Name: Shrinivas Vasant Shanbhag
UIN: 934008523

Social Networking Application (LinkStream)

1. Introduction:

In today's interconnected world, the ability to reach and communicate with people across the globe has become essential. Achieving this level of connectivity requires robust distributed computing practices and reliable web applications capable of supporting real-time interactions. With this goal in mind, I developed a distributed social networking application designed to demonstrate how modern distributed systems can power global, real-time communication.

The application (LinkStream) consists of both a frontend and a backend, each playing a critical role in the overall user experience. The frontend provides a clean, responsive, and user-friendly interface through which users can sign in, follow others, send messages, and view their timeline. The backend is composed of three coordinated microservices. The authentication service manages user registration, login, and the assignment of users to specific chat servers. It also oversees the distributed architecture by adding new servers when needed and continuously monitoring chat server health.

The chat server is responsible for handling user communications, follow/unfollow actions, and generating personalized timelines. It operates in a master–slave configuration to ensure high availability and fault tolerance. If the primary server fails, the backup server seamlessly takes over, maintaining uninterrupted service so that users remain unaware of any internal failover events. This distributed design ensures reliability, scalability, and resilience—key qualities for any real-time social networking platform. Finally we have a database service, which will be used to persist all the data into the database.

2. Key Learnings:

Working on this project gave me the chance to grow significantly in both frontend and backend development. I set out to improve my skills across the full stack, and this project provided the right level of challenge and hands-on learning.

On the frontend, I spent time creating an interactive, user-friendly interface using TypeScript. I also refined my approach to building clean and responsive layouts with modern CSS, making sure the client felt intuitive and worked well across different screen sizes.

On the backend, I deepened my knowledge of Java and became more comfortable building server-side applications using the Spring framework. I also learned more about how microservices communicate, how to monitor their health, and how to organize backend logic in a modular way.

The most impactful part of the project was working with distributed systems. I gained practical experience setting up a master–slave architecture, handling server failover, and ensuring the system stays available even when individual components fail. I also learned different strategies for dealing with distributed system challenges such as synchronization, consistency, state sharing, and fault tolerance. These lessons helped me build a more reliable and scalable application and gave me a much clearer understanding of how large distributed systems operate in real-world environments.

3. Architectural Design:

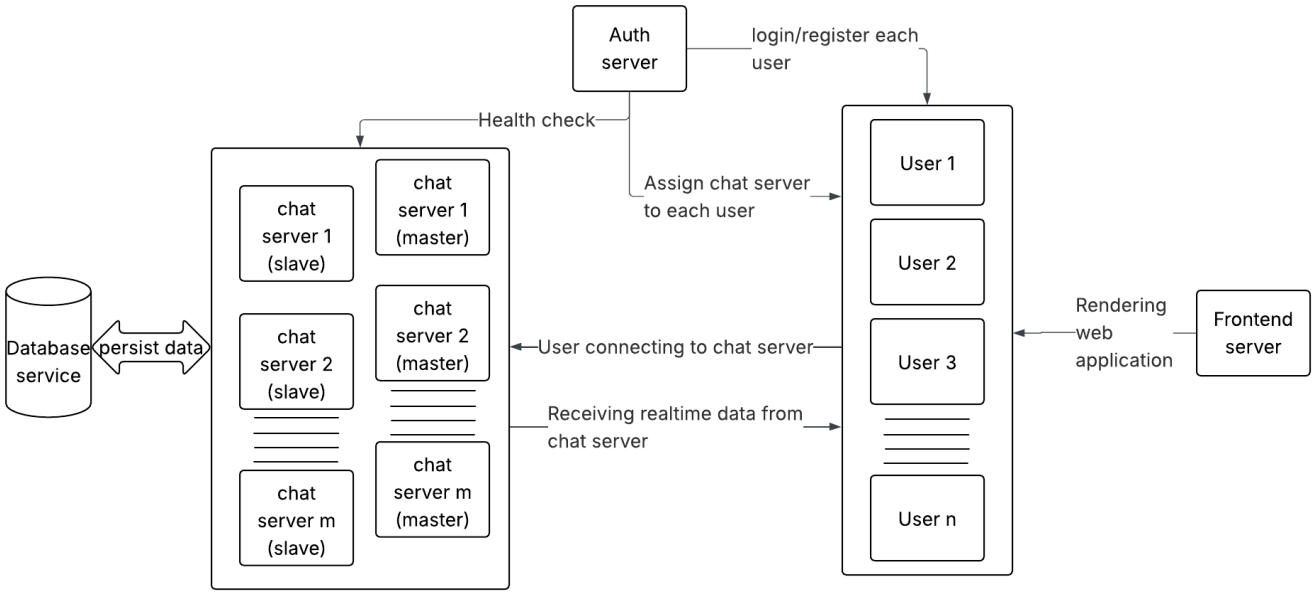

Figure 1: Architectural Design

The system architecture for this project is designed to support a distributed, highly available social networking platform. It breaks the application into independent services that work together to deliver real-time functionality while ensuring reliability and fault tolerance. The diagram above provides an overview of how these components interact within the overall system.

