreign Key linked to users)
   - `text`
   - `created_at`
   - `updated_at`
4. friendships Table
   - `user_id_1` (Foreign Key linked to users)
   - `user_id_2` (Foreign Key linked to users)
   - `status` (e.g., pending, accepted)
   - `created_at`
   - `updated_at`
5. likes Table
   - `like_id` (Primary Key)
   - `post_id` (Foreign Key linked to posts)
   - `user_id` (Foreign Key linked to users)
6. hashtags Table
   - `hashtag_id` (Primary Key)
   - `text`
7. postHashtags Table
   - `post_id` (Foreign Key linked to posts)
   - `hashtag_id` (Foreign Key linked to hashtags)
8. userHashtags Table
   - `user_id` (Foreign Key linked to users)
   - `hashtag_id` (Foreign Key linked to hashtags)
9. chats Table
   - `chat_id` (Primary Key)
   - `name`
   - `created_at`
   - `updated_at`
10. chatUser Table
    - `chat_id (Foreign Key linked to chats)`
    - `user_id (Foreign Key linked to users)`
    - `status (pending, joined)`
11. chatMessages Table
    - `message_id` (Primary Key)
    - `chat_id` (Foreign Key linked to chats)
    - `sender_id` (Foreign Key linked to users)
    - `text`
    - `timestamp`
    - `created_at`
    - `updated_at`
12. groups Table
    - `group_id` (Primary Key)
    - `group_name`
    - `created_at`
    - `updated_at`
13. groupMemberships Table
    - `group_id` (Foreign Key linked to groups)
    - `user_id` (Foreign Key linked to users)
    - `role` (e.g., admin, member)
    - `joined_at`

API Specification

User Management

- Create User: `POST /api/users/register`
- Login User: `POST /api/users/login`
- User Top 5 Similar Actors: `POST /api/users/actors`
- Update User Profile: `PATCH /api/users/{user_id}`
- Delete User: `DELETE /api/users/{user_id}`

Posts Management

- Create Post: `POST /api/posts`
- Get User Posts: `GET /api/posts/user/{user_id}`
- Get Post: `GET /api/posts/{post_id}`
- Delete Post: `DELETE /api/posts/{post_id}`
- Like a Post: `POST /api/posts/{post_id}/likes`

Hashtag Management

- Get Top 10 Hashtags: `GET /api/hashtags`

Groups Management

- Create group: `POST /api/groups`
- Add member to group: `POST /api/groups/{group_id}/members`
- Fetch group posts: `GET /api/groups/{group_id}/posts`

Search and Recommendations

- Search Users/Posts: `GET /api/search`
- Fetch trending posts: `GET /api/posts/trending`
- Fetch recommendations: `GET /api/recommendations`

Password Management

- Request password reset: `POST /api/users/forgot-password`
- Reset password: `POST /api/users/reset-password/{token}`

Privacy Settings

- Update post visibility: `PATCH /api/posts/{post_id}/visibility`

Comments Management

- Add Comment to Post: `POST /api/posts/{post_id}/comments`
- Get Comments for Post: `GET /api/posts/{post_id}/comments`

Friendships and Interactions

- Get Friendships: `GET /api/friends/{user_id}`
- Create Friendship: `POST /api/friends/{user_id}`
- Update Friendship Status: `PATCH /api/friends/{user_id}`
- Remove friendship: `DELETE /api/friends/{user_id1, user_id2}`
-

Chat and Communication

- Create chat session: `POST /api/chats`
- Accept chat session: `POST /api/chats/requests`
- Send chat session: `POST /api/chats/send`
- Get chat sessions: `GET /api/chats`
- Send message: `POST /api/chats/{chat_id}/messages`
- Fetch chat messages: `GET /api/chats/{chat_id}/messages`