SocialPulse - Team W

grade: 98
comments: one of the top reports in the class, with 2 minor glitches in petri and E/R.
I'm expecting a great system delivered by this team.

SocialPulse
Software Requirements Specification
For SocialPulse

Version 1.0

Revision History

Date	Version	Description	Author
11/21/2023	1.0	Added Section 5 mockups	Selma Doganata
11/21/2023	1.0	ER Diagram / Pseudo Code	Georgios Ioannou
11/21/2023	1.0	Added collaboration diagrams/petri-nets and Purpose	Leon Belegu
11/21/2023	1.0	Added collaboration diagrams/petri-nets/	Jolie Huang
11/21/2023	1.0	ER Diagram / Pseudo Code	Christian Rasmussen

Table of Contents

1. Introduction

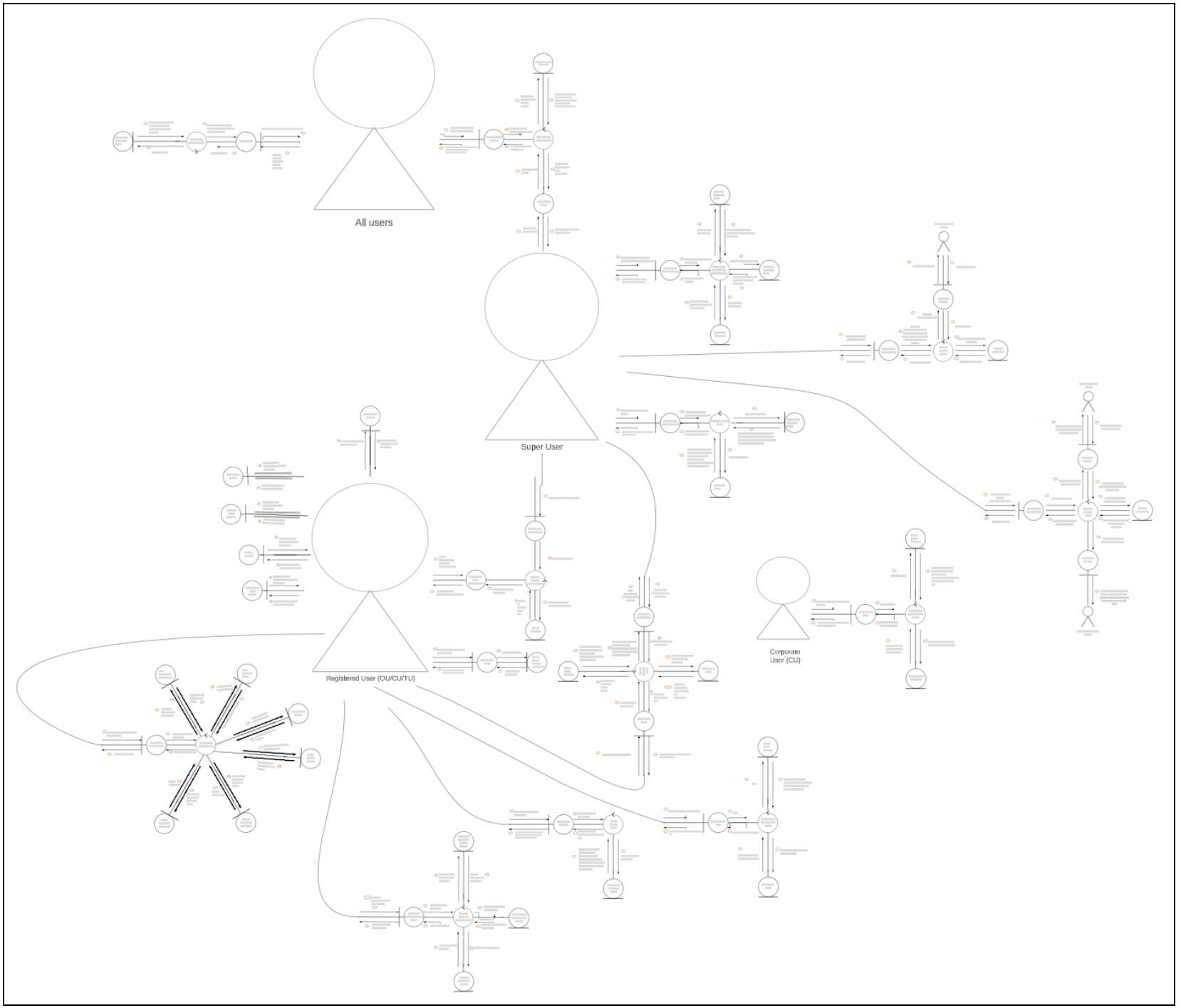
1.1 Purpose

This report's objective is to give a general overview of the logic and data structure of the system in order to implement the features required by the specifications. With the use of Petri net, E-R, and collaborative class diagrams, this report will present a comprehensive overview of the system. Subsequently, this report will include pictures and explanations of each page along with an explanation of its layout and features. Lastly, the work distribution and advancement of the SocialPulse system, along with the duties and obligations of each team member, will all be covered in this report.

1.2 System Collaboration Diagram

The following diagram is a system collaboration diagram depicting how features of our system will interact with one another.

- Link to the LucidChart file: https://lucid.app/lucidchart/9a2ea01e-49e1-4f5a-a3b8-20f228090aac/edit?viewport_loc=-18386%2C1767%2C20554%2C10377%2C0_0&invitationId=inv_55314f33-cfd0-4293-853a-f591d97e9415
- Link to a downloadable PDF of the diagram: https://drive.google.com/file/d/1hhhwVzqEjAB4fr-Ui-PySFClz6U6W3-l/view?usp=sharing


All users
Super User
Registered User (OU/CU/TU)
Corporate User (CU)

