Intro

The Social Media project is designed to create a fully functional social media website and database for the client Edwin Rodriguez.

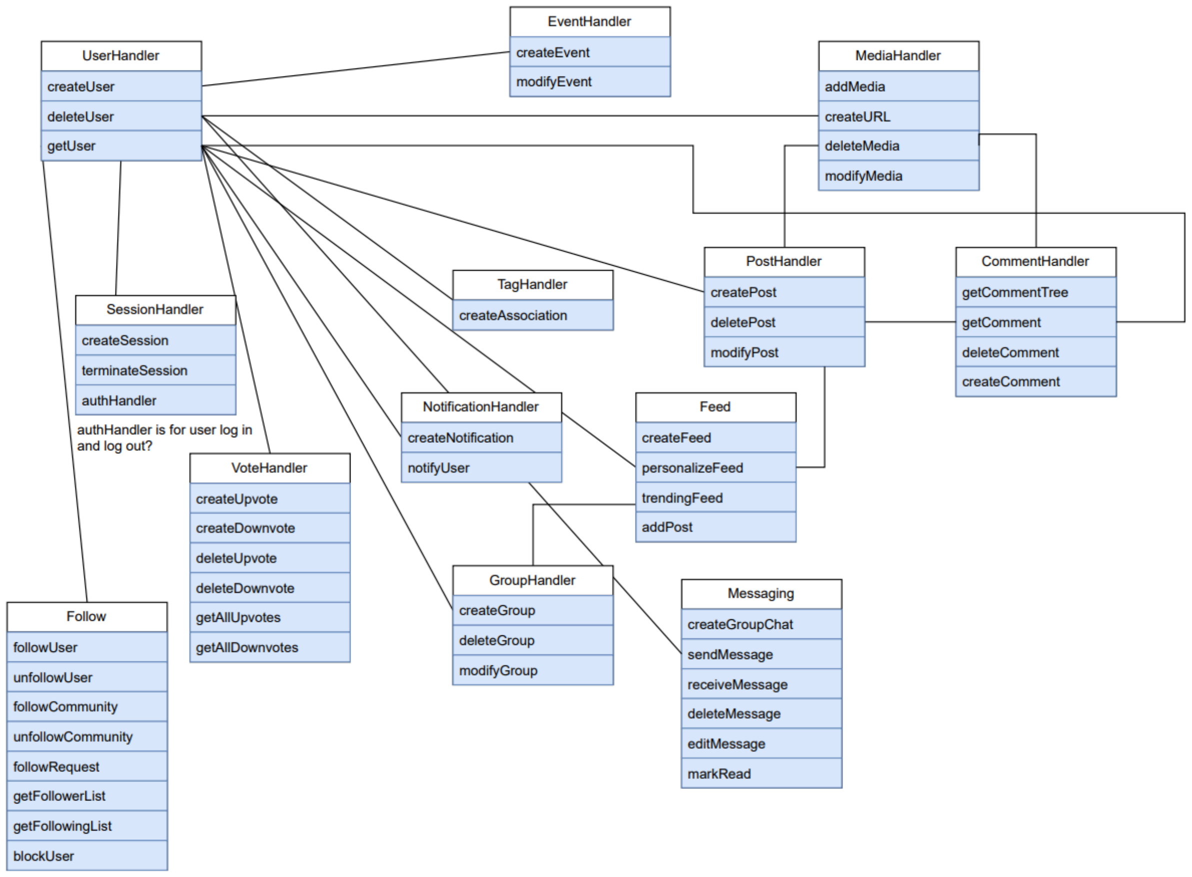
Problem Description

The client, Edwin Rodriguez, is seeking to break into the Social Media space and wants a social media website. The client wishes for the website to have features similar to other companies in the industry. These features are for users to create posts, comment on posts, like/dislike posts, create a friends and follower list, direct messaging from one user to another, and communities.

Solution Approach

To solve the problem, we used the agile method to develop and complete the social media website. We clarified what the software will do, clarified functional and nonfunctional requirements, and any concepts or goals when developing the SRS. We then implemented the features one by one in sprints to complete everything. The sprints consisted of regular face-to-face meetings and updates to ensure the objective of each sprint is completed.

Design/Architecture Description



Handler Descriptions

H1. User Handler

Purpose: The user handler is responsible for managing all aspects of user interactions and profiles. It processes user registrations, logins, and logouts, and security with authentication through password management and verification. The user handler also manages profile creation and updates, allowing users to edit their information, upload profile pictures, and customize settings. It also manages user preferences, such as privacy settings and notification controls, ensuring that users have a personalized experience. The user handler processes interactions such as sending friend requests, blocking users, and reporting inappropriate behavior. It may also retrieve and display user specific data, like posts, likes, and comments, to create a tailored feed. By efficiently managing these aspects, the user handler enhances user engagement, ensures security, and fosters a positive community environment within the app.

H2. Session Handler

Purpose: The session handler is essential for managing user sessions and enhancing the overall user experience. It customizes session handling beyond Flasks default functionality by creating, maintaining, and terminating user sessions, allowing the app to track interactions such as likes, comments, and messages over time. The session handler stores user specific data, including profile information, friend lists, and activity history, ensuring that this information remains accessible across multiple visits. Security is also an important function for the session handler because it implements measures like generating unique session IDs, managing session expiration, and protecting against session hijacking to protect user data. Additionally, it facilitates efficient storage of session data whether in memory and in a database optimizing performance by reducing the need for repetitive data fetching. Overall, the session handler has an important part in creating a secure experience for users, allowing them to interact with the app.

H3. Follow Handler

Purpose: The follow handler is responsible for managing user relationships, but mainly handles the actions related to following and unfollowing other users. It processes follow and unfollow requests, updating the database to show changes in user relationships, and notifies users when someone follows or unfollows them. The follow handler also manages the retrieval of follower and following lists, allowing users to view who they follow and who follows them. Additionally, it implements features such as mutual follow notifications, suggestions for new accounts to follow based on interests, and privacy settings that control who can follow whom. By efficiently handling these interactions, the follow handler plays a critical role in enhancing user engagement and building a connected community within the app.

H4. Tag Handler

Purpose: The tag handler is responsible for managing the creation, association, and retrieval of tags within user content. It allows users to create tags when posting content, such as photos or status updates, which helps categorize and allow discoverability. The tag handler processes requests to add or remove tags from posts, making sure that the associations are correctly updated in the database. It also handles the retrieval of content based on tags, enabling users to search for posts related to specific interests or topics. The tag handler may also implement features like trending tags, tag suggestions, and user notifications when content is tagged, improving user engagement. By efficiently managing tags, the tag handler plays a crucial role in organizing content, improving user experience, and improving community interaction within the website.

H5. Media Handler

Purpose: The media handler is responsible for managing all aspects of media content, including images, videos, and audio files. It processes uploads, and stores media files to be optimized for performance. The media handler often handles file compression. The media handler also manages the retrieval and display of media content in user feeds, profiles, and messages, improving the experience for users. The media handler may also implement features like tagging media, adding captions, and managing privacy settings for shared content. It can also support content moderation by flagging or removing inappropriate media based on community guidelines. By efficiently managing media, the media handler enhances user engagement and enriches the overall content experience within the app.

H6. Comment Handler

Purpose: The comment handler is responsible for managing all aspects of user comments on posts, photos, and videos. Its primary functions include processing the submission of new comments, updating the database, and ensuring that comments are displayed correctly beneath the relevant content. The comment handler also facilitates editing and deleting comments, allowing users to manage their contributions easily. It also implements features such as nested replies, which enable users to engage in discussions, and notifications that alert users when someone replies to their comments. The comment handler also includes moderation tools to flag or remove inappropriate comments. By effectively managing comments, the comment handler can improve community interaction, enhance user engagement, and contribute to meaningful conversations within the website.

H7. Feed Handler

Purpose: The feed handler is responsible for managing the content displayed in a user's feed, ensuring that it is relevant, engaging, and timely. Its primary functions include retrieving and organizing posts from friends, followed accounts, and suggested content based on user interests and interactions. The feed handler utilizes algorithms to prioritize which posts appear first, taking into account factors like engagement levels, recency, and user preferences.

It also processes user interactions with the feed, such as likes, shares, and comments, updating the display in real time to reflect these activities. The feed handler may also manage content filtering options, allowing users to customize what they see based on categories, topics, or specific users. By efficiently curating and delivering content, the feed handler enhances user engagement, promotes meaningful interactions, and contributes to a personalized experience within the app.

H8. Event Handler

Purpose: The event handler is responsible for managing user interactions and system events, ensuring that the application responds appropriately to various actions. Its primary functions include capturing events such as clicks, swipes, or keyboard inputs and triggering the corresponding responses, like opening menus, submitting forms, or navigating to different pages. The event handler also manages real-time notifications, ensuring that users receive alerts for messages, friend requests, or updates without needing to refresh the app. Additionally, it can handle custom events, such as user-generated actions like tagging friends in posts or creating events, and ensure that these interactions are processed smoothly.

H9. Notification Handler

Purpose: The notification handler is responsible for managing user notifications, ensuring that users are promptly informed about relevant activities and interactions. Its primary functions include generating notifications for events such as new friend requests, likes, comments, messages, and mentions. The notification handler processes these events in real time, allowing users to stay updated without needing to refresh the app. Additionally, it manages the display of notifications, organizing them in a way that prioritizes the most relevant or recent activities. Users can often customize their notification preferences through the handler, deciding which types of notifications they want to receive and how they want to be alerted. By effectively managing notifications, the notification handler enhances user engagement, promotes interaction, and ensures that users feel connected to their social network within the website.

H10. Group Handler

Purpose: The group handler is responsible for managing all aspects of user-created groups or communities. Its primary functions include facilitating the creation of new groups, allowing users to set privacy settings (public, private, or secret), and managing group membership by processing join and leave requests. The group handler also oversees group interactions, such as posting updates, sharing media, and organizing events, ensuring that all activities are properly tracked and displayed in the group feed. Additionally, it may implement moderation tools that allow group admins to manage content, enforce guidelines, and handle member disputes.

H11. Vote Handler

Purpose: The vote handler is responsible for managing user interactions related to voting on various content, such as polls, questions, or comments. Its primary functions include processing vote submissions, updating the database to reflect the current vote counts, and ensuring that results are displayed accurately in real time. The vote handler also manages features like limiting users to one vote per item, allowing users to change their votes, and potentially implementing voting periods for time-sensitive polls. Additionally, it may track and display user participation in voting activities, providing insights into how many users engaged with a particular poll or question.

H12. Messaging Handler

Purpose: The messaging handler is responsible for managing all aspects of user messaging and communication features. Its primary functions include facilitating the sending and receiving of direct messages between users, ensuring that messages are delivered in real time and displayed in a user-friendly interface. The messaging handler also handles features such as group chats, message notifications, and message threading, allowing users to engage in multiple conversations seamlessly. It may implement security measures, such as end-to-end encryption, to protect user privacy and ensure that messages remain confidential.

H13. Post Handler

Purpose: The post handler is responsible for managing all aspects of user-generated content, including creating, editing, deleting, and displaying posts. Its primary functions include processing new posts submitted by users, ensuring that content is properly formatted and stored in the database, and facilitating any associated media uploads, such as images or videos. The post handler also manages interactions related to posts, such as likes, shares, comments, and tagging other users. It ensures that these interactions are updated in real time and reflected in users' feeds and notifications. Additionally, it may implement moderation features to flag or remove inappropriate content, ensuring that community guidelines are upheld.

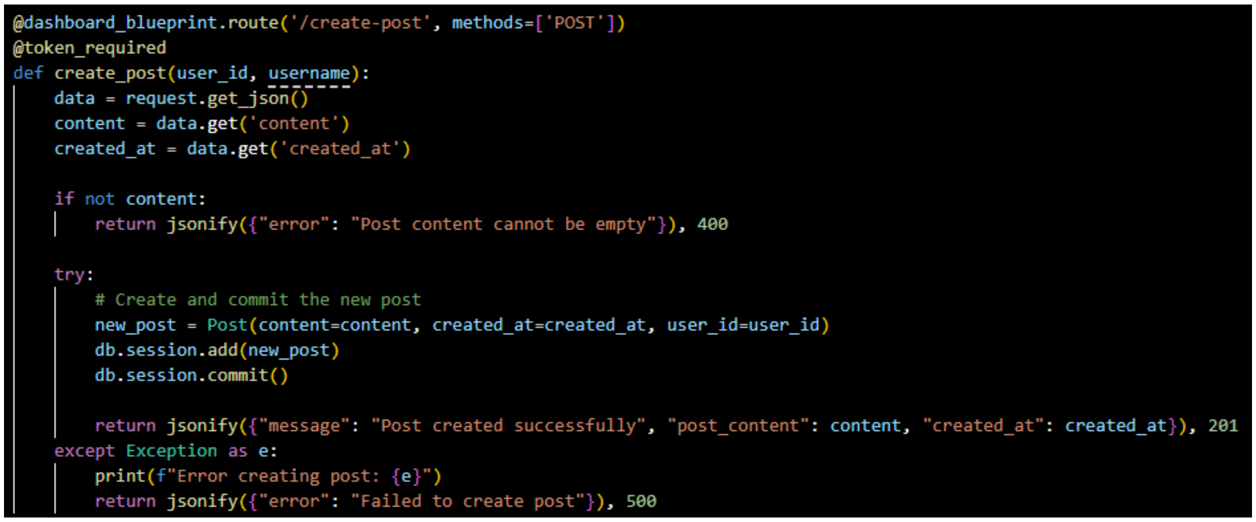
Implementation Discussion

The implementation of the Horizon social media platform was structured to ensure continuous functionality and scalability. The development process followed agile methodologies, breaking down the implementation into focused sprints, each dedicated to specific features and components.

Backend Implementation

The backend is built using Flask, a lightweight Python framework, ensuring modularity and scalability. It comprises several endpoints designed to handle specific user interactions and system operations. Below are some of the key methods, supported by SQLAlchemy ORM for database management:

Creating a Post
/create-post
Method: POST
Description: Creates a new post for the logged-in user.

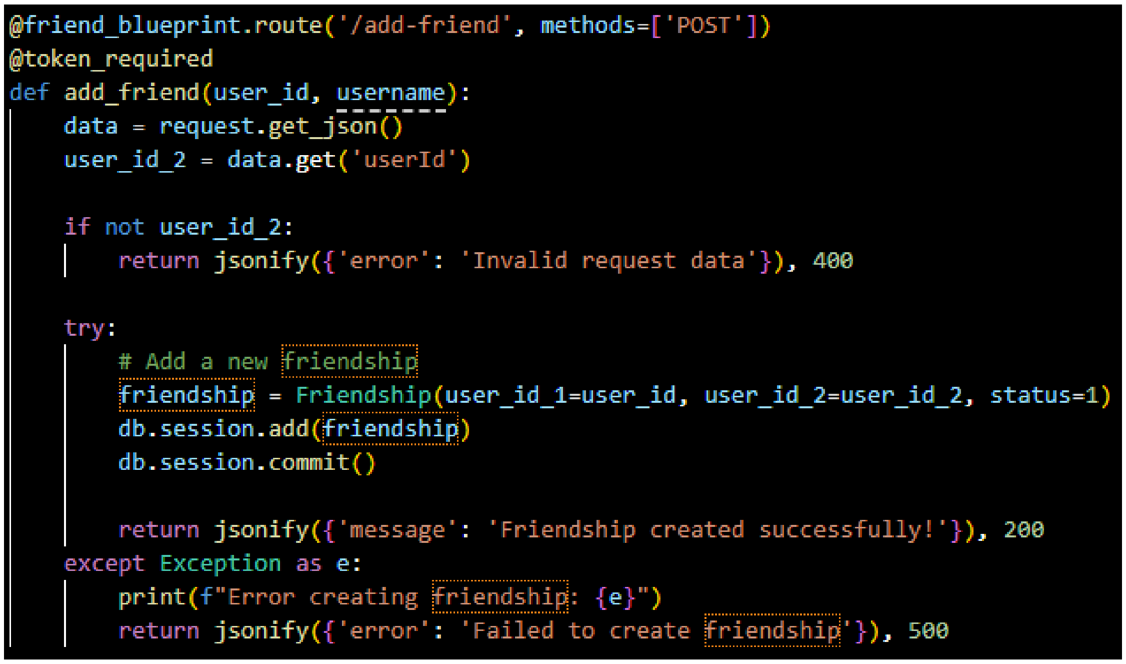


```python
@dashboard_blueprint.route('/create-post', methods=['POST'])
@token_required
def create_post(user_id, username):
    data = request.get_json()
    content = data.get('content')
    created_at = data.get('created_at')

    if not content:
        return jsonify({"error": "Post content cannot be empty"}), 400

    try:
        # Create and commit the new post
        new_post = Post(content=content, created_at=created_at, user_id=user_id)
        db.session.add(new_post)
        db.session.commit()

        return jsonify({"message": "Post created successfully", "post_content": content, "created_at": created_at}), 201
    except Exception as e:
        print(f"Error creating post: {e}")
        return jsonify({"error": "Failed to create post"}), 500
```

Explanation: Validates the presence of the content field. Creates a new Post object linked to the logged-in user's ID using token_required for authentication. Saves the post to the database with a timestamp.

