VIETNAM INTERNATIONAL UNIVERSITY – HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

WEB APPLICATION DEVELOPMENT PROJECT

ONLINE FURNITURE STORE

By

Huỳnh Thị Ngọc Trâm - ITCSIU21238

Hoàng Gia Huy - ITCSIU21186

Phạm Anh Huy - ITCSIU21133

Advisor: Dr. Nguyen Van Sinh

A report submitted to the School of Computer Science and Engineering in partial fulfillment of the requirements for the Final Project in Web Application Development course

Ho Chi Minh city, Vietnam, 2024

I. INTRODUCTION

A. Background

Social media has become an integral part of modern life, enabling people around the world to connect, share content, and engage with one another. Facebook, one of the largest and most widely used social media platforms, has pioneered many of the key features that have defined the social media experience for billions of users. This project aims to create a new social media platform that replicates the core functionality of Facebook, providing users with a familiar and intuitive experience for connecting, sharing, and interacting.

B. Scope and objective

The scope of this project is to create a social media system focused at optimizing user experience and the system function to handle large amounts of data. Objectives include:

- Ability for users to create personalized accounts and profiles, including the ability to upload profile photos, provide biographical information, and change the bio information.
- Functionality for users to create and share various types of content, including text-based status updates, photo and video uploads, links, and more.
- Private messaging capabilities that allow users to send direct messages, hold conversations, and take photos in its own device camera with individual connections or groups.
- The ability to "follow" other users' profiles, build a network of connections, and receive updates on their activity.
- Security measures to protect user data, prevent malicious activity, and ensure a safe and trustworthy environment.

The primary objectives of this social media project are:

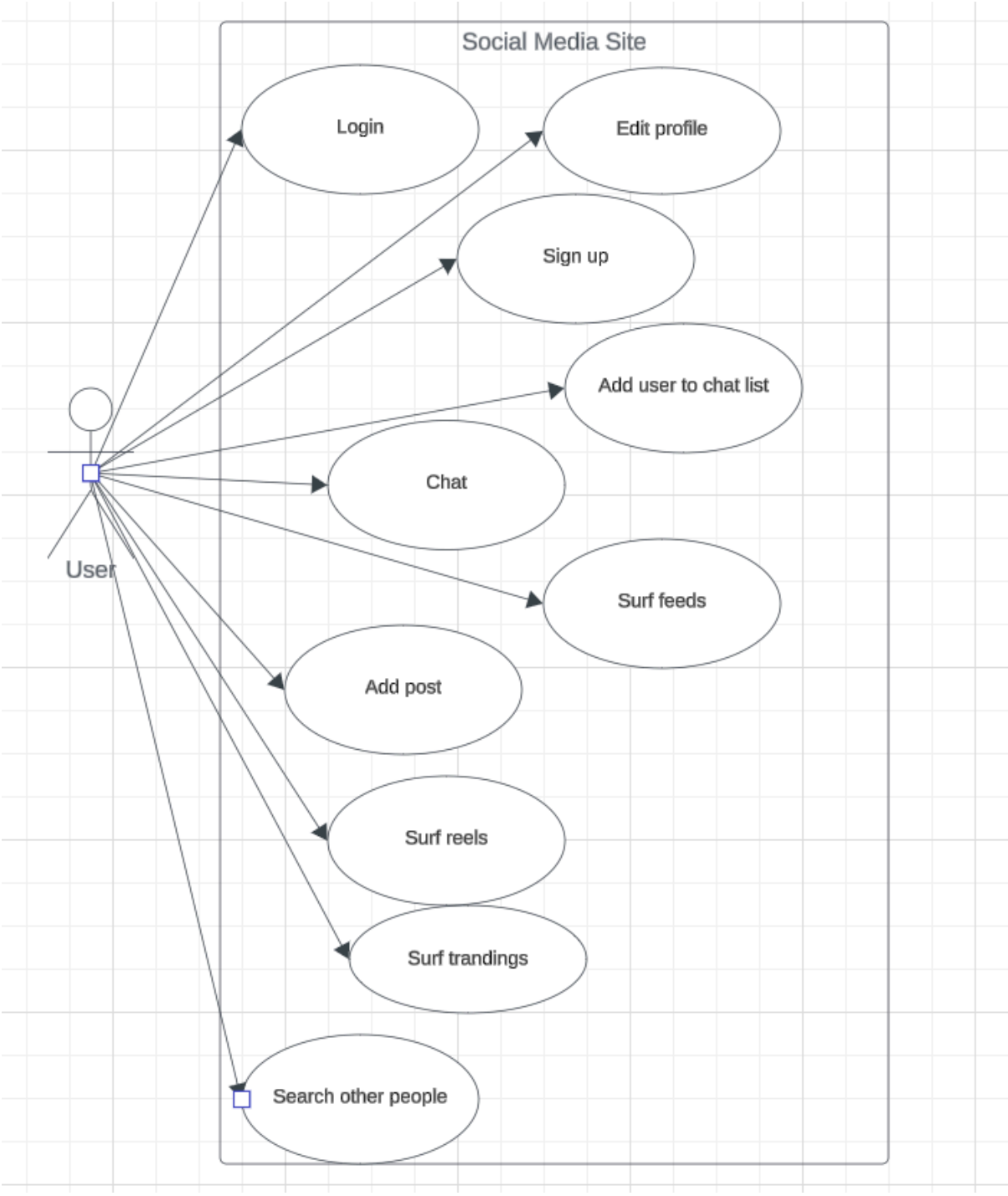
- To create a user-friendly and intuitive social media platform.
- To enable seamless and engaging content sharing, connection, and interaction between users, fostering a sense of community and belonging.
- To develop a secure and privacy-conscious platform that empowers users to control the visibility of their personal information and online activity.

II. REQUIREMENT ANALYSIS AND DESIGN

A. Requirement Analysis

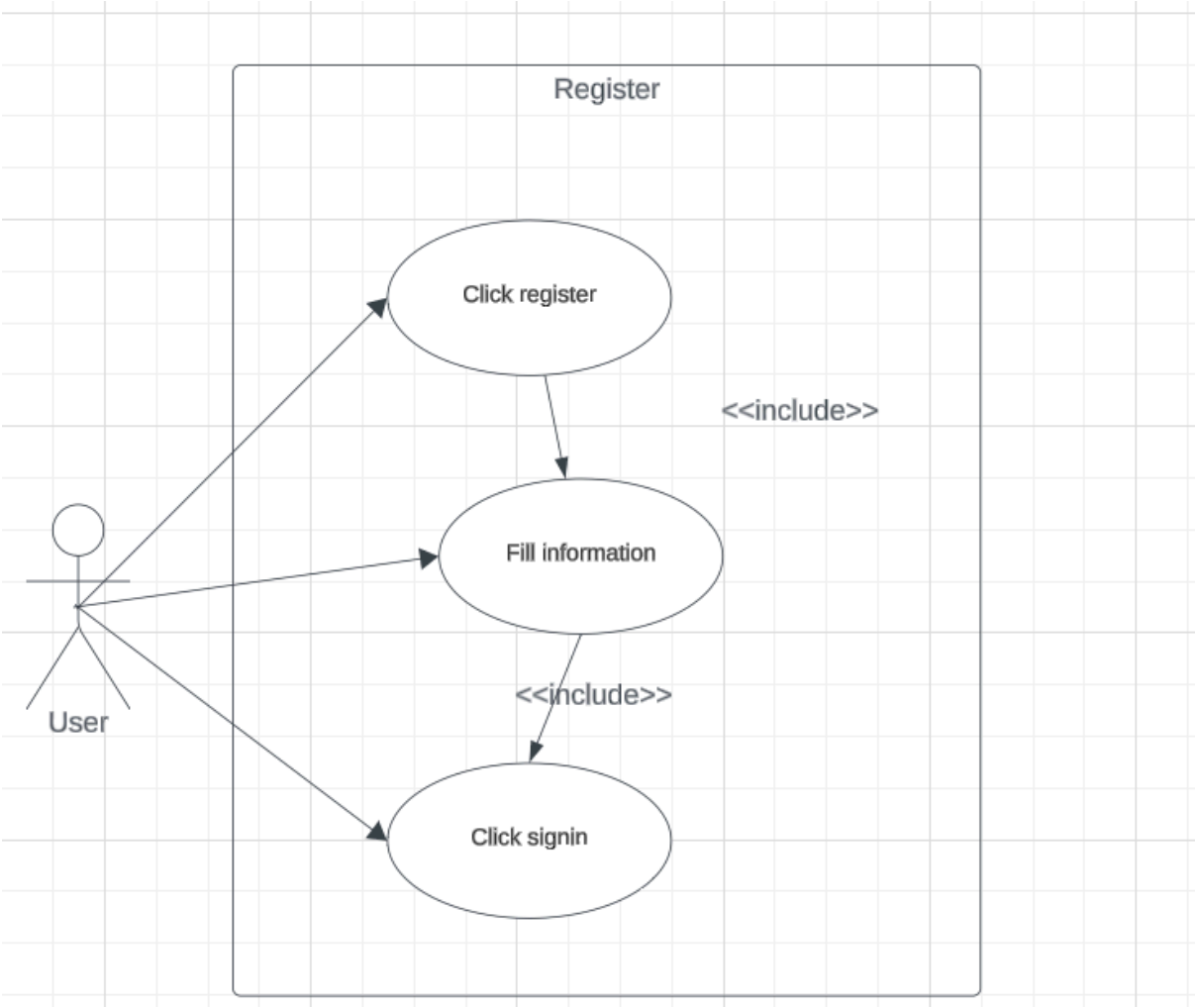
1. Use-case

A. Summary Goal



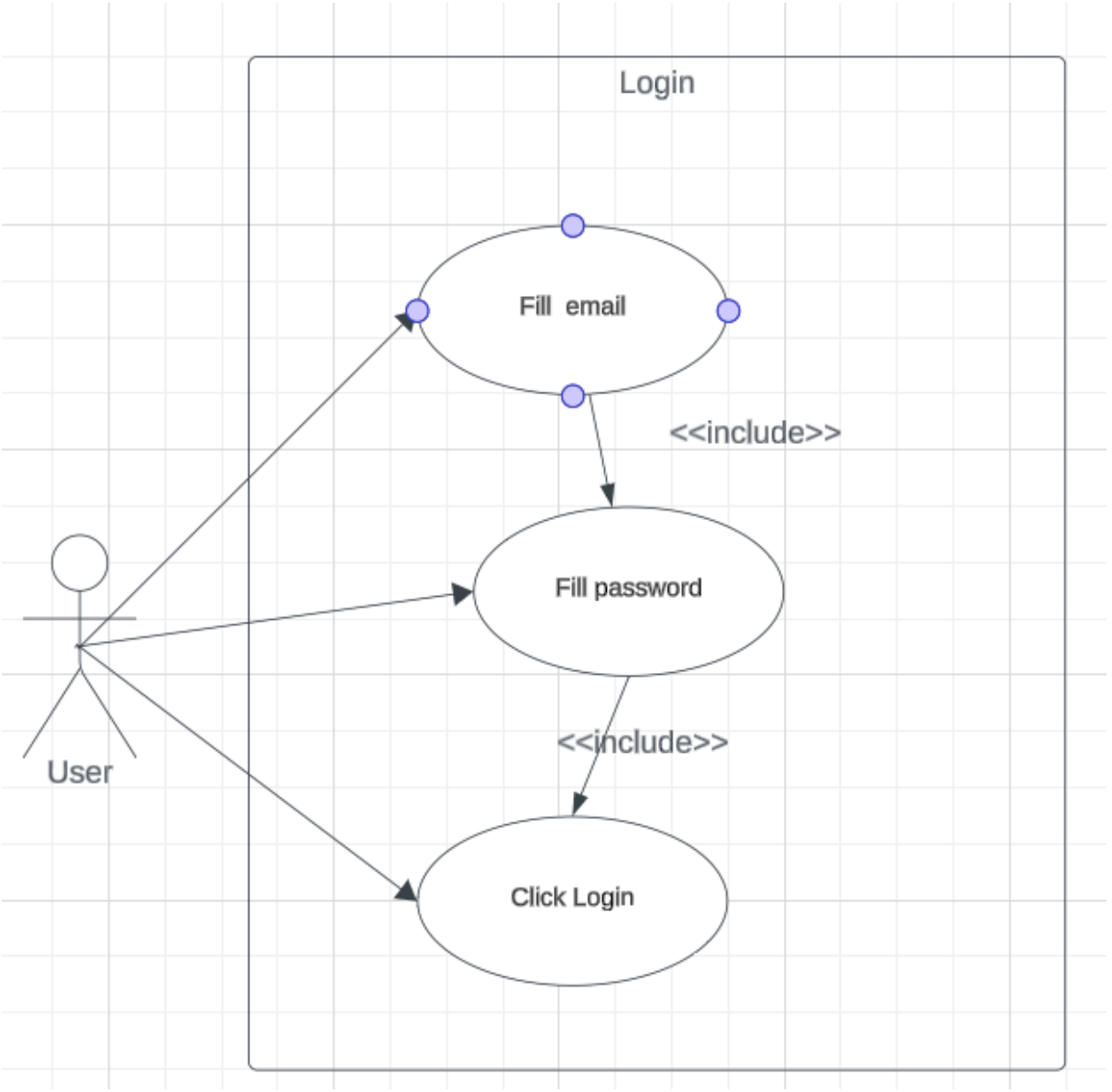
B. User Goal

- **Sign up**



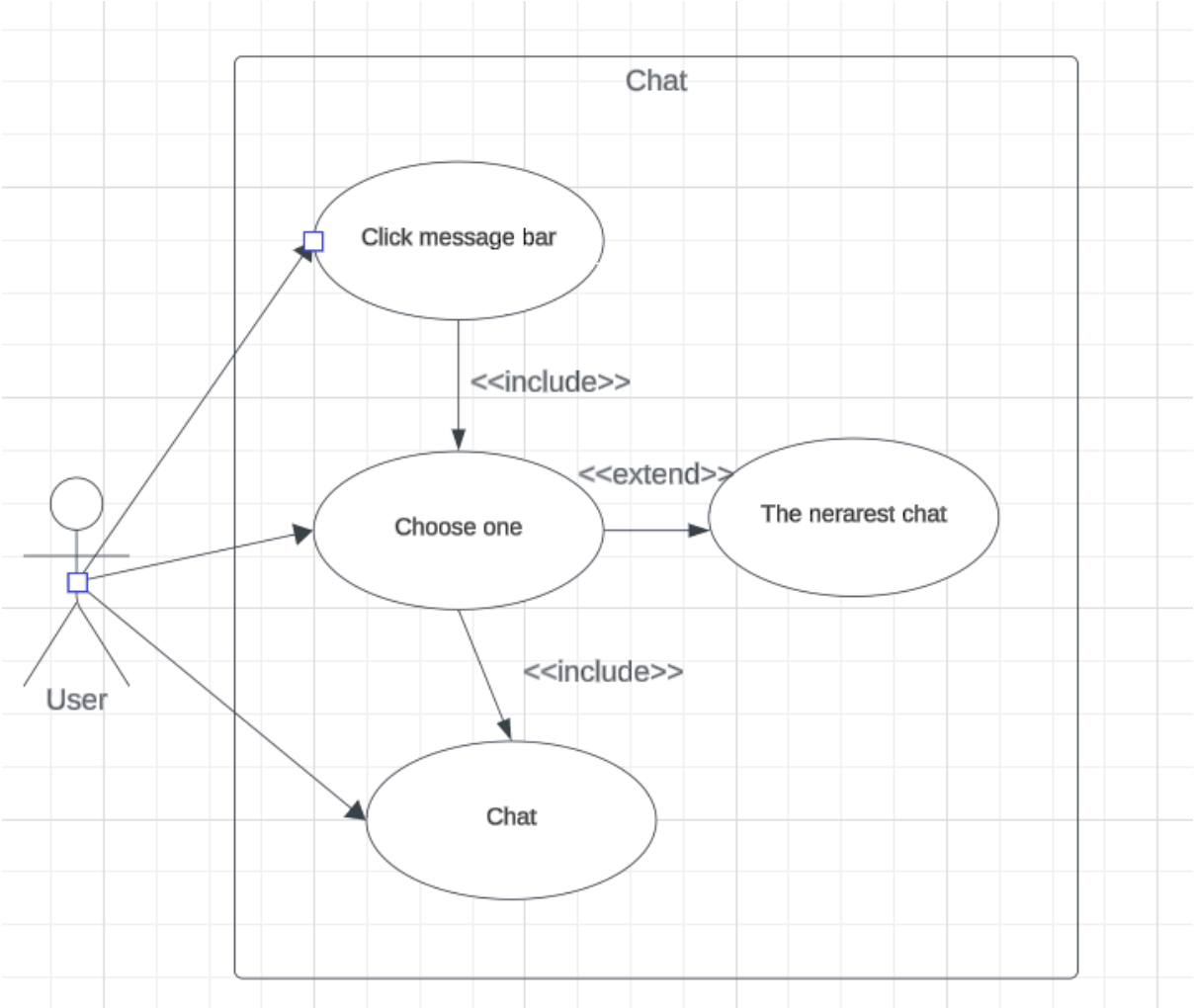
- ➔ Actor: User
- ➔ Pre-condition: None
- ➔ Post-condition: None
- ➔ Flow: Click register -> Fill form -> Click sign in
- ➔ Alternative flow: None

- **Login**



Actor: User
Pre-condition: Already have account
Post-condition: Login success
Flow: Enter email-> Enter password-> Click login
Alternative flow: None