2. Use Cases

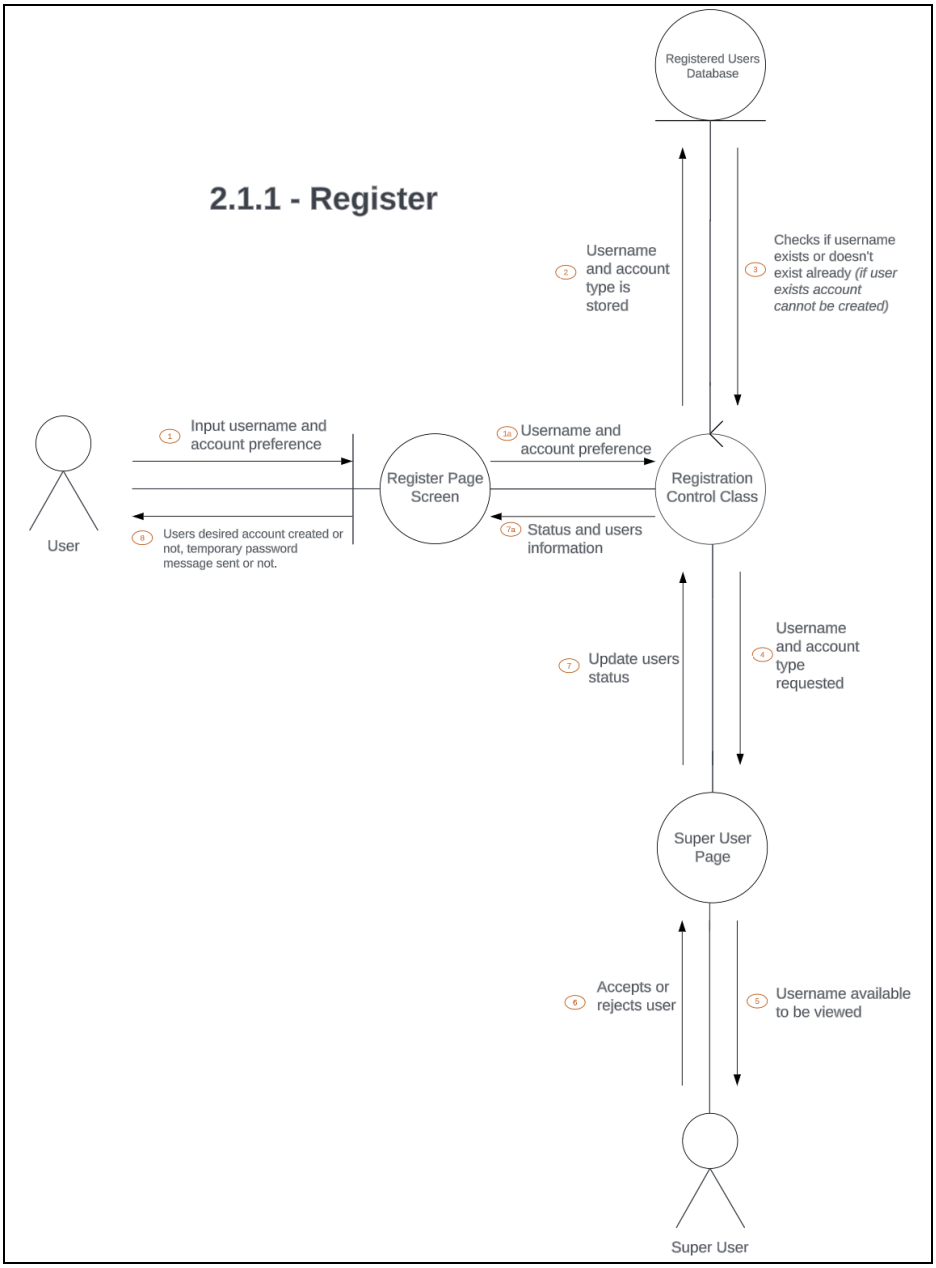
Section 2.1 of the Design Report will serve as an overview of all use cases including a description of each, their different scenarios, and an accompanying collaboration or sequence class diagram. Section 2.2 will display petri-nets to showcase how a use-case interacts with another.

2.1 Scenarios

2.1.1 Register

Description: This is a function that is accessible and available when a surfer and registered user first access the site. They will be prompted to register for an account (ordinary user or corporate user), in which they will declare their username. The super user will then either confirm or deny new users.

- Normal:
    - The normal case is that the username a surfer wants to use is not taken/valid.
- Exceptional:
    - The username has been taken already (meaning they are already a registered user).