Output example:
```
{
   "message": "Post created successfully",
   "post_content": "Hello, world!",
   "created_at": "2024-11-22T10:00:00Z"
}
```

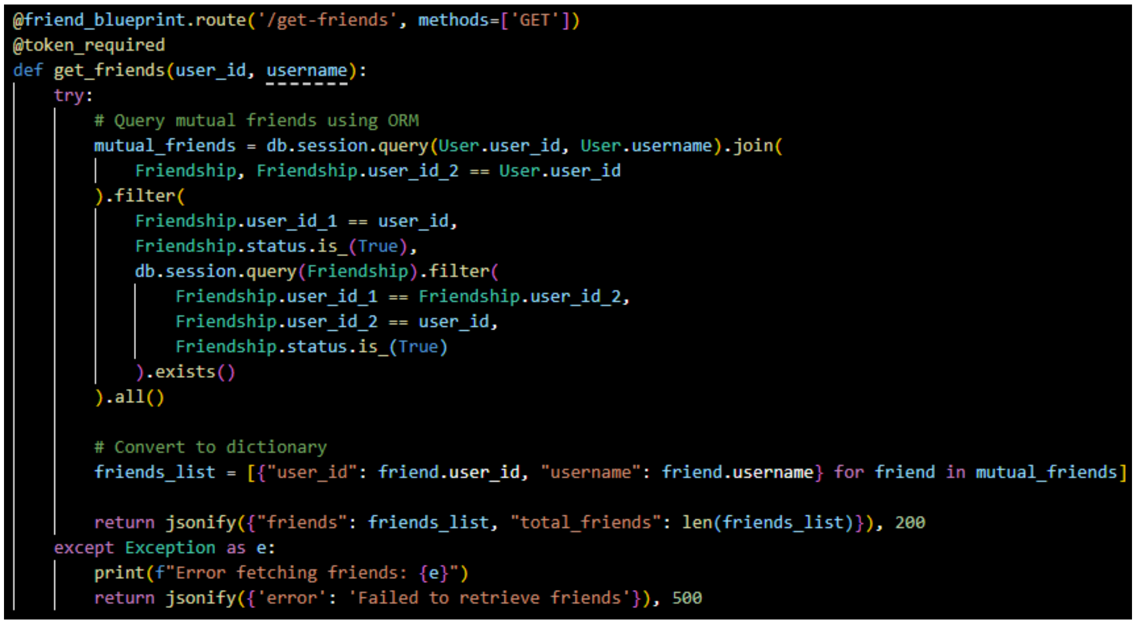
Adding a friend
/add-friend
Method: POST
Description: Creates a new friendship between two users.



```python
@friend_blueprint.route('/add-friend', methods=['POST'])
@token_required
def add_friend(user_id, username):
    data = request.get_json()
    user_id_2 = data.get('userId')

    if not user_id_2:
        return jsonify({'error': 'Invalid request data'}), 400

    try:
        # Add a new friendship
        friendship = Friendship(user_id_1=user_id, user_id_2=user_id_2, status=1)
        db.session.add(friendship)
        db.session.commit()

        return jsonify({'message': 'Friendship created successfully!'}), 200
    except Exception as e:
        print(f"Error creating friendship: {e}")
        return jsonify({'error': 'Failed to create friendship'}), 500
```

Explanation: Checks for valid userId in the request body. Inserts a new friendship record into the Friendship table, marking the relationship as active (status=1).

Returning Mutual Friends
/get-friends
Method: GET
Description: Fetches the list of mutual friends for the logged-in user.

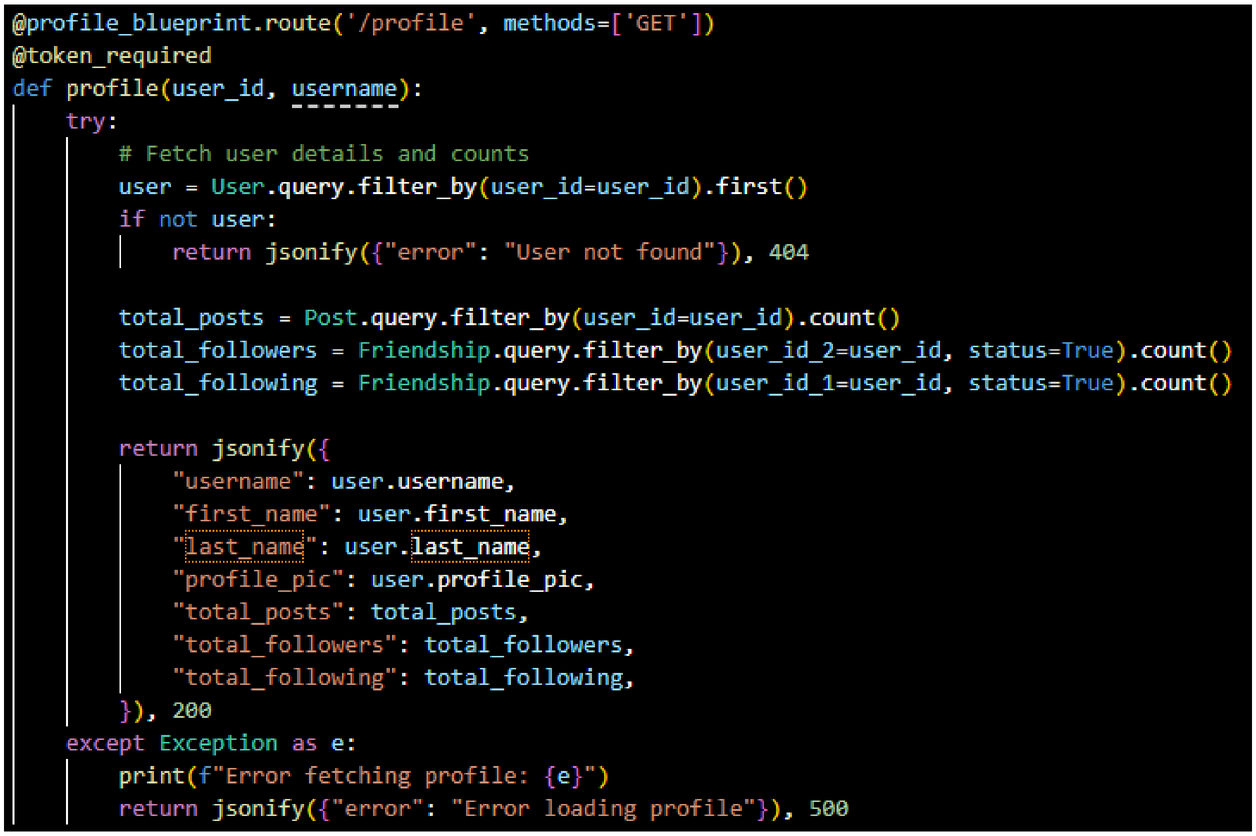


```python
@friend_blueprint.route('/get-friends', methods=['GET'])
@token_required
def get_friends(user_id, username):
    try:
        # Query mutual friends using ORM
        mutual_friends = db.session.query(User.user_id, User.username).join(
            Friendship, Friendship.user_id_2 == User.user_id
        ).filter(
            Friendship.user_id_1 == user_id,
            Friendship.status.is_(True),
            db.session.query(Friendship).filter(
                Friendship.user_id_1 == Friendship.user_id_2,
                Friendship.user_id_2 == user_id,
                Friendship.status.is_(True)
            ).exists()
        ).all()

        # Convert to dictionary
        friends_list = [{"user_id": friend.user_id, "username": friend.username} for friend in mutual_friends]

        return jsonify({"friends": friends_list, "total_friends": len(friends_list)}), 200
    except Exception as e:
        print(f"Error fetching friends: {e}")
        return jsonify({'error': 'Failed to retrieve friends'}), 500
```

Explanation: Fetches mutual friends for a logged-in user by querying the Friendship table using SQLAlchemy, filtering for active friendships where both users are connected. The query performs two joins to check for mutual connections and returns the results as a list of dictionaries containing each mutual friend's user_id and username. The data is returned as a JSON response with the list of friends and the total count

Example output:
```
{
   "friends": [
      {"user_id": 2, "username": "alice_smith"},
      {"user_id": 3, "username": "bob_johnson"}
   ],
   "total_friends": 2
}
```

Get Profile
/profile
Method: GET
Description: Retrieves profile details for the logged-in user, including counts of posts, followers, following, and friends.

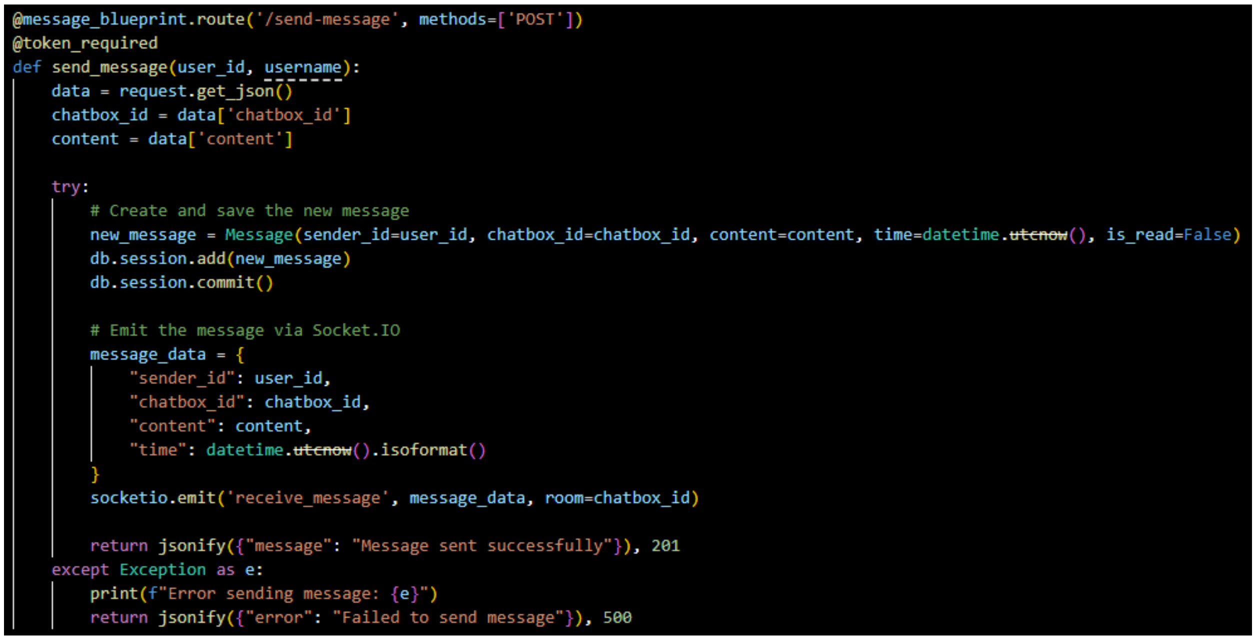


```python
@profile_blueprint.route('/profile', methods=['GET'])
@token_required
def profile(user_id, username):
    try:
        # Fetch user details and counts
        user = User.query.filter_by(user_id=user_id).first()
        if not user:
            return jsonify({"error": "User not found"}), 404

        total_posts = Post.query.filter_by(user_id=user_id).count()
        total_followers = Friendship.query.filter_by(user_id_2=user_id, status=True).count()
        total_following = Friendship.query.filter_by(user_id_1=user_id, status=True).count()

        return jsonify({
            "username": user.username,
            "first_name": user.first_name,
            "last_name": user.last_name,
            "profile_pic": user.profile_pic,
            "total_posts": total_posts,
            "total_followers": total_followers,
            "total_following": total_following,
        }), 200
    except Exception as e:
        print(f"Error fetching profile: {e}")
        return jsonify({"error": "Error loading profile"}), 500
```

Explanation: Queries the database for user details and related counts. Returns aggregated data such as total posts, followers, and following.

Output example:
```
{
   "username": "john_doe",
   "first_name": "John",
   "last_name": "Doe",
   "profile_pic": "/images/john_doe.jpg",
   "total_posts": 15,
   "total_followers": 50,
   "total_following": 30
}
```

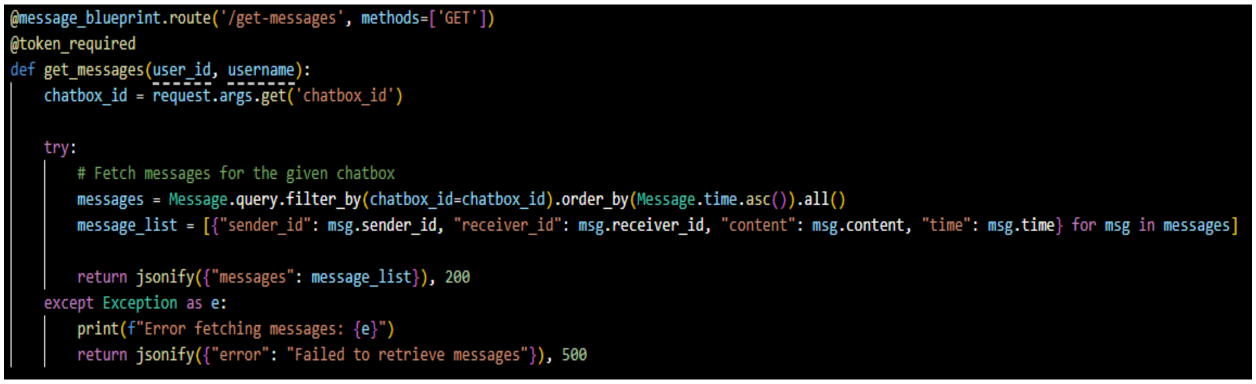
Sending Message
/send-message
Method: POST
Description: Sends a message in a chatbox and emits it to the chat room.



```python
@message_blueprint.route('/send-message', methods=['POST'])
@token_required
def send_message(user_id, username):
    data = request.get_json()
    chatbox_id = data['chatbox_id']
    content = data['content']

    try:
        # Create and save the new message
        new_message = Message(sender_id=user_id, chatbox_id=chatbox_id, content=content, time=datetime.~~utcnow~~(), is_read=False)
        db.session.add(new_message)
        db.session.commit()

        # Emit the message via Socket.IO
        message_data = {
            "sender_id": user_id,
            "chatbox_id": chatbox_id,
            "content": content,
            "time": datetime.~~utcnow~~().isoformat()
        }
        socketio.emit('receive_message', message_data, room=chatbox_id)

        return jsonify({"message": "Message sent successfully"}), 201
    except Exception as e:
        print(f"Error sending message: {e}")
        return jsonify({"error": "Failed to send message"}), 500
```

Explanation: Inserts the message into the Message table with metadata like chatbox_id and timestamp (time). Uses Flask-SocketIO to broadcast the message to users in the same chatroom (room=chatbox_id).

Get Chatbox Messages
/get-messages
Method: GET
Description: Fetches all messages for a specific chatbox

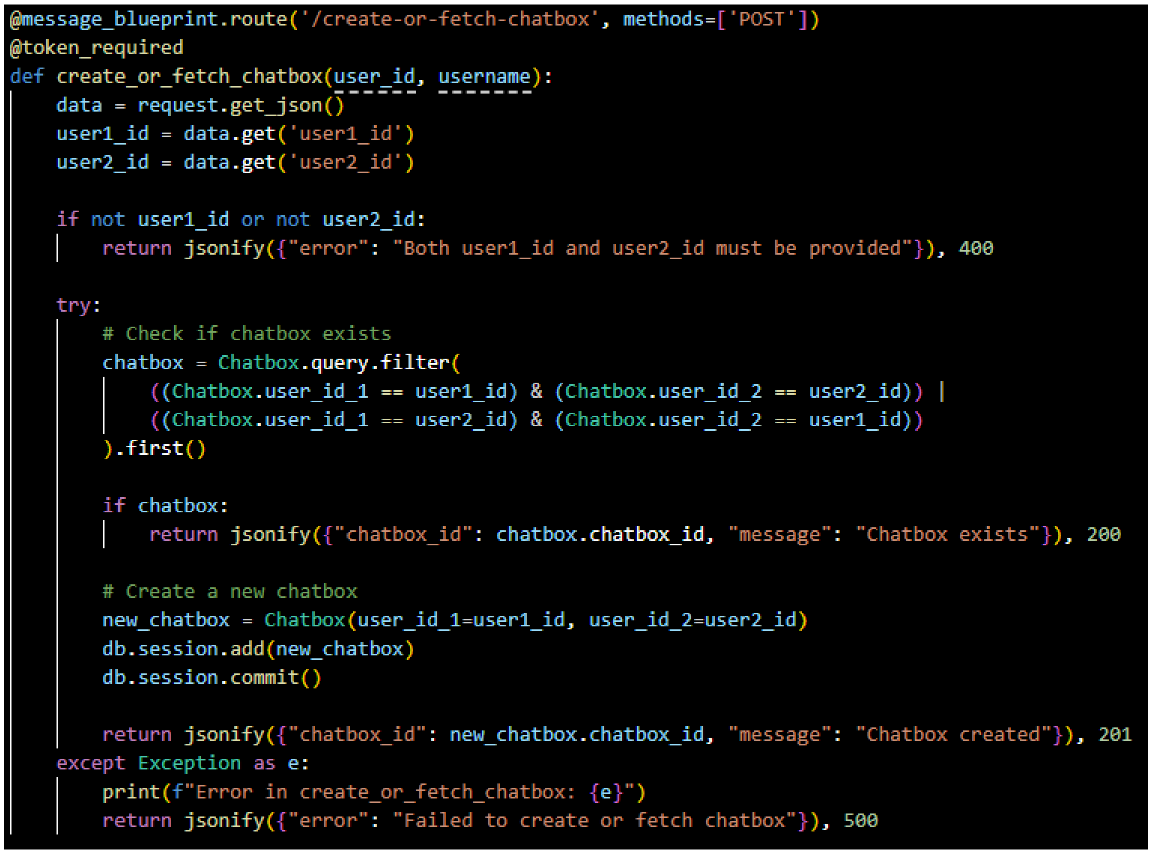


```python
@message_blueprint.route('/get-messages', methods=['GET'])
@token_required
def get_messages(user_id, username):
    chatbox_id = request.args.get('chatbox_id')

    try:
        # Fetch messages for the given chatbox
        messages = Message.query.filter_by(chatbox_id=chatbox_id).order_by(Message.time.asc()).all()
        message_list = [{"sender_id": msg.sender_id, "receiver_id": msg.receiver_id, "content": msg.content, "time": msg.time} for msg in messages]

        return jsonify({"messages": message_list}), 200
    except Exception as e:
        print(f"Error fetching messages: {e}")
        return jsonify({"error": "Failed to retrieve messages"}), 500
```