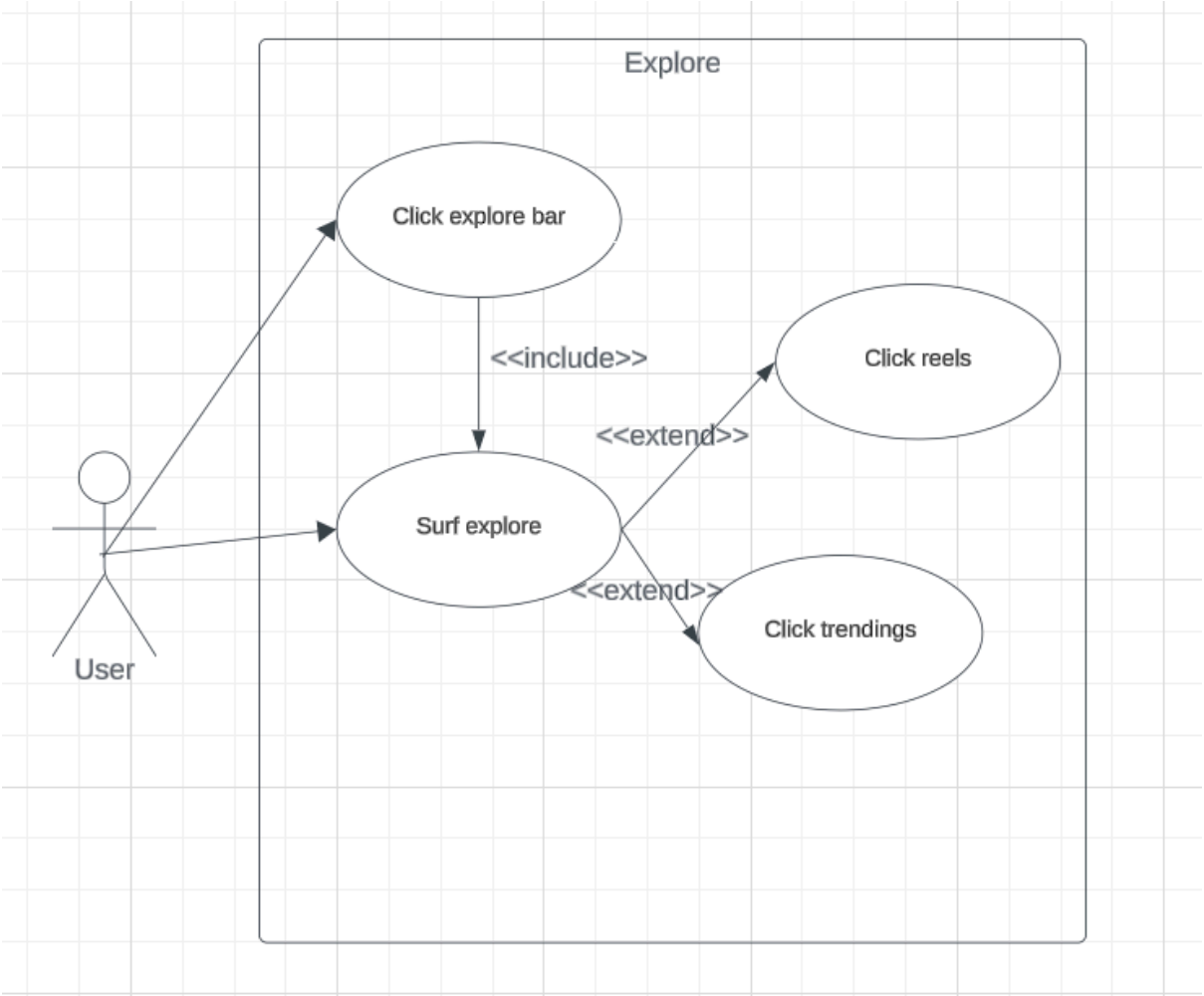
- **Chat**



Actor: User
Pre-condition: Already have account
- Login to account

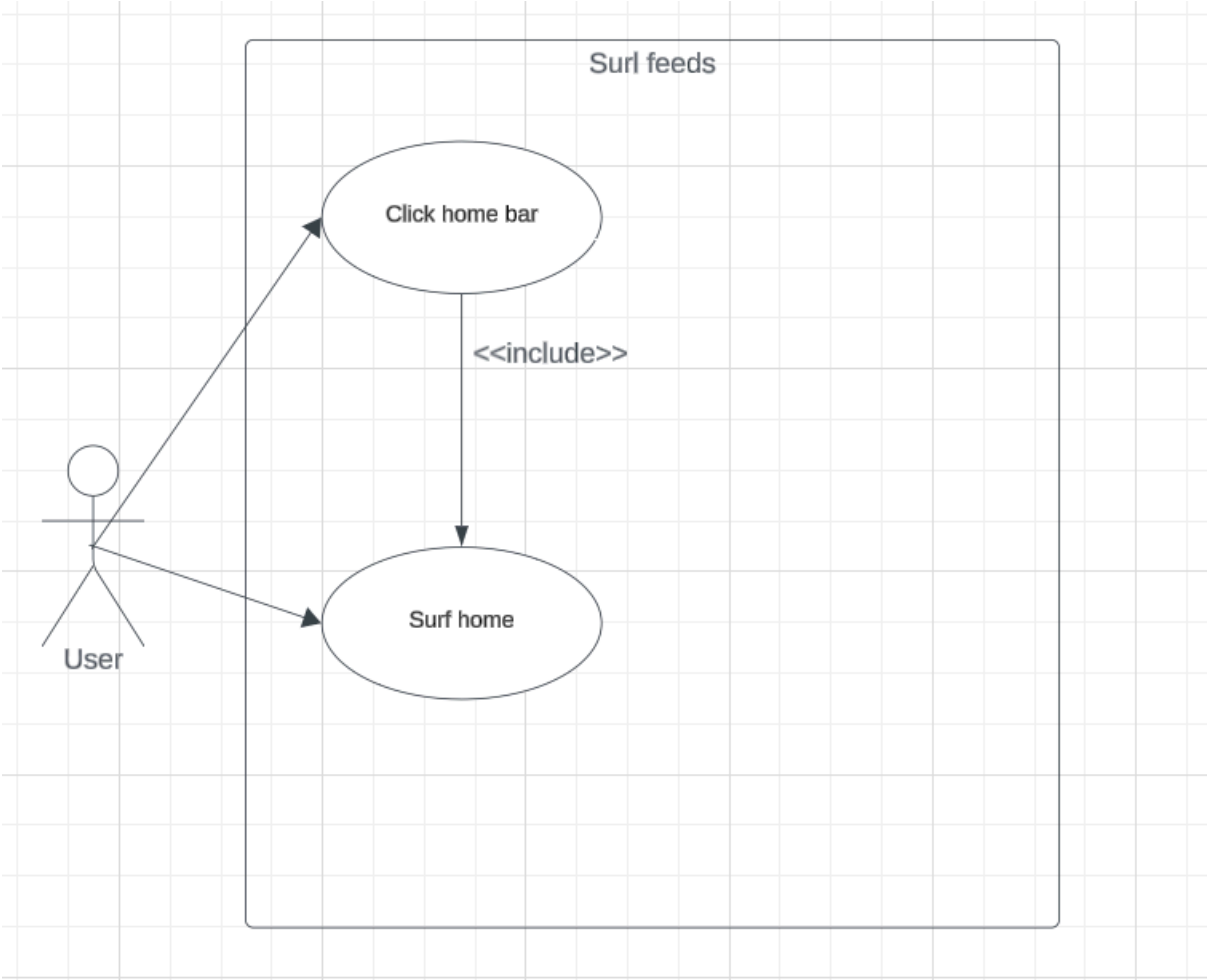
Post-condition: Send message success
Flow: Click message bar -> choose one to chat -> enter message
Alternative flow: None

- **Surf explore**



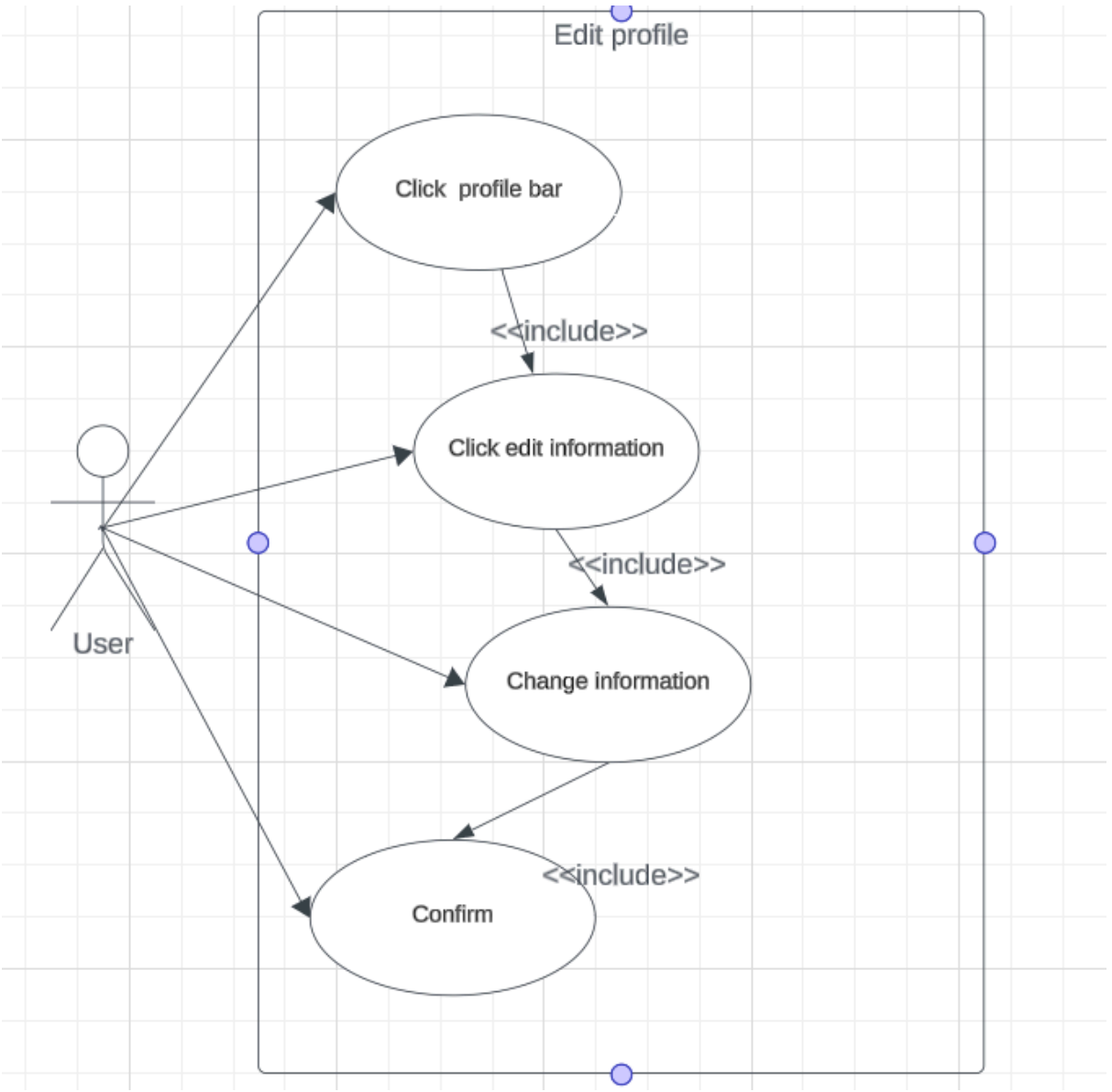
Actor: User
Pre-condition: Already have account
-logined to account
Post-condition: None
Flow: Click explore bar -> choose trending -> surf
Alternative flow:choose reels - > sulf

- **Surf feeds**



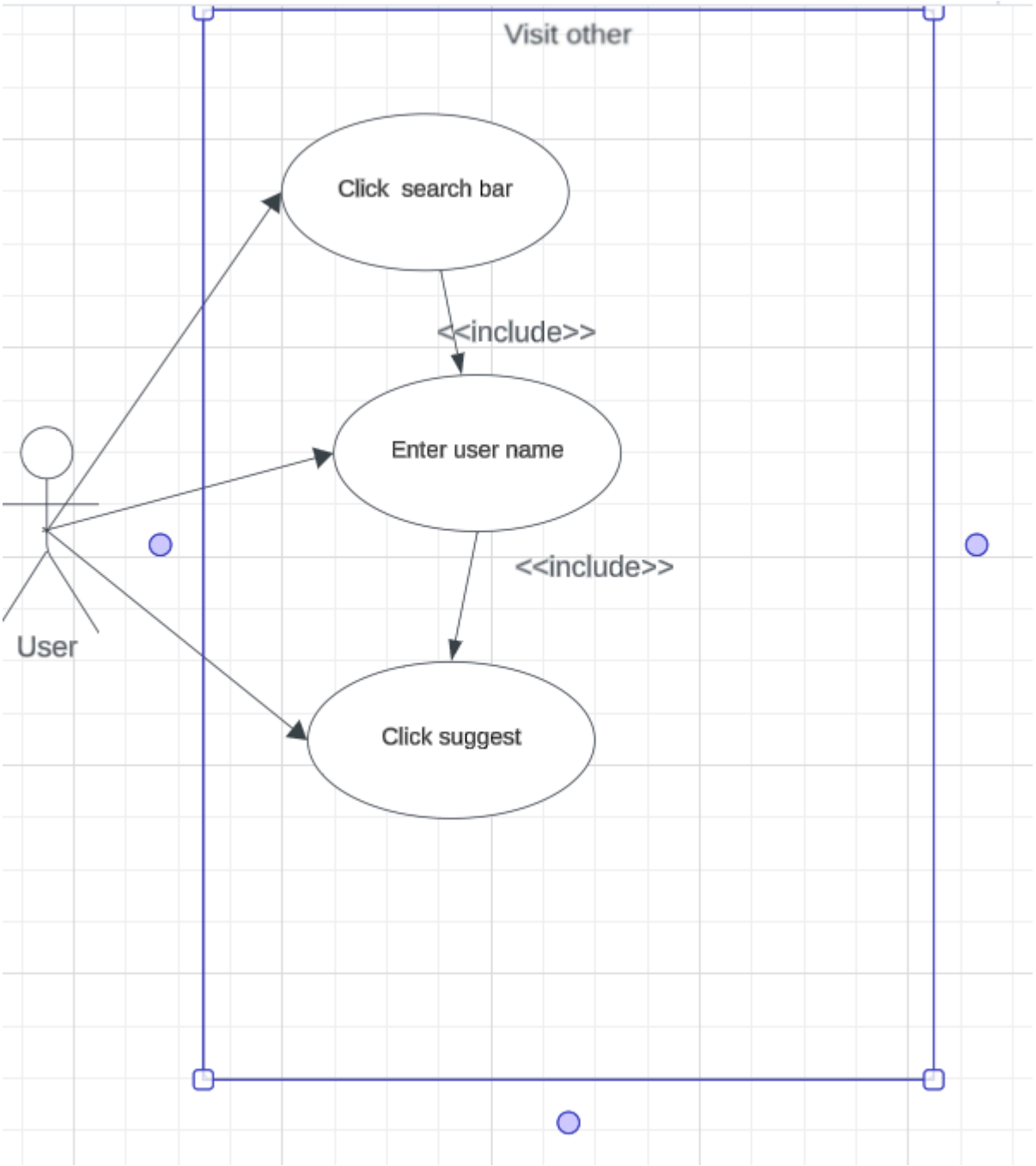
Actor: User
Pre-condition: Already have account
-login already
Post-condition: None
Flow: Click register -> Fill form -> Click sign in
Alternative flow: None

- **Edit profile**



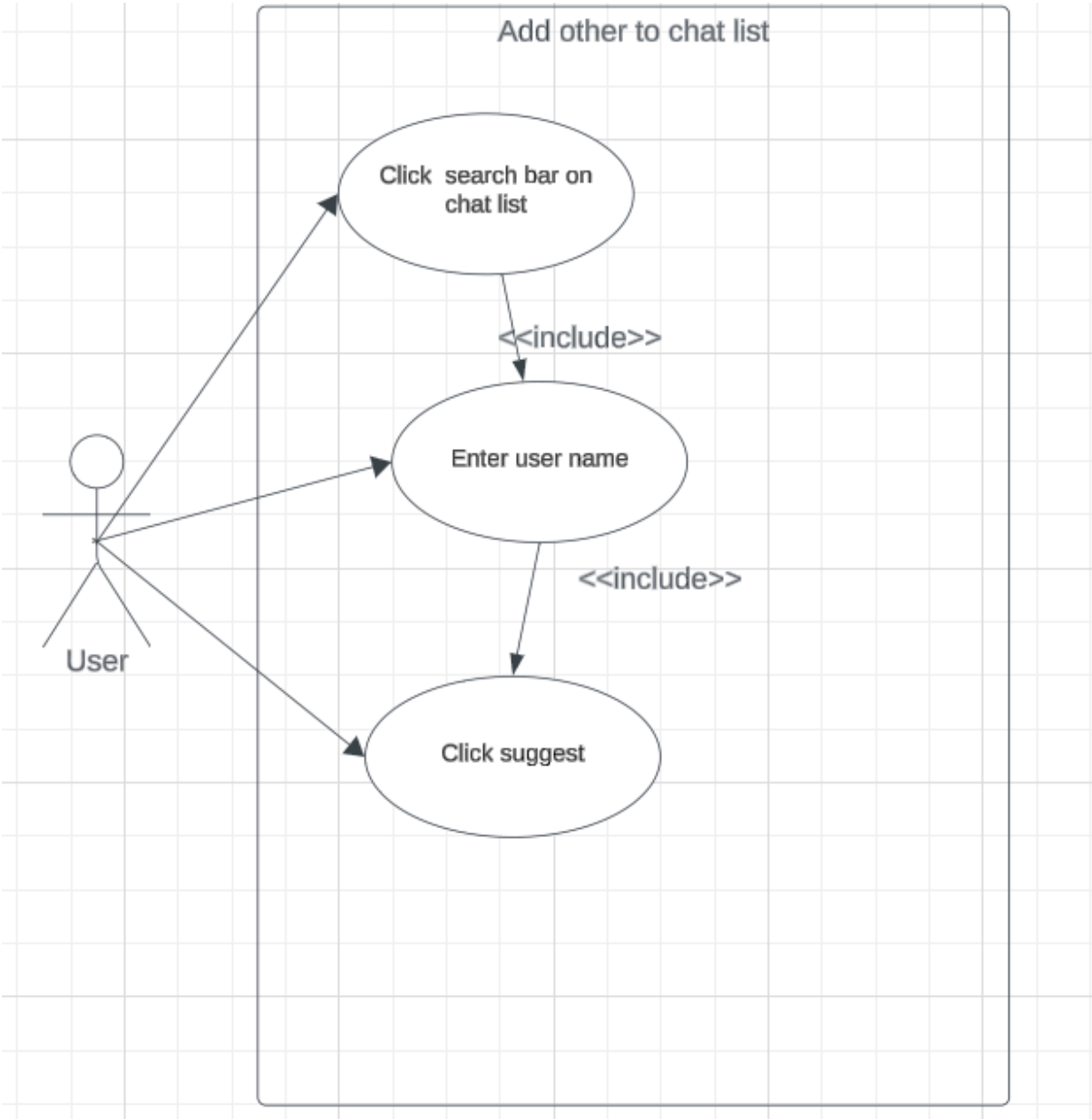
Actor: User
Pre-condition: Already have account
-Login already
Post-condition: None
Flow: Click profile bar -> click edit information -> change information -> Confirm
Alternative flow: None