The architecture consists of the following key components:

i. Frontend server:

This server renders the web app pages to the user's browser and manages all client-side interactions. It is built using TypeScript and CSS, providing a clean, responsive, and interactive user interface. The frontend also uses WebSockets to maintain real-time communication with the backend servers for live updates.

ii. Users:

In this system, multiple users begin by connecting to the frontend server, which delivers the web application interface to their browsers. From there, users can log in using their existing credentials or register as new users. The authentication process is handled by the auth-server, which validates the credentials and then assigns each user to a specific chat server in the distributed network. This ensures that users are efficiently distributed across available chat servers for better scalability and performance.

iii. Auth server:

This is the single server, developed using java and spring boot framework. It is using Rest APIs to connect and share information with web clients and backend servers. This server takes care of user authentication, then assigns chat servers to each of the registered users. It monitors the health of each of the chat servers, if any master/main chat server is down, then it will assign its backup server as a new master server, and communicate this new master server ip address to all the users connected to that server. It stores user information in the database using database service.

iv. Chat server:

This server operates as part of a master–slave setup in a distributed system, where multiple master–slave pairs handle requests from different sets of users. It is implemented using Java Spring Boot and leverages both REST APIs and WebSocket connections to communicate with the database service, auth server, and frontend clients.The master server handles all user interactions, including chat messages, timeline updates, and follow/unfollow requests. It sends periodic heartbeats to the auth server, enabling the system to detect failures and promote the slave to master if needed. The master also keeps track of database updates and synchronizes with other servers on periodic polling basis to ensure consistency, so that any user trying to connect to another user on a different server is properly routed.

The slave server maintains a real-time copy of the master's state by syncing with the database. It does not directly communicate with users, but it is ready to take over immediately if the master fails, ensuring high availability and fault tolerance. REST endpoints in the server are used for retrieving static or queryable information (e.g., list of followed users, chat history), while WebSocket connections provide real-time updates for chats, timeline posts, and user interactions. This combination ensures that the system remains consistent, responsive, and resilient under distributed workloads.

v. Database service:

The database service serves as the central data store for the application, managing users' followers, chat messages, and posts. Built with Java Spring Boot, it provides REST APIs for other backend services to read and update data, ensuring consistency across the distributed

system. By persisting data in JSON files, it maintains the state reliably, supporting real-time interactions and coordination between chat servers and the frontend.

4. Key features and Use Cases:

i. User Registration & Login:

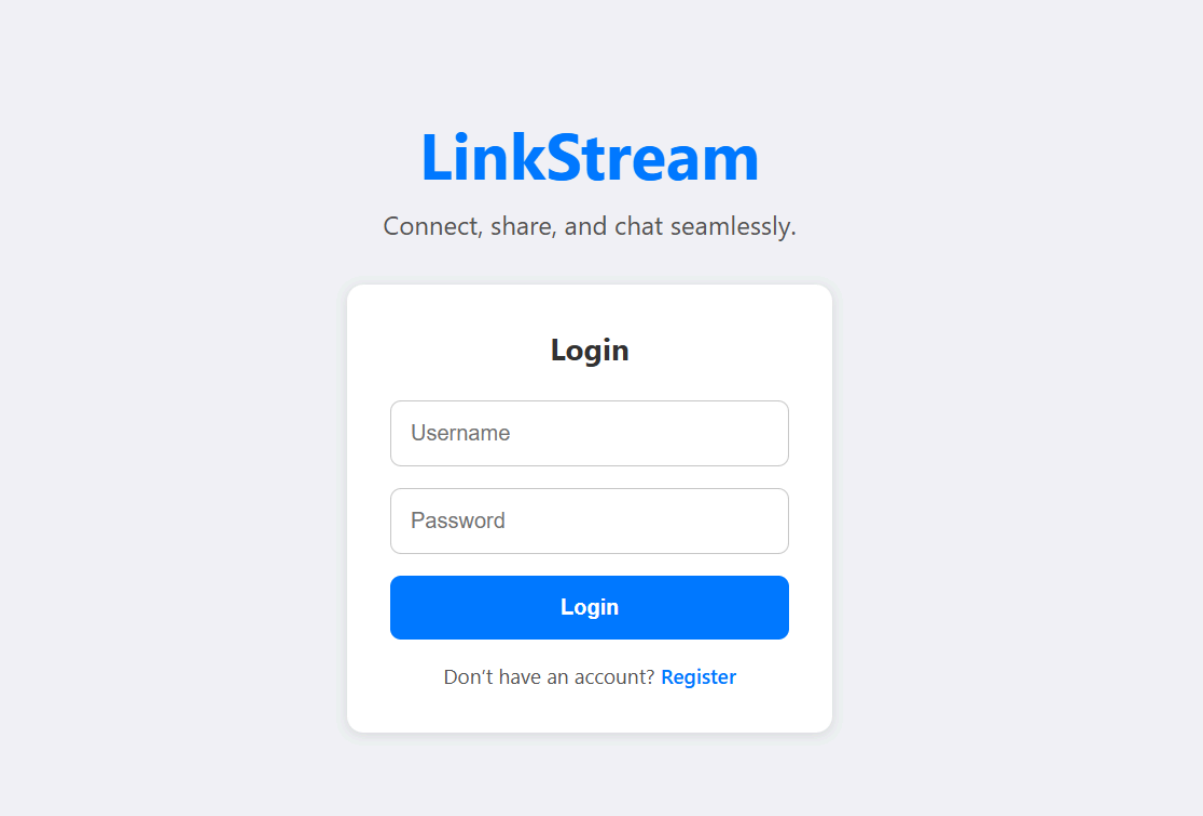

Figure 2: login page

In this web app, the first page will be the login page, it looks like Figure 2. Here users can give valid username and password to validate and access other features/pages of this app. If the user is a new user, then they can click on the register button given in the login page and it will take them to the registration page as shown in Figure 2. These two pages will be making direct Rest API calls to the auth server, once they get authenticated by the auth server, it gets a chat server ip address from auth server. Then the next page will directly connect to a particular chat server ip address. If the user name or password is wrong, then a new pop up page will come telling either username or password is wrong as shown in Figure 2.1.