Explanation: Fetches all messages for a specific chatbox, ordered by time. It retrieves the chatbox_id from the request arguments and queries the Message table for all messages related to that chatbox, ordering them by Message.time. The result is returned as a JSON list containing each message's sender ID, receiver ID, content, and timestamp.
Example output:

```
{
   "messages": [
      {
         "sender_id": 1,
         "receiver_id": 2,
         "content": "Hey, how's it going?",
         "time": "2024-11-22T12:00:00"
      },
      {
         "sender_id": 2,
         "receiver_id": 1,
         "content": "Terrible! How about you?",
         "time": "2024-11-22T12:05:00"
      }
   ]
}
```

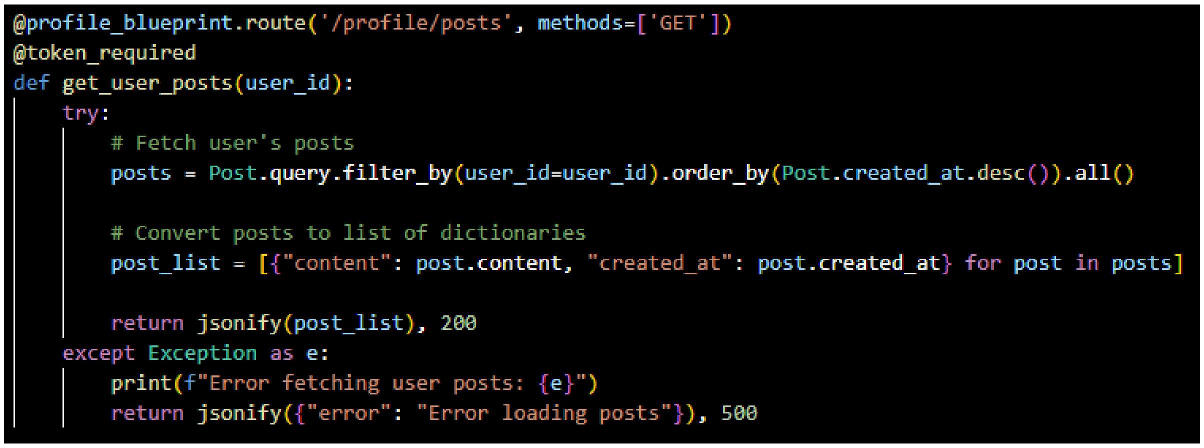
Creating a chat box
/create-or-fetch-catchbox
Method: POST
Description: Fetches an existing chatbox or creates a new one if it doesn't exist.



```
@message_blueprint.route('/create-or-fetch-chatbox', methods=['POST'])
@token_required
def create_or_fetch_chatbox(user_id, username):
    data = request.get_json()
    user1_id = data.get('user1_id')
    user2_id = data.get('user2_id')

    if not user1_id or not user2_id:
        return jsonify({"error": "Both user1_id and user2_id must be provided"}), 400

    try:
        # Check if chatbox exists
        chatbox = Chatbox.query.filter(
            ((Chatbox.user_id_1 == user1_id) & (Chatbox.user_id_2 == user2_id)) |
            ((Chatbox.user_id_1 == user2_id) & (Chatbox.user_id_2 == user1_id))
        ).first()

        if chatbox:
            return jsonify({"chatbox_id": chatbox.chatbox_id, "message": "Chatbox exists"}), 200

        # Create a new chatbox
        new_chatbox = Chatbox(user_id_1=user1_id, user_id_2=user2_id)
        db.session.add(new_chatbox)
        db.session.commit()

        return jsonify({"chatbox_id": new_chatbox.chatbox_id, "message": "Chatbox created"}), 201
    except Exception as e:
        print(f"Error in create_or_fetch_chatbox: {e}")
        return jsonify({"error": "Failed to create or fetch chatbox"}), 500
```

Explanation: first checks if both user1_id and user2_id are provided, and then queries the database to see if a chatbox already exists between the users. If it does, it returns the existing chatbox's ID; otherwise, it creates a new chatbox, adds it to the database, and returns the new chatbox ID.

Get Profiles posts
/profile/posts
Method: GET
Description: Retrieves all posts created by the logged-in user



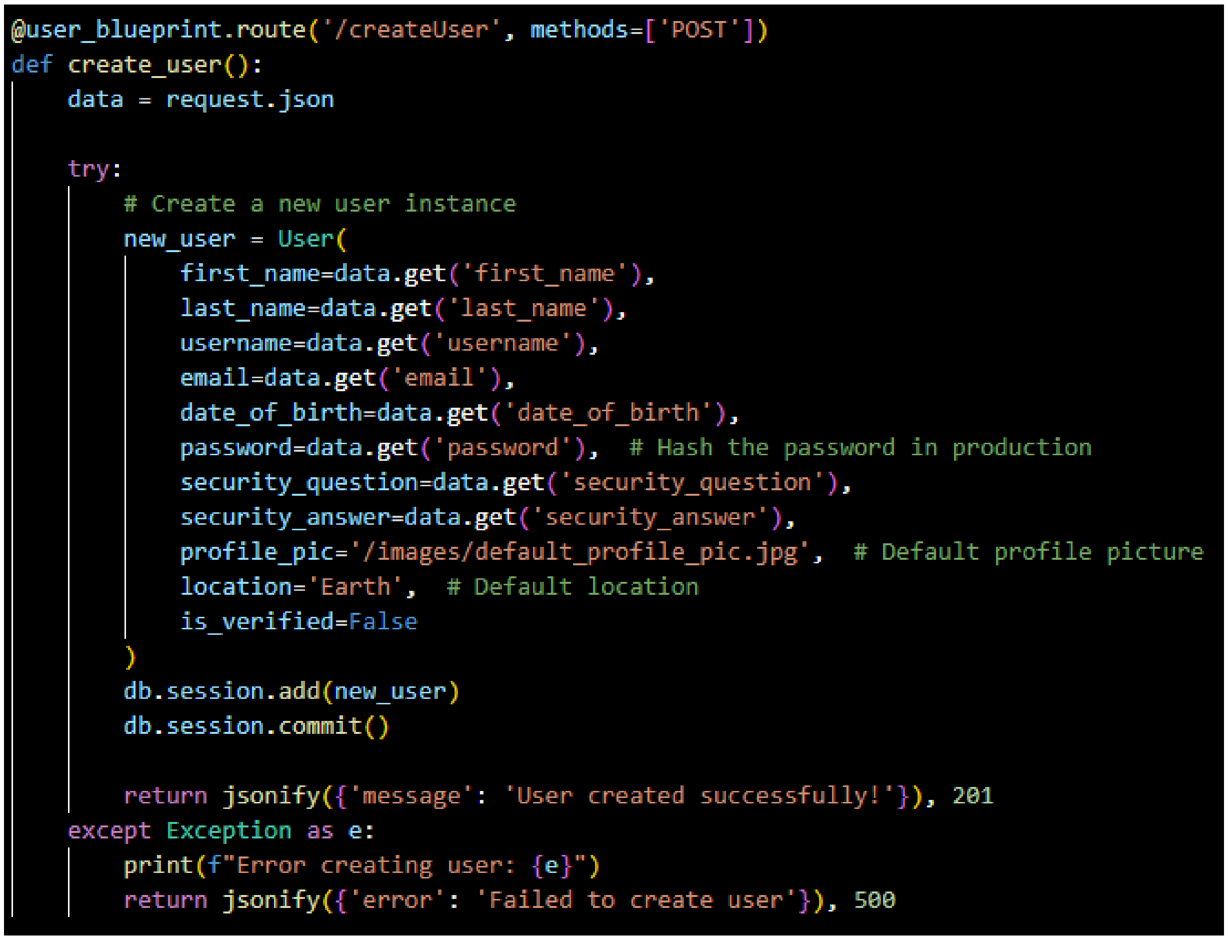
```python
@profile_blueprint.route('/profile/posts', methods=['GET'])
@token_required
def get_user_posts(user_id):
    try:
        # Fetch user's posts
        posts = Post.query.filter_by(user_id=user_id).order_by(Post.created_at.desc()).all()

        # Convert posts to list of dictionaries
        post_list = [{"content": post.content, "created_at": post.created_at} for post in posts]

        return jsonify(post_list), 200
    except Exception as e:
        print(f"Error fetching user posts: {e}")
        return jsonify({"error": "Error loading posts"}), 500
```

Explanation: Fetches posts for the user from the database. Formats the response with content and created_at.

Output Example:

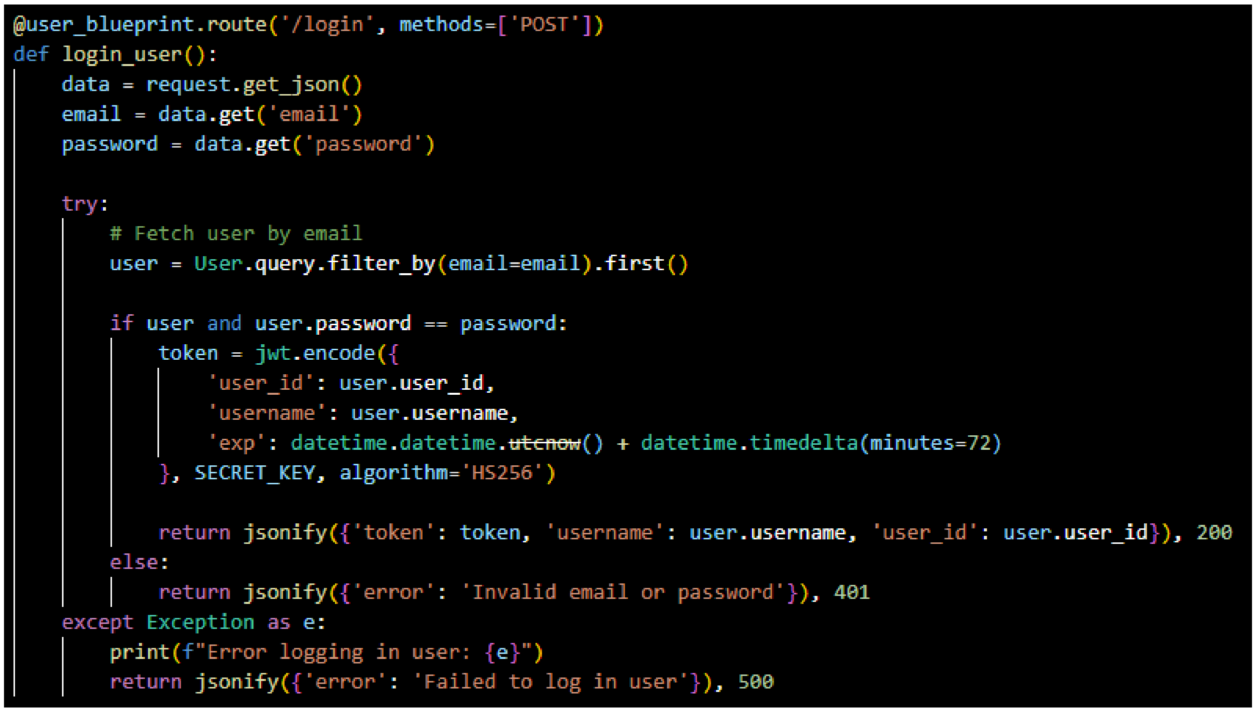
```
[
   {"content": "Third post!", "created_at": "2024-11-21T10:00:00Z"},
   {"content": "Another post", "created_at": "2024-11-20T12:00:00Z"}
]
```

New User Creation
/createUser
Method: Post
Description: Creates a new user account with the provided information.



```python
@user_blueprint.route('/createUser', methods=['POST'])
def create_user():
    data = request.json

    try:
        # Create a new user instance
        new_user = User(
            first_name=data.get('first_name'),
            last_name=data.get('last_name'),
            username=data.get('username'),
            email=data.get('email'),
            date_of_birth=data.get('date_of_birth'),
            password=data.get('password'),  # Hash the password in production
            security_question=data.get('security_question'),
            security_answer=data.get('security_answer'),
            profile_pic='/images/default_profile_pic.jpg',  # Default profile picture
            location='Earth',  # Default location
            is_verified=False
        )
        db.session.add(new_user)
        db.session.commit()

        return jsonify({'message': 'User created successfully!'}), 201
    except Exception as e:
        print(f"Error creating user: {e}")
        return jsonify({'error': 'Failed to create user'}), 500
```

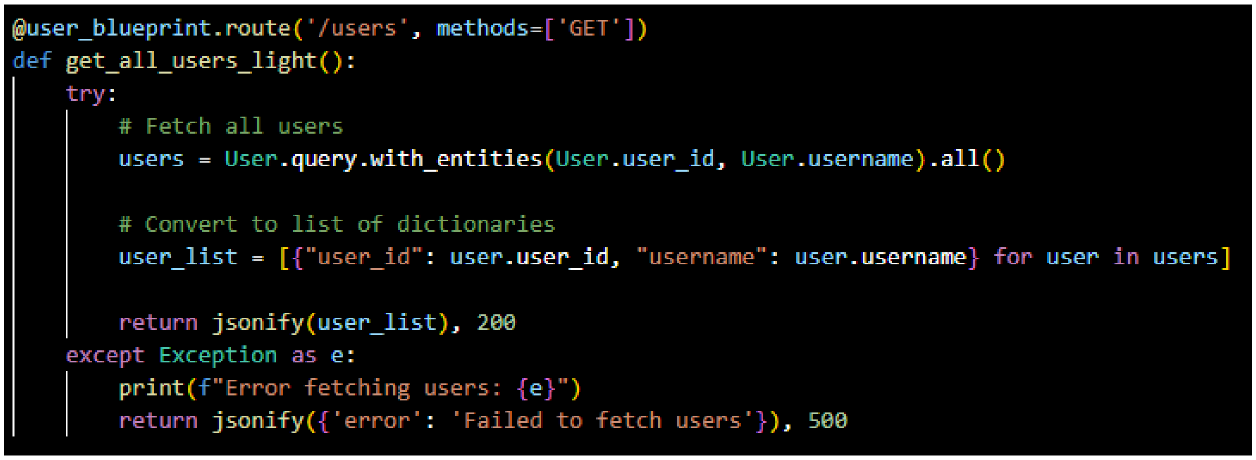
Logging in
/login
Method: Post
Description: Authenticates a user and generates a JWT token for session management.



```
@user_blueprint.route('/login', methods=['POST'])
def login_user():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')

    try:
        # Fetch user by email
        user = User.query.filter_by(email=email).first()

        if user and user.password == password:
            token = jwt.encode({
                'user_id': user.user_id,
                'username': user.username,
                'exp': datetime.datetime.~~utcnow~~() + datetime.timedelta(minutes=72)
            }, SECRET_KEY, algorithm='HS256')

            return jsonify({'token': token, 'username': user.username, 'user_id': user.user_id}), 200
        else:
            return jsonify({'error': 'Invalid email or password'}), 401
    except Exception as e:
        print(f"Error logging in user: {e}")
        return jsonify({'error': 'Failed to log in user'}), 500
```

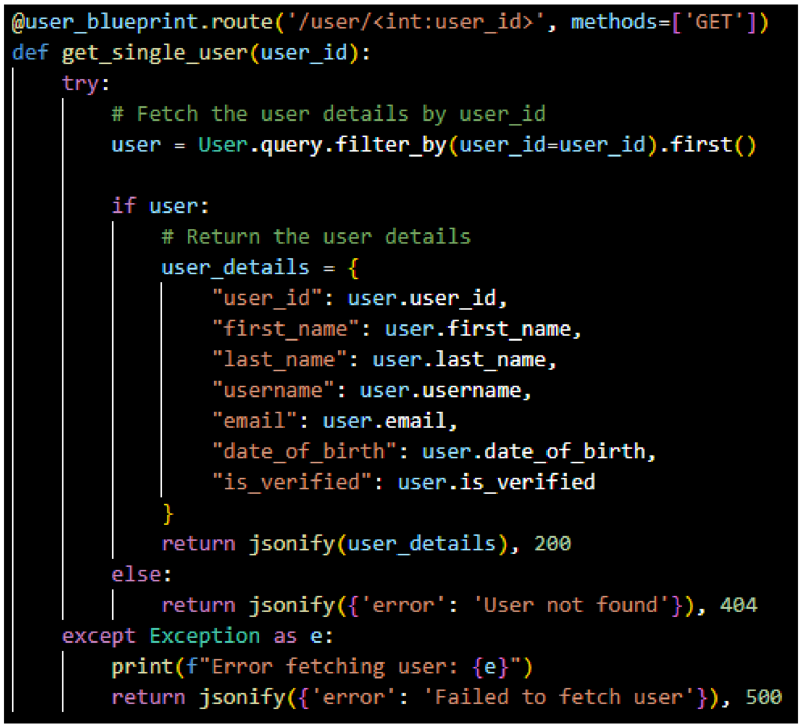
Explanation: The backend receives the email and password from the frontend request. We try to fetch a user from the database with the same email, which is unique among all users. If the user and the password of the user match with the request, the user is logged in and a token is created with the user_id, username, time created, a secret key, and an encryption algorithm. If the password or email does not match, an error will be returned.

/users
Method: GET
Description: Retrieves a list of all users with their IDs and usernames.