- **Visit other user**



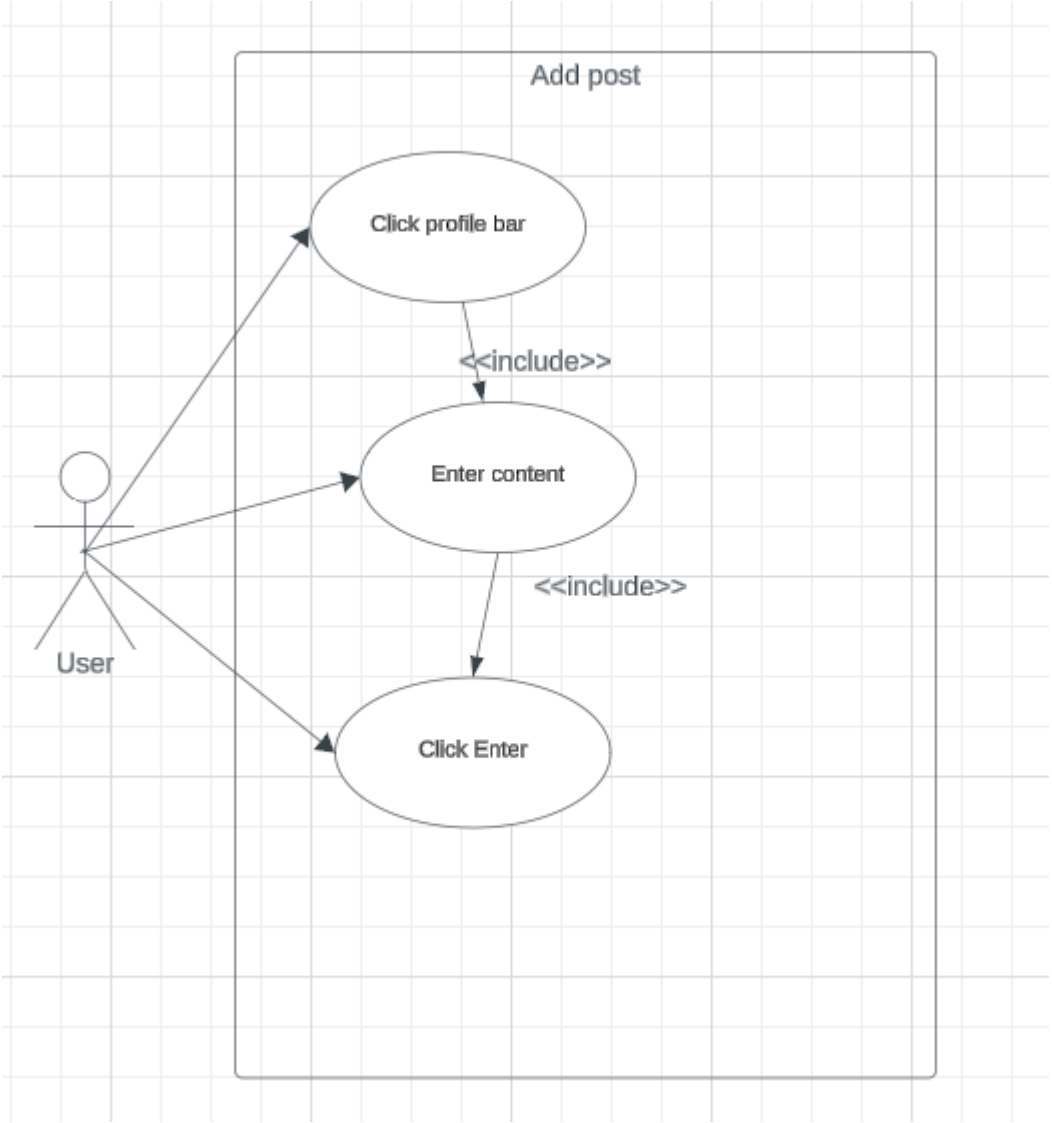
Actor: User
Pre-condition: Already have account
-login account
Post-condition: None
Flow: Click search bar -> enter account name -> click suggestion
Alternative flow: Click enter to search

- **Add user to chat list**



- Actor: User
  Pre-condition: Already have account
  -Login already
  Post-condition: None
  Flow: Click search bar in chat list -> enter account name -> click suggestion

  Alternative flow: Click enter to add
- **Add posts**



Actor: User
Pre-condition: Already have account
-Login already
Post-condition: None
Flow: Click profile bar -> add content -> Click post
Alternative flow: None

2. **Non-functional Requirement**

- **Scalability**:

- The platform must be able to scale seamlessly to accommodate a growing user base, with the ability to handle millions of concurrent users and billions of interactions.

- **Reliability:**

- The platform must be available and accessible to users 24/7, with a target uptime of at least 99.9% per year.

- The system must be designed to handle sudden spikes in user traffic and activity without significant performance degradation.
- The system can store a great amount of users' information and chat messages without error.

- **Usability:**

- The user interface must be intuitive, visually appealing, and consistent across all platforms and devices (web, mobile, etc.).
- The onboarding and account management processes must be streamlined and user-friendly, minimizing the effort required for new users to get started.

- **Maintainability:**

- The platform must undergo regular maintenance, updates, and security patches to ensure optimal performance, stability, and protection against emerging threats.

B. Design

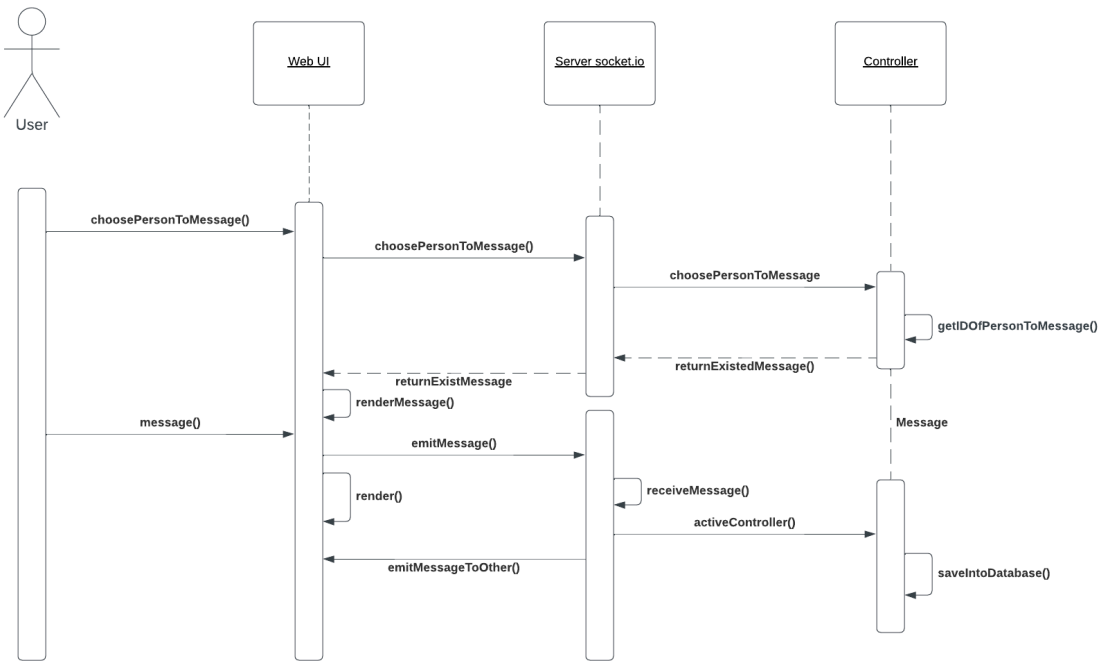
1. Database design

In traditional relational database projects, an Entity-Relationship Diagram (ERD) is commonly used to visually represent the data model, including the relationships between different entities. However, our project, SugarCube, utilizes a NoSQL database (MongoDB) which follows a different paradigm for data modeling compared to relational databases.

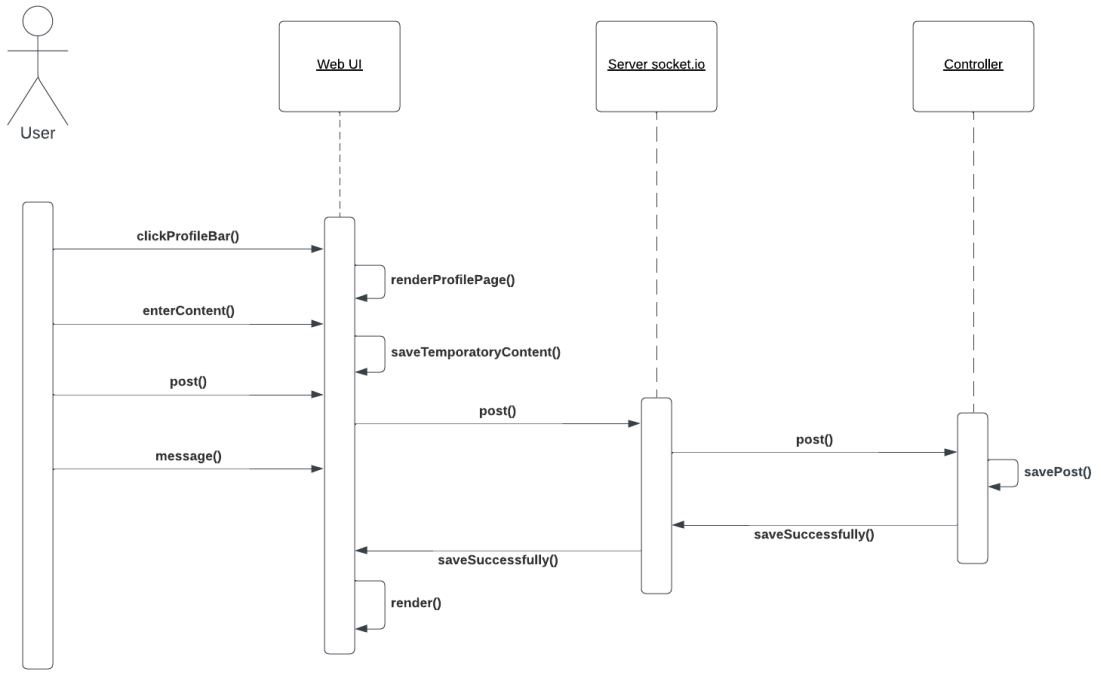
2. Class Diagram

3. 3 features

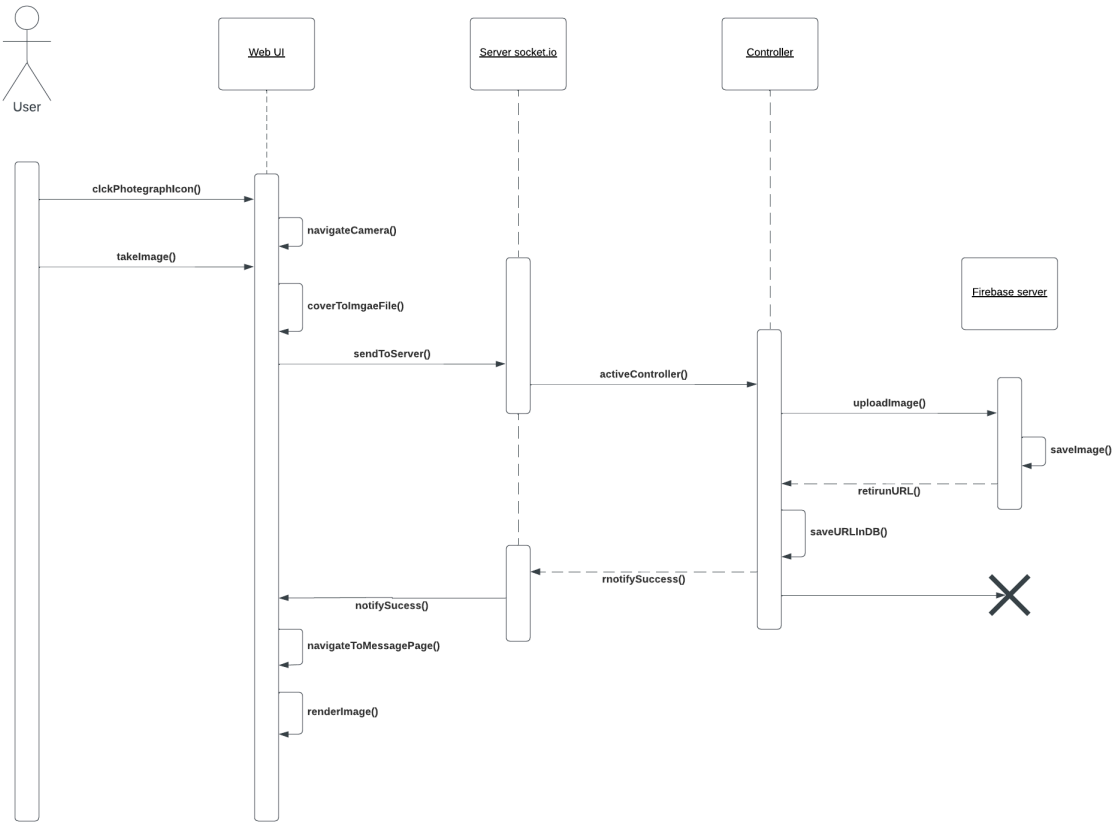
❖ Message



❖ Post



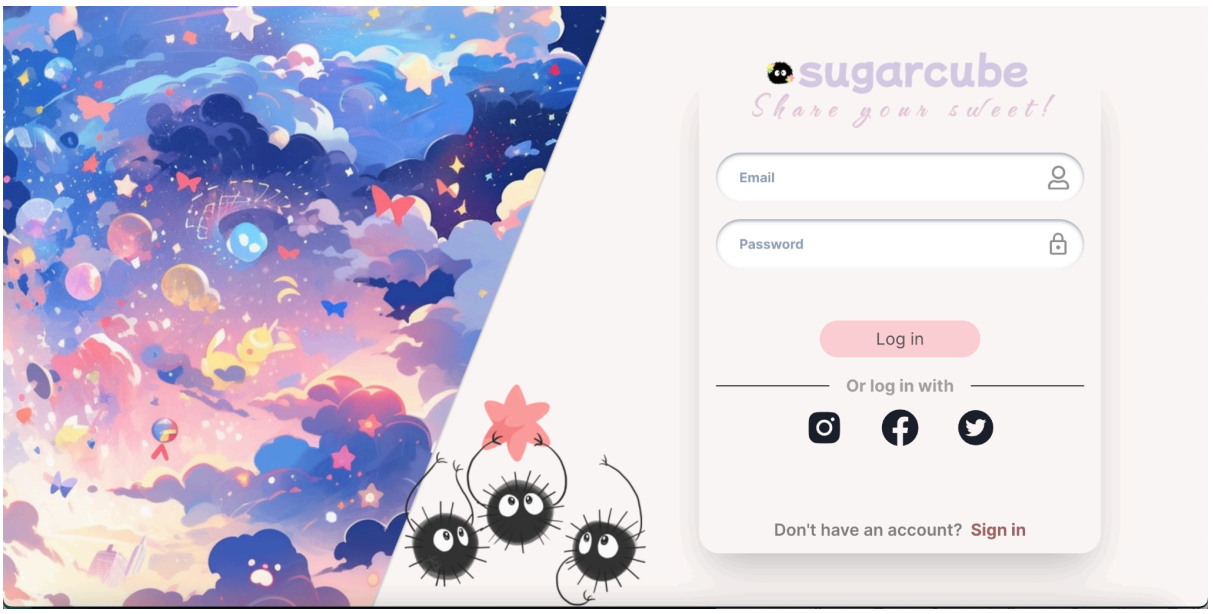
❖ Take Picture



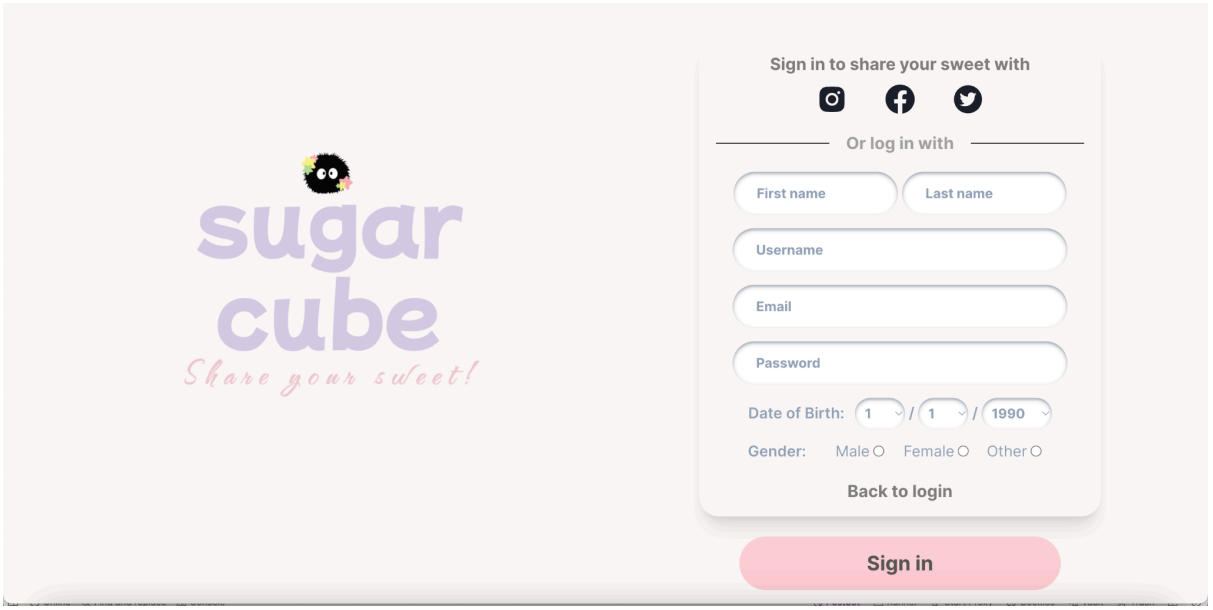
III. IMPLEMENTATION

A. Authentication

1. Demo:



The Login Page is the first interface that users will encounter if they do not log in. Users will input their Email and Password and have the option to go to the sign in page to create an account.