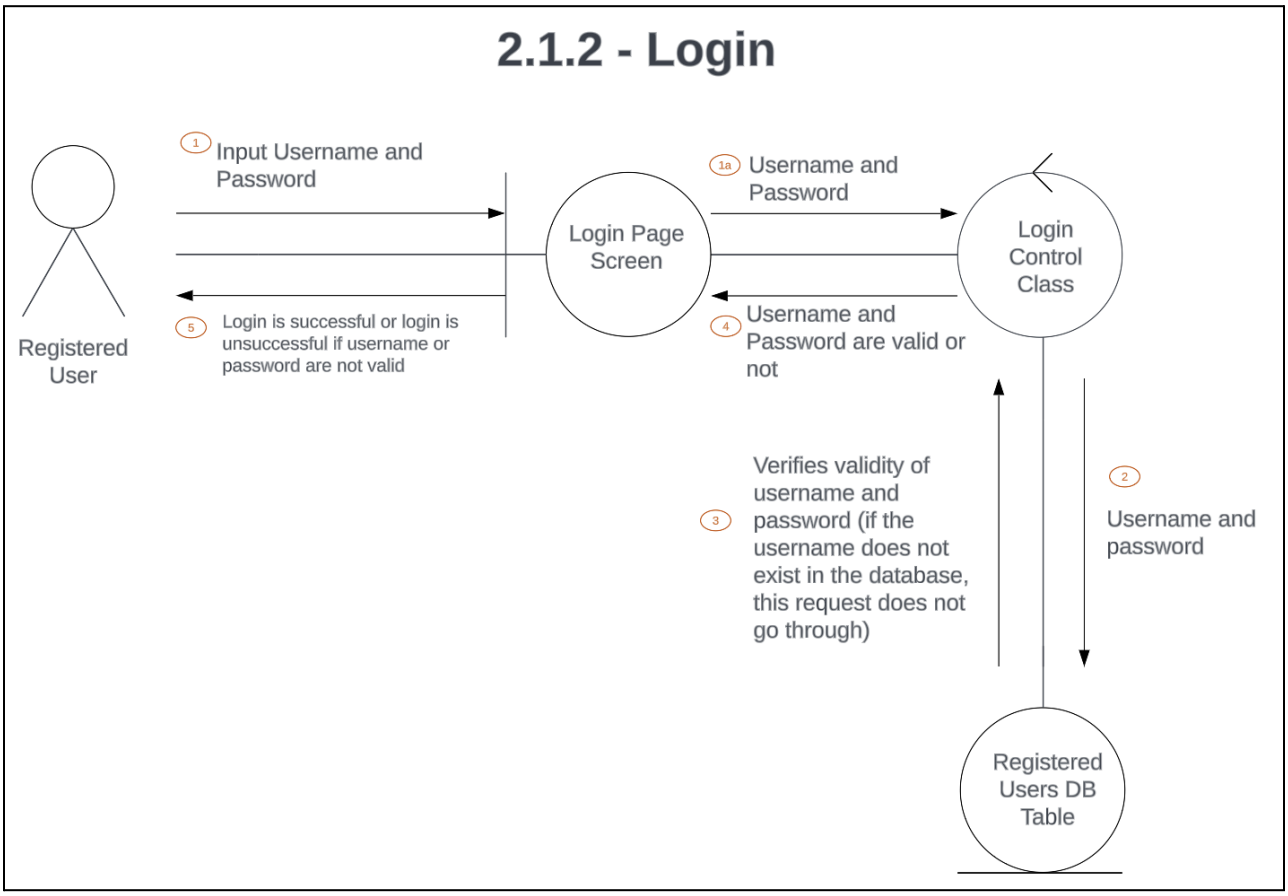


2.1.2 Login

Description: This is a function accessible to returning users, where they will have the option to login to their existing account to access the site.

- Normal:
    - The normal case is that the returning user provides a valid username and password.
- Exceptional:

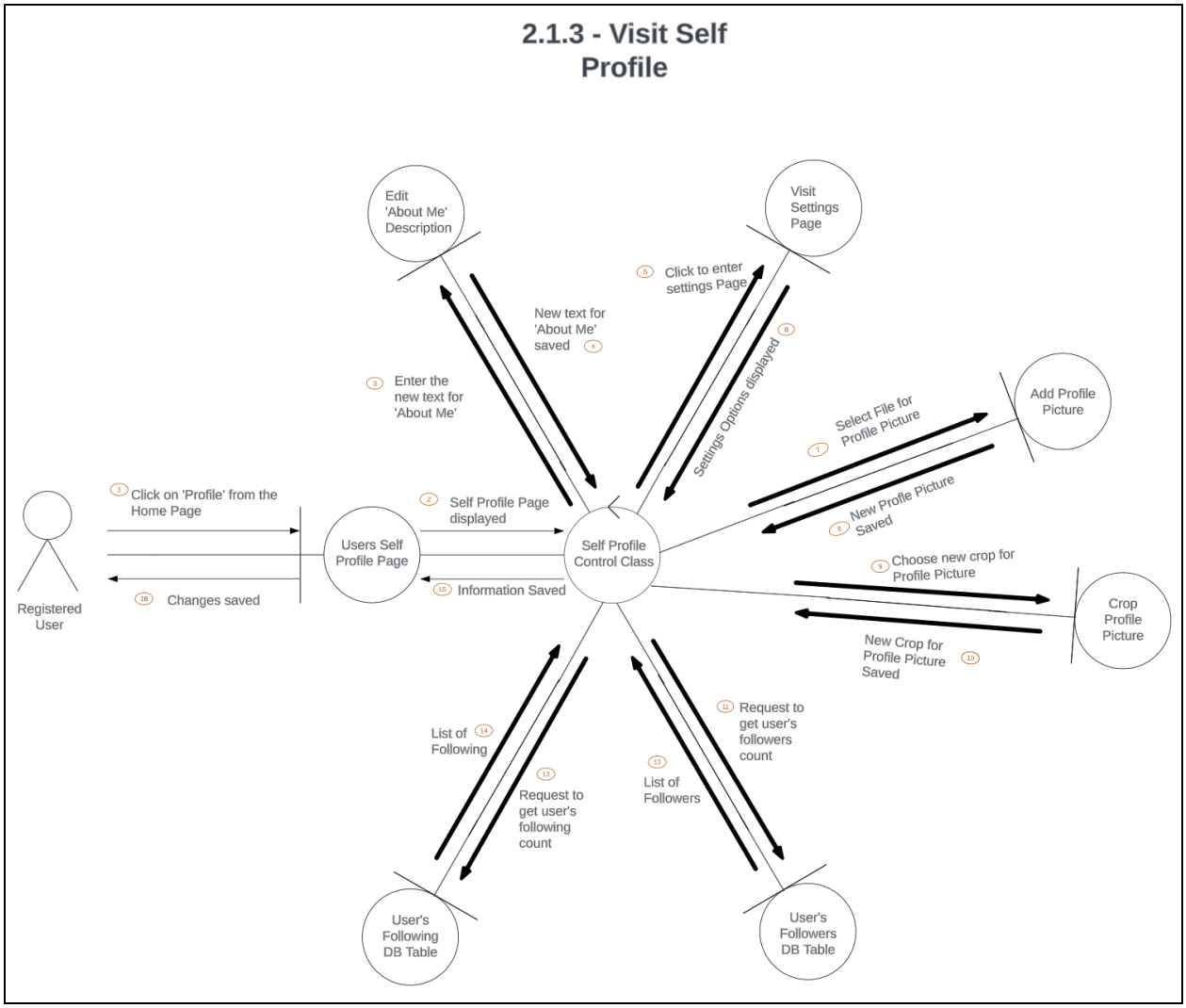
- The returning user provides an invalid username and/or password.



2.1.3 Visit Self Profile

Description: Registered users can view their own profile.