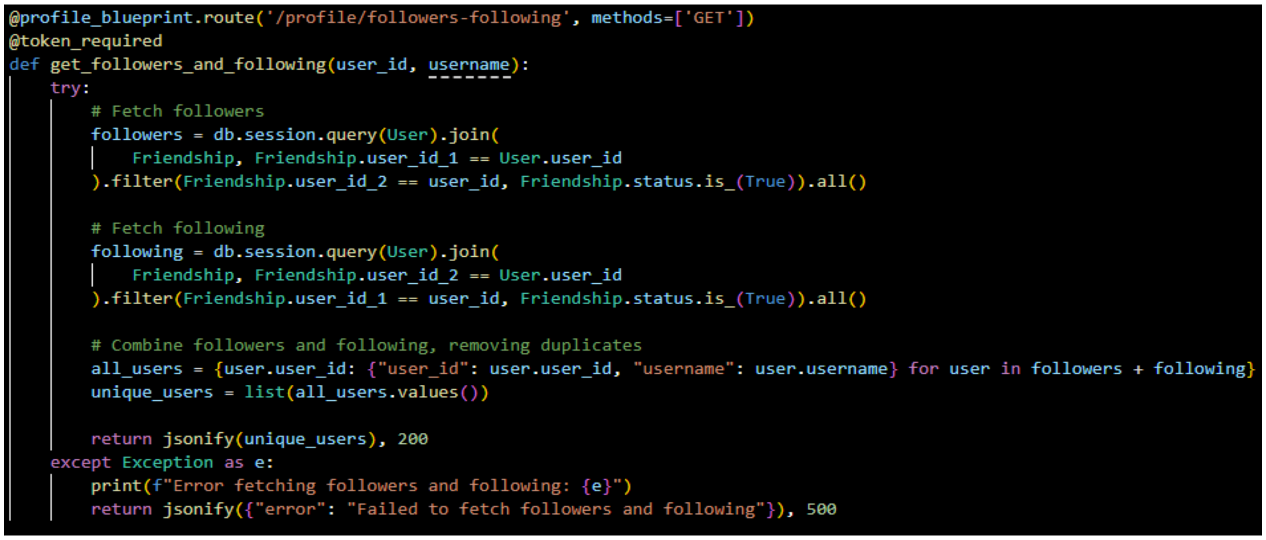
```
@user_blueprint.route('/users', methods=['GET'])
def get_all_users_light():
    try:
        # Fetch all users
        users = User.query.with_entities(User.user_id, User.username).all()

        # Convert to list of dictionaries
        user_list = [{"user_id": user.user_id, "username": user.username} for user in users]

        return jsonify(user_list), 200
    except Exception as e:
        print(f"Error fetching users: {e}")
        return jsonify({'error': 'Failed to fetch users'}), 500
```

/user/<int:user_id>
Method: GET
Description: Fetches detailed information for a specific user by user_id.



```python
@user_blueprint.route('/user/<int:user_id>', methods=['GET'])
def get_single_user(user_id):
    try:
        # Fetch the user details by user_id
        user = User.query.filter_by(user_id=user_id).first()

        if user:
            # Return the user details
            user_details = {
                "user_id": user.user_id,
                "first_name": user.first_name,
                "last_name": user.last_name,
                "username": user.username,
                "email": user.email,
                "date_of_birth": user.date_of_birth,
                "is_verified": user.is_verified
            }
            return jsonify(user_details), 200
        else:
            return jsonify({'error': 'User not found'}), 404
    except Exception as e:
        print(f"Error fetching user: {e}")
        return jsonify({'error': 'Failed to fetch user'}), 500
```

Get friends
/profile/followers-following
Method: GET
Description: Fetches a combined list of followers for the logged-in user.



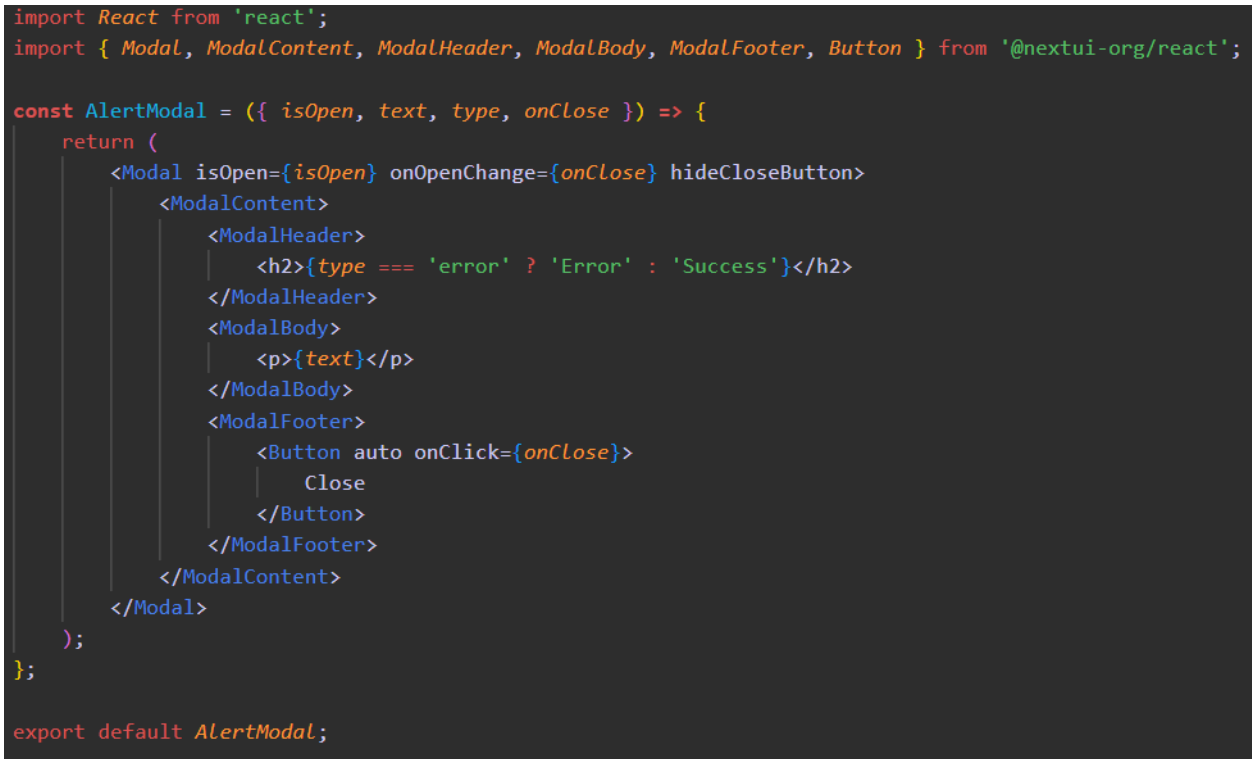
```python
@profile_blueprint.route('/profile/followers-following', methods=['GET'])
@token_required
def get_followers_and_following(user_id, username):
    try:
        # Fetch followers
        followers = db.session.query(User).join(
            Friendship, Friendship.user_id_1 == User.user_id
        ).filter(Friendship.user_id_2 == user_id, Friendship.status.is_(True)).all()

        # Fetch following
        following = db.session.query(User).join(
            Friendship, Friendship.user_id_2 == User.user_id
        ).filter(Friendship.user_id_1 == user_id, Friendship.status.is_(True)).all()

        # Combine followers and following, removing duplicates
        all_users = {user.user_id: {"user_id": user.user_id, "username": user.username} for user in followers + following}
        unique_users = list(all_users.values())

        return jsonify(unique_users), 200
    except Exception as e:
        print(f"Error fetching followers and following: {e}")
        return jsonify({"error": "Failed to fetch followers and following"}), 500
```

Frontend Implementation

The front end is developed using React, a JavaScript library that is great for building user interfaces. It has a component based architecture will break down each user interaction into smaller and reusable components to increase modularity:

/assets/components/AlertModal.js

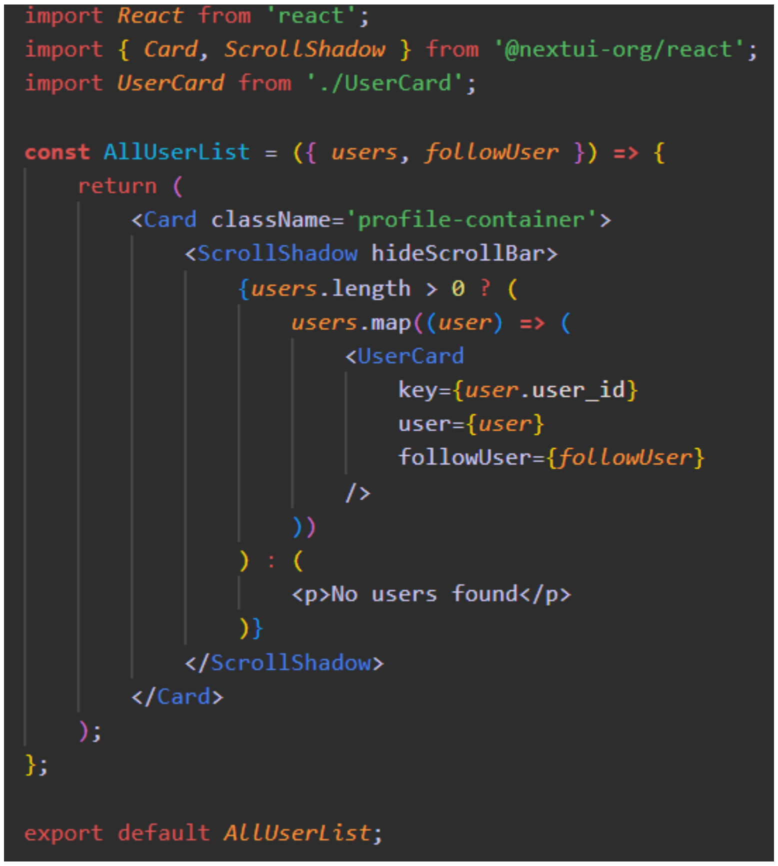
Description: Displays a modal based on its type: displaying error if type is error and success otherwise. Modal will close when the close button or backdrop is pressed.
Methods: isOpen



```
import React from 'react';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button } from '@nextui-org/react';

const AlertModal = ({ isOpen, text, type, onClose }) => {
    return (
        <Modal isOpen={isOpen} onOpenChange={onClose} hideCloseButton>
            <ModalContent>
                <ModalHeader>
                    <h2>{type === 'error' ? 'Error' : 'Success'}</h2>
                </ModalHeader>
                <ModalBody>
                    <p>{text}</p>
                </ModalBody>
                <ModalFooter>
                    <Button auto onClick={onClose}>
                        Close
                    </Button>
                </ModalFooter>
            </ModalContent>
        </Modal>
    );
};

export default AlertModal;
```

/assets/components/AllUserList.js

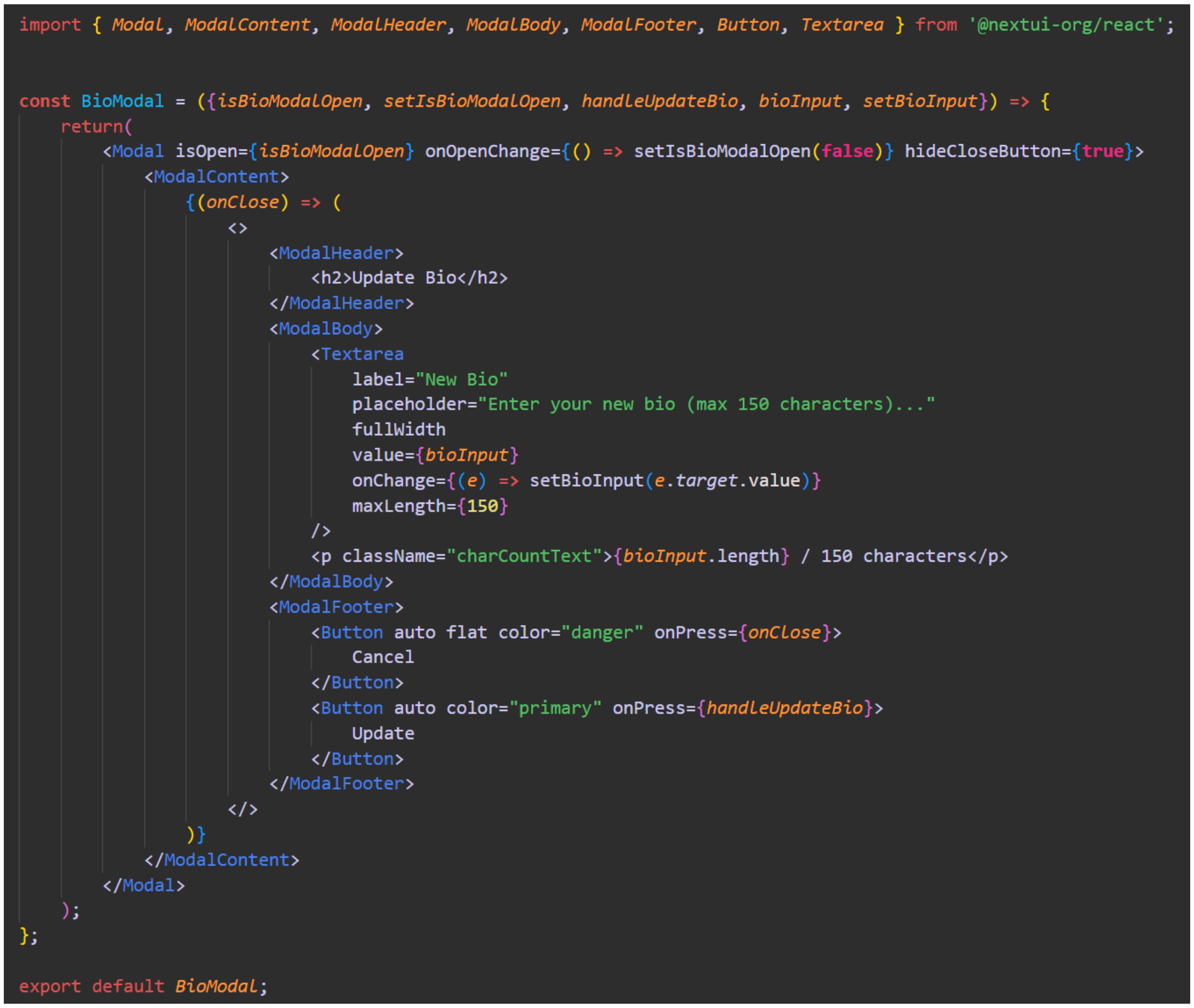
Description: Displays every user within a given list of users using the UserCard component.



```
import React from 'react';
import { Card, ScrollShadow } from '@nextui-org/react';
import UserCard from './UserCard';

const AllUserList = ({ users, followUser }) => {
    return (
        <Card className='profile-container'>
            <ScrollShadow hideScrollBar>
                {users.length > 0 ? (
                    users.map((user) => (
                        <UserCard
                            key={user.user_id}
                            user={user}
                            followUser={followUser}
                        />
                    ))
                ) : (
                    <p>No users found</p>
                )}
            </ScrollShadow>
        </Card>
    );
};

export default AllUserList;
```

/assets/components/BioModal.js

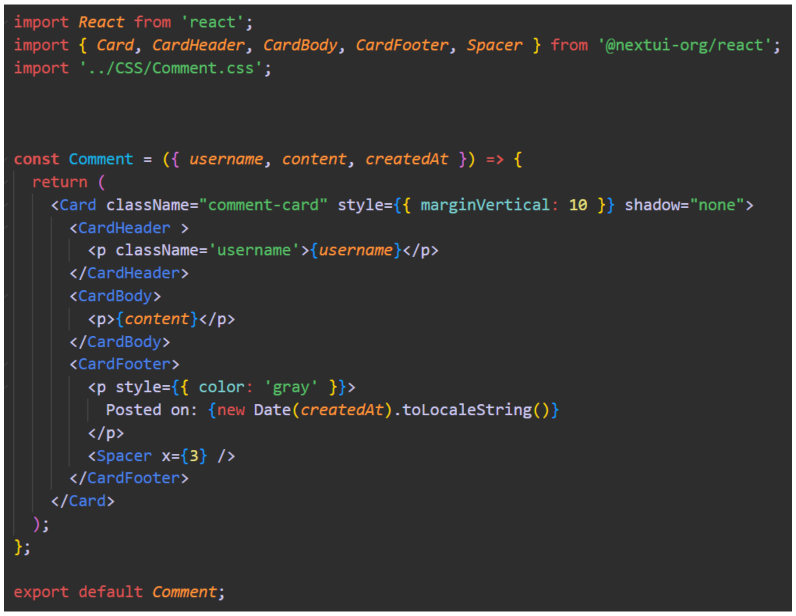
Description: Displays a text field where a user's current bio can be viewed and edited. Has a close button and an update button.



```
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button, Textarea } from '@nextui-org/react';


const BioModal = ({isBioModalOpen, setIsBioModalOpen, handleUpdateBio, bioInput, setBioInput}) => {
    return(
        <Modal isOpen={isBioModalOpen} onOpenChange={() => setIsBioModalOpen(false)} hideCloseButton={true}>
            <ModalContent>
                {(onClose) => (
                    <>
                        <ModalHeader>
                            <h2>Update Bio</h2>
                        </ModalHeader>
                        <ModalBody>
                            <Textarea
                                label="New Bio"
                                placeholder="Enter your new bio (max 150 characters)..."
                                fullWidth
                                value={bioInput}
                                onChange={(e) => setBioInput(e.target.value)}
                                maxLength={150}
                            />
                            <p className="charCountText">{bioInput.length} / 150 characters</p>
                        </ModalBody>
                        <ModalFooter>
                            <Button auto flat color="danger" onPress={onClose}>
                                Cancel
                            </Button>
                            <Button auto color="primary" onPress={handleUpdateBio}>
                                Update
                            </Button>
                        </ModalFooter>
                    </>
                )}
            </ModalContent>
        </Modal>
    );
};

export default BioModal;
```