The Sign Page is in charge of creating user accounts, the user will input their corresponding information and click on the Sign In button. If the user already has an account and goes to the sign in page by accident. There is a “Back to login” button that navigates the user back to the Login page.

- Implementation:
- Server-side:
    - The registerPost api will help user create account, the user will input their information in and it will be added to the database

```
exports.registerPost = async (req, res) => {
    try {
      const { username, email, password, firstName, lastName, gender, dob } = req.body;

      if (!username || !email || !password || !firstName || !lastName || !gender || !dob) {
        return res.status(400).json({ message: "All fields are required" });
      }

      const validGenders = ["male", "female", "other"];
      if (!validGenders.includes(gender)) {
        return res.status(400).json({ message: "Invalid gender" });
      }

      const salt = await bcrypt.genSalt(10);
      const hashedPassword = await bcrypt.hash(password, salt);

      const newUser = new User({
        username,
        email,
        password: hashedPassword,
        firstName,
        lastName,
        gender,
        dob,
      });

      const user = await newUser.save();
      res.status(200).json(user);
    } catch (err) {
      res.status(500).json({ message: err.message });
    }
  };
```

- The loginPost api will help user login and check for the existence of an account,if that account exists then the login process will execute, or else the login api will not execute and the client side will return an error. If login is successful, it will return to the user the information of that account and a token that will be used for all other api. Without that token, the other api will return an error and the user needs that token to use the page.

```
exports.loginPost = async (req, res) => {
  try {
    console.log(req.body.user);
      const user = await User.findOne({ email: req.body.email });
      if (!user) {
          return res.status(404).json("Wrong Email!");
      }

      const validPassword = bcrypt.compare(req.body.password, user.password);
      console.log(req.body.password);
      console.log(user.password)
      if (!validPassword) {
          return res.status(404).json("Wrong password");
      }

      if (user && validPassword) {
          const Token = jwt.sign(
              {
                  id: user._id,
              },
              process.env.JWT_ACCESS_KEY,
              { expiresIn: "1y" }
          );

          const { profilePicture, coverPicture, followers, followings, ...others } = user._doc;
          return res.status(200).json({ ...others, Token });
      }
  } catch (err) {
      return res.status(500).json(err);
  }
```

- Client-side:
  - First, in the main app we will create a protected route that will return the user to the login page if they are trying to change the path and navigate to another page without login.

```
function App() {
    return (
        <Router>
            <Routes>
                {publicRoutes.map((route, index) =>{
                    const Page = route.component;

                    let Layout;
                    if(route.layout === null)
                        Layout = Fragment;
                    else
                        Layout = route.layout;
                    return <Route key={index} path={route.path} element={<Layout><Page /></Layout>}/>
                })}
                {privateRoutes.map((route, index) =>{
                    const Page = route.component;

                    let Layout;
                    if(route.layout === null)
                        Layout = Fragment;
                    else
                        Layout = route.layout;
                    return (
                        <Route key={index} element={<ProtectedRoutes/>}>
                            <Route key={index} path={route.path} element={<Layout><Page/></Layout>}/>
                        </Route>
                    )
                })}
```

  - The sign up page will handle user information and fetch the api to create an account for the user. The page will also check for empty input

that if the user does not enter enough information, the api will not process.

```
function SigninForm(props) {

    const handleSubmit = async (err) => {
        err.preventDefault();
        if (!formValues.email || !formValues.username || !formValues.password || !formValues.firstname || !formValues.lastname || !formValues.gender) {
            console.log("error");
        } else {
            try {
                const response = await fetch('https://sugar-cube.onrender.com/register', {
                    method: 'post',
                    headers: {
                        Accept: 'application/json',
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        firstName: formValues.firstname,
                        lastName: formValues.lastname,
                        username: formValues.username,
                        email: formValues.email,
                        password: formValues.password,
                        dob: formValues.year + "-" + formValues.month + "-" + formValues.day,
                        gender: formValues.gender,
                        profilePicture: formValues.profilePicture,  // Include profile picture
                    })
                });

                console.log(response);

                if (response.ok) {
                    console.log("sign up successfully");
                    setIsSubmit(true);
                } else {
                    throw new Error("Could not fetch resource");
                }
            } catch (error) {
                console.error(error);
            }
        }
    };
```

- The login page will handle the login information, if the user already have an account, the login process will be executed with flying colors, or else it will fail and return an on screen message to the user. If the login process is successfully executed, it will store all of the necessary information of the user like userId, username and the token. It will also turn off the protected route and user can go to any page they want, the protected route can only be turned on when the user click on the “Logout” button in the right nav bar.

```
function LoginForm() {
    const handleSubmit = async (err) => {
        err.preventDefault();
        if(!formValues.email || !formValues.password){
            setErrormessage("Please fill out all information");
        }
        else{
            try{

                const response = await fetch('https://sugar-cube.onrender.com/login', {
                    method: 'post',
                    headers: {
                        Accept: 'application/json',
                        'Content-Type': 'application/json',
                    },
                    body: JSON.stringify({
                        email: formValues.email,
                        password: formValues.password
                    })
                });

                if(response.ok){
                    const data = await response.json();
                    Cookies.set('token', (data.Token));
                    Cookies.set('username', (data.username));
                    Cookies.set('userId', (data._id));
                    Cookies.set('user', 'true');
                    localStorage.setItem('token', data.Token);
                    localStorage.setItem('username', data.username);
                    localStorage.setItem('name', data.firstName + " " + data.lastName);
                    localStorage.setItem('userId', data._id);
                    localStorage.setItem('user', 'true');
                    setIsSubmit(true);
                }
                else{
                    throw new Error("Could not fetch resource");
                }
            }
```

B. Page Home

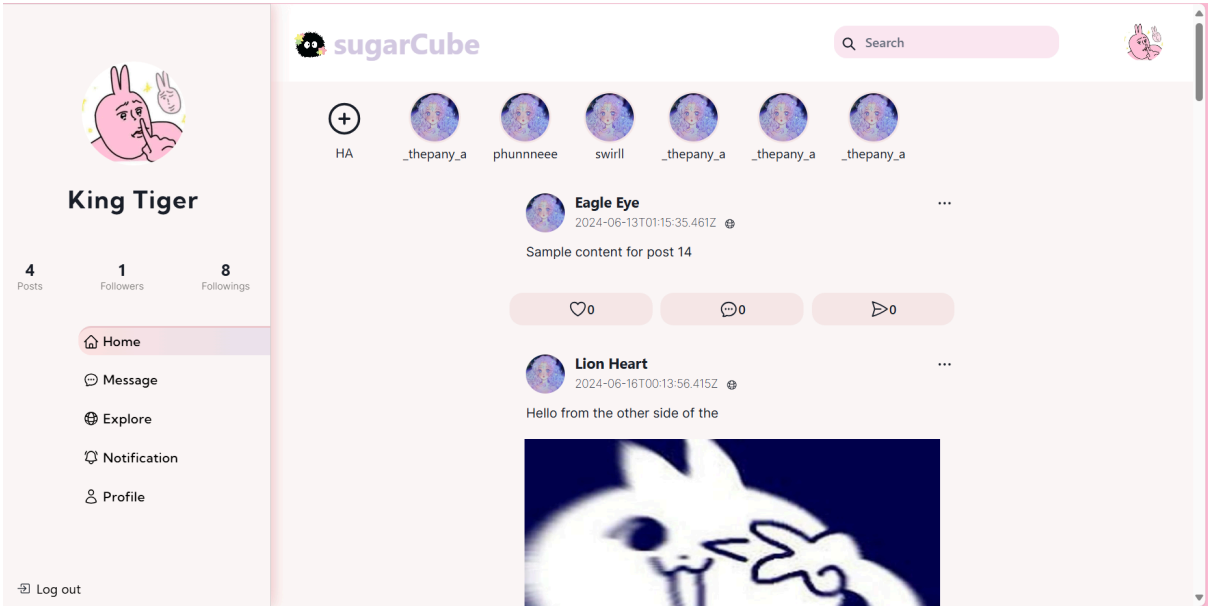
1. Demo:

   The "Home Page" is the initial interface that users encounter upon successfully logging into Suga Cube, designed to offer a distinct and engaging social networking experience.

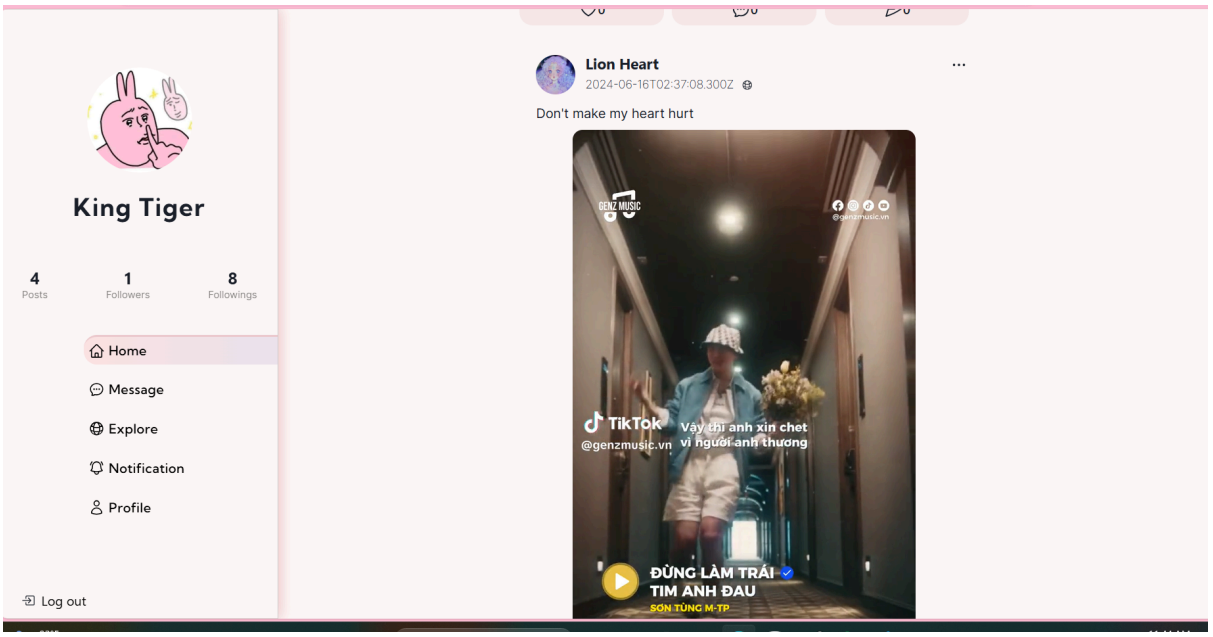
   At the very top of the Home Page is the Story bar, intended to display stories from the past 24 hours of the accounts users follow. Although this feature is currently under development, it is anticipated to provide vibrant and engaging content in the near future.

   Following the Story bar is the newfeed section, where posts are presented in a randomized order. This section includes both public posts and those set to following mode from the accounts users are following. The randomized presentation ensures a fresh and diverse array of content with each visit, fostering easier connection and interaction within the user's community.

   In summary, the "Home Page" of Suga Cube is designed to enhance the social networking experience by offering dynamic content and facilitating meaningful interactions.



As users scroll down, they can continue to see additional posts until all posts from the accounts they follow have been displayed.



2. Implementation:
   - *Sever-side*: The server-side logic is responsible for retrieving posts from accounts that the user is following. The core function is getFollowingPosts, which interacts with the database to fetch the relevant posts.

```
const getFollowingPosts = async (req, res) => {
    const userId = req.params.objectId;
    try {
      const user = await UserModel.findById(userId);
      if (!user) {
        return res.status(404).json({ message: "User not found" });
      }

      const followingIds = user.followings.map(following => following.following_id);

      const posts = await PostModel.find({
        Object_id: { $in: followingIds },
      }).populate("Object_id");

      const allPosts = posts.reduce((acc, cur) => {
        if (cur.Object_id && cur.posts && cur.posts.length > 0) {
          cur.posts.forEach(post => {
            if (post.privacyLevel === 'Public' || post.privacyLevel === 'Following') {
              const userId = cur.Object_id && cur.Object_id._id ? cur.Object_id._id.toString() : null;
              acc.push({
                userId: userId,
                post: post
              });
            }
          });
        }
        return acc;
      }, []);

      const shuffledPosts = shuffle(allPosts);

      res.json(shuffledPosts);
    } catch (error) {
      console.error("Error fetching posts:", error);
      res.status(500).json({ message: "Internal server error" });
    }
```

- User Validation: The function starts by retrieving the user's ID from the request parameters and checking if the user exists in the database.
- Following IDs Extraction: It then extracts the IDs of the accounts the user is following.
- Posts Retrieval: Posts from the followed accounts are retrieved using PostModel.find(), with the condition that the Object_id should be among the following IDs. The populate method is used to include related data.
- Filtering and Aggregation: The retrieved posts are filtered based on their privacy level (Public or Following) and aggregated into a list.
- Shuffling: The list of posts is shuffled to randomize the order.
- Response: Finally, the shuffled posts are sent back as a JSON response.

- Client Side: The client-side logic involves fetching the posts from the server and rendering them on the Home Page. This is handled by the PostZone component in React.

```
import React, { useState, useEffect } from 'react';  6.9k (gzipped: 2.7k)
import { Flex, Spacer } from '@chakra-ui/react';  38.6k (gzipped: 12.4k)
import Post from './Post';

Codiumate: Options | Test this function
function PostZone() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchPosts = async () => {
      const userId = localStorage.getItem('userId');
      if (!userId) {
        console.error('User ID not found in localStorage');
        return;
      }

      try {
        const token = localStorage.token;
        const response = await fetch(`http://localhost:3000/post/followings/${userId}`, {
          headers: {
            'Authorization': `Bearer ${token}`
          }
        });

        if (!response.ok) {
          throw new Error('Failed to fetch posts');
        }

        const data = await response.json();
        setPosts(data);
      } catch (error) {
        console.error('Error fetching posts:', error);
      }
    };
```

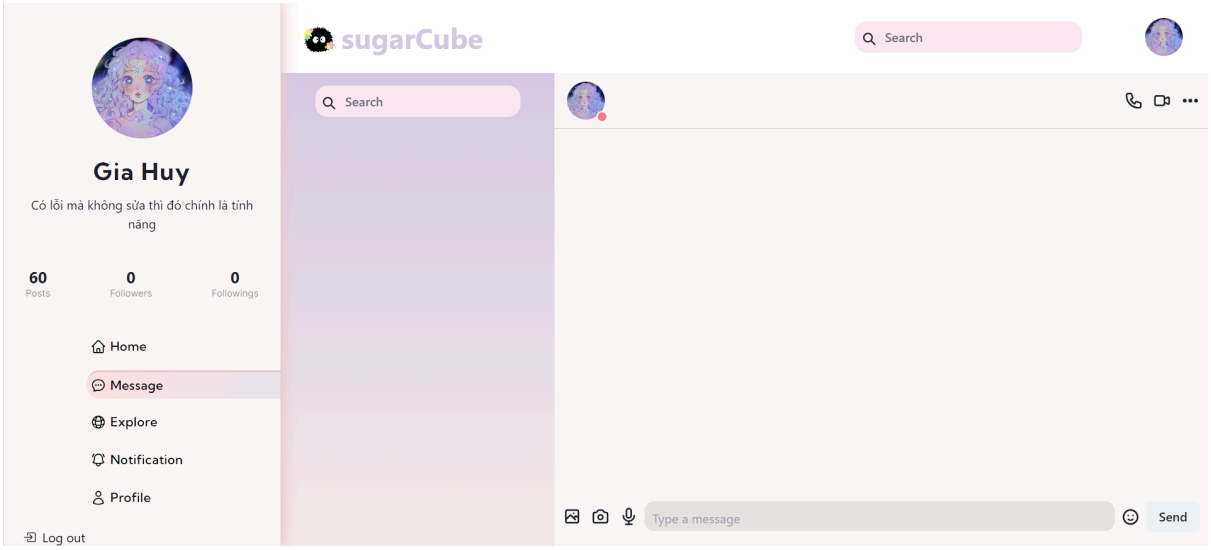
- **State Initialization**: The component initializes a state variable posts to store the fetched posts.
- **Effect Hook**: The useEffect hook is used to fetch posts when the component mounts.
    - **User ID Retrieval**: The user's ID is retrieved from localStorage.
    - **Token and Fetch Request**: The user's token is also retrieved from localStorage, and a fetch request is made to the server to get the posts of the followed accounts.
    - **Error Handling**: Errors during the fetch process are caught and logged to the console.

The "Home Page" implementation in SugaCube effectively combines server-side and client-side logic to fetch and display posts from followed accounts. The server handles the database interactions and filtering of posts, while the client manages the rendering and user interactions. This approach ensures a dynamic and engaging experience for users as they scroll through their feed.

C. Page Message

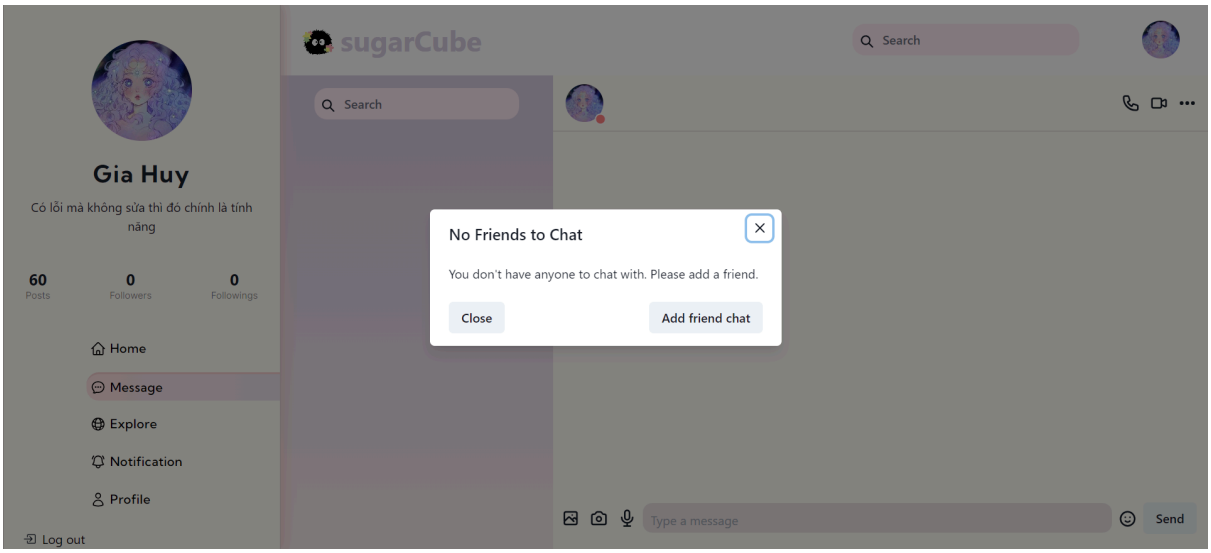
1. Demo

-When you entering the message bar, the user interface would look like the image below.