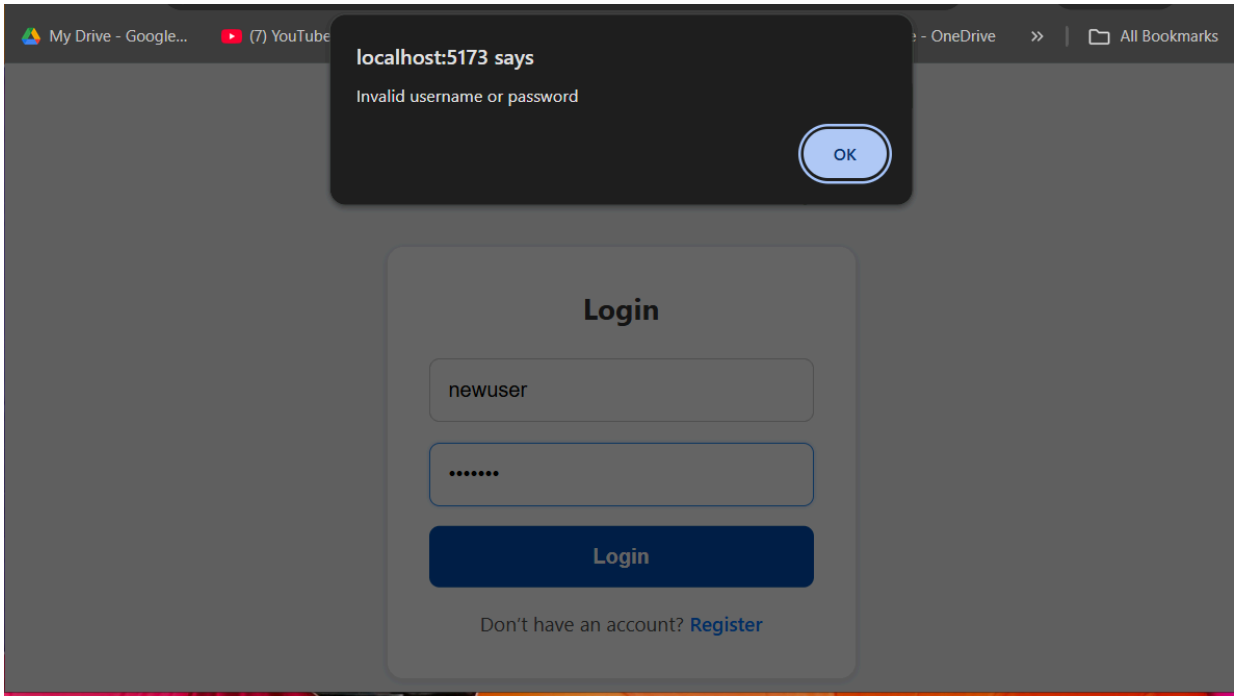

Figure 2.1: invalid username or password

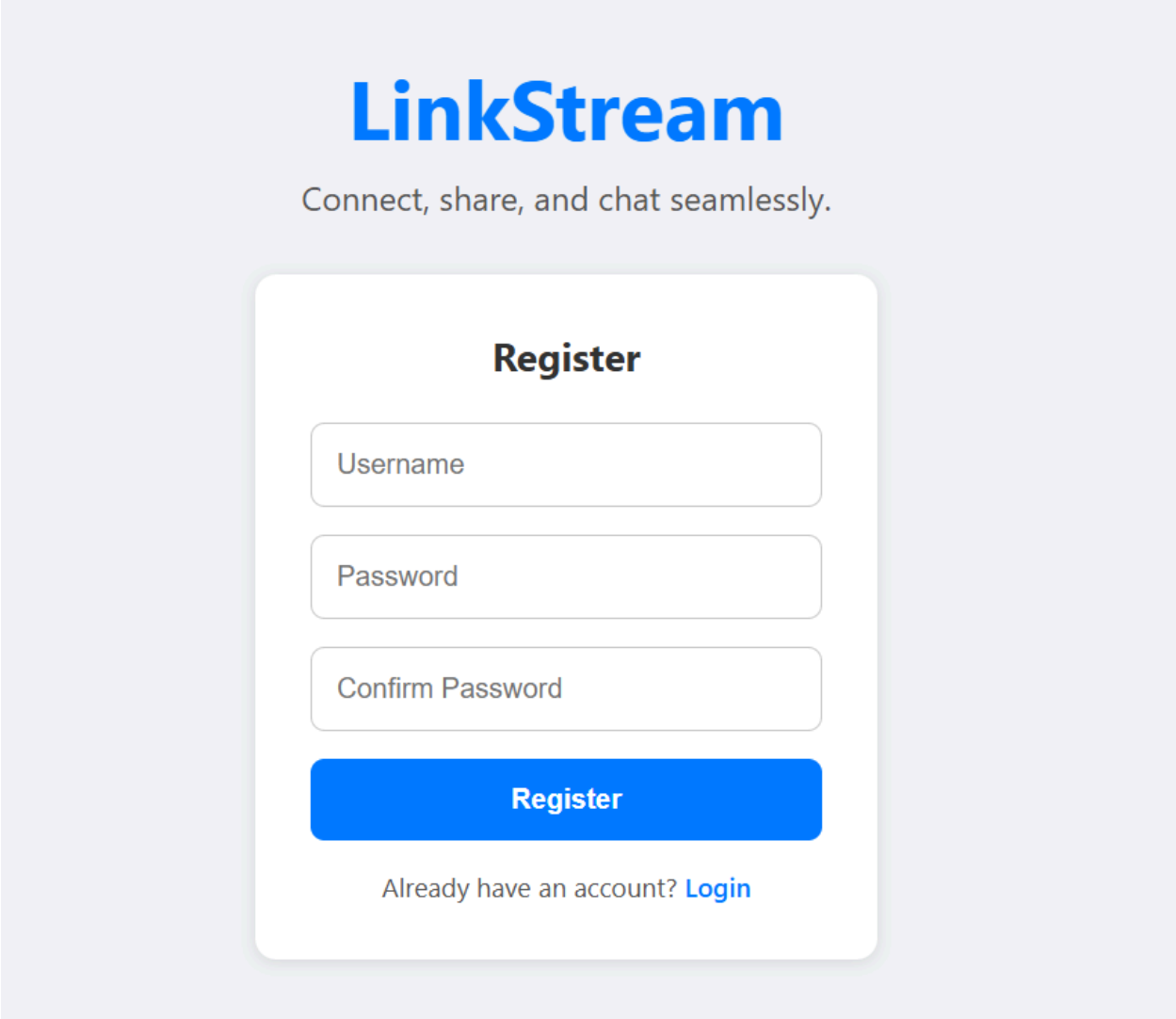

Figure 3: New user registration page

ii. Main page layout:

Figure 4 is the main page after user login is successful. Here on the left top we can see logged in user name, and logout option below it. Below the logout button, there we can see all the users currently using the web application. If we click on the