/assets/components/Comment.js

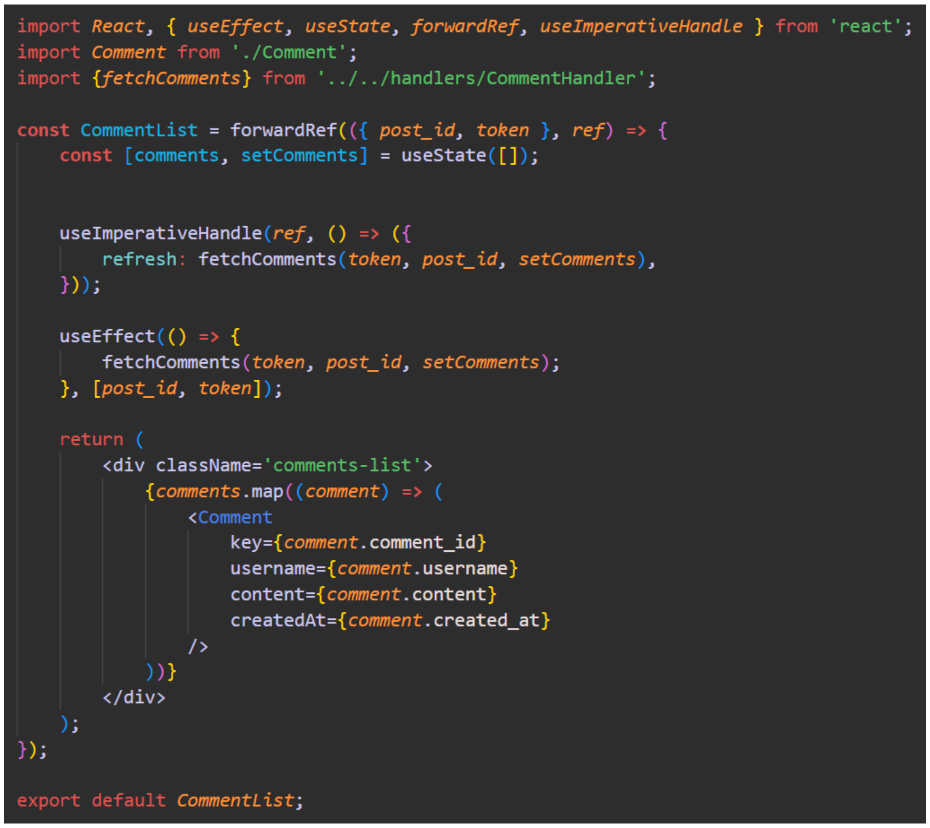
Description: Displays a user’s username, content of their comment, and the datetime when it was made.



```
import React from 'react';
import { Card, CardHeader, CardBody, CardFooter, Spacer } from '@nextui-org/react';
import '../CSS/Comment.css';


const Comment = ({ username, content, createdAt }) => {
  return (
    <Card className="comment-card" style={{ marginVertical: 10 }} shadow="none">
      <CardHeader >
        <p className='username'>{username}</p>
      </CardHeader>
      <CardBody>
        <p>{content}</p>
      </CardBody>
      <CardFooter>
        <p style={{ color: 'gray' }}>
          Posted on: {new Date(createdAt).toLocaleString()}
        </p>
        <Spacer x={3} />
      </CardFooter>
    </Card>
  );
};

export default Comment;
```

/assets/components/CommentList.js

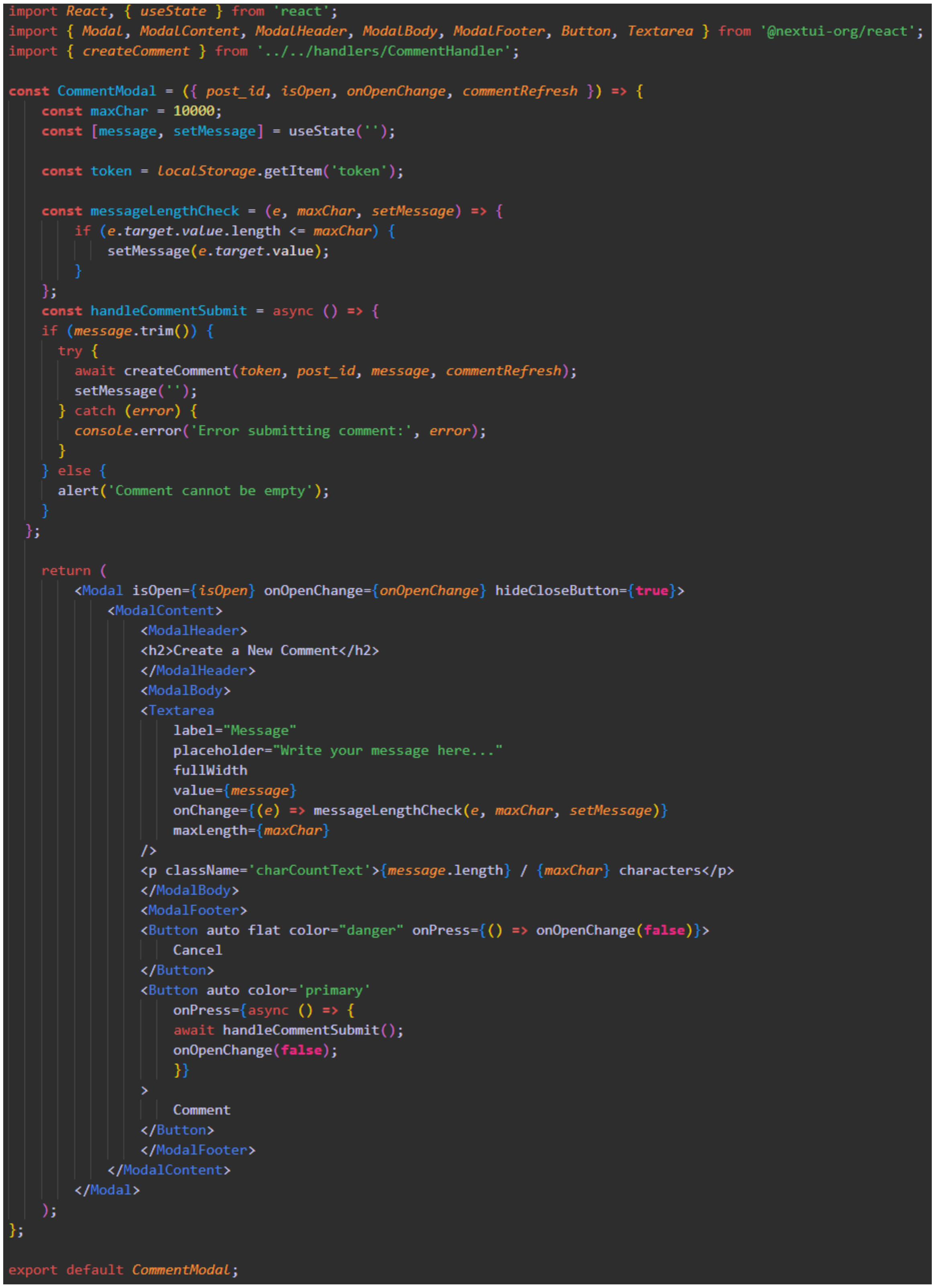
Description: Fetches a list of comments of a post with the given post_id. Displays the comments as a list of the Comment component above.



```
import React, { useEffect, useState, forwardRef, useImperativeHandle } from 'react';
import Comment from './Comment';
import {fetchComments} from '../../handlers/CommentHandler';

const CommentList = forwardRef(({ post_id, token }, ref) => {
    const [comments, setComments] = useState([]);


    useImperativeHandle(ref, () => ({
        refresh: fetchComments(token, post_id, setComments),
    }));

    useEffect(() => {
        fetchComments(token, post_id, setComments);
    }, [post_id, token]);

    return (
        <div className='comments-list'>
            {comments.map((comment) => (
                <Comment
                    key={comment.comment_id}
                    username={comment.username}
                    content={comment.content}
                    createdAt={comment.created_at}
                />
            ))}
        </div>
    );
});

export default CommentList;
```

/assets/components/CommentModal.js

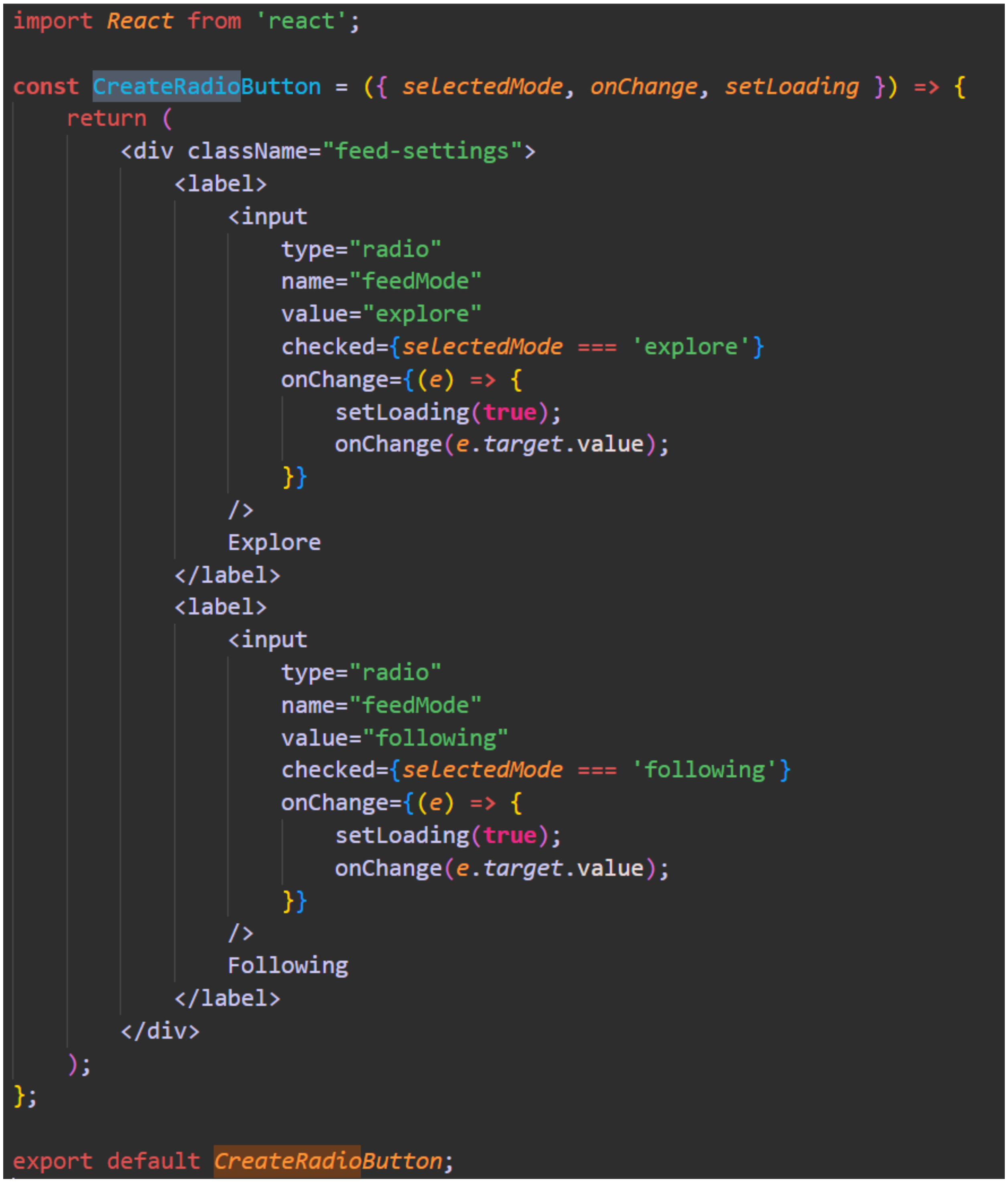
Description: Displays a modal where a user can write a message to comment on another post. Contains a cancel button to close and a Comment button to attach a comment to a post.



```
import React, { useState } from 'react';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button, Textarea } from '@nextui-org/react';
import { createComment } from '../../handlers/CommentHandler';

const CommentModal = ({ post_id, isOpen, onOpenChange, commentRefresh }) => {
    const maxChar = 10000;
    const [message, setMessage] = useState('');

    const token = localStorage.getItem('token');

    const messageLengthCheck = (e, maxChar, setMessage) => {
        if (e.target.value.length <= maxChar) {
            setMessage(e.target.value);
        }
    };
    const handleCommentSubmit = async () => {
    if (message.trim()) {
      try {
        await createComment(token, post_id, message, commentRefresh);
        setMessage('');
      } catch (error) {
        console.error('Error submitting comment:', error);
      }
    } else {
      alert('Comment cannot be empty');
    }
  };

    return (
        <Modal isOpen={isOpen} onOpenChange={onOpenChange} hideCloseButton={true}>
            <ModalContent>
                <ModalHeader>
                <h2>Create a New Comment</h2>
                </ModalHeader>
                <ModalBody>
                <Textarea
                    label="Message"
                    placeholder="Write your message here..."
                    fullWidth
                    value={message}
                    onChange={(e) => messageLengthCheck(e, maxChar, setMessage)}
                    maxLength={maxChar}
                />
                <p className='charCountText'>{message.length} / {maxChar} characters</p>
                </ModalBody>
                <ModalFooter>
                <Button auto flat color="danger" onPress={() => onOpenChange(false)}>
                    Cancel
                </Button>
                <Button auto color='primary'
                    onPress={async () => {
                    await handleCommentSubmit();
                    onOpenChange(false);
                    }}
                >
                    Comment
                </Button>
                </ModalFooter>
            </ModalContent>
        </Modal>
    );
};

export default CommentModal;
```

/assets/components/CreateRadioButton.js

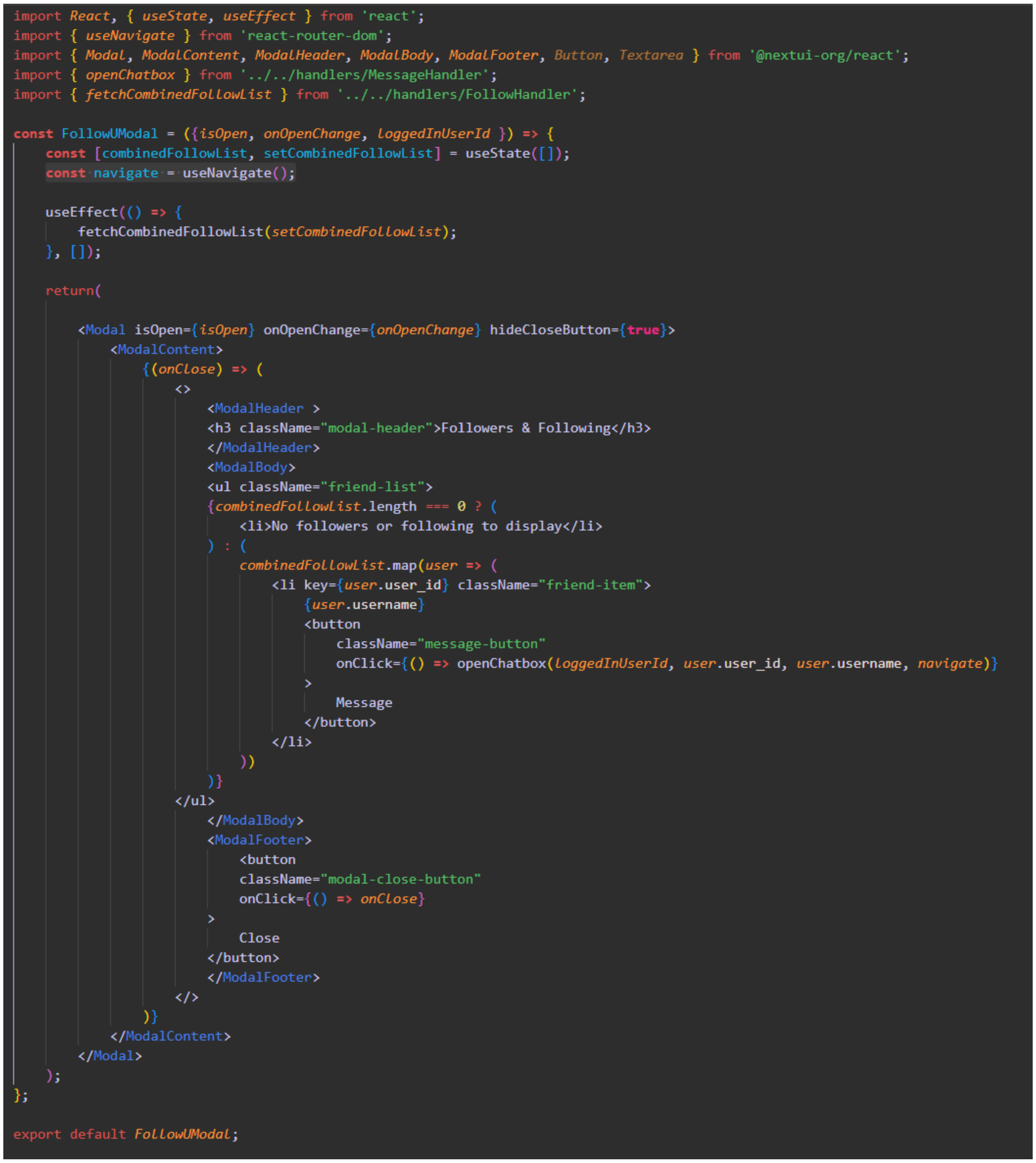
Description: Displays two buttons that switch the feed between all posts and following posts.



```
import React from 'react';

const CreateRadioButton = ({ selectedMode, onChange, setLoading }) => {
    return (
        <div className="feed-settings">
            <label>
                <input
                    type="radio"
                    name="feedMode"
                    value="explore"
                    checked={selectedMode === 'explore'}
                    onChange={(e) => {
                        setLoading(true);
                        onChange(e.target.value);
                    }}
                />
                Explore
            </label>
            <label>
                <input
                    type="radio"
                    name="feedMode"
                    value="following"
                    checked={selectedMode === 'following'}
                    onChange={(e) => {
                        setLoading(true);
                        onChange(e.target.value);
                    }}
                />
                Following
            </label>
        </div>
    );
};

export default CreateRadioButton;
```