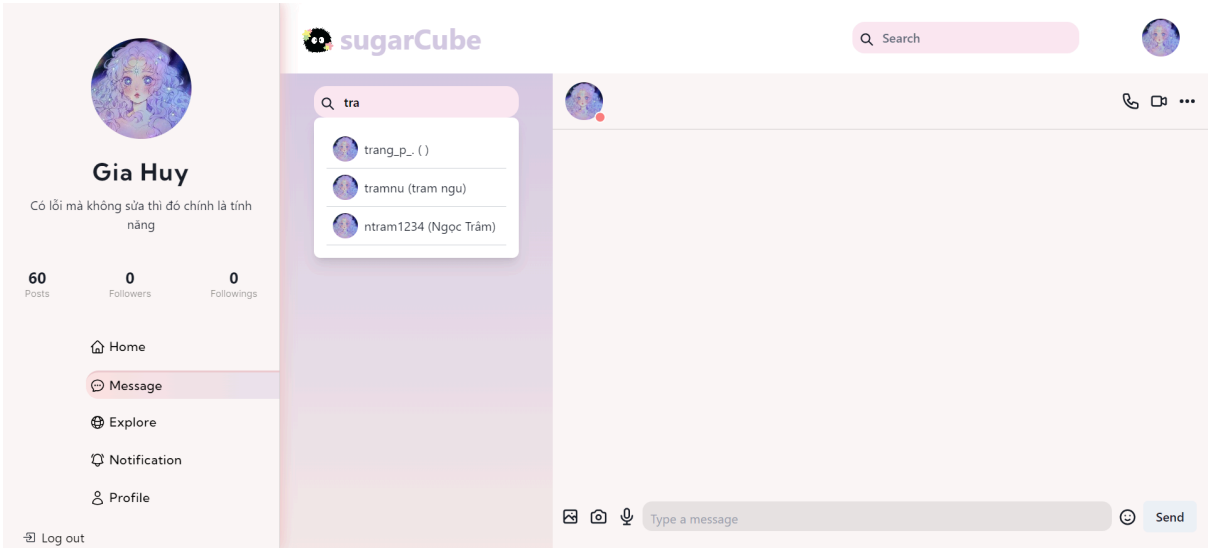
Img.3.C.1: UI of chat message function

-If this is your first time, the system would ask you to add another user to your friend list.



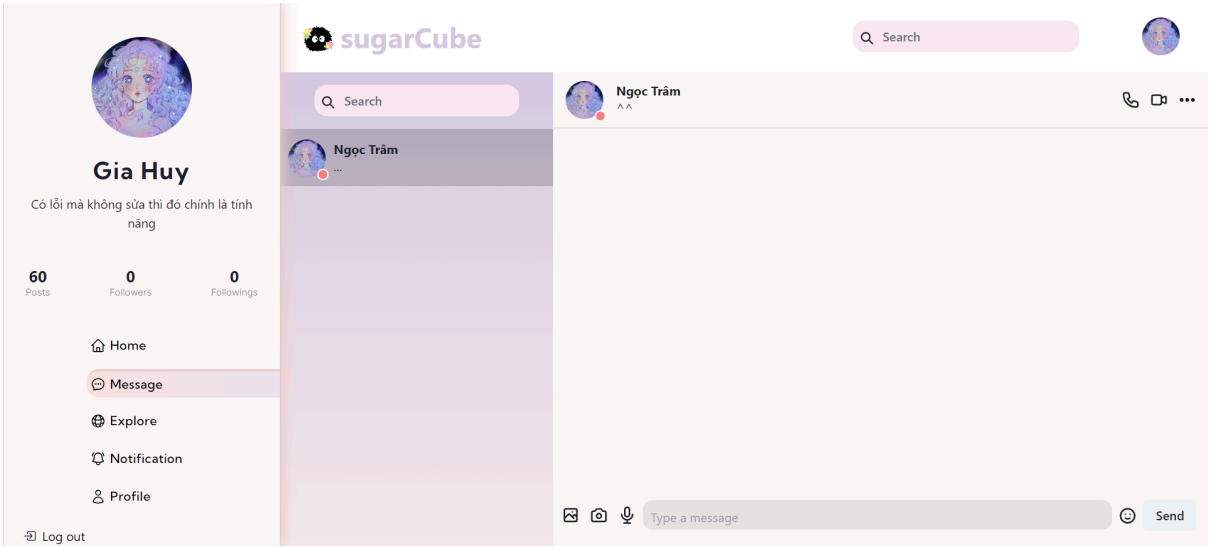
Img.3.C.2: add friend modal

-There are two option, the the cancel would close the box, the ‘’Add friend chat” would focus use to the Search bar, where you could add other people to your chat list.
-If you enter the user name that you want to chat with, the system, alos suggest base on the similarity like the image below.



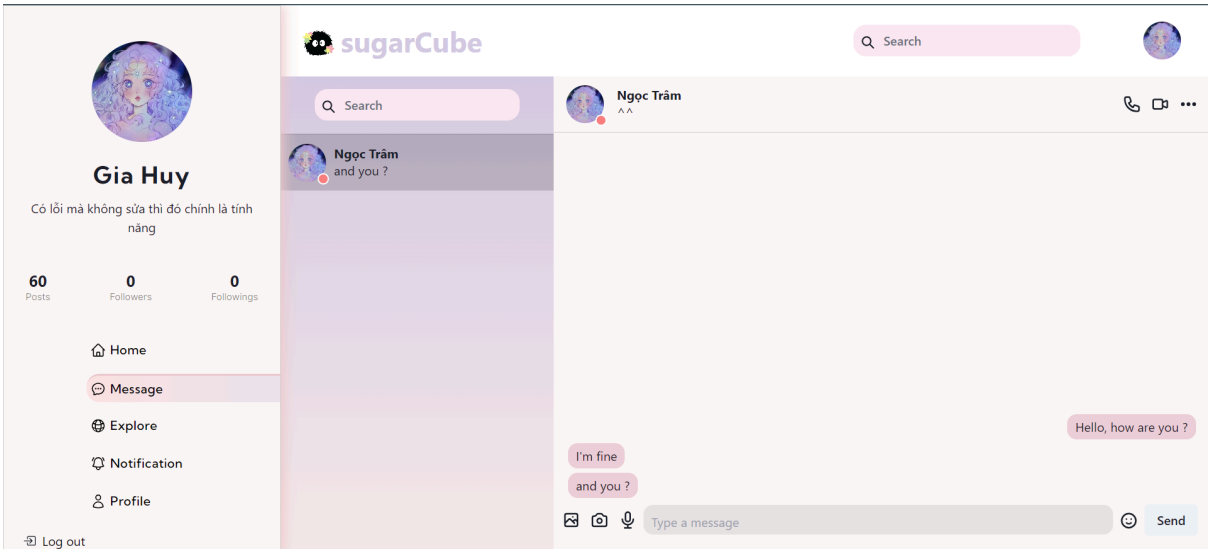
Img.3.C.3: Suggestion if type any key

-If you click to the one that you want to chat, this action would add this user to your chat list.



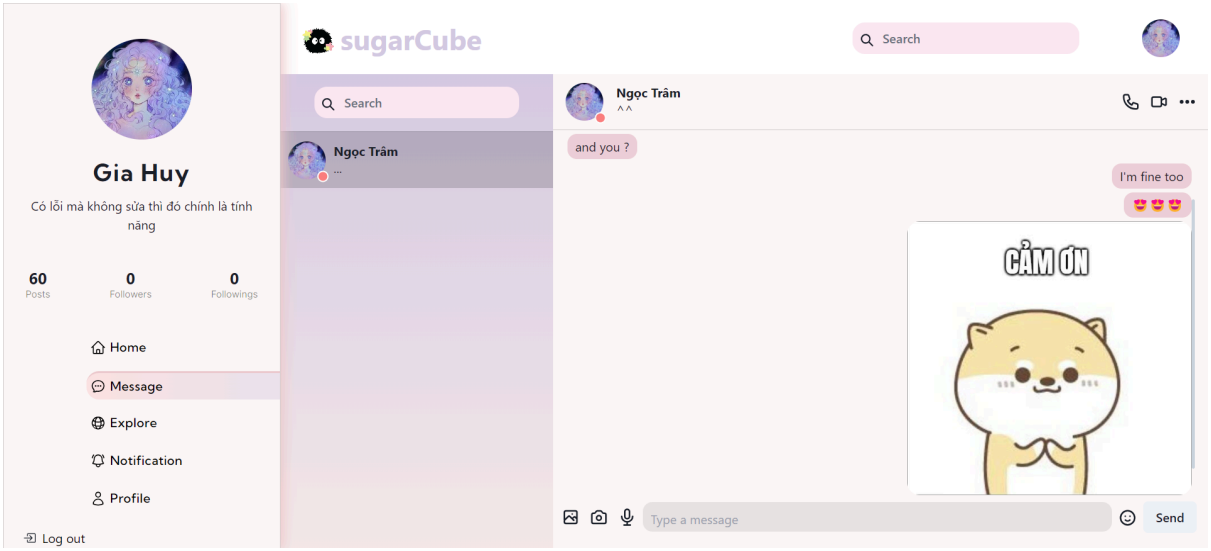
Img.3.C.4: After add other to chat list.

-From now on, you could chat with your new friend.



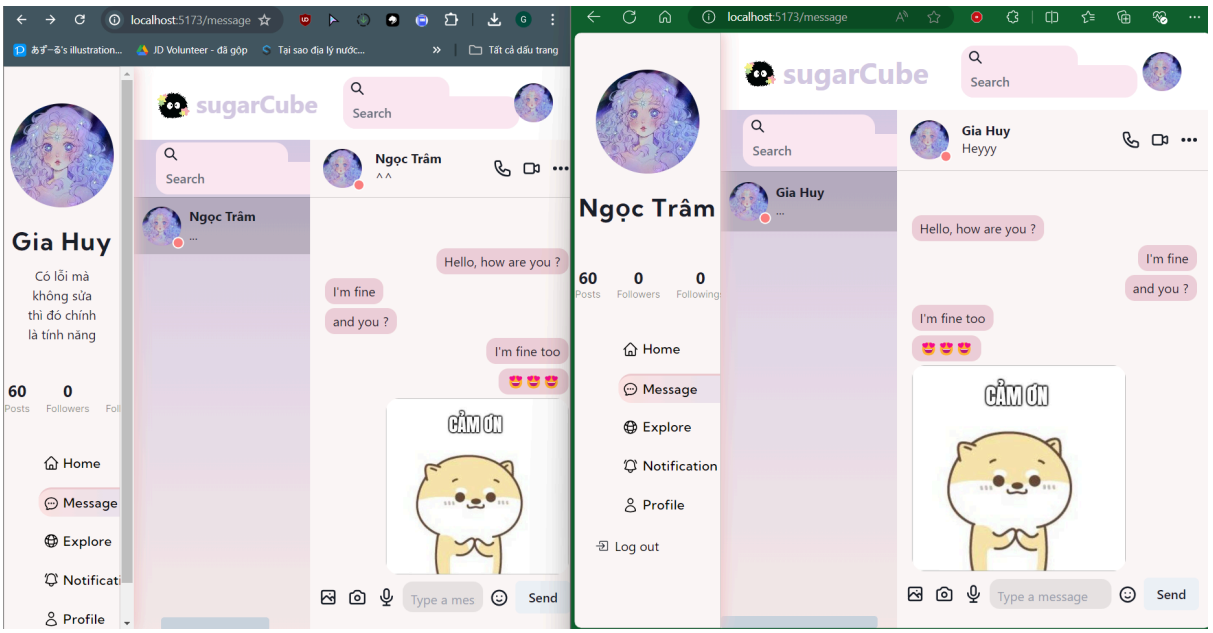
Img.3.C.5: Chat message

-So on, in the message, this could also include icon and image.



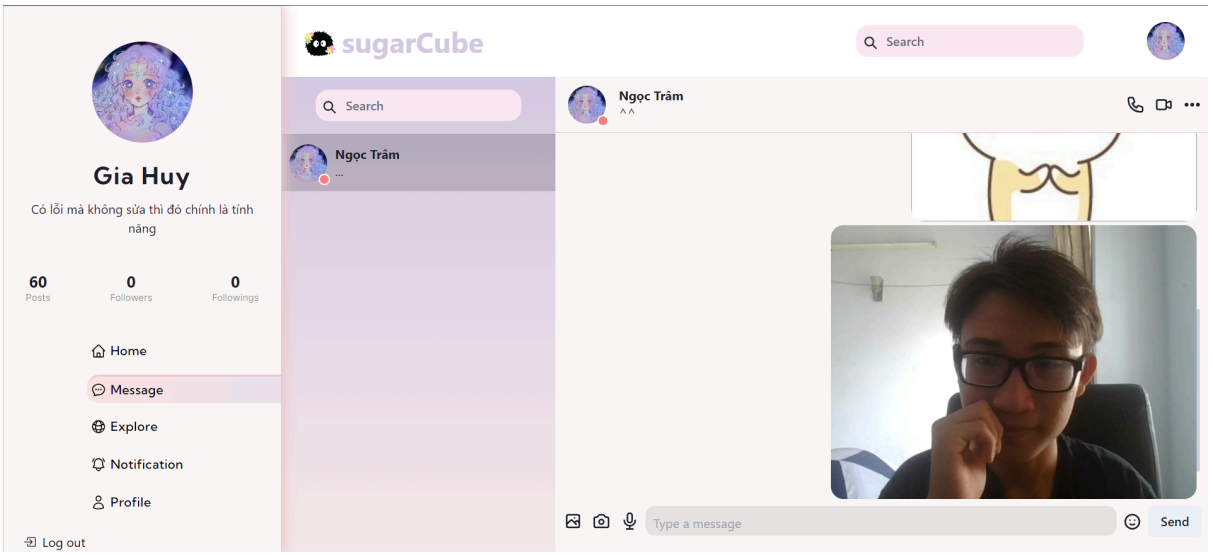
Img.3.C.5: Send Icon and Image

-But the most important thing ís that, you can chat with your friend in real life.



Img.3.C.6: One-to-one online message

-Moreover, you also take image from your camera to your friend by click to the photograph icon on the left side of chat part.



Img.3.C.7: Take picture function

2. Implementation

-First of all, we have to set up socket.io.

```
import io from 'socket.io-client';

let socket;

const connectSocket = (userID) => {
  if (!socket) {
```

```
    socket = io("http://localhost:3000", {
      query: `userID=${userID}`,
      reconnection: true,
      reconnectionAttempts: Infinity,
      reconnectionDelay: 1000,
    });

    socket.on('disconnect', () => {
      console.log('Socket disconnected');
    });
  }
};

export { socket, connectSocket };
```

Img.3.C.8: socket.io set up environment

-Basicly, the code above is to export the socket and the connectSocket function, which would connect to socket server anytime that you login.
-Another one is send message function.

```
const handleSendMessage = async () => {
    if (text === "" && !image.file) return;

    let imgURL = null;

    if (image.file) {
      try {
        imgURL = await uploadImage(image.file);
      } catch (error) {
        console.error("Image upload failed: ", error);
        return;
      }
    }

    const message = {
      userId1: currentUserId,
      userId2: recepientID,
      username: currentUsername,
```

```
        content: text,
        imageURL: imgURL,
      };

      try {
        await apiRequestPost(
          "http://localhost:3000/message",
          accessToken,
          message
        );

        dispatch(
          changeLastMessage({
            content: text,
            recepientID: recepientID,
          })
        );
        dispatch(
          addMessage({
            content: text,
            imageURL: imgURL,
            username: currentUsername,
          })
        );

        if (socket) {
          socket.emit("send-message", message);
        }

        const audio = new Audio("sound/sent_message.mp3");
        audio.play();

        setText("");
        setImage({
          file: null,
          url: "",
        });
```

```
    } catch (error) {
      console.error("Failed to send message: ", error);
    }
  };
```

Img.3.C.9: handle send message function

-Anytime that you enter a message and click send, this logic would happen. Firstly, it check if the text or the image exist or not, if not just return, else continue. If the image exist, then it would save the image to Firebase server through uploadImage() method. Next, it would create the message concrete, and save to MongoDB cloud through the API. Then it would update the state management (using redux with slice reducer) to load the message/image to above. The most important things here is to send the message to socket.io server. The server would then send this message to target user. The image illustrate how the socket server send to target user to keep one-to-one online chatting. Next, it make a sound to note the message have already sent. Finally, it would reset the image and text state.

```
socket.on("send-message", async (message) => {
    try {
      const senderID = message.userId1;
      const recepientID = message.userId2;

      const senderUser = await User.findById(senderID)
        .select("username")
        .lean();
      const senderUsername = senderUser ? senderUser.username : null;

      const packageMessage = {
        content: message.content,
        imageURL: message.imageURL,
        username: senderUsername,
        recepientID: recepientID,
      };

      const toSocketUser = await User.findById(recepientID)
        .select("socket_id")
        .lean();
      const toSocketID = toSocketUser ? toSocketUser.socket_id : null;

      const fromSocketUser = await User.findById(senderID)
        .select("socket_id")
```

```
            .lean();
        const fromSocketID = fromSocketUser ? fromSocketUser.socket_id : null;

        if (toSocketID) {
            io.to(toSocketID).emit("new-message", packageMessage);
        }

        if (fromSocketID) {
            io.to(fromSocketID).emit("verify-sent", {
                message: "Sent message successfully",
            });
        }
    } catch (error) {
        console.log("Error in send-message event:", error);
    }
});
```

Img.3.C.10: socket.io server receives event from client

-For the chat list, it would first fetch the current data (chat list in user account)

```
const currentUserId = Cookies.get('userId');
  const accessToken = Cookies.get("token");

  const [data, setData] = useState([]);
  const searchRef = useRef();
  const fetchedRef = useRef(false);
  const { isOpen, onOpen, onClose } = useDisclosure(); // useDisclosure hook to control
modal

  const fetchData = async () => {
    try {
      const userData = await fetchChatList(currentUserId, accessToken);
      setData(userData);
      if (userData.length === 0) {
        onOpen(); // Open modal if no data
      }
    } catch (error) {
      console.error("Error happened when fetching data: ", error);
```

```
    }
  };

  const handleSearch = () => {
    console.log('SearchBar ref:', searchRef.current);
    if (searchRef.current) {
      searchRef.current.focus(); // Use focus method from SearchBar
      onClose();
    }
  };

  useEffect(() => {
    if (currentUserId) {
      if (!fetchedRef.current) {
        fetchedRef.current = true;
        fetchData();
      }
    }
  }, [currentUserId]);
```

Img.3.C.11: Fetch chat list of the current user

-If chat list is empty, then it would pop up a modal to notify you to add other users to your chat list.

```
<Modal isOpen={isOpen} onClose={onClose} isCentered>
            <ModalOverlay />
            <ModalContent>
              <ModalHeader>No Friends to Chat</ModalHeader>
              <ModalCloseButton />
              <ModalBody>
                You don't have anyone to chat with. Please add a
                friend.
              </ModalBody>
              <ModalFooter>
                <Flex
                  flexDir={"row"}
                  justifyContent="space-between"
                  w="100%"
                >
```

```
                    <Button onClick={onClose}>Close</Button>
                    <Button onClick={handleSearch}>
                      Add friend chat
                    </Button>
                  </Flex>
                </ModalFooter>
              </ModalContent>
```

Img.3.C.12: Model that pop up if chat list is empty

-In this pop up modal, there two button, first one is to close, the second option is to add friend to your chat list by the handleSearch() method.
- The logic behind add other to your chat list is that it would use the api (add other to chat list base on their _id). The image below show the logic.