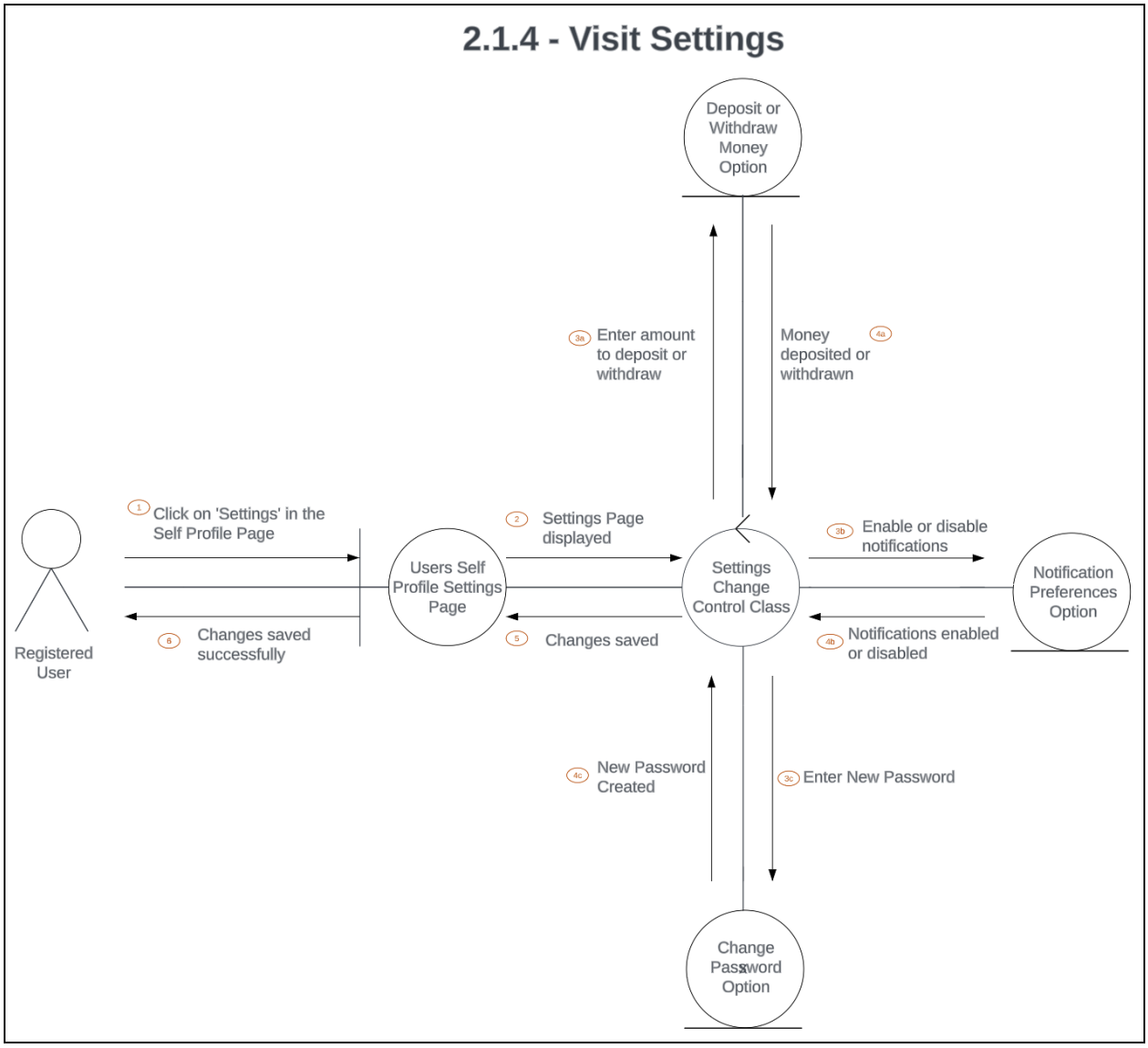
- Normal: Registered users have options to take the following actions within their self profile, including:
  - Edit “About Me”/bio
  - Add profile picture
  - Crop profile picture
  - See number of people you follow
  - See number of followers you have
  - Visit Settings



2.1.4 Visit Settings

Description: In the Settings page of a registered user's Self Profile, they can do the following:

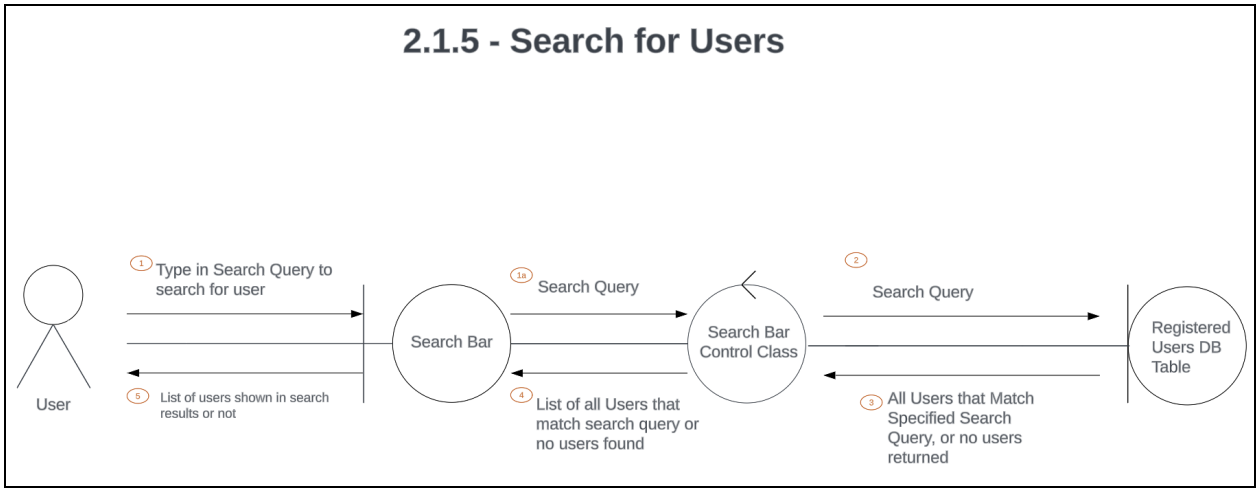
- Normal:
  - Deposit/withdraw money
  - Change password
  - Pick whether or not you want notifications on



2.1.5 Search for User

Description: Any type of user (Surfers, ordinary, trendy, and corporate users) can search for registered users through the search bar.

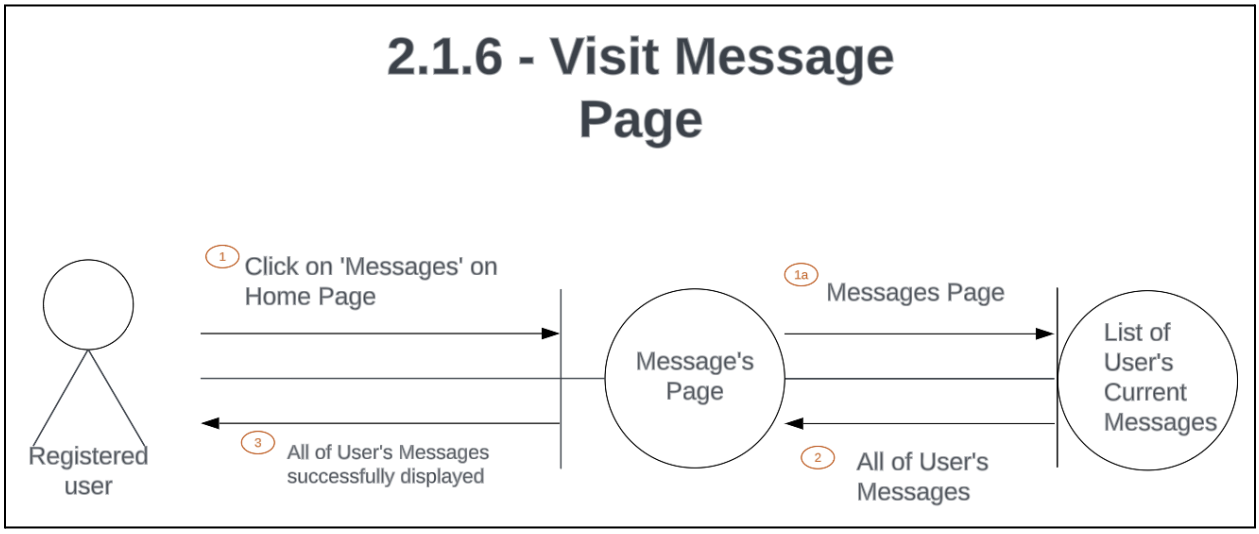
- Normal:
    - The user they are looking for exists
- Exceptional:
    - The user they are looking for does not exist



2.1.6 Visit Message Page

Description: In this message page, registered users can view their 1-to-1 messages with other users. They can also view the warnings they have received from the super user, and information on active disputes.