/assets/components/FollowUModal.js

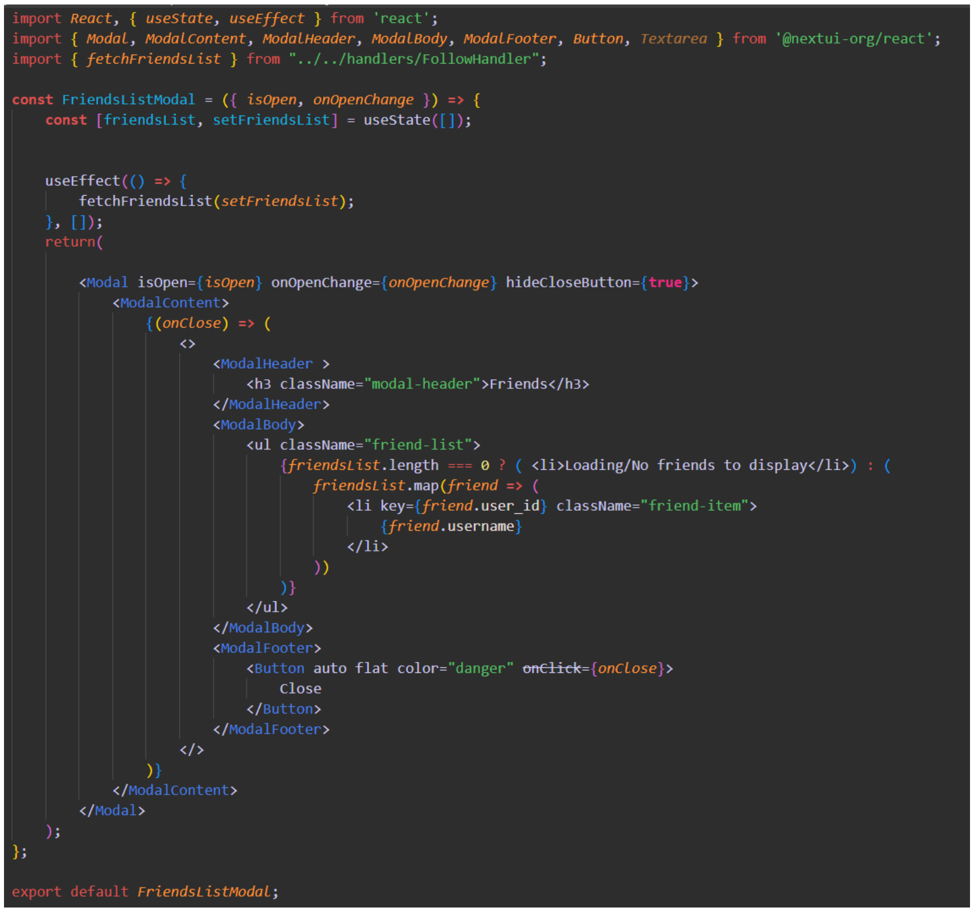
Description: Displays a modal that shows a list of followers and following users..



```
import React, { useState, useEffect } from 'react';
import { useNavigate } from 'react-router-dom';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button, Textarea } from '@nextui-org/react';
import { openChatbox } from '../../handlers/MessageHandler';
import { fetchCombinedFollowList } from '../../handlers/FollowHandler';

const FollowUModal = ({isOpen, onOpenChange, loggedInUserId }) => {
    const [combinedFollowList, setCombinedFollowList] = useState([]);
    const navigate = useNavigate();

    useEffect(() => {
        fetchCombinedFollowList(setCombinedFollowList);
    }, []);

    return(

        <Modal isOpen={isOpen} onOpenChange={onOpenChange} hideCloseButton={true}>
            <ModalContent>
                {(onClose) => (
                    <>
                        <ModalHeader >
                        <h3 className="modal-header">Followers & Following</h3>
                        </ModalHeader>
                        <ModalBody>
                        <ul className="friend-list">
                        {combinedFollowList.length === 0 ? (
                            <li>No followers or following to display</li>
                        ) : (
                            combinedFollowList.map(user => (
                                <li key={user.user_id} className="friend-item">
                                    {user.username}
                                    <button
                                        className="message-button"
                                        onClick={() => openChatbox(loggedInUserId, user.user_id, user.username, navigate)}
                                    >
                                        Message
                                    </button>
                                </li>
                            ))
                        )}
                    </ul>
                        </ModalBody>
                        <ModalFooter>
                            <button
                            className="modal-close-button"
                            onClick={() => onClose}
                        >
                            Close
                        </button>
                        </ModalFooter>
                    </>
                )}
            </ModalContent>
        </Modal>
    );
};

export default FollowUModal;
```

/assets/components/FriendsListModal.js

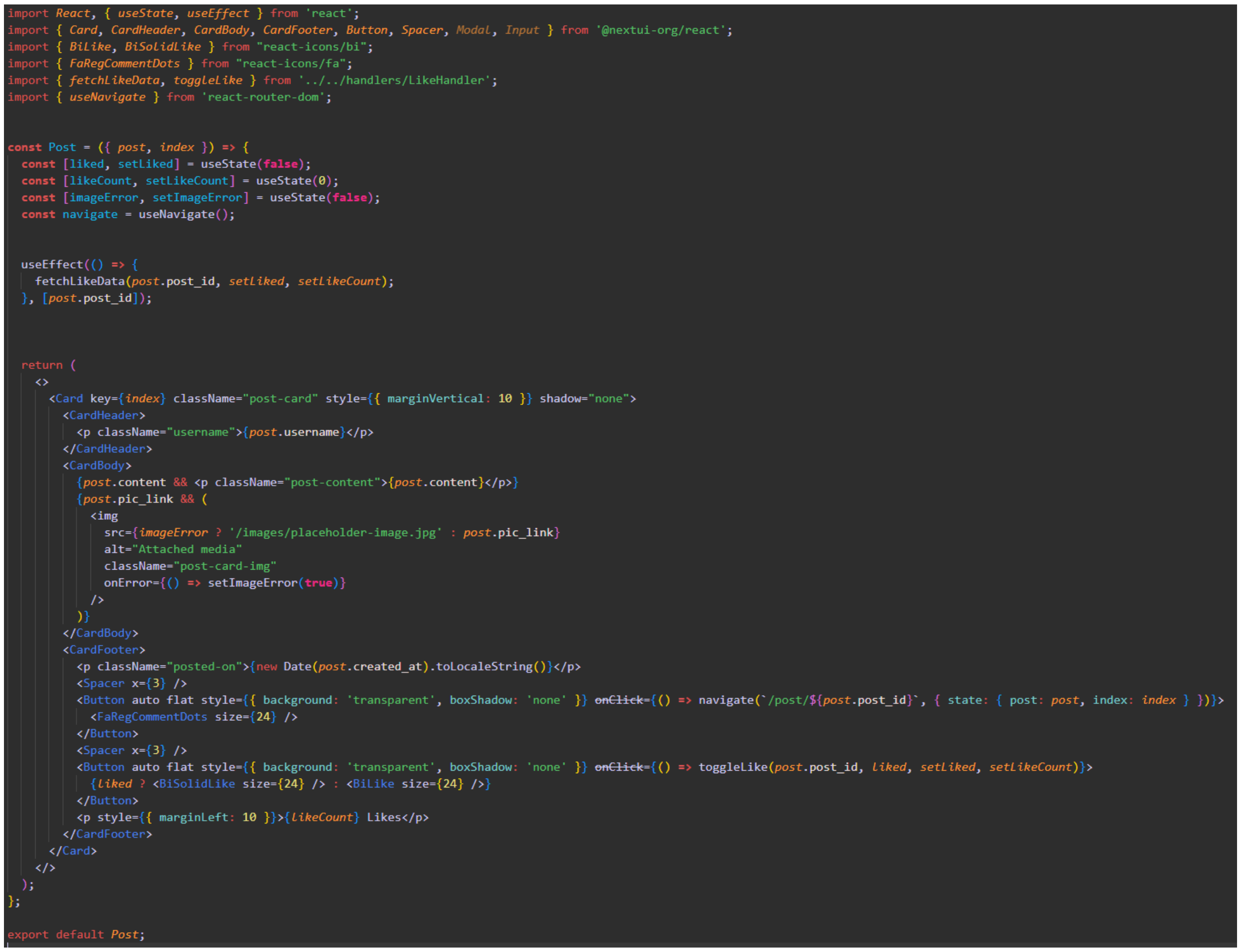
Description: Displays amodal that shows a list of users who follow the current user and the current user also follows that same user.



```
import React, { useState, useEffect } from 'react';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button, Textarea } from '@nextui-org/react';
import { fetchFriendsList } from "../../handlers/FollowHandler";

const FriendsListModal = ({ isOpen, onOpenChange }) => {
    const [friendsList, setFriendsList] = useState([]);


    useEffect(() => {
        fetchFriendsList(setFriendsList);
    }, []);
    return(

        <Modal isOpen={isOpen} onOpenChange={onOpenChange} hideCloseButton={true}>
            <ModalContent>
                {(onClose) => (
                    <>
                        <ModalHeader >
                            <h3 className="modal-header">Friends</h3>
                        </ModalHeader>
                        <ModalBody>
                            <ul className="friend-list">
                                {friendsList.length === 0 ? ( <li>Loading/No friends to display</li>) : (
                                    friendsList.map(friend => (
                                        <li key={friend.user_id} className="friend-item">
                                            {friend.username}
                                        </li>
                                    ))
                                )}
                            </ul>
                        </ModalBody>
                        <ModalFooter>
                            <Button auto flat color="danger" ~~onClick~~={onClose}>
                                Close
                            </Button>
                        </ModalFooter>
                    </>
                )}
            </ModalContent>
        </Modal>
    );
};

export default FriendsListModal;
```

/assets/components/Post.js

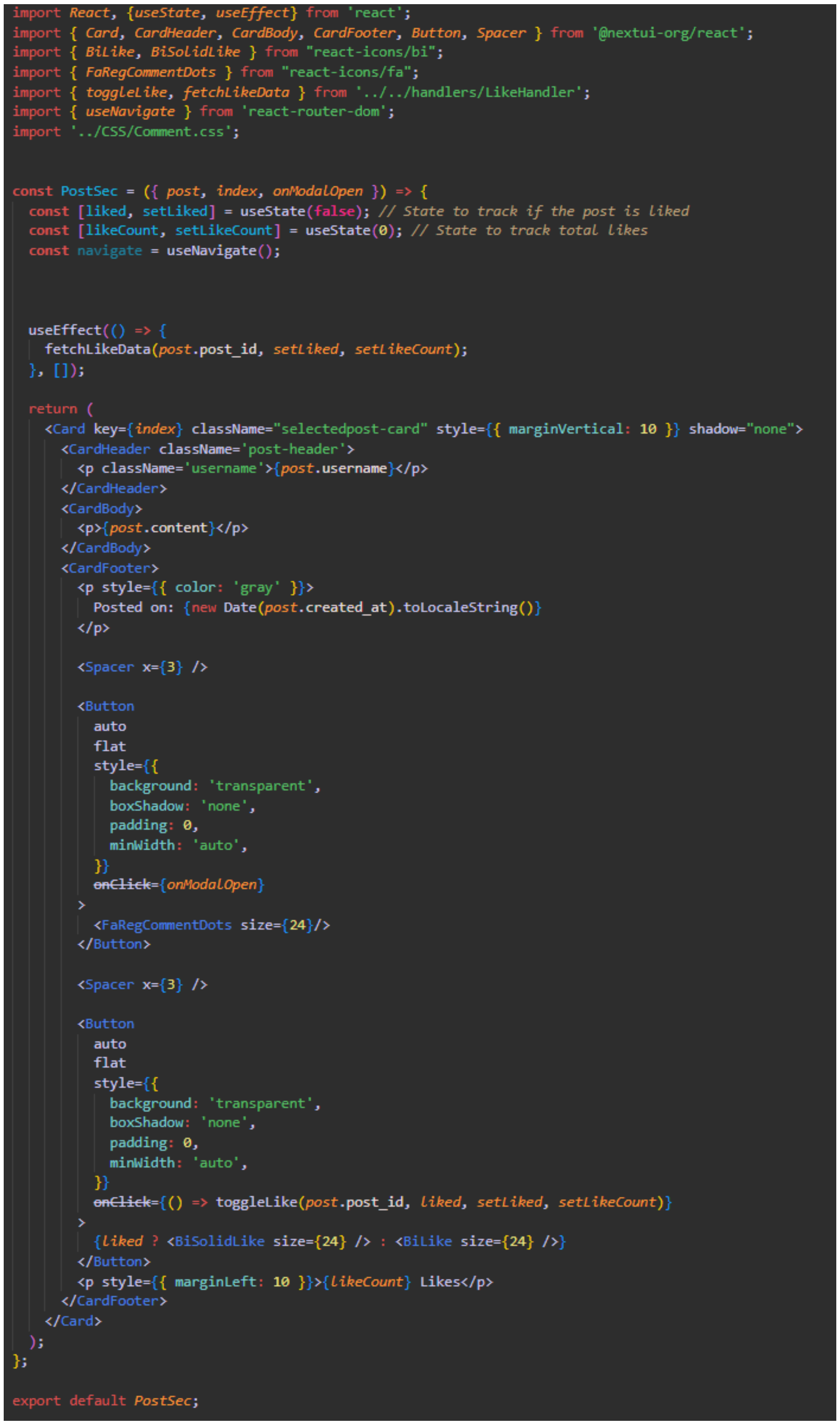
Description: Displays a post with the given post object. Has the user who posted username, date posted, and the contents of the post. An image will also be displayed if it is a part of the post. The user can press the like button to like, and the comment button to direct to another page to display the post's comments.



```
import React, { useState, useEffect } from 'react';
import { Card, CardHeader, CardBody, CardFooter, Button, Spacer, Modal, Input } from '@nextui-org/react';
import { BiLike, BiSolidLike } from "react-icons/bi";
import { FaRegCommentDots } from "react-icons/fa";
import { fetchLikeData, toggleLike } from '../../handlers/LikeHandler';
import { useNavigate } from 'react-router-dom';


const Post = ({ post, index }) => {
  const [liked, setLiked] = useState(false);
  const [likeCount, setLikeCount] = useState(0);
  const [imageError, setImageError] = useState(false);
  const navigate = useNavigate();


  useEffect(() => {
    fetchLikeData(post.post_id, setLiked, setLikeCount);
  }, [post.post_id]);



  return (
    <>
      <Card key={index} className="post-card" style={{ marginVertical: 10 }} shadow="none">
        <CardHeader>
          <p className="username">{post.username}</p>
        </CardHeader>
        <CardBody>
          {post.content && <p className="post-content">{post.content}</p>}
          {post.pic_link && (
            <img
              src={imageError ? '/images/placeholder-image.jpg' : post.pic_link}
              alt="Attached media"
              className="post-card-img"
              onError={() => setImageError(true)}
            />
          )}
        </CardBody>
        <CardFooter>
          <p className="posted-on">{new Date(post.created_at).toLocaleString()}</p>
          <Spacer x={3} />
          <Button auto flat style={{ background: 'transparent', boxShadow: 'none' }} ~~onClick~~={() => navigate(`/post/${post.post_id}`, { state: { post: post, index: index } })}>
            <FaRegCommentDots size={24} />
          </Button>
          <Spacer x={3} />
          <Button auto flat style={{ background: 'transparent', boxShadow: 'none' }} ~~onClick~~={() => toggleLike(post.post_id, liked, setLiked, setLikeCount)}>
            {liked ? <BiSolidLike size={24} /> : <BiLike size={24} />}
          </Button>
          <p style={{ marginLeft: 10 }}>{likeCount} Likes</p>
        </CardFooter>
      </Card>
    </>
  );
};

export default Post;
```