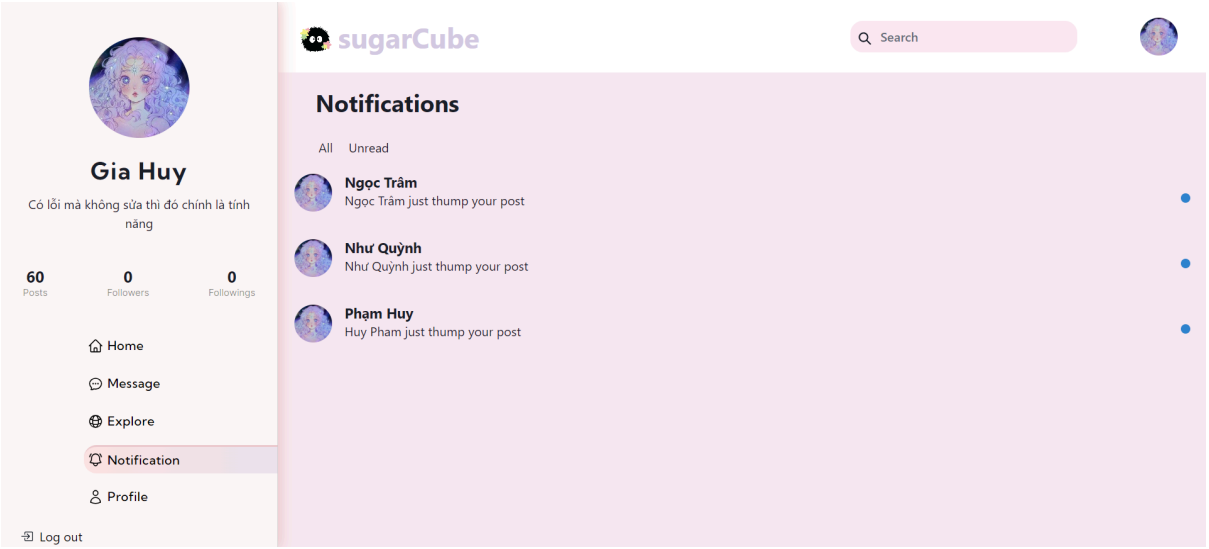
```
const handleUserClick = async (username, userID) => {
    if (location === "Message") {
      try {
        await apiRequestPost(
          "http://localhost:3000/user/chatlist",
          accessToken,
          {
            currentUserID,
            addUserID: userID,
          }
        );
      } catch (error) {
        console.log("Error when add chatlist: ", error);
        return;
      }
      setQuery('');
      navigate("/message");
      window.location.reload();
    } else {
      navigate(`/user/${username}`);
      window.location.reload();
    }
  }
```

Img.3.C.13: Logic to handle add new chat user.

D. Page Notification

1. Demo

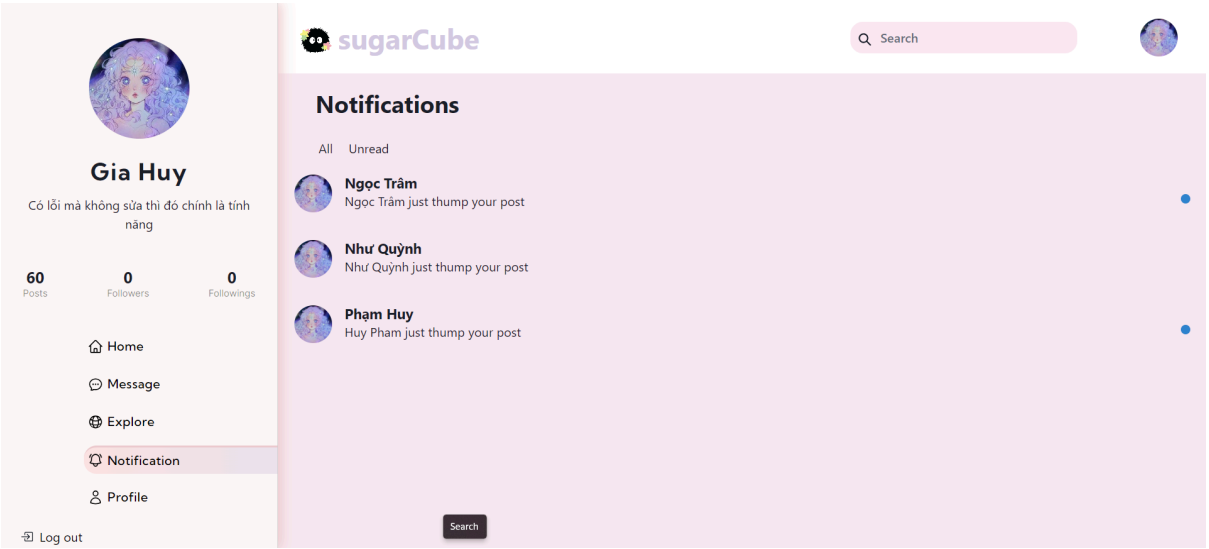
-When you click into the Notification bar, it would result in the UI like image Img3.D.1



Img3.D.1: UI of the notification page

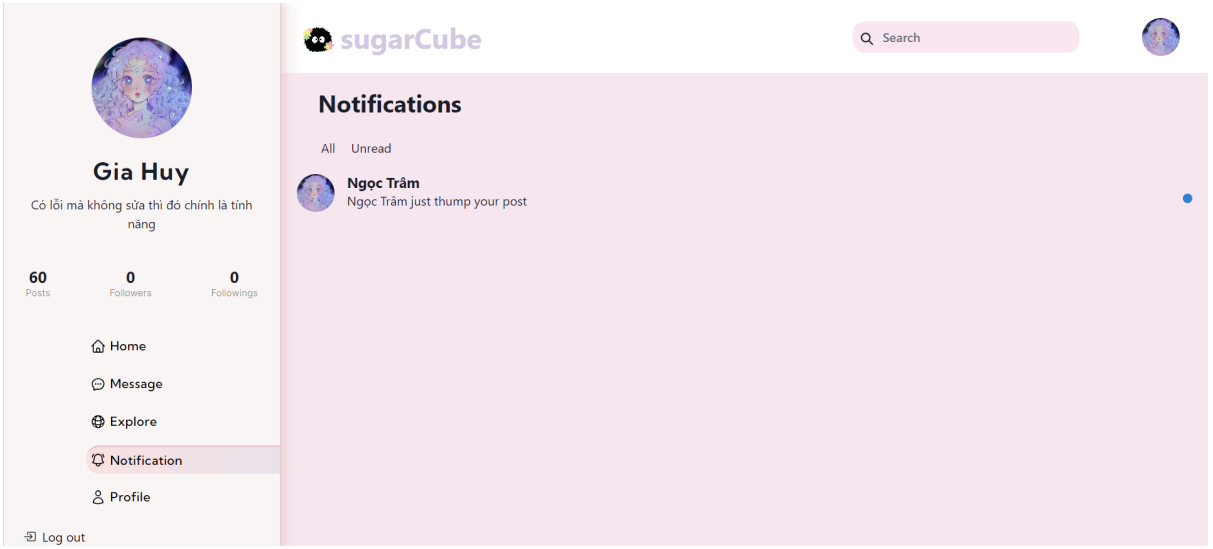
-Anytime that one thump your post this would result in this one. And when you access, there a blue peanut on the right side that notice the notification still not see.

-When you click into the the notification, the blue peanut would disappear, this notify that this notification had beed read. (the Img3.D.2)



Img3.D.2: When click the blue peanut disappear

-The left top side of the page, there are two options, one to select all the notification, another is select one that still not been read.



Img.D.3: The one that read disappear

2. Implementation

-The Notification component, basic is done by using Chakra UI and some HTML code.

```
<div
    className="h-full w-full max-h-full overflow-auto overscroll-auto"
    style={bgColor}
>
    <div
      className="text-left text-3xl font-bold pt-5 mb-5 ml-12"
      style={bgColor}
    >
      Notifications
    </div>

    <Flex ml={"42px"} flexDir={"row"}>
      <ButtonNot contentText="All" onClick={handleAll} />
      <ButtonNot contentText="Unread" onClick={() => handleUnread(data)} />
    </Flex>

    {data.map((postData, index) => (
      <NotificationItem key={index} data={postData} />
    ))}
</div>
```

Img.3.D.4: Notification component

-The function featchData response for fetch the notifications from the database. This fetchData() function was revoke in the useEffect hook (Img3.D.6)

```
const fetchData = async () => {
    try {
      const userData = await fetchNotification(
        currentUserID,
        accessToken
      );
      const notifications = userData.map((item) => ({
        avatar: item.sender_id.profilePicture,
        name: `${item.sender_id.firstName} ${item.sender_id.lastName}`,
        content: item.contentNot,
        read: item.read,
        notificationID: item.notification_id,
      }));
      setData(notifications);
    } catch (error) {
      console.error("Error happened when fetching data: ", error);
    }
  };
```

Img3.D.5: fetchData() function

```
useEffect(() => {
    if (currentUserID) {
      if (!fetchedRef.current) {
        fetchedRef.current = true;
        fetchData();
      }
    }
  }, []);
```

Img3.D.6: useEffect() function to call fetchData()

-Also, the logic the make the blue peanut to disappear is just using a fillter function that loop through the data just fetch before and fillter all the unread() notification. This action also make the page to re-render.

```
const handleUnread = (allNotifications) => {
    const unreadNotifications = allNotifications.filter(
      (notification) => !notification.read
```

```
    );
    setData(unreadNotifications);
  };
```

Imge3.D.7: handleUnread() method

- And if the other user thump the your post, the would result in create a new notification, through an API as follow.

```
async function addNotification(req, res) {
  const { userID, contentNotification, senderID } = req.body;

  try {
    let notification = await NotificationModel.findOne({ user_id: userID });

    if (!notification) {
      notification = new NotificationModel({
        user_id: userID,
        content: [],
      });
    }

    const notification_id = notification.content.length + 1;

    const newNotification = {
      notification_id,
      sender_id: senderID,
      contentNot: contentNotification,
      read: false,
      timestamp: new Date(),
    };

    notification.content.push(newNotification);

    await notification.save();

    res.status(201).json({
      message: "Notification added successfully",
      notification,
    });
```

```
  } catch (error) {
    res.status(500).json({ message: "Failed to add notification", error });
  }
}
```

Img.3.D.8: add notification function

-And the one to get notification is below

```
async function getNotification(req, res) {
  const { user_id } = req.params;

  try {
    const notification = await NotificationModel.findOne({
      user_id,
    }).populate("content.sender_id");

    if (!notification) {
      return res
        .status(200)
        .json([]);
    }

    notification.content.sort((a, b) => b.timestamp - a.timestamp);

    res.status(200).json(notification.content);
  } catch (error) {
    res.status(500).json({ message: "Failed to get notifications", error });
  }
}
```

Img.3.D.9: get notification function

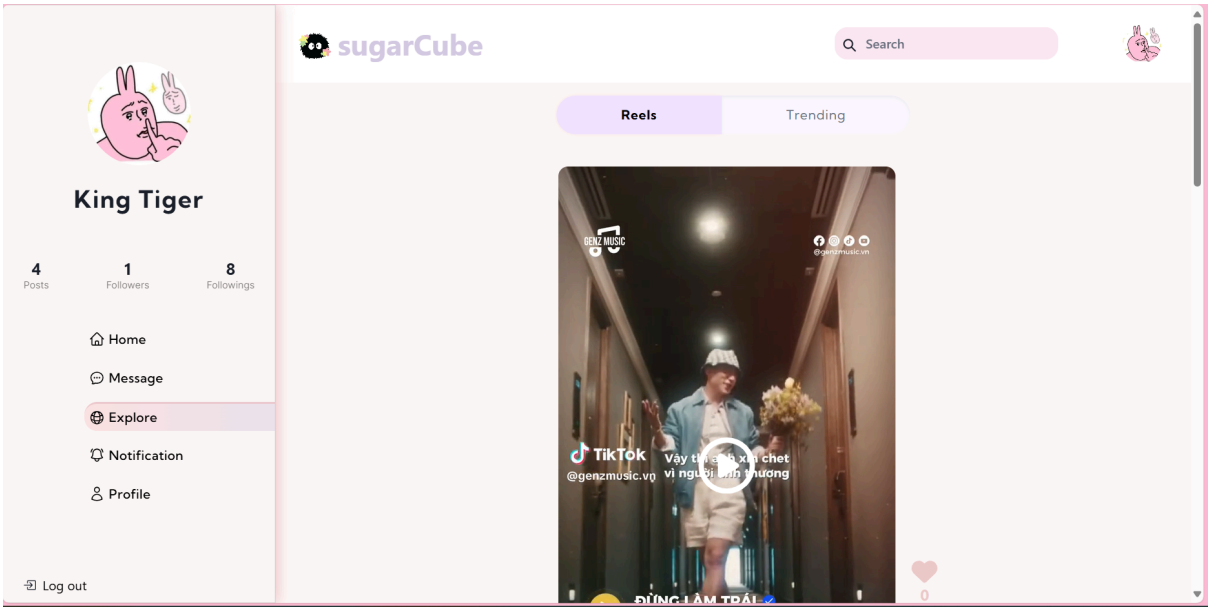
E. Page Explore

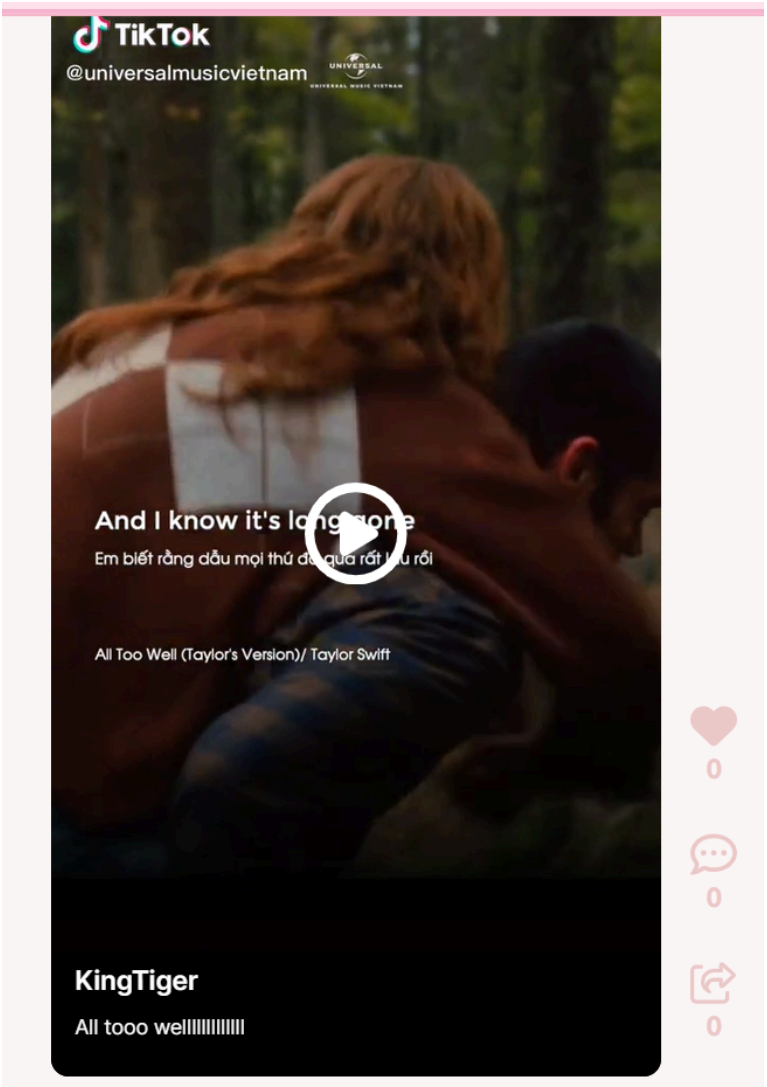
1. Demo

The Explore page on Suga Cube is the perfect destination for users to discover diverse and exciting content from the community. This page is divided into two main sections: Explore Reels and Explore Trending.

- Explore reels: The Explore Reels section showcases a variety of videos randomly selected from our database. All videos displayed here are set to public mode, ensuring that everyone can view and engage with them. The

videos are arranged randomly to provide a fresh and rich exploration experience for users.

