InstaLite Project Report

Andrew Chang, Daniel Li, Grace Deng, Wesley Liu

Overview

InstaLite is a full-stack, Instagram inspired social media platform with support for posting images, real time chat, and personalized feeds powered by social connections and content-based ranking. The frontend end is a React application that enables users to sign up, log in, and browse from an infinite-scroll personalized feed. There are also secondary screens for profile management, friend lists, chat interface, and natural language search. On the backend, we use Node.js to handle authentication, session management, and REST endpoints for connecting the frontend and backend. All of our data is stored on the cloud via AWS, providing security and scalability for our system.

Technical Components

Frontend

The frontend serves as the interface between our users and our instalite application. It is written in Typescript, uses the React framework, and has custom components (ActorCard, CreatePost, Post, and SearchForm) to display information on our different screens (Login, Signup, Profile, Home/Feed, Create Post, Chat, Natural Language Search, and Friends). The frontend queries our backend routes using axios to obtain and then display information, in order to separate our frontend and backend logic. Our styling was implemented using TailwindCSS for ease of development.

One extra credit component we implemented on the frontend was infinite scrolling in the feed. We utilized React's infinite scrolling component, which repeatedly calls our feed data fetching endpoint with different parameters for the offset and limit to correctly get and display new elements.

Backend

The backend serves as the core interface between our frontend application, persistent data storage, and external compute services. Built with Node.js and Express, it supports user authentication, social graph operations, feed interactions, chat, and personalized recommendations. Two major architectural enhancements include the integration of Kafka for real-time streaming and Apache Spark via Livy for scalable batch computation.

We defined multiple routes for authentication and user interactions. New users register with a username, password, nconst, first name, last name, email, birthday, and affiliation; upon registration, users upload a selfie photo, whose embedding is matched against actors' photos' embeddings stored in ChromaDB. User-uploaded selfie photos are stored in an S3 bucket, which can be accessed for displaying the user profile and for changing profile photos. To ensure sensitive user information is stored securely, we used bcrypt to salt and hash the user password, which is then stored in DynamoDB.

We integrated Kafka for real-time streaming via producer and consumer event handlers. The consumer runs in the background and processes posts from the stream in real time and sends them to our application's databases. The producer sends posts whenever the route for post creation is used.

These days, social media platforms are well known for their "algorithms." In our app, we use three recommendation systems to personalize each user's experience:

1. Social Ranking: Using a PageRank-adjacent algorithm on the friend network, this computes the most "popular" or influential users in the graph. These scores are used to rank friends and affect the visibility of posts from those users.
2. Post Ranking: Based on a label adsorption algorithm, this operates over a heterogeneous graph of users, posts, hashtags, likes, and comments. Labels start on user nodes and diffuse throughout the graph to assign each post a relevance score, personalized to the user viewing it.
3. Friend Recommendations: This is based on a friends-of-friends graph traversal, identifying likely connections based on mutual relationships and proximity in the social graph.

Because these algorithms can be computationally intensive, we offload them to a remote Spark cluster. Jobs are written in a Java-based Spark module and submitted via Livy, which is hosted on an AWS EMR cluster. To avoid performance issues associated with large memory operations (e.g., .collect()), the Spark jobs read from and write directly to our AWS RDS instance via JDBC. This design allows us to scale recommendation computations without burdening the backend.

Additionally, our backend includes a CRON scheduler that periodically triggers these Spark jobs to refresh the social rank, post weights, and friend recommendations. This lets us deliver timely and relevant content to users while keeping the app responsive and performant.

We used WebSockets to implement the Chat Mode feature, rather than polling, since WebSockets offered the ability to see other users' messages and respond in real-time. Users connect to a socket right when they log in, so that we can update the database of online users and send a real-time message to all other online users. This allows us to determine which users are online (for creating group chats, adding friends, etc.) instantly, instead of having to refresh or poll every few minutes. The online users table also stored the corresponding socket id of each user, so that we would be able to send messages to those sockets.

Design Decisions

One major component of our project was how we chose to design the databases. Nearly every part of our project interfaces with data in some way, and so our goal was to build a clean and robust database system to support it. At a high level, we rely on three different types of databases: Blob Storage (S3), Vector Storage (ChromaDB), and Relational Databases (RDS).

We use Amazon S3 for user generated media like post images, profile photos, and selfie uploads. Uploading image data to S3 keeps our relational databases clean and avoids overloading the system with raw image data. Instead, we can simply store a URL in the relational databases, which is not only much more efficient but also allows us to easily display these photos back to the user.

ChromaDB powers two core features: face matching & RAG (for natural language search). Since ChromaDB is a vector database, we can pre-index a lot of data beforehand, and, at runtime, easily query ChromaDB to return similar embeddings. This would've been very difficult if we had just stored this information in a relational database.

The bulk of our data, however, is stored in relational databases:

User profiles are captured in the *users* table, which stores authentication data (username & hashed password), profile metadata (name, email, birthday, affiliation), and user interests (hashtags). Additionally, the *users* table also contains links to S3 objects for the user's profile & selfie pictures.

Content tables (*posts, comments, likes*) store information that eventually gets displayed on the feed. In the *posts* table, we include all the post content as well as a column for 'source' to differentiate local content from federated/Bluesky content, allowing our ranking logic to properly compute a user's feed. Since *comments* and *likes* both follow a one-to-many pattern (each post can have many comments or likes), each table has a foreign key reference to a unique post_id. Additionally, because InstaLite supports threaded comments, each entry in the comment table has a unique comment ID and a parent ID, which essentially creates a forest of comments for each post. This relationship allows us to easily render nested comments in the frontend, as we can simply recursively render each comment tree.

Friendship and friendship recommendations are also stored in edge tables (*friends, friend_recs*). This schema allows us to easily compute friend recommendations and feed recommendations (*social_rank, post_weights)* from the *friends* table while also allowing us to track existing relationships.

Finally, our chat system relies on the database *chat_rooms, chat_members, chat_messages,* and *chat_invites.* These tables collectively store data in a normalized form and efficiently track any room metadata, membership, history, and pending invites. The foreign key constraints we added between tables allows us to make sure that we are correctly updating all tables in the backend whenever users connect or disconnect, as well. Finally, the *online_users* table leverages web sockets to track users that are currently logged into the system, allowing users to see who else is online.

Overall, we spent a lot of time fleshing out designs for our databases. We wanted to ensure that data was stored properly and queries would execute efficiently as many of our endpoints construct complex queries over our data.

Changes & Lessons Learned

Serialization for Spark + Livy

A key challenge in running Spark jobs via Livy was ensuring serialization compatibility. Since Livy sends jobs from our backend to a remote Spark cluster, all passed objects must be serializable—something that breaks if Spark executors can't access local environment context like .env files.

To solve this, we avoided passing our full AppConfig object. Instead, we passed a simple Map<String, String> of environment variables, then reconstructed AppConfig inside the Spark job itself, ensuring all setup occurred within the executor JVM and avoiding NotSerializableException.

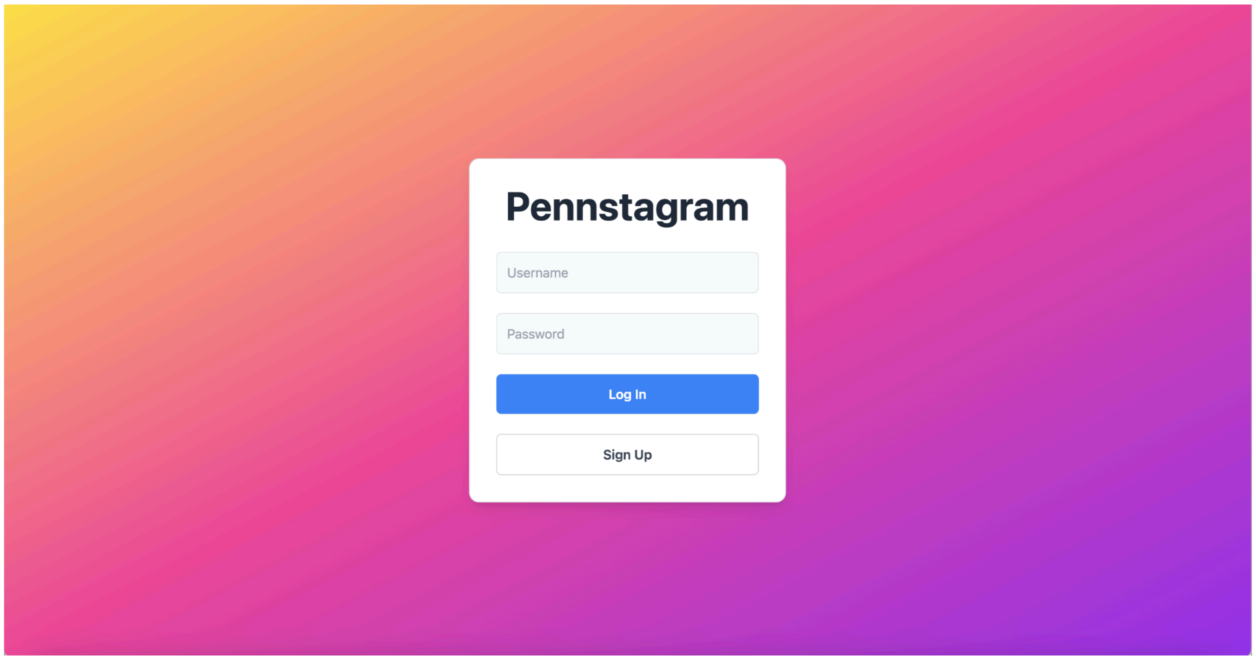
We also used Maven's shade plugin to bundle all dependencies into a single .jar, allowing seamless deployment and execution on the cluster. This approach kept backend and Spark logic cleanly separated and made the Spark module portable and easy to test.

Extra Credit

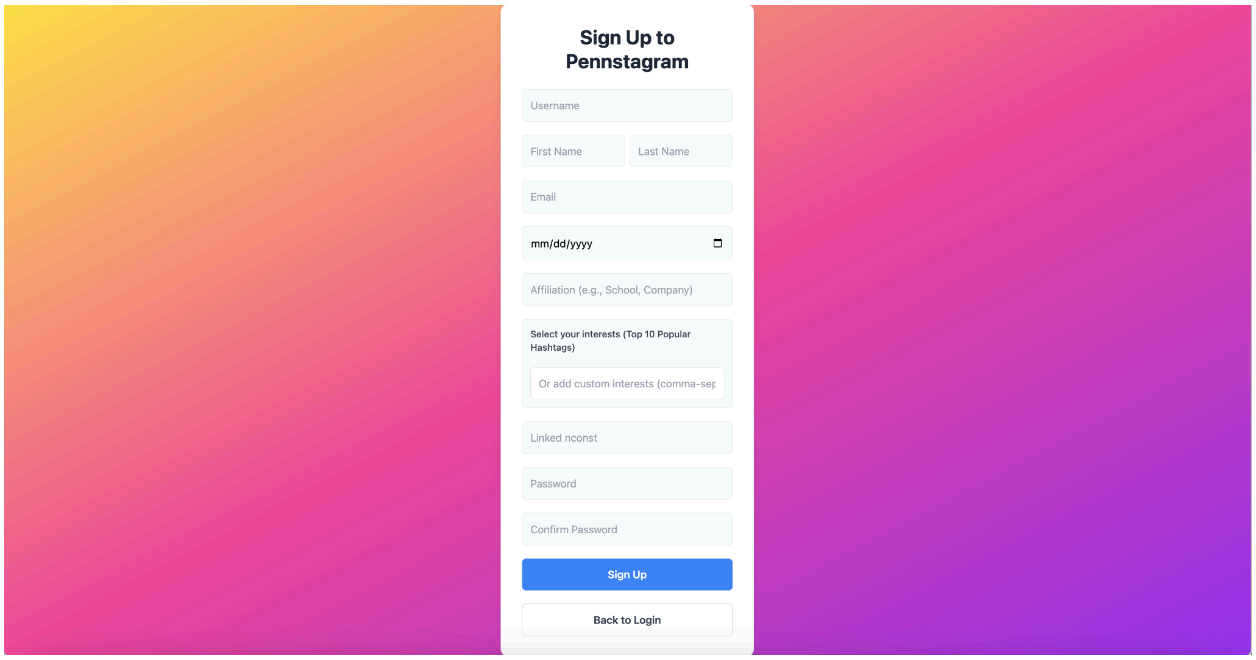
(+2) Infinite scrolling
We implemented infinite scrolling in the frontend for the feed page. Whenever a user scrolls to the bottom, our backend route will automatically be queried to retrieve more posts (with the right offset and limit), and then display the results when the user continues scrolling down.

(+3) WebSockets for chat;
We used WebSockets in the backend for chats. When users log in, they automatically connect to a WebSocket. Chats between users will be real-time and instant, as the user does not have to refresh the page or wait for some sort of polling method for the new messages to show up.

Images



Login Page



Signup Page

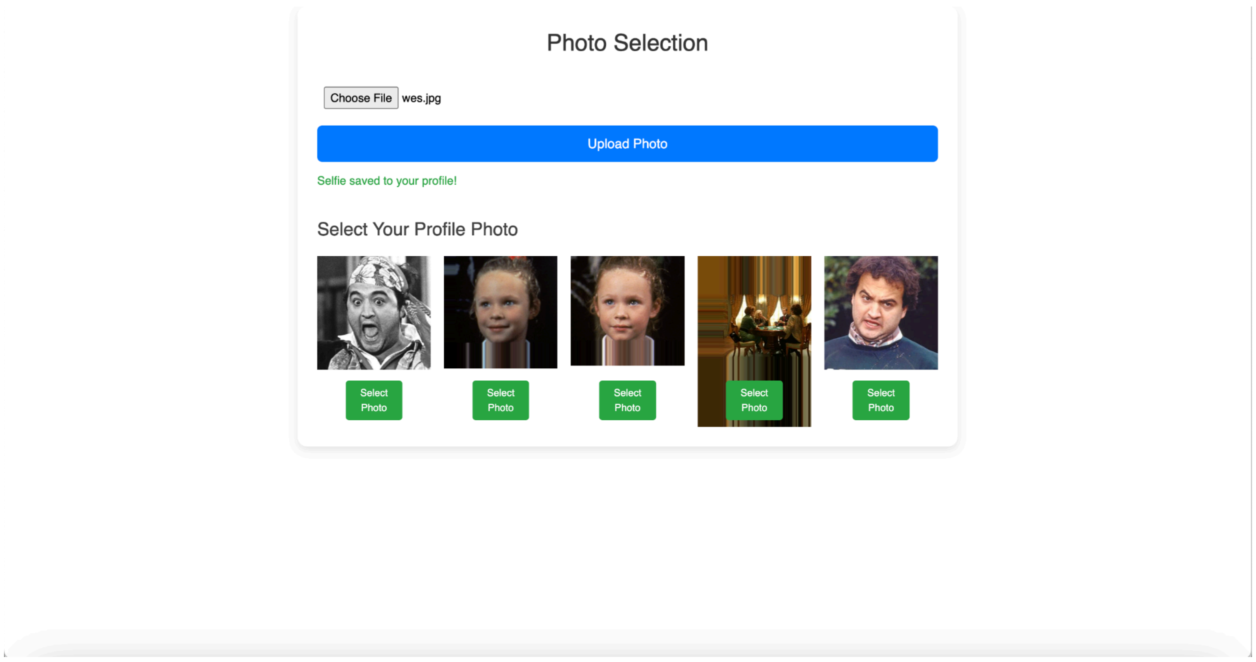


Photo Selection

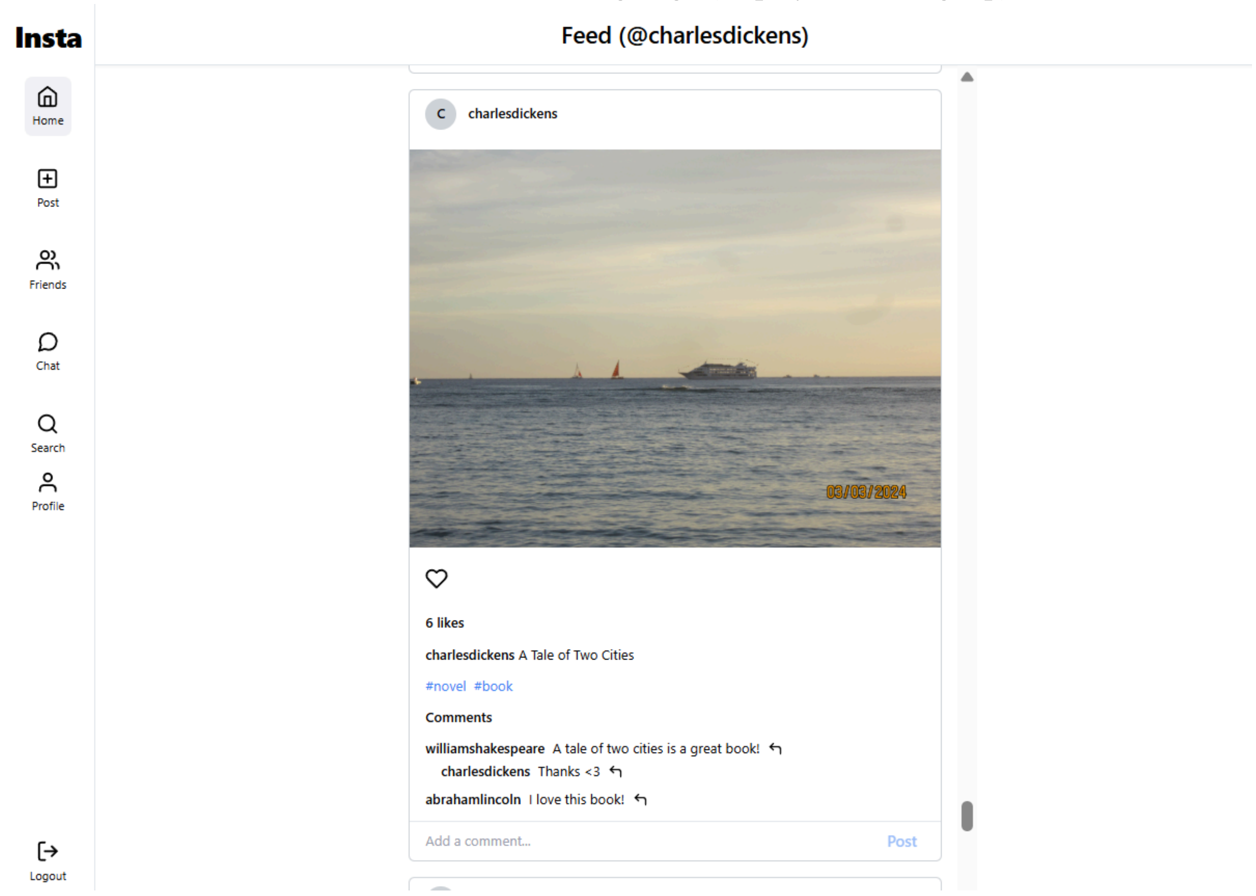
Choose File wes.jpg

Upload Photo

Selfie saved to your profile!

Select Your Profile Photo

Photo Selection/Actor Linking Page (displayed after signup)



Feed (@charlesdickens)

charlesdickens

6 likes

charlesdickens A Tale of Two Cities

#novel #book

Comments

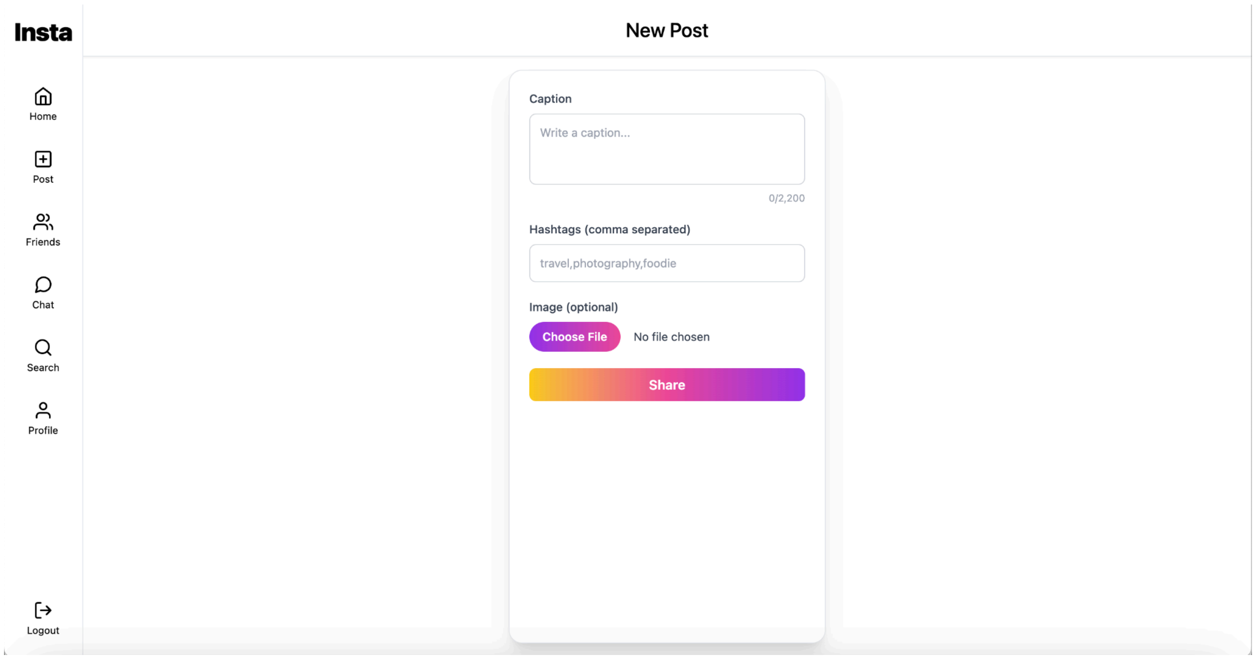
williamshakespeare A tale of two cities is a great book!

charlesdickens Thanks <3

abrahamlincoln I love this book!

Add a comment... Post

Home/Feed Page


Insta

Home
Post
Friends
Chat
Search
Profile
Logout

New Post

Caption

Write a caption...

0/2,200

Hashtags (comma separated)

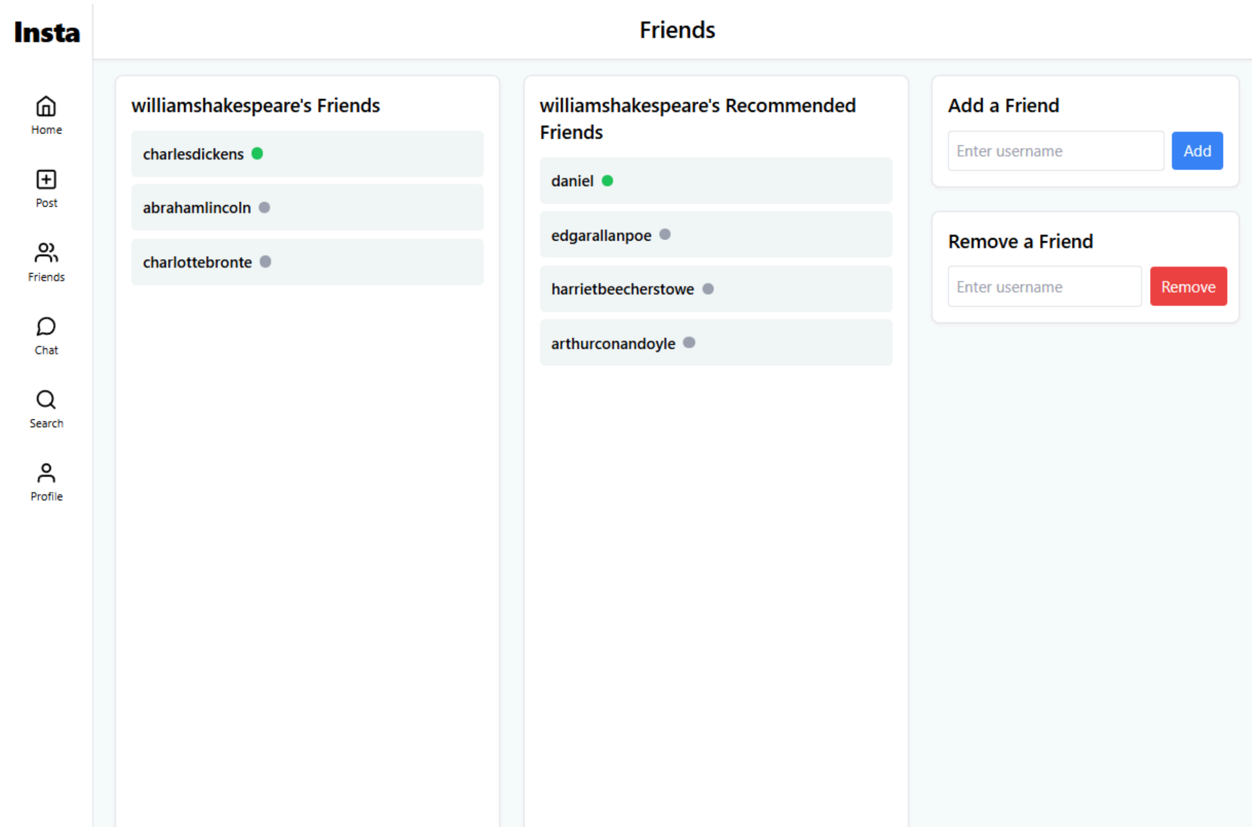
travel,photography,foodie

Image (optional)

Choose File No file chosen

Share

Create Post Page


Insta

Home
Post
Friends
Chat
Search
Profile

Friends

williamshakespeare's Friends

charlesdickens ●
abrahamlincoln ●
charlottebronte ●

williamshakespeare's Recommended Friends

daniel ●
edgarallanpoe ●
harrietbeecherstowe ●
arthurconandoyle ●

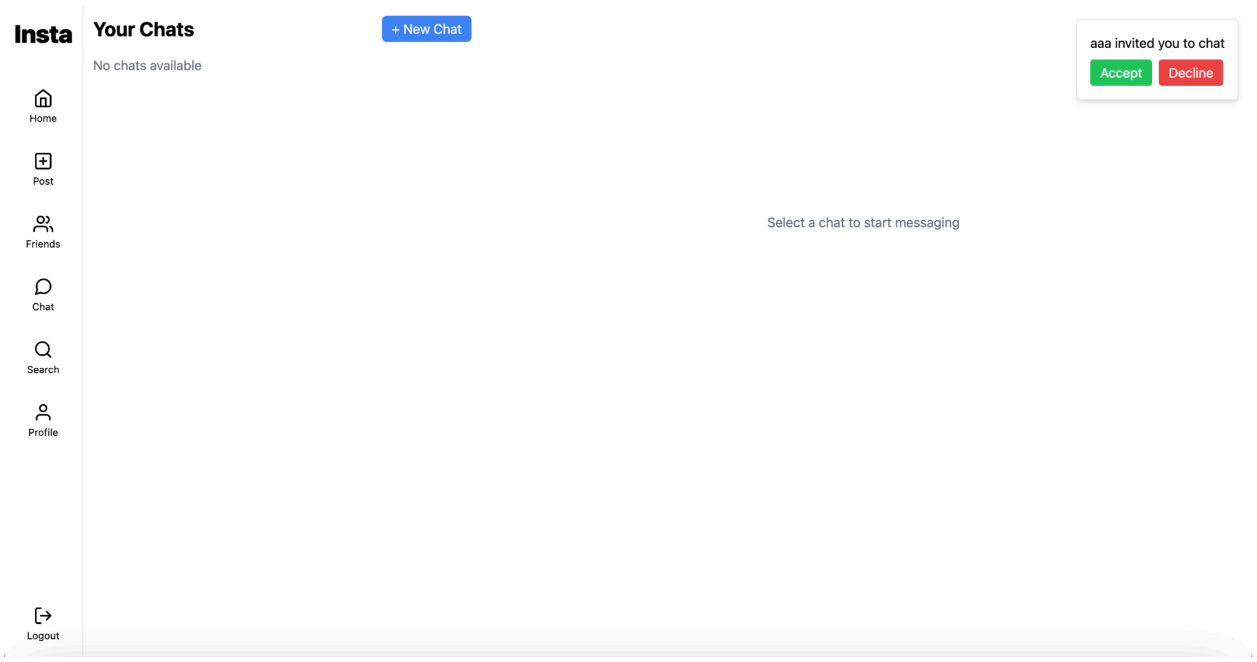
Add a Friend

Enter username Add

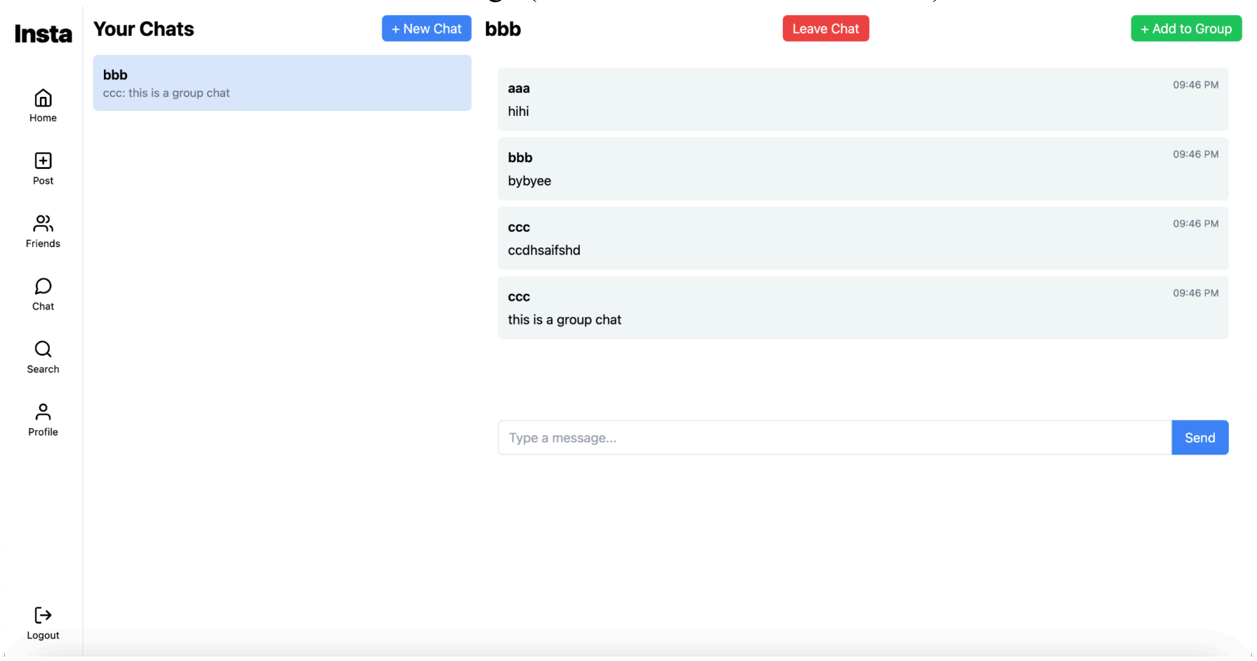
Remove a Friend

Enter username Remove

Friend Page



Chat Mode Page (when a user is invited to a chat)



bbb
ccc: this is a group chat

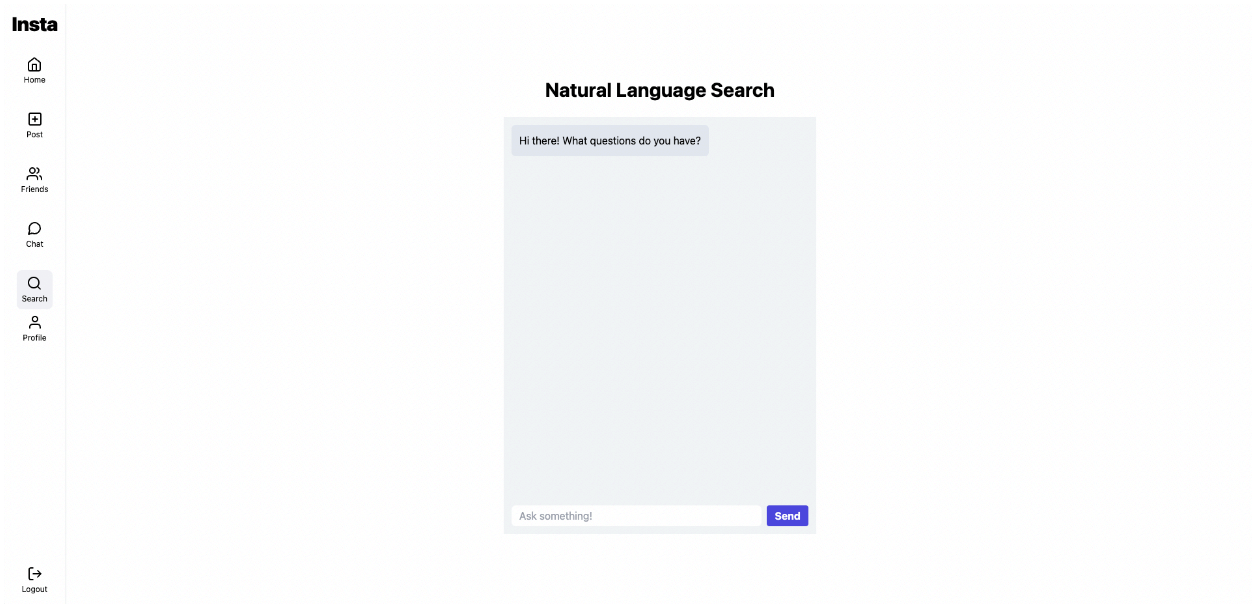
aaa 09:46 PM
hihi

bbb 09:46 PM
bybyee

ccc 09:46 PM
ccdhsaifshd

ccc 09:46 PM
this is a group chat

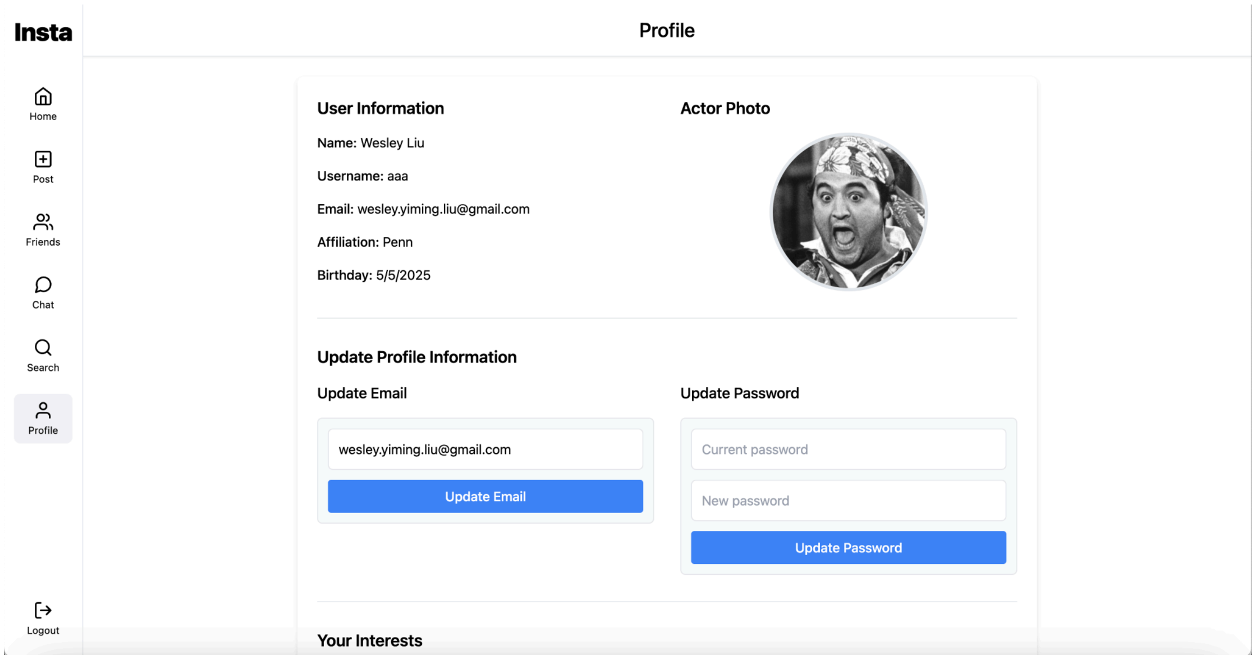
Chat Mode Page (after a user accepts a group invite and begins chatting)



Natural Language Search

Hi there! What questions do you have?

Natural Language Search Page



Profile

User Information

Name: Wesley Liu

Username: aaa

Email: wesley.yiming.liu@gmail.com

Affiliation: Penn

Birthday: 5/5/2025

Actor Photo

Update Profile Information

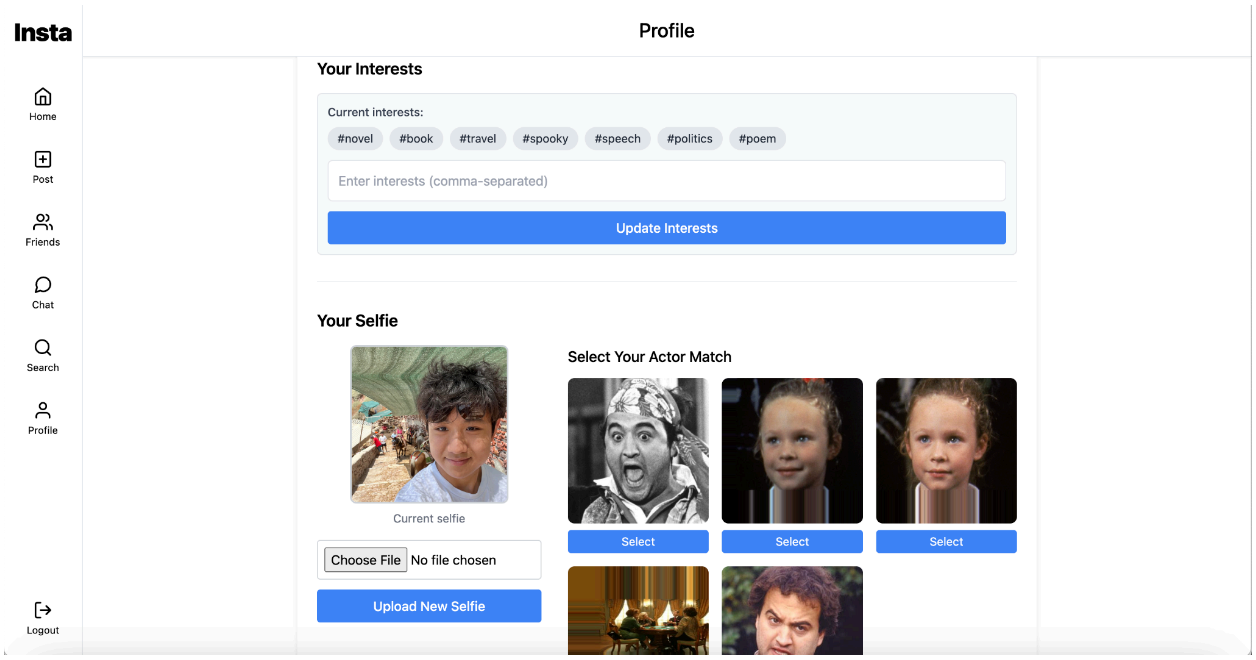
Update Email

wesley.yiming.liu@gmail.com

Update Password

Your Interests

Profile Page (top part with information and changing information section)



Profile

Your Interests

Current interests:

#novel #book #travel #spooky #speech #politics #poem

Enter interests (comma-separated)

Update Interests

Your Selfie

Current selfie

Choose File No file chosen

Upload New Selfie

Select Your Actor Match

Select Select Select

Profile Page (switching what actor they are linked to and changing hashtags)