/assets/components/PostSec.js

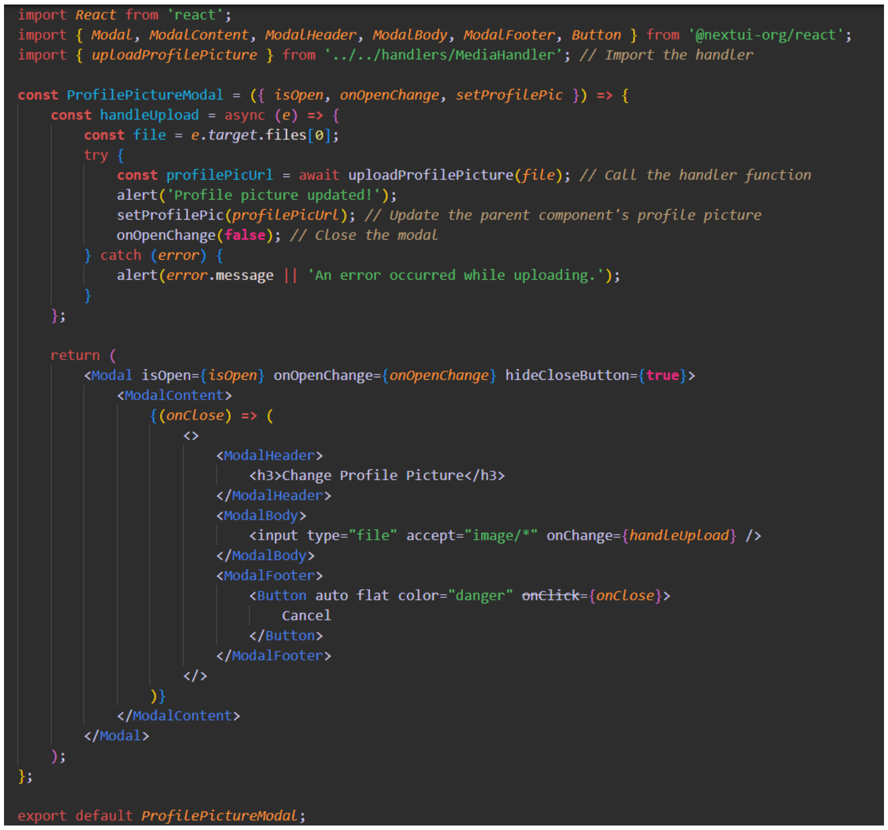
Description: Same contents as the post, but is specific to the selected post page. The comment button will instead allow a user to add a comment.



```
import React, {useState, useEffect} from 'react';
import { Card, CardHeader, CardBody, CardFooter, Button, Spacer } from '@nextui-org/react';
import { BiLike, BiSolidLike } from "react-icons/bi";
import { FaRegCommentDots } from "react-icons/fa";
import { toggleLike, fetchLikeData } from '../../handlers/LikeHandler';
import { useNavigate } from 'react-router-dom';
import '../CSS/Comment.css';


const PostSec = ({ post, index, onModalOpen }) => {
  const [liked, setLiked] = useState(false); // State to track if the post is liked
  const [likeCount, setLikeCount] = useState(0); // State to track total likes
  const navigate = useNavigate();



  useEffect(() => {
    fetchLikeData(post.post_id, setLiked, setLikeCount);
  }, []);

  return (
    <Card key={index} className="selectedpost-card" style={{ marginVertical: 10 }} shadow="none">
      <CardHeader className='post-header'>
        <p className='username'>{post.username}</p>
      </CardHeader>
      <CardBody>
        <p>{post.content}</p>
      </CardBody>
      <CardFooter>
        <p style={{ color: 'gray' }}>
          Posted on: {new Date(post.created_at).toLocaleString()}
        </p>

        <Spacer x={3} />

        <Button
          auto
          flat
          style={{
            background: 'transparent',
            boxShadow: 'none',
            padding: 0,
            minWidth: 'auto',
          }}
          ~~onClick~~={onModalOpen}
        >
          <FaRegCommentDots size={24}/>
        </Button>

        <Spacer x={3} />

        <Button
          auto
          flat
          style={{
            background: 'transparent',
            boxShadow: 'none',
            padding: 0,
            minWidth: 'auto',
          }}
          ~~onClick~~={() => toggleLike(post.post_id, liked, setLiked, setLikeCount)}
        >
          {liked ? <BiSolidLike size={24} /> : <BiLike size={24} />}
        </Button>
        <p style={{ marginLeft: 10 }}>{likeCount} Likes</p>
      </CardFooter>
    </Card>
  );
};

export default PostSec;
```

/assets/components/ProfilePictureModal.js

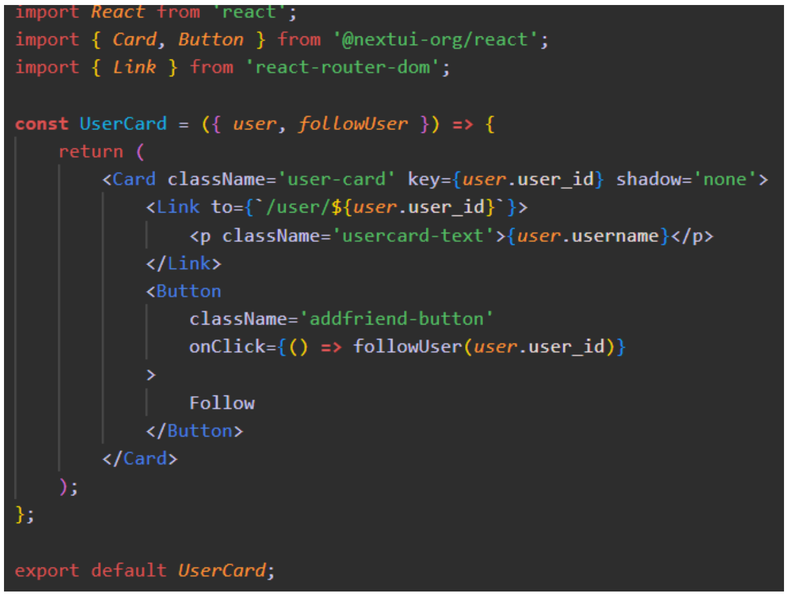
Description: Displays a modal where a user can change their profile picture. The button allows the user to select an image and will change their picture.



```
import React from 'react';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button } from '@nextui-org/react';
import { uploadProfilePicture } from '../../handlers/MediaHandler'; // Import the handler

const ProfilePictureModal = ({ isOpen, onOpenChange, setProfilePic }) => {
    const handleUpload = async (e) => {
        const file = e.target.files[0];
        try {
            const profilePicUrl = await uploadProfilePicture(file); // Call the handler function
            alert('Profile picture updated!');
            setProfilePic(profilePicUrl); // Update the parent component's profile picture
            onOpenChange(false); // Close the modal
        } catch (error) {
            alert(error.message || 'An error occurred while uploading.');
        }
    };

    return (
        <Modal isOpen={isOpen} onOpenChange={onOpenChange} hideCloseButton={true}>
            <ModalContent>
                {(onClose) => (
                    <>
                        <ModalHeader>
                            <h3>Change Profile Picture</h3>
                        </ModalHeader>
                        <ModalBody>
                            <input type="file" accept="image/*" onChange={handleUpload} />
                        </ModalBody>
                        <ModalFooter>
                            <Button auto flat color="danger" ~~onClick~~={onClose}>
                                Cancel
                            </Button>
                        </ModalFooter>
                    </>
                )}
            </ModalContent>
        </Modal>
    );
};

export default ProfilePictureModal;
```

/assets/components/UserCard.js

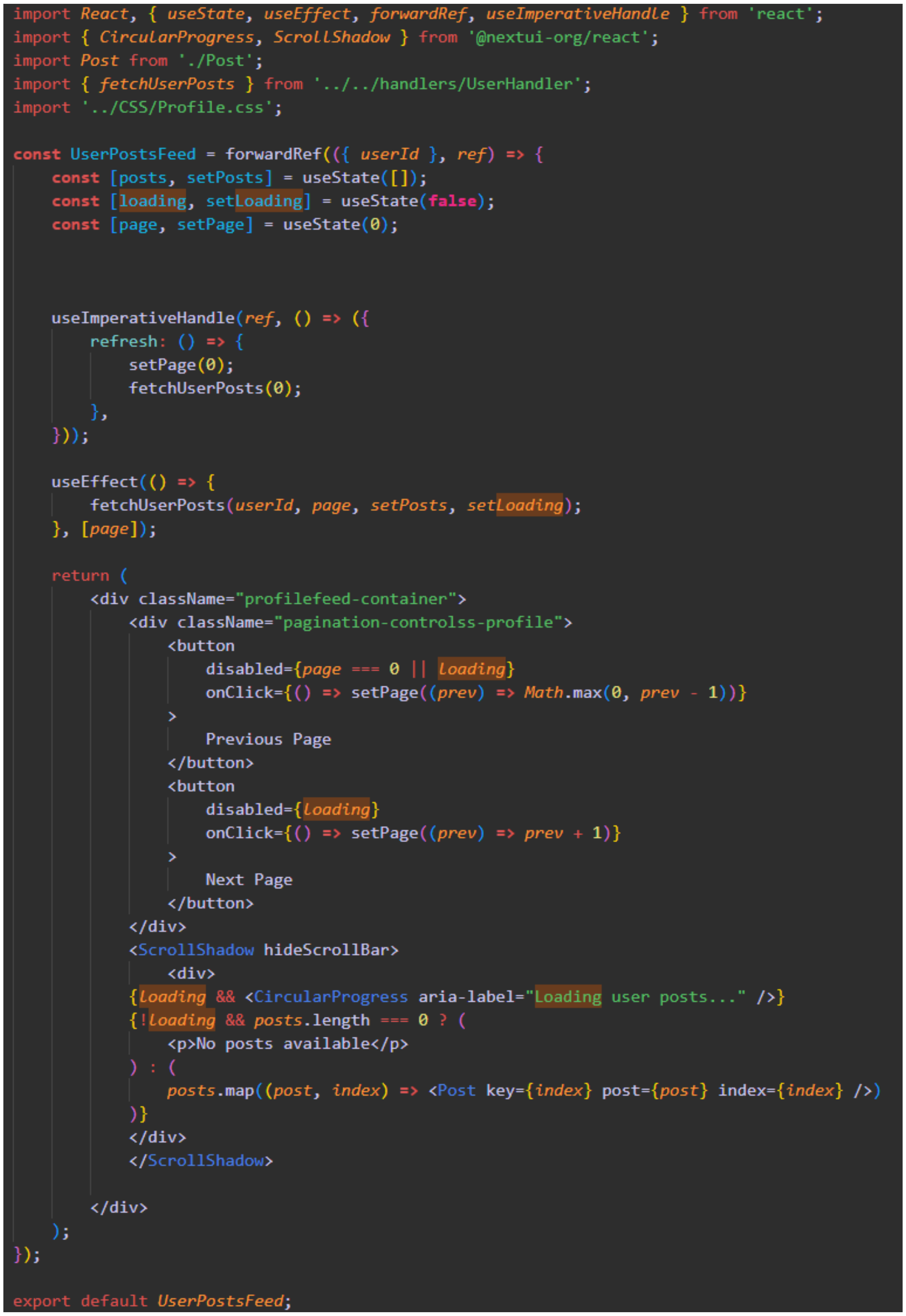
Description: Displays a given user’s name and a follow button to the right.



```
import React from 'react';
import { Card, Button } from '@nextui-org/react';
import { Link } from 'react-router-dom';

const UserCard = ({ user, followUser }) => {
    return (
        <Card className='user-card' key={user.user_id} shadow='none'>
            <Link to={`/user/${user.user_id}`}>
                <p className='usercard-text'>{user.username}</p>
            </Link>
            <Button
                className='addfriend-button'
                onClick={() => followUser(user.user_id)}
            >
                Follow
            </Button>
        </Card>
    );
};

export default UserCard;
```

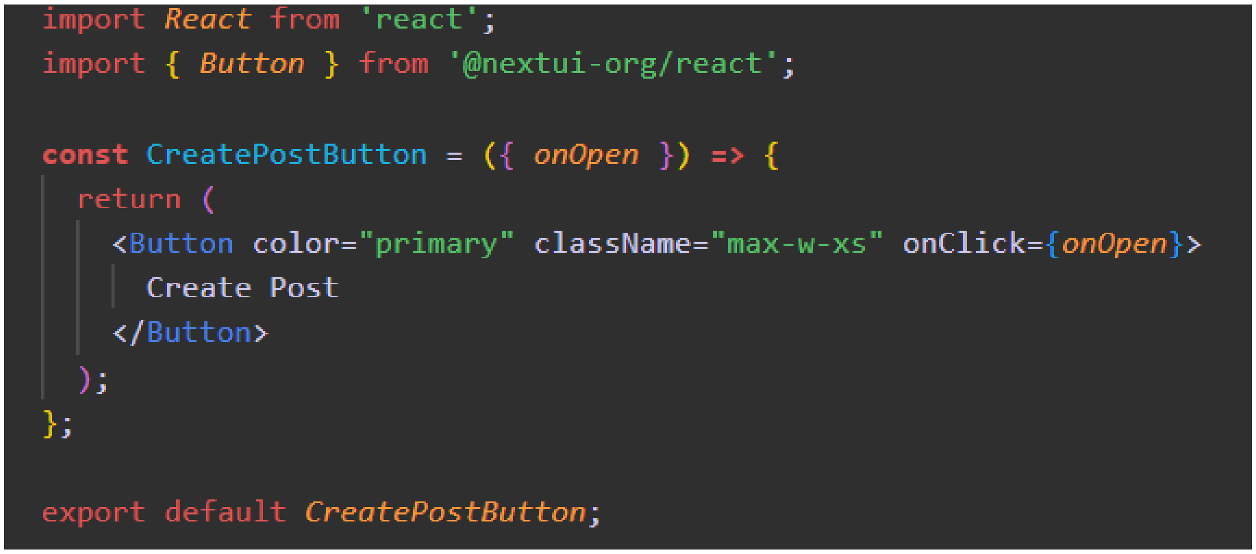
/assets/components/UserPostsFeed.js

Description: fetches a user's posts with their given user_id. Displays the posts and has paginations so the posts don't all load at once.

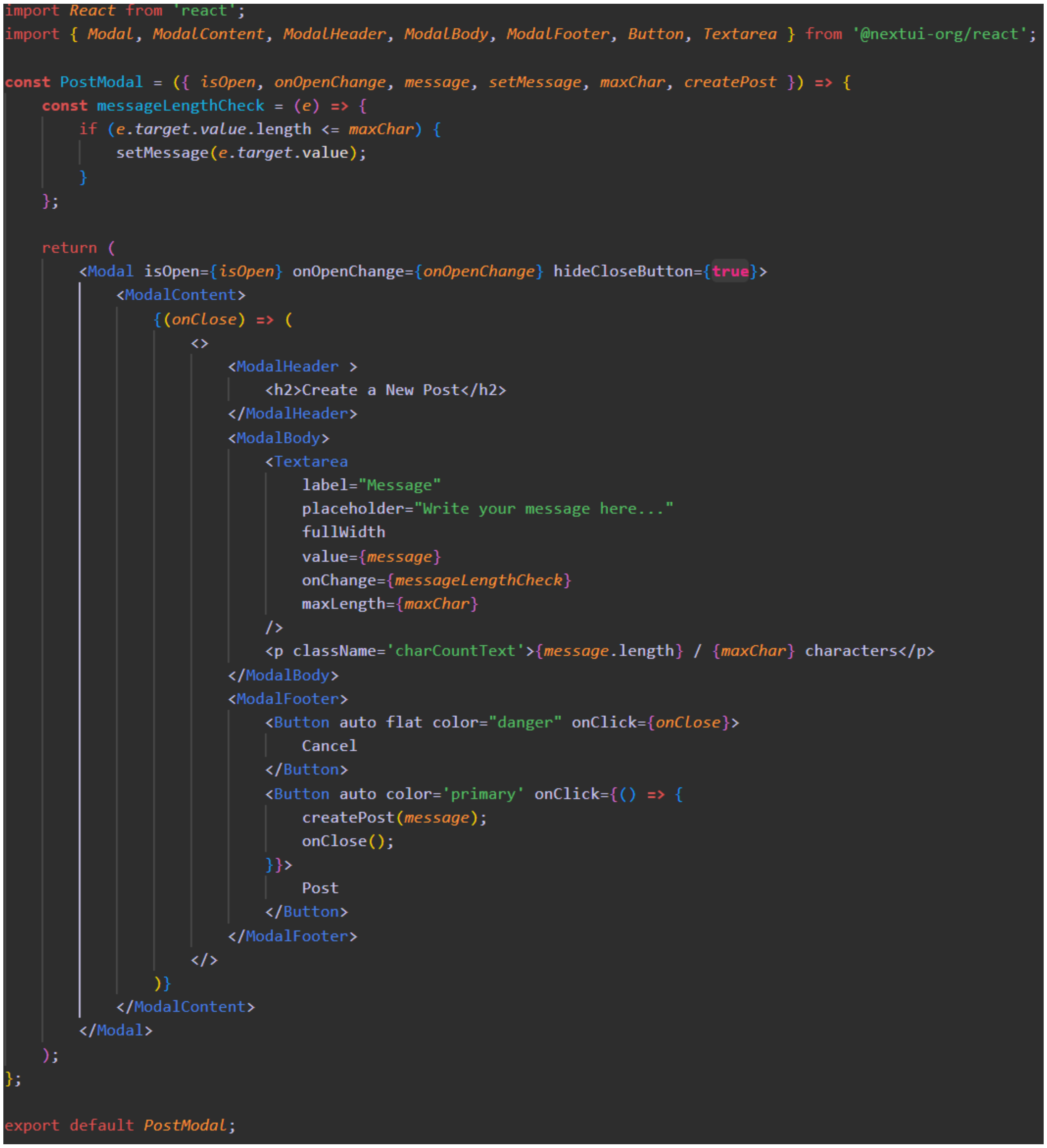


```
import React, { useState, useEffect, forwardRef, useImperativeHandle } from 'react';
import { CircularProgress, ScrollShadow } from '@nextui-org/react';
import Post from './Post';
import { fetchUserPosts } from '../../handlers/UserHandler';
import '../CSS/Profile.css';

const UserPostsFeed = forwardRef(({ userId }, ref) => {
    const [posts, setPosts] = useState([]);
    const [loading, setLoading] = useState(false);
    const [page, setPage] = useState(0);



    useImperativeHandle(ref, () => ({
        refresh: () => {
            setPage(0);
            fetchUserPosts(0);
        },
    }));

    useEffect(() => {
        fetchUserPosts(userId, page, setPosts, setLoading);
    }, [page]);

    return (
        <div className="profilefeed-container">
            <div className="pagination-controlss-profile">
                <button
                    disabled={page === 0 || loading}
                    onClick={() => setPage((prev) => Math.max(0, prev - 1))}
                >
                    Previous Page
                </button>
                <button
                    disabled={loading}
                    onClick={() => setPage((prev) => prev + 1)}
                >
                    Next Page
                </button>
            </div>
            <ScrollShadow hideScrollBar>
                <div>
            {loading && <CircularProgress aria-label="Loading user posts..." />}
            {!loading && posts.length === 0 ? (
                <p>No posts available</p>
            ) : (
                posts.map((post, index) => <Post key={index} post={post} index={index} />)
            )}
            </div>
            </ScrollShadow>

        </div>
    );
});

export default UserPostsFeed;
```

/assets/components/CreatePostButton.js
Description: Displays “Create Post” button that will open the Post Modal when pressed.