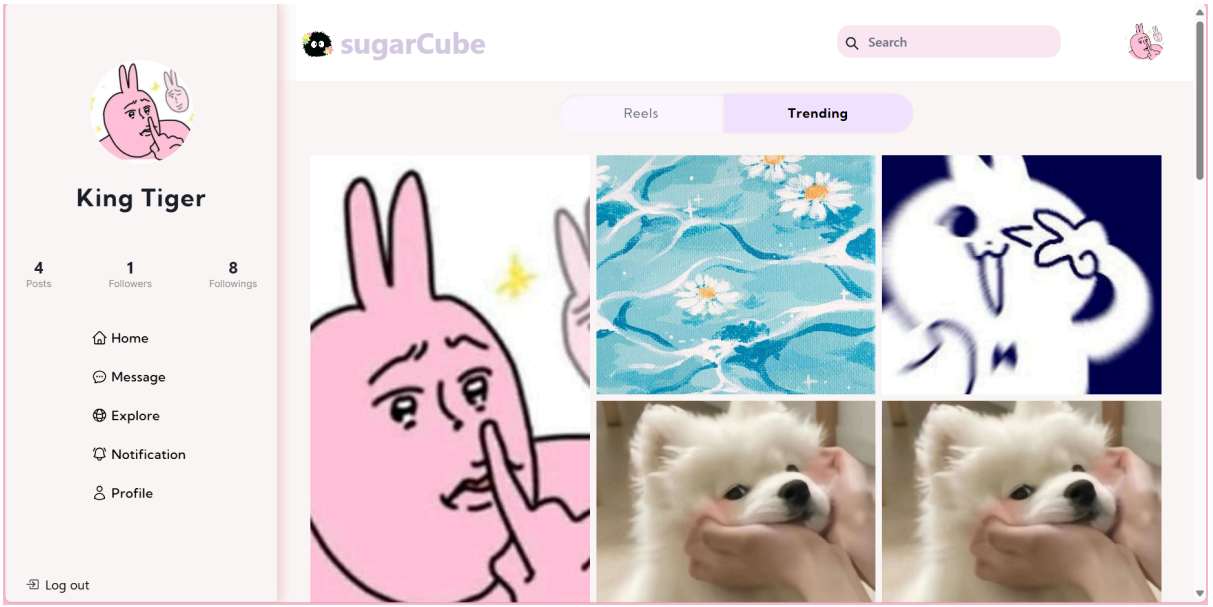




For each video rendered in the Explore Reels section, the following elements are displayed:

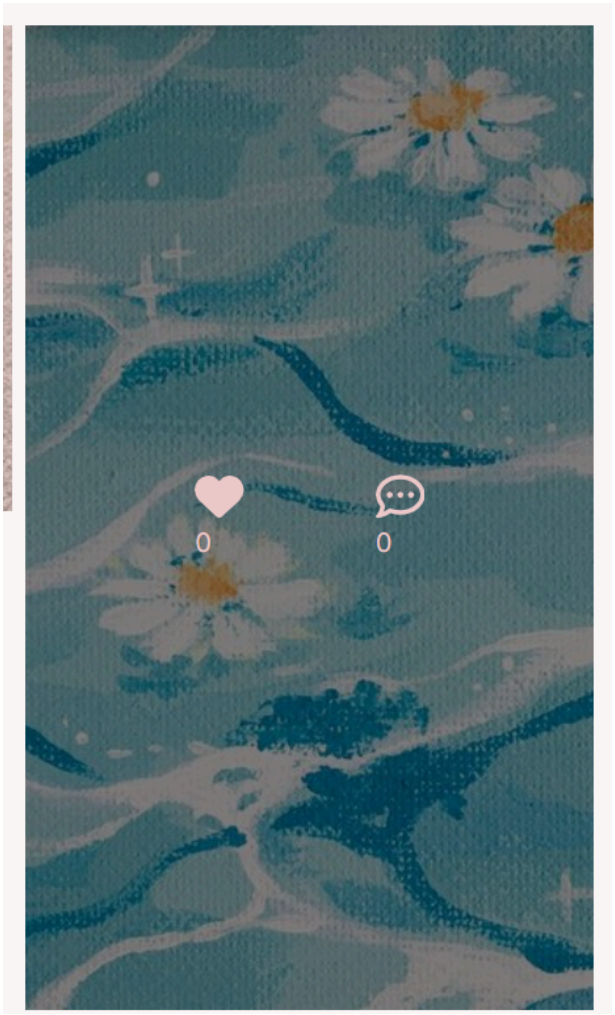
- User Name: The name of the user who posted the video.
- Caption: A description or caption of the video.
- Interaction Metrics: To the right of each video, users can see the number of interactions, including likes, shares, and comments.

- Explore Trending: The Explore Trending section highlights popular images randomly selected from our database. Similar to Explore Reels, all images here are in public mode and arranged randomly. This allows users to easily find stunning photos and discover the latest trends within the Suga Cube community.



For each image rendered in the Explore Trending section:

- Hover Interaction Metrics: When a user hovers over an image, they can see interaction metrics such as the number of likes and comments.



2. Implementation:
   Explore Reels and Explore Trending. Both sections utilize the getAllPublicPosts API function on the server side to fetch public posts from the database, and they present the data differently on the client side.

- Server - side:
  The getAllPublicPosts function is responsible for retrieving all public posts from the database and preparing the data to be used by the client-side components.

```
async function getAllPublicPosts(req, res) {
    try {
        // Aggregate posts with privacyLevel "Public" and join with User collection
        const result = await PostModel.aggregate([
            { $unwind: '$posts' },
            { $match: { 'posts.privacyLevel': 'Public' } },
            {
                $lookup: {
                    from: 'users',
                    localField: 'Object_id',
                    foreignField: '_id',
                    as: 'user'
                }
            },
            { $unwind: '$user' },
            {
                $project: {
                    _id: 0,
                    userId: '$user._id',
                    username: '$user.username',
                    post: '$posts'
                }
            }
        ]);

        console.log("Result from aggregation:", JSON.stringify(result, null, 2));

        if (result.length === 0) {
            return res.status(404).json({ message: 'No public posts found' });
        }

        const publicPosts = shuffle(result);
        res.json(publicPosts);
    } catch (error) {
        console.error("Error fetching public posts:", error);
        res.status(500).json({ message: 'Internal server error' });
    }
```

- **Aggregation**: The function aggregates posts with the privacy level "Public" and joins the posts with the User collection to get user details.
- **Unwind and Match**: The posts are unwound to separate documents, and only public posts are matched.
- **Lookup**: A lookup is performed to join the posts with the corresponding user details from the User collection.
- **Project**: The projection stage formats the output to include userId, username, and the post details.
- **Shuffle and Respond**: The resulting posts are shuffled to ensure randomness and sent back in the response.

- Client - side:
  Explore Reels: The ExploreReels component fetches video posts from the server and displays them.

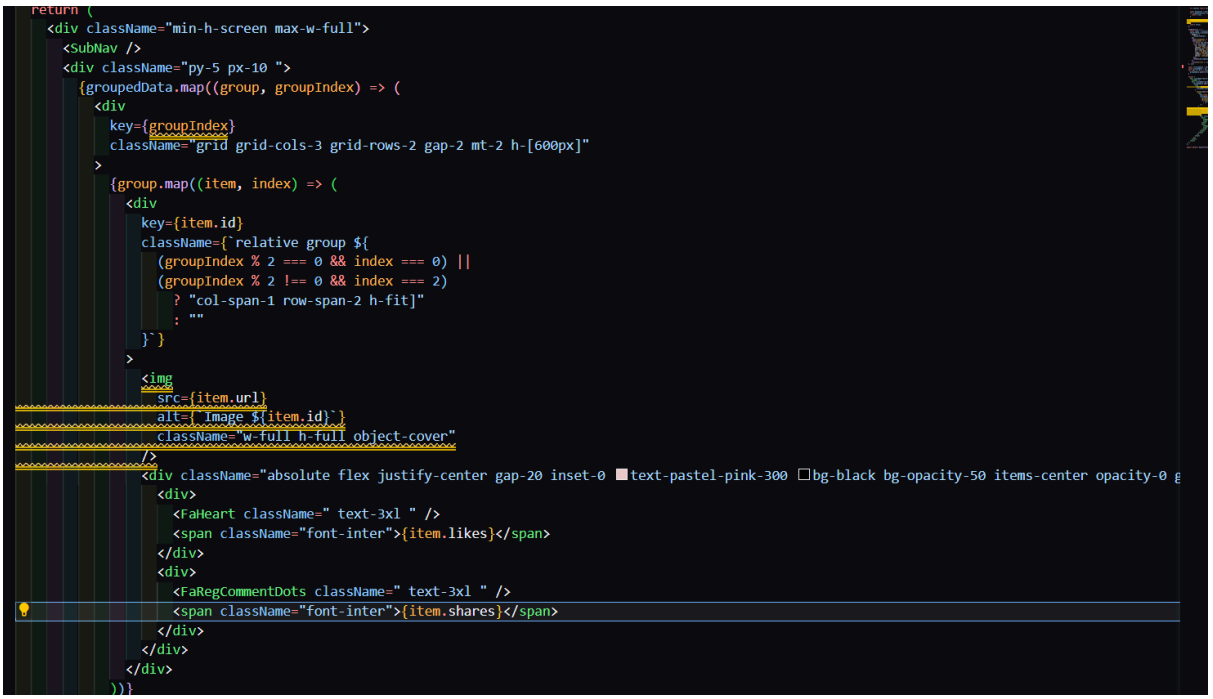
```
Codiumate: Options | Test this function
function ExploreReels() {
  const [data, setData] = useState([]);

  useEffect(() => {
    async function fetchData() {
      const token = localStorage.token;
      try {
        const response = await fetch('http://localhost:3000/post/', {
          headers: {
            'Authorization': `Bearer ${token}`
          }
        });
        const result = await response.json();
        const videoPosts = result.filter(video => isVideo(video.post.mediaURL));
        setData(videoPosts);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    }
    fetchData();
  }, []);
```

```
return (
  <div className="min-h-screen overflow-hidden">
    <SubNav />
    <div className="flex-1 h-full relative rounded-xl pl-14 py-10 flex flex-col space-y-10 items-center justify-center">
      {data.map((video, index) => (
        <Video
          key={index}
          channel={video.username}
          description={video.post.content}
          url={video.post.mediaURL}
          likes={video.post.like}
          comment={video.post.comment}
          shares={video.post.share}
        />
      ))}
    </div>
  </div>
);
}
```

- State Initialization: The component initializes a state variable data to store the fetched posts.
- Fetch Data: An useEffect hook is used to fetch data when the component mounts. The fetch request retrieves posts from the server, and the response is filtered to include only video posts using the isVideo function.
- Render Videos: The filtered video posts are rendered using the Video component, which displays the video along with the username, description, and interaction metrics (likes, comments, and shares).

Explore Trending: The ExploreTrending component fetches image posts from the server and displays them in a grid. When an image is hovered over, interaction metrics are shown.

```
useEffect(() => {
  const token = localStorage.token;
  fetch("http://localhost:3000/post/", {
    headers: {
      'Authorization': `Bearer ${token}`
    }})
    .then((response) => response.json())
    .then((data) => {
      // Parse the fetched data to match the required format
      const parsedData = data.map((item) => ({
        id: item.post._id,
        channel: item.username,
        description: item.post.content,
        url: item.post.mediaURL,
        likes: item.post.like,
        comment: item.post.comment,
        shares: item.post.share,
      }));
      setData(parsedData);
    })
    .catch((error) => console.error("Error fetching data:", error));
}, []);

const filteredData = data.filter(item => !isVideo(item.url));
const groupedData = [];
for (let i = 0; i < filteredData.length; i += 5) {
  groupedData.push(filteredData.slice(i, i + 5));
}
```



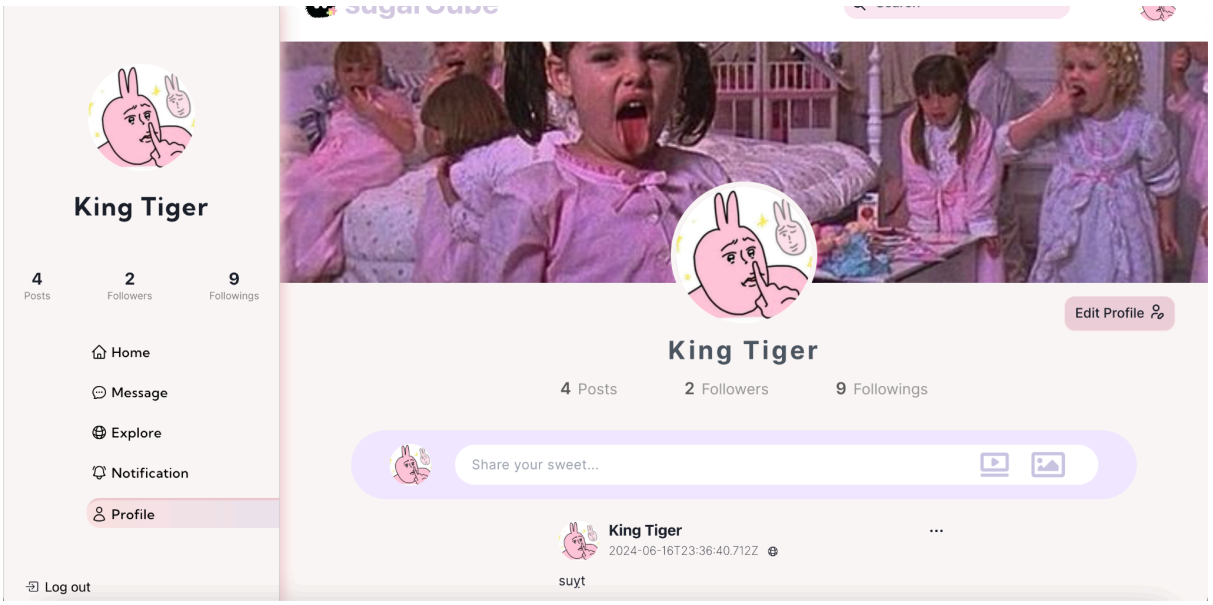
```
return (
  <div className="min-h-screen max-w-full">
    <SubNav />
    <div className="py-5 px-10 ">
      {groupedData.map((group, groupIndex) => (
        <div
          key={groupIndex}
          className="grid grid-cols-3 grid-rows-2 gap-2 mt-2 h-[600px]"
        >
          {group.map((item, index) => (
            <div
              key={item.id}
              className={`relative group ${
                (groupIndex % 2 === 0 && index === 0) ||
                (groupIndex % 2 !== 0 && index === 2)
                  ? "col-span-1 row-span-2 h-fit]"
                  : ""
              }`}
            >
              <img
                src={item.url}
                alt={`Image ${item.id}`}
                className="w-full h-full object-cover"
              />
              <div className="absolute flex justify-center gap-20 inset-0 text-pastel-pink-300 bg-black bg-opacity-50 items-center opacity-0 g
                <div>
                  <FaHeart className=" text-3xl " />
                  <span className="font-inter">{item.likes}</span>
                </div>
                <div>
                  <FaRegCommentDots className=" text-3xl " />
                  <span className="font-inter">{item.shares}</span>
                </div>
              </div>
            </div>
          </div>
        ))}
```

- **State Initialization**: The component initializes a state variable data to store the fetched posts.
- **Fetch Data**: An useEffect hook is used to fetch data when the component mounts. The fetch request retrieves posts from the server, and the response is mapped to the required format.
- **Filter and Group Data**: The posts are filtered to exclude videos using the isVideo function. The remaining image posts are grouped into chunks of five.
- **Render Images**: The grouped image posts are rendered in a grid layout. Each image has interaction metrics (likes and comments) displayed when hovered over, using CSS transitions for smooth effects.

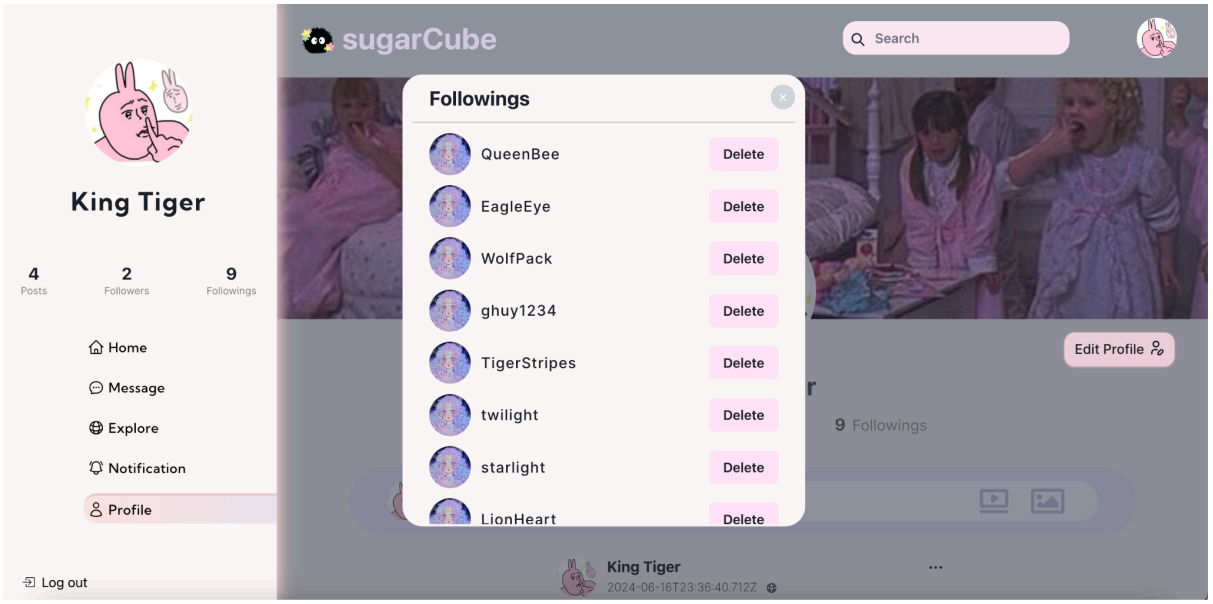
The implementation of the "Explore" page on SugaCube involves fetching public posts from the server and displaying them in two distinct sections: Explore Reels and Explore Trending. The server-side function getAllPublicPosts aggregates and prepares the data, while the client-side components handle the presentation and interaction logic. This setup ensures a dynamic and engaging user experience for exploring videos and trending images on the platform.

F. Page Profile

- Demo:



Click on the Profile button in the navigation bar on the right side, the profile page of our account will show up. The profile page will contain the avatar, cover picture, the post status bar, the number of post followings followers and the post that we have posted. We can check for the number of followings, followers and see the posts that we have posted. Click on the work following follower can show us the list of followings and followers and their information.



- We can also delete the user we just follow. Click on the “following” word, it will show up a list of pages we have followed. We can then click the delete button to delete that user.