user's name, then it will open up a chat page. Here we can follow any user or unfollow any previously followed users. On the right side we can see the timeline for the current logged in user. Here we are using Rest API calls to chat servers to get all the user information, like who I currently follow and who I don't. Then logout api call to auth server.

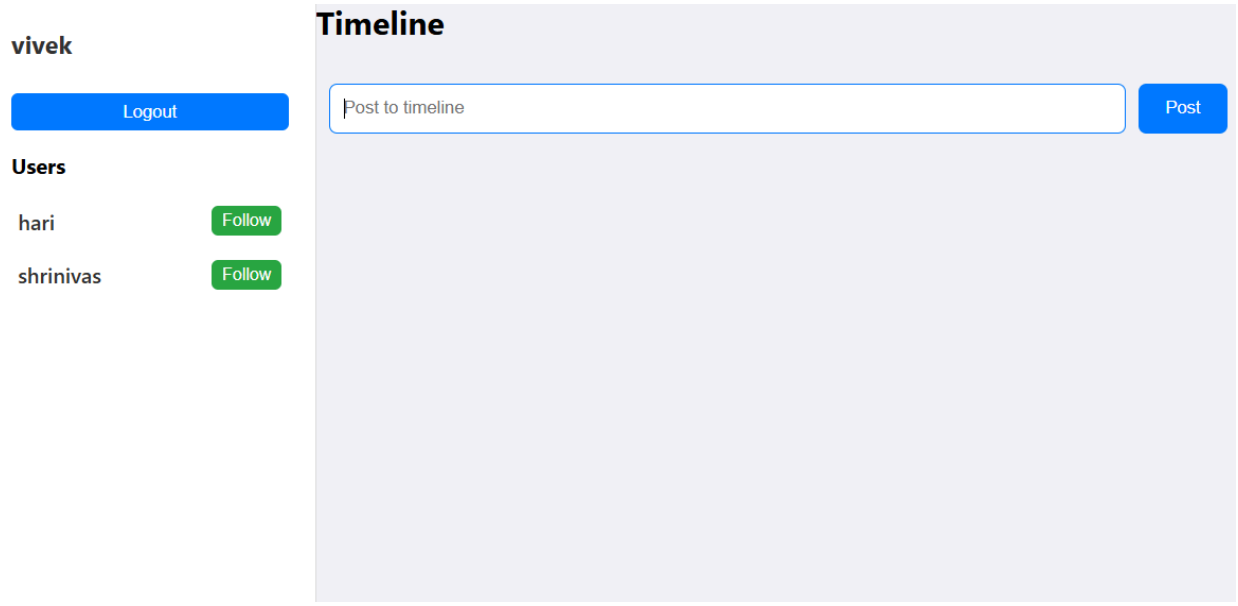

Figure 4: main page

iii. Chat page:

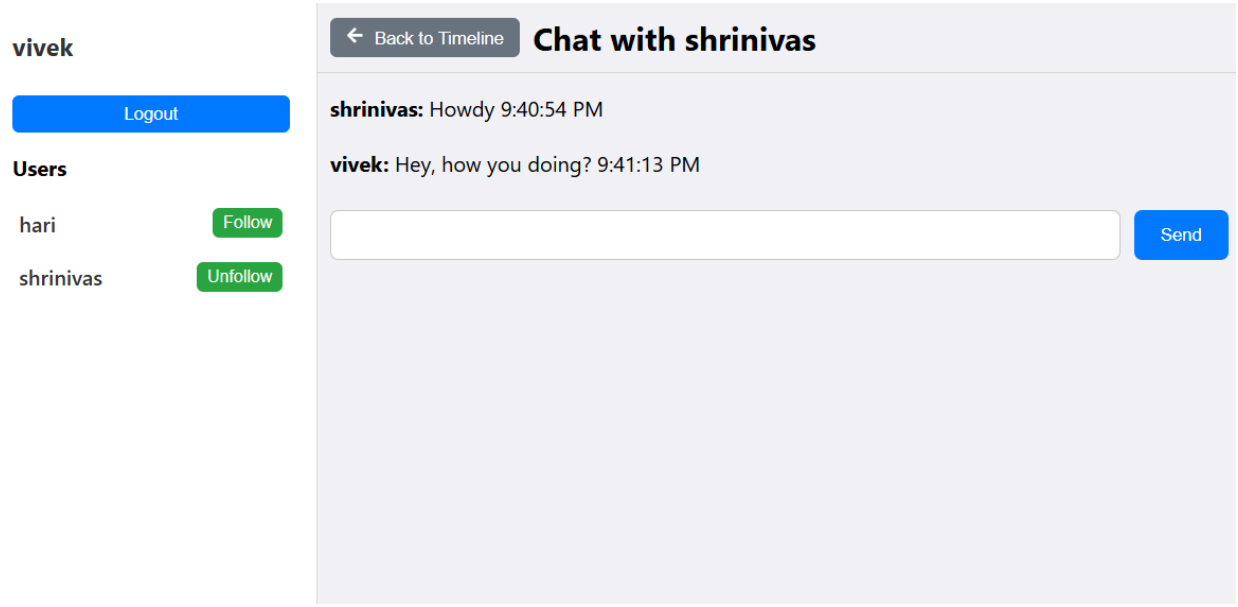

Figure 5: Chat page

If I want to chat with some user, then I just need to click on their name from the users list, then the chat page will open as shown in Figure 5. Here instead of the timeline we get a chat section, where we can see our old messages. This is one-on-one chat or we can call it direct message, where each of the messages are sorted according to the time it was sent. Even the time of the message sent is visible for every chat created. Here if these two users are handled by two

different servers, then when one user sends message, the message will be stored in the data base, and other server will poll the database in periodic time interval, and if it finds some new message then it will send it to the user, this way even though two users are handled by different servers we can achieve eventual consistency in sending chat messages. At this page if the user wants to go back to timeline view, then they need to click on the “Back to timeline” button. Here websockets are used between chat server and web app, this way any updates from chat server will be easily updated in the web page, and users don’t need to reload the page. This way we are achieving a realtime chatting system.

iv. Timeline generation:

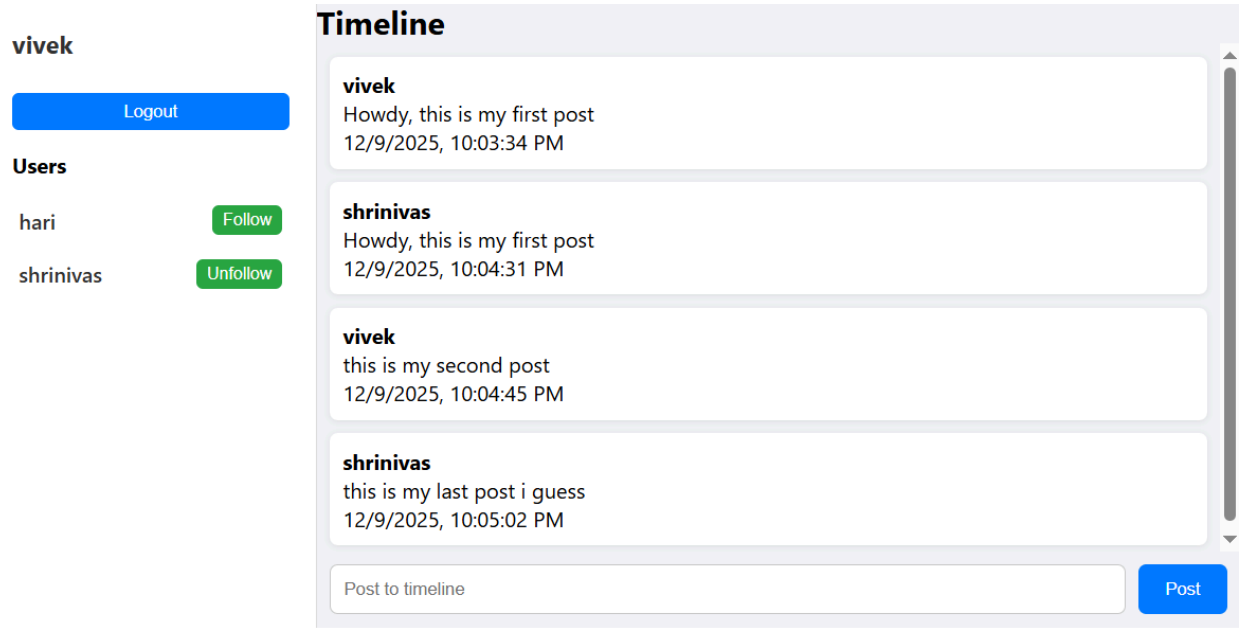

Figure 6: timeline generation:

In Figure 6 we can see what the timeline view looks like. Each user can post anything on their timeline and it will be visible to them on their timeline along with the time it was posted. If a user “vivek” follows another user “shrinivas”, then this user “vivek” will get all the future posts from “shrinivas”. And when “vivek” unfollows “shrinivas” then onwards no moe posts of “shrinivas” will be visible to “vivek”. This way we are generating a timeline sorted by time it was posted, and it includes posts from those who the current user is following. If a few users who I follow are handled by different servers, then their posts will be sent to the database by their server, and my server will eventually read them from the database and it uses web sockets to connect to users web applications and in realtime it will post to me. Here bi-directional streams are achieved using live websocket sessions, this way any updates found in the database can be dynamically sent to a connected web app in real time. This way we are achieving eventual consistency.

v. Follow/Unfollow users:

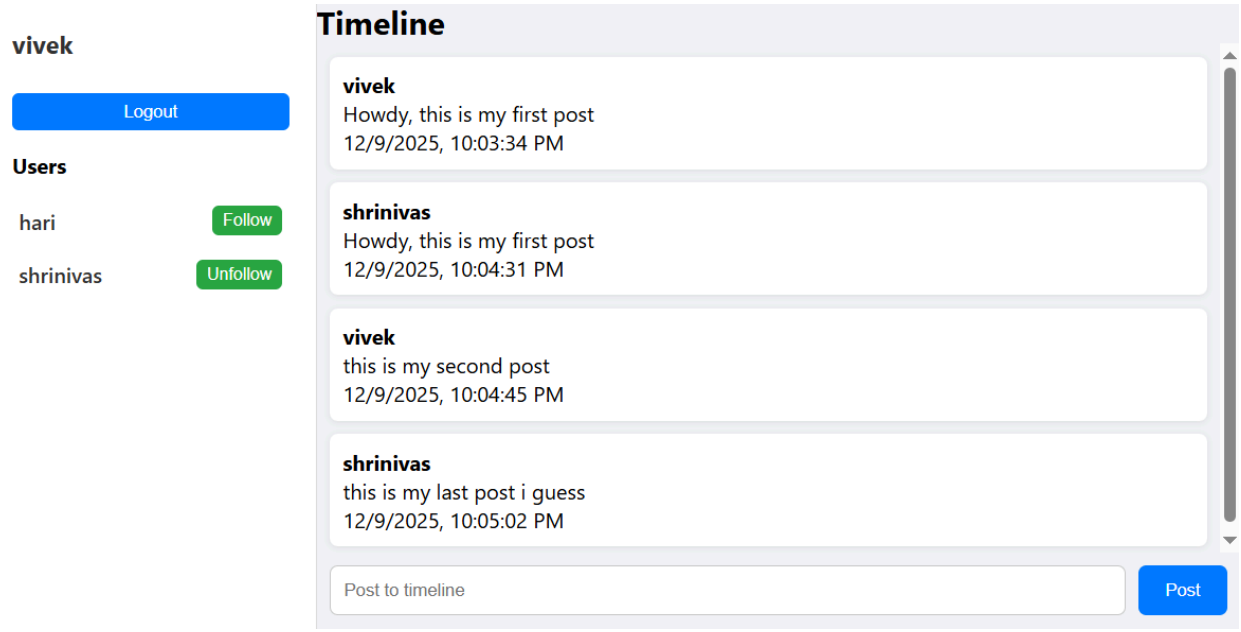

Figure 7: Follow and unfollow page

In Figure 7 we can see on the left side that “vivek” has started following “shrinivas” and has unfollowed hari. Here the “follow” button will make a Rest API call to the chat server telling the current user wants to follow another user. This is then sent to the database, and other servers will read this new change from the database and update their status. This way all the user web applications will be having updated follower/followee information.

vi. Master-Slave Failover and High Availability:

We used localhost:9090 as the master server and localhost:9091 as its slave. As shown in Figure 8.1, all requests initially go to localhost:9090. When this master server crashes, the web application fails to connect and then asks the auth server for an updated master. The auth server promotes the slave (localhost:9091) to the new master and returns its IP address to the client. Figure 8.2 shows the request to the auth server, and Figure 8.3 shows localhost:9091 successfully serving as the new master for the same user. Here the user interface won’t be reloaded so the user won’t be knowing anything about the backend server crash, this way we are achieving high availability. And data is persisted by the database service independent of the chat server, so we are able to achieve fault tolerance during chat server crashes.

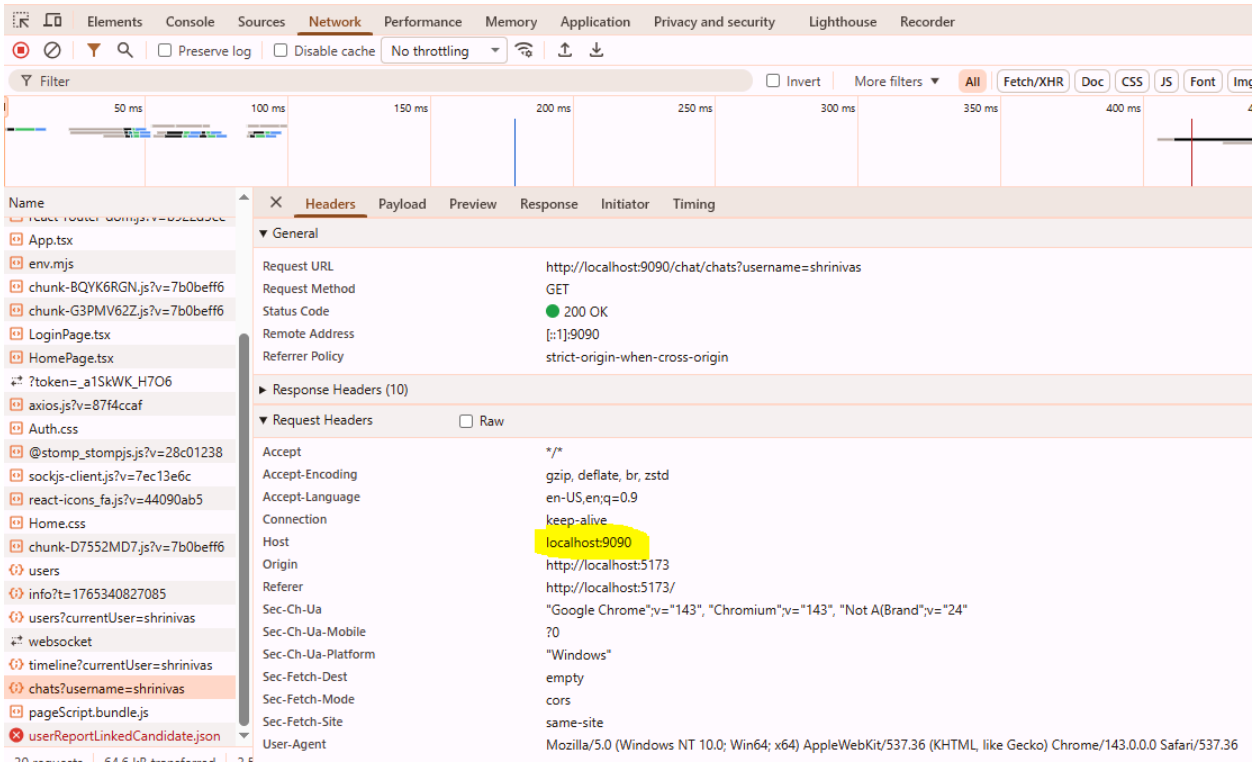

Figure 8.1: Before master server localhost:9090 crash

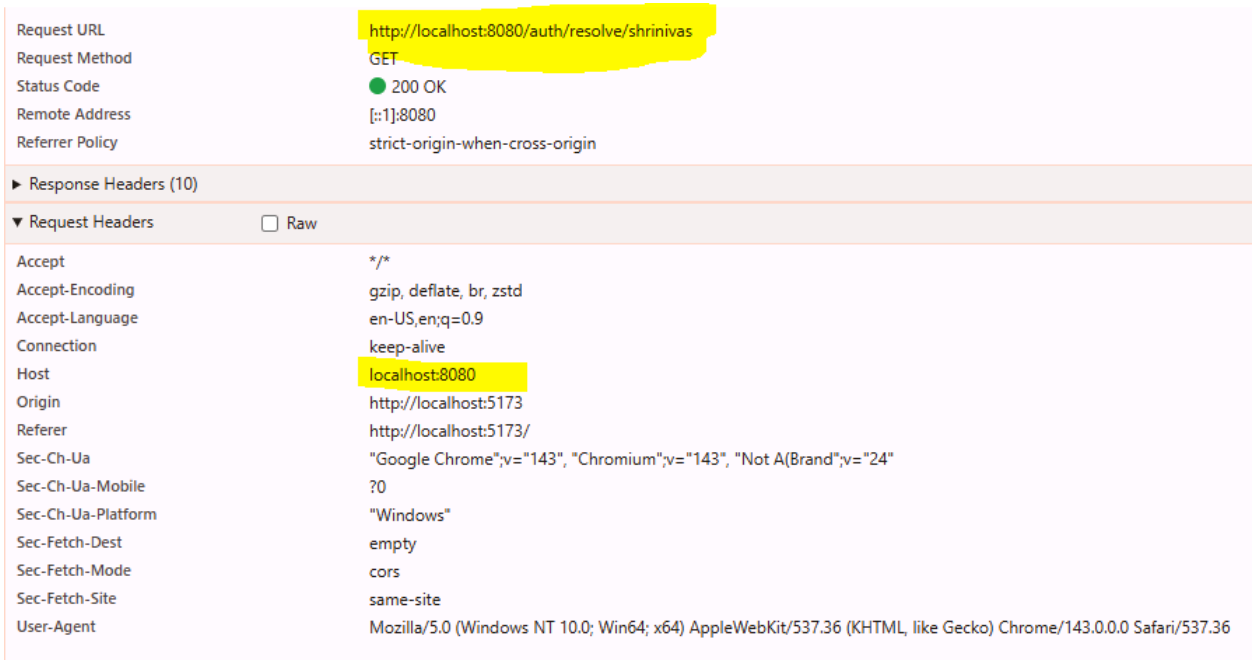

Figure 8.2: Resolving new master server ip address on crash

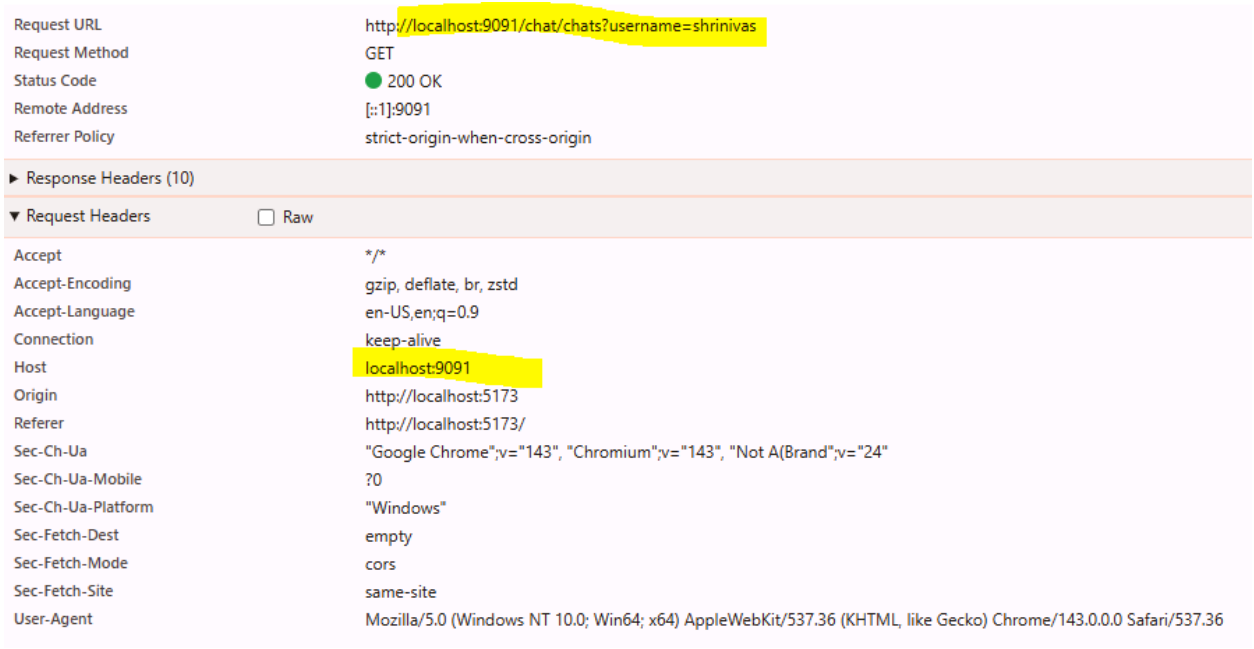

Figure 8.2: After master server localhost:9090 crash

5. Results:

To validate the behavior of the system under real distributed conditions, we manually conducted several experiments focusing on fault tolerance, state synchronization, and real-time communication. One key experiment involved crashing the master server running on localhost:9090 while users were actively connected. The moment the master went down, the client's WebSocket connection dropped, which triggered a request to the auth server for a new master. Since the auth server continuously monitors the health of all chat servers, it had already promoted the slave (localhost:9091) to the new master. The client reconnected to this new master automatically, without forcing the user to re-login. This confirmed that the failover mechanism worked smoothly and the system remained operational even during server failures.

We also evaluated how well the system maintained consistency across distributed chat servers. The master frequently updated the database with its current state, while the slave regularly pulled these updates to maintain a synced snapshot. Because of this, once the failover occurred, the newly promoted master already had the latest user relationships, message data, and timeline information. Message exchanges during high concurrency also converged correctly using the shared database as the source of truth. Overall, the tests demonstrated that the system reliably maintained eventual consistency, handled failover gracefully, and continued to deliver real-time communication without interruptions.

6. Conclusion and Future work:

This project successfully implemented a distributed social networking application with automatic failover, real-time communication, and consistent data synchronization across servers. The experiments showed that the system can recover from server crashes without significantly affecting the user experience. By combining a master–slave architecture with health checks, database-backed synchronization, and WebSocket-based messaging, the system was able to maintain both availability and consistency under realistic workloads. These results highlight that the architecture is robust, scalable, and well-suited for distributed environments.

Although the system performed well in our experiments, there are several areas where it can be further enhanced. Introducing load balancing across multiple master–slave pairs would help distribute traffic more evenly, improving scalability under heavy use. Implementing more advanced consistency models or conflict-resolution strategies could allow the system to support even more complex distributed workflows. Adding stronger monitoring tools, improved security layers, and deployment using container orchestration (like Kubernetes) would make the system production-ready. Finally, expanding the system with additional social networking features—such as media sharing or push notifications—would further strengthen its real-world applicability.

7. Github repo link:

https://github.com/Shrinivas-Shanbhag/Distributed-Social-Networking-Application