.

```
import React from 'react';
import { Button } from '@nextui-org/react';

const CreatePostButton = ({ onOpen }) => {
  return (
    <Button color="primary" className="max-w-xs" onClick={onOpen}>
      Create Post
    </Button>
  );
};

export default CreatePostButton;
```

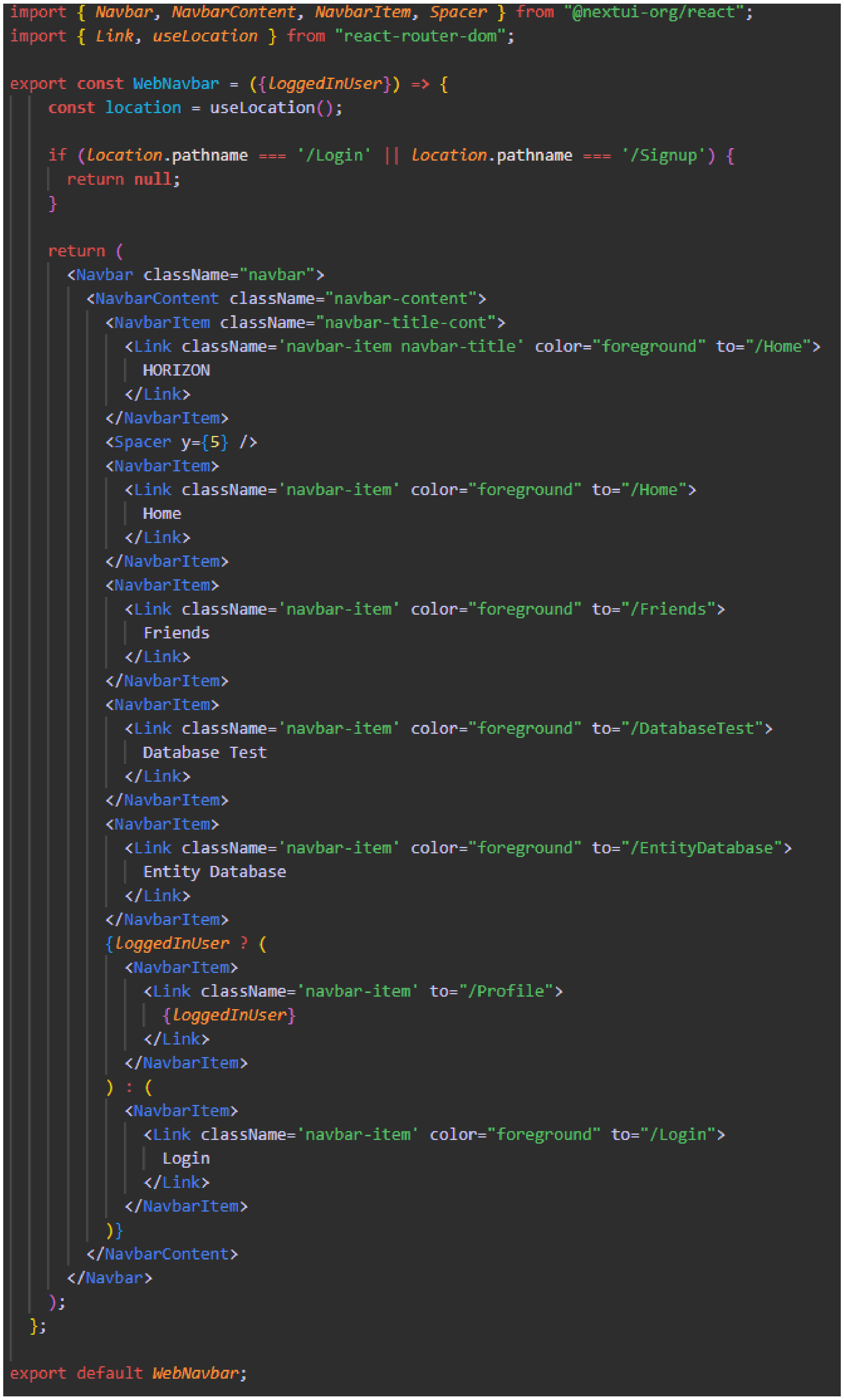
/assets/components/PostModal.js
Description: Displays Post Modal that allows users to write a message to post. The modal will also check if the content character length does not exceed the “maxChar”.Once finished, the “Post” button will publish the post to the feed.



```
import React from 'react';
import { Modal, ModalContent, ModalHeader, ModalBody, ModalFooter, Button, Textarea } from '@nextui-org/react';

const PostModal = ({ isOpen, onOpenChange, message, setMessage, maxChar, createPost }) => {
    const messageLengthCheck = (e) => {
        if (e.target.value.length <= maxChar) {
            setMessage(e.target.value);
        }
    };

    return (
        <Modal isOpen={isOpen} onOpenChange={onOpenChange} hideCloseButton={true}>
            <ModalContent>
                {(onClose) => (
                    <>
                        <ModalHeader >
                            <h2>Create a New Post</h2>
                        </ModalHeader>
                        <ModalBody>
                            <Textarea
                                label="Message"
                                placeholder="Write your message here..."
                                fullWidth
                                value={message}
                                onChange={messageLengthCheck}
                                maxLength={maxChar}
                            />
                            <p className='charCountText'>{message.length} / {maxChar} characters</p>
                        </ModalBody>
                        <ModalFooter>
                            <Button auto flat color="danger" onClick={onClose}>
                                Cancel
                            </Button>
                            <Button auto color='primary' onClick={() => {
                                createPost(message);
                                onClose();
                            }}>
                                Post
                            </Button>
                        </ModalFooter>
                    </>
                )}
            </ModalContent>
        </Modal>
    );
};

export default PostModal;
```

/assets/components/WebNavbar.js

Description: Displays the navbar that contains a link to: Home page, Friends page, both Database tests, and either a login link or the logged in user's Profile page.



```jsx
import { Navbar, NavbarContent, NavbarItem, Spacer } from "@nextui-org/react";
import { Link, useLocation } from "react-router-dom";

export const WebNavbar = ({loggedInUser}) => {
    const location = useLocation();

    if (location.pathname === '/Login' || location.pathname === '/Signup') {
      return null;
    }

    return (
      <Navbar className="navbar">
        <NavbarContent className="navbar-content">
          <NavbarItem className="navbar-title-cont">
            <Link className='navbar-item navbar-title' color="foreground" to="/Home">
              HORIZON
            </Link>
          </NavbarItem>
          <Spacer y={5} />
          <NavbarItem>
            <Link className='navbar-item' color="foreground" to="/Home">
              Home
            </Link>
          </NavbarItem>
          <NavbarItem>
            <Link className='navbar-item' color="foreground" to="/Friends">
              Friends
            </Link>
          </NavbarItem>
          <NavbarItem>
            <Link className='navbar-item' color="foreground" to="/DatabaseTest">
              Database Test
            </Link>
          </NavbarItem>
          <NavbarItem>
            <Link className='navbar-item' color="foreground" to="/EntityDatabase">
              Entity Database
            </Link>
          </NavbarItem>
          {loggedInUser ? (
            <NavbarItem>
              <Link className='navbar-item' to="/Profile">
                {loggedInUser}
              </Link>
            </NavbarItem>
          ) : (
            <NavbarItem>
              <Link className='navbar-item' color="foreground" to="/Login">
                Login
              </Link>
            </NavbarItem>
          )}
        </NavbarContent>
      </Navbar>
    );
  };

export default WebNavbar;
```

Deployment Infrastructure

The application is hosted on AWS Cloud Infrastructure for scalability and reliability:

- **Route 53**: Handles domain registration and DNS routing.
- **CloudFront**: Delivers content with low latency.
- **S3**: Hosts the React frontend and stores user-uploaded media.
- **RDS**: Manages the MySQL database, ensuring automated backups and high availability.
- **Lambda**: Executes serverless functions, reducing overhead costs.
- **API Gateway**: Serves as a single point of entry for backend APIs.

Testing Approach

The testing process was crucial to ensuring the Horizon social media platform met functional and nonfunctional requirements. The following testing methodologies and frameworks were used:

1. Goal of Testing

The objective was to validate the functionality, reliability, and user experience by testing key features in the system. The primary focus was on:

- Backend API correctness and robustness using pytest and unittest.
- Frontend component rendering and interactivity using Jest and React Testing Library.

2. Use Cases

The testing strategy was centered around real-world user scenarios outlined in the Horizon Test Document. Each use case included:

- **Test Number**: Unique identifier for each test.
- **Scenario**: Description of the action being tested.
- **Location**: Section of the website where the action takes place.
- **Expected Outcome**: The anticipated behavior upon executing the test.
- **Pass/Fail**: Status of the test result.
- **Category**: Functional grouping (e.g., Login, Social Interaction).
- **Unit Test Name**: Path to the relevant test file.

3. Categories

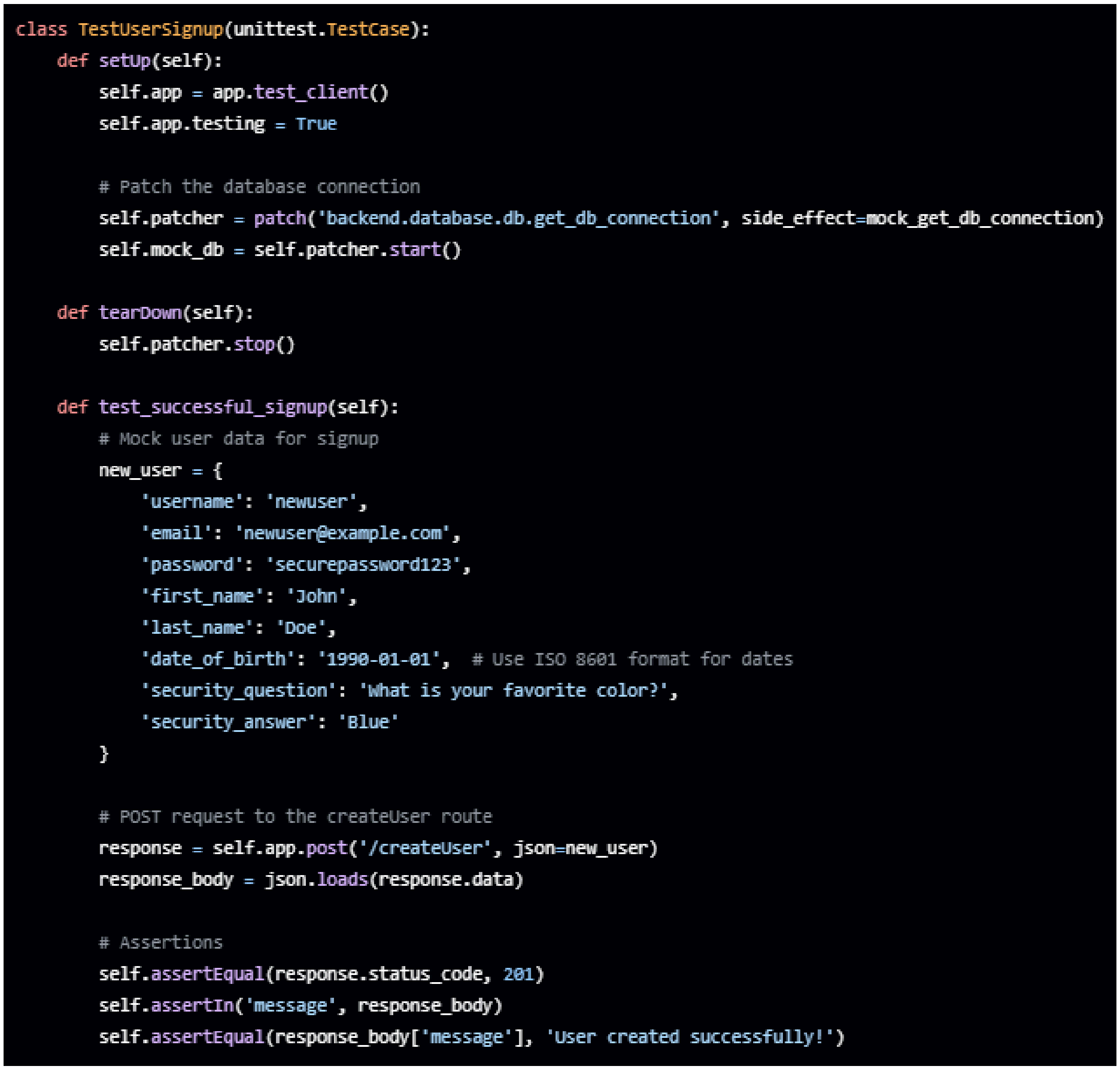
Tests were divided into the following categories based on their purpose and features:

- **Account Login**: Validating secure authentication.

- **Account Creation**: Ensuring new users can register with valid credentials.
- **Content Creation**: Testing posts, comments, and interactions.
- **Social Interaction**: Friendships, followers, and notifications.
- **Account Settings**: Managing profile updates and preferences.
- **Groups**: Handling group creation and participation.
- **Navigation**: Ensuring seamless page transitions.
- **Searching**: Validating user and content search functionality.
- **Alerts**: Ensuring error and success modals function correctly.

4. Backend Testing

The backend was rigorously tested using pytest and unittest. Examples:

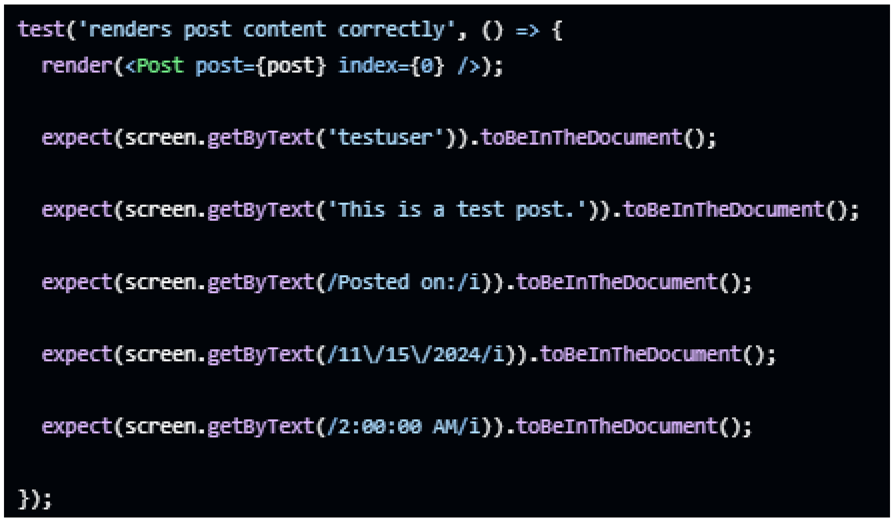


```python
class TestUserSignup(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

        # Patch the database connection
        self.patcher = patch('backend.database.db.get_db_connection', side_effect=mock_get_db_connection)
        self.mock_db = self.patcher.start()

    def tearDown(self):
        self.patcher.stop()

    def test_successful_signup(self):
        # Mock user data for signup
        new_user = {
            'username': 'newuser',
            'email': 'newuser@example.com',
            'password': 'securepassword123',
            'first_name': 'John',
            'last_name': 'Doe',
            'date_of_birth': '1990-01-01',  # Use ISO 8601 format for dates
            'security_question': 'What is your favorite color?',
            'security_answer': 'Blue'
        }

        # POST request to the createUser route
        response = self.app.post('/createUser', json=new_user)
        response_body = json.loads(response.data)

        # Assertions
        self.assertEqual(response.status_code, 201)
        self.assertIn('message', response_body)
        self.assertEqual(response_body['message'], 'User created successfully!')
```

Mock Database Testing: Simulated database interactions using unittest.mock to ensure proper handling of edge cases.
JWT Token Validation: Verified secure session management using token expiration and claims.

5. Frontend Testing

Front-end tests emphasized visual correctness, interaction, and component rendering using **Jest**. Example:



```
test('renders post content correctly', () => {
  render(<Post post={post} index={0} />);

  expect(screen.getByText('testuser')).toBeInTheDocument();

  expect(screen.getByText('This is a test post.')).toBeInTheDocument();

  expect(screen.getByText(/Posted on:/i)).toBeInTheDocument();

  expect(screen.getByText(/11\/15\/2024/i)).toBeInTheDocument();

  expect(screen.getByText(/2:00:00 AM/i)).toBeInTheDocument();

});
```

6. Automation Frameworks
Backend: Automated API tests using pytest for fast execution and easy maintenance.
Frontend: Component and integration tests were automated with Jest, ensuring immediate feedback on UI changes.

Conclusion and Suggestions

The Horizon social media platform is in the process of being fully successfully developed and tested, delivering:

- An interactive user interface that complete user experience
- Connection, engagement, and collaboration for users
- Scalable code for more additions to the website

The testing phase validated the system's reliability and performance under various scenarios. Key functionalities such as content creation, messaging, and social interactions passed tests, ensuring readiness for deployment.

Suggestions

Though the social media platform has completed the objectives and goals, a couple features could be added to improve the website.

Enhanced Testing Coverage:

Introduce performance testing for high-traffic scenarios.

Expand test cases for accessibility.

User Feedback:

Deploy beta testing for a select group of users to gather real-world feedback.

Incorporate feedback loops for continuous improvement

Enhance User Experience:

Conduct regular testing to identify points that can optimize the user interface.

Introduce personalized features such as content recommendations and advanced filtering options.

Appendices

Appendix A: Test Case Table

Appendix B: Key Libraries