- Implementation:

- Server-side:
  - The getFollowings and getFollowers api is responsible for getting the information of who we are followed and who are following us from the

database. The server will search through the database and return us all information of followings and followers. The information will return in an array form.

```
getFollowings = async (req, res) => {
    try {
        const user = await User.findOne({
            username: req.params.username,
        }).populate("followings.following_id", "username profilePicture");
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        res.status(200).json(user.followings);
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
};
```

```
  getFollowers = async (req, res) => {
      console.log(req.params.username);
      try {
          const user = await User.findOne({
              username: req.params.username,
          }).populate("followers.follower_id", "username profilePicture");

          if (!user) {
              return res.status(404).json({ message: "User not found" });
          }

          res.status(200).json(user.followers);
      } catch (err) {
          res.status(500).json({ message: err.message });
      }
  };
```

```
async function getAllPostsByUser(req, res) {
    const userId = req.params.userId;
    console.log(userId)
    try {
        // Fetch posts for the specific userId and populate the Object_id field
        const posts = await PostModel.find({ Object_id: userId }).populate('Object_id');

        // Initialize an array to store all posts
        const allPosts = [];

        // Iterate through fetched posts and construct the desired format
        posts.forEach(post => {
            if (post.Object_id && post.posts && post.posts.length > 0) {
                post.posts.forEach(p => {
                    allPosts.push({
                        userId: userId,
                        post: {
                            post_id: p.post_id,
                            content: p.content,
                            imageURL: p.imageURL,
                            tags: p.tags,
                            like: p.like,
                            privacyLevel: p.privacyLevel,
                            comment: p.comment,
                            share: p.share,
                            mediaURL: p.mediaURL,
                            timestamp: p.timestamp,
                            _id: p._id
                        }
                    });
                });
            }
        });

        // Sorting posts by timestamp in descending order (newest to oldest)
        allPosts.sort((a, b) => new Date(b.post.timestamp) - new Date(a.post.timestamp));

        res.json(allPosts);
    } catch (error) {
        console.error("Error fetching posts:", error);
        res.status(500).json({ message: "Internal server error" });
    }
};
```

- The getAllPostsByUser api is responsible for getting all the posts of the user from the database, it will show us all the post of that specific user have posted and will be showed in the profile page of that user.

```
removeFollowing = async (req, res) => {
    try {
        const user = await User.findOne({ username: req.params.username });
        const following = await User.findById(req.body.following_id);
        if (!user || !following) {
            return res
                .status(404)
                .json({ message: "User or following not found" });
        }

        user.followings = user.followings.filter(
            (following) =>
                following.following_id.toString() !== req.body.following_id
        );

        following.followers = following.followers.filter(
            (follower) =>
                follower.follower_id.toString() !== user._id.toString()
        );

        await user.save();
        await following.save();

        res.status(200).json({ message: "Following removed successfully" });
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
};
```

- Client-side:
    - The Profile will show us all information of the post the user has posted, and will show the ProfileHeader and the StatusBar.

```
function Profile() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetchPosts();
  }, []);

  const fetchPosts = async () => {
    const userId = localStorage.getItem('userId');
    if (!userId) {
      console.error('User ID not found in localStorage');
      return;
    }

    try {
      const token = localStorage.token;
      const response = await fetch(`https://sugar-cube.onrender.com/post/${userId}`, {
        headers: {
          'Authorization': `Bearer ${token}`
        }
      });

      if (!response.ok) {
        throw new Error('Failed to fetch posts');
      }

      const data = await response.json();
      setPosts(data);
      console.log('Posts updated:', data); // Log updated data to check if state is updated correctly
    } catch (error) {
      console.error('Error fetching posts:', error);
    }
};
```

- The ProfileHeader will show us all information including the number of posts the user has posted, the number of followers the user has followed and the number of followers the user has followed. Each will fetch the api of that specific task and will return the arrays, the arrays will then be mapped into a specific component.

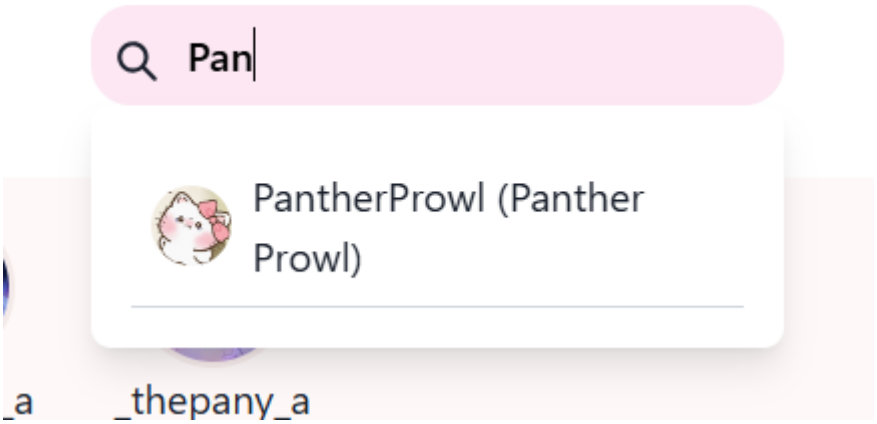
```
115  const ProfileHeader = () => {
199
200    return (
201      <>
202        <FollowInfor
203          isFollowers={isFollowers}
204          isFollowings={isFollowings}
205          handleClose={handleClose}
206          dataFollowers={dataFollowers}
207          dataFollowings={dataFollowings}
208          userData={user}
209        />
210        <div className="w-full relative font-inter">
211          <div className="bg-white h-72 relative group">
212            <img
213              src={userData.coverPicture || BannerImage}
214              alt="Profile Banner"
215              className="w-full h-full object-cover"
216            />
217            <input
218              type="file"
219              id="cover-input"
220              className="hidden"
221              onChange={handleCoverChange}
222            />
223            <label
224              htmlFor="cover-input"
225              className="absolute top-2 z-50 right-2 bg-gray-800 bg-opacity-50 font-inter text-white text-sm py-1 px-2 rounded-xl cursor-pointer opacity-0 hover:opacity
226            >
227              Edit Cover
228            </label>
229            <div className="flex justify-end py-4 px-10">
230              <button
231                onClick={handleEditProfile}
232                className="bg-pastel-pink-200 z-10 rounded-xl py-2 px-3 flex flex-row gap-1 font-medium shadow-sm shadow-pastel-pink-300 hover:bg-pastel-pink-300 hove
233              >
234                Edit Profile
235                <TbUserEdit className="h-5 w-5" />
```

- The FollowInfor will act like an pop up page that show us the list of the information of the following and follower. We will set for the specific variable that if it is true, it will return that pop up page or else it will return null.

```
const FollowInfor = ({ isFollowers, isFollowings, handleClose, dataFollowers, dataFollowings, userData }) => {


    if (!isFollowers && !isFollowings) return null;

    console.log(dataFollowings);
    console.log(dataFollowers);

    const handleDelete = (userDataId) => {
        console.log(userDataId);
        if (isFollowings){
            fetchDeleteFollowings(userData, userDataId);
        }
    };

    if(isFollowers){
        return (
            <div className="fixed inset-0 bg-gray-800 bg-opacity-50 flex justify-center items-center z-50">
            <div className="bg-pastel-pink-100 p-3 rounded-3xl h-3/4 w-1/3 relative">
                <p className="inline-block font-bold text-2xl pl-5">Followers</p>
                <button
                type="button"
                className="bg-gray-300 text-white text-lg px-[9px] pb-[0.1rem] rounded-full hover:bg-gray-400 absolute right-3"
                onClick={handleClose}
                >
                &times;
                </button>
                <div className="flex flex-row pt-3">
                <div className="h-0.5 bg-gray-300 w-full" />
                </div>
                <div className="h-[93%] w-full overflow-auto overscroll-auto">
                    {dataFollowers.map((data, index) => (
                        <div key={index} className="flex justify-start items-center gap-3 pt-3 px-5 h-[13%]">
                            <div className="relative w-[50px] h-[50px] rounded-full overflow-hidden">
                                <img
                                src="/img/avatar.png"
                                alt="avatar"
```

G. Extra

1. Searching bar:
   - Demo:



Type the username, first name, or last name of the user you are looking for. The search will filter results in real-time.The results will appear in a dropdown below the search bar. Click on a user to view their profile.

- Implementation:

- Server-side:
  - The removeFollowing api is responsible for removing users from the database. The data includes username and the id of the user we want to unfollow, and the server will remove the id of the user we have followed from the database.

```
useEffect(() => {
    // Fetch the user list from the API
    const fetchUsers = async () => {
        try {
            const token = localStorage.getItem("token");
            const response = await axios.get(
                "https://sugar-cube.onrender.com/user/userlist",
                {
                    headers: {
                        Authorization: `Bearer ${token}`,
                    },
                }
            );
            setUsers(response.data);
        } catch (error) {
            console.error("Error fetching user list:", error);
        }
    };
    fetchUsers();
```

- **User Retrieval**: The function queries the database for all users, selecting only the username, firstName, lastName, and profilePicture fields.
- **Response Handling**: If the query is successful, the user data is sent back to the client with a status code of 200. If an error occurs, a 500 status code and an error message are returned.

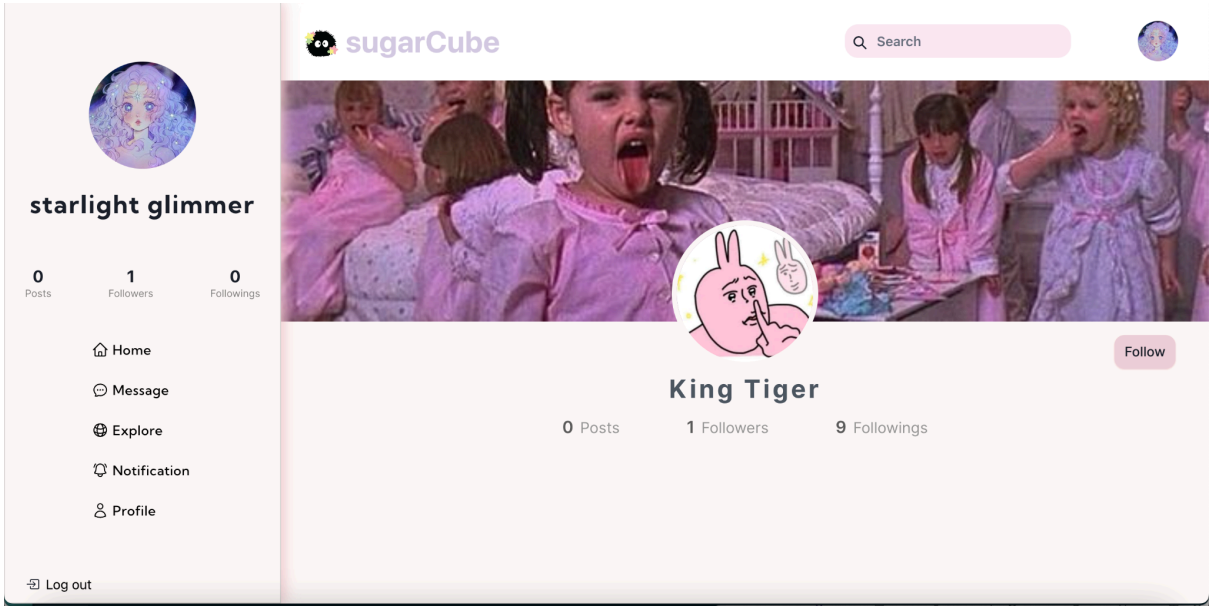
- Client-side: The SearchBar component is responsible for rendering the search bar and handling user interactions, such as inputting search queries and displaying filtered results.

```
useEffect(() => {
    // Fetch the user list from the API
    const fetchUsers = async () => {
        try {
            const token = localStorage.getItem("token");
            const response = await axios.get(
                "http://localhost:3000/user/userlist",
                {
                    headers: {
                        Authorization: `Bearer ${token}`,
                    },
                }
            );
            setUsers(response.data);
        } catch (error) {
            console.error("Error fetching user list:", error);
        }
    };
    fetchUsers();
}, []);

const handleSearch = (event) => {
    const query = event.target.value.toLowerCase();
    setQuery(query);
    const filtered = users.filter(
        (user) =>
            (user.username &&
                user.username.toLowerCase().includes(query)) ||
            (user.firstName &&
                user.firstName.toLowerCase().includes(query)) ||
            (user.lastName && user.lastName.toLowerCase().includes(query))
    );
    setFilteredUsers(filtered);
};
```

The search bar implementation involves initializing states for storing the list of users (users), filtered users (filteredUsers), and the search query (query). Upon component mount, the useEffect hook fetches the users from the server, storing them in the users state with an authorization token included in the request. The handleSearch function updates the query state and filters the users list based on the input, storing the results in filteredUsers. The focusInput function focuses the input field when the search icon is clicked, and handleClickOutside clears the search query if the user clicks outside the component. The search bar includes a SearchIcon for focusing the input field, an input field for capturing the query, and a SearchModal component that displays filtered results in a dropdown when a query is present.

2. Follow other page and delete that following page:

- Demo:



Type the username, first name, or last name of the user you are looking for in the search box in the previous section and click on that box of the user we want to follow. The server will navigate the user to that corresponding page of that user. we can see the number of followers and followings of that person. Click on the follow button and the button will become active to mark that we have followed that user.

- Implementation:
- Server-side:
    - The addFollowing api is responsible for adding users to the database. The data includes username and the id of the user we want to follow, and the server will add the id of the user we have followed to the database.

```
addFollowing = async (req, res) => {
    try {
        const user = await User.findOne({ username: req.params.username });
        const following = await User.findById(req.body.following_id);
        if (!user || !following) {
            return res
                .status(404)
                .json({ message: "User or following not found" });
        }

        user.followings.push({ following_id: req.body.following_id });

        following.followers.push({ follower_id: user._id });

        await user.save();
        await following.save();

        res.status(200).json({ message: "Following added successfully" });
    } catch (err) {
        res.status(500).json({ message: err.message });
    }
};
```

- Client-side: The ProfileUser component is responsible for rendering the page of the other user according to the username param. There is a path that leads to the page of each user according to their username, and the page will be created for each user.

```
function ProfileUser() {
    const { username } = useParams();
    console.log(username);

    return (
    <div className="min-h-screen w-full">
        <div classNam="">
        <ProfileUserHeader username={username}/>
        </div>
    </div>
    );
}

export default ProfileUser;
```

- The ProfileUserHeader component will show us the avatar, cover picture and the follow button according to which user. The component will be passed in the username of that user from the param to fetch information of that user.

```
useEffect(() => {
    const fetchIfFollow = async () => {
        const data = await fetchFollowingsData(user);
        data.forEach((item) => {
            if (item.following_id.username === username) {
                handleButtonClick();
            }
        });
    };
    const fetchUser = async () => {
        const data = await fetchUserData(username);
        setName(data.firstName+" "+data.lastName);
        console.log(data);
        setUserData(data);
    };
    const fetchDataFollowers = async () => {
        const data = await fetchFollowersData(username);
        setDataFollowers(data);
    };
    const fetchDataFollowings = async () => {
        const data = await fetchFollowingsData(username);
        setDataFollowings(data);
    };
    fetchUser();
    fetchDataFollowers();
    fetchDataFollowings();
    fetchIfFollow();

}, []);
```

- The ProfileUserHeader will fetch the information of that user which is the followers' information of that user, the followings' information of that user and some information like that user userId to pass into the addFollowing api

```
const fetchDeleteFollowings = async (user, following_id) => {
    try {
        const token = Cookies.get("token");
        const response = await fetch(`https://sugar-cube.onrender.com/user/${user}/followings`, {
            method: 'delete',
            headers: {
                Accept: 'application/json',
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${token}` // Example of adding an authorization header
            },
            body: JSON.stringify({
                following_id: following_id,
            })
        });
        if (response.ok) {
            window.location.reload;
        }
        if (!response.ok) {
            throw new Error("Network response was not ok");
        }
    } catch (error) {
        console.error("There was a problem with the fetch operation:", error);
    }
};
```

- Correspondingly, The ProfileHeader will fetch the information of that user which is the followers' information of that user, the followings' information of that user and some information like that user userId to pass into the addFollowing api

IV. DISCUSSION AND CONCLUSION

- Discussion

Developing SugarCube, a social media platform, has been a significant learning experience for both me and my teammates. Our project aimed to create a space where users can share pictures, videos, and chat with friends. Throughout this journey, we have utilized a variety of technologies and methodologies that have expanded our technical and soft skills.

- Technical Skills:
    - Frontend Development: We leveraged ReactJS and Tailwind CSS to create a responsive and dynamic user interface. ReactJS allowed us to build reusable components and manage the application state effectively. Tailwind CSS provided a utility-first approach to styling, enabling us to design a visually appealing and consistent user experience.
    - Backend Development: Using Node.js and MongoDB, we developed a robust server-side architecture. Node.js offered a scalable and efficient runtime environment for our server, while MongoDB provided a flexible and scalable NoSQL database solution. We implemented RESTful APIs to handle data transactions between the frontend and backend.
    - API Integration: Integrating various APIs, such as user authentication and data fetching, was crucial for the functionality of our application. This experience helped us understand the importance of secure and efficient API design.
- Soft Skills

    - Teamwork: Collaborating on SugarCube required effective communication, task delegation, and conflict resolution. We learned to leverage each team member's strengths and work together towards common goals.
    - Project Management: Managing our project timeline, setting milestones, and adapting to changes were essential aspects of our development process. We employed agile methodologies, such as sprint planning and daily stand-ups, to keep the project on track.
    - Despite our progress, some features, like the story functionality, remain incomplete. This highlights the importance of scope management and prioritization in project development. Nevertheless, the challenges we faced and the solutions we devised have significantly contributed to our growth as developers.

- Conclusion

In conclusion, the development of SugarCube has been an enriching experience that has provided us with invaluable insights and skills. We have gained a deeper understanding of both frontend and backend technologies, enhanced our problem-solving abilities, and improved our collaboration and project management skills. While the project is still a work in progress, the knowledge and experience we have acquired are undeniable. We are confident that with continued effort and perseverance, we will be able to complete SugarCube and deliver a fully functional social media platform. This project has not only prepared us for future technical challenges but also strengthened our ability to work effectively as a team in any professional environment.

V. REFERENCES

- https://github.com/safak/youtube.git
- https://youtu.be/_ee38nL13mE
- https://v2.tailwindcss.com/docs
- https://v2.chakra-ui.com/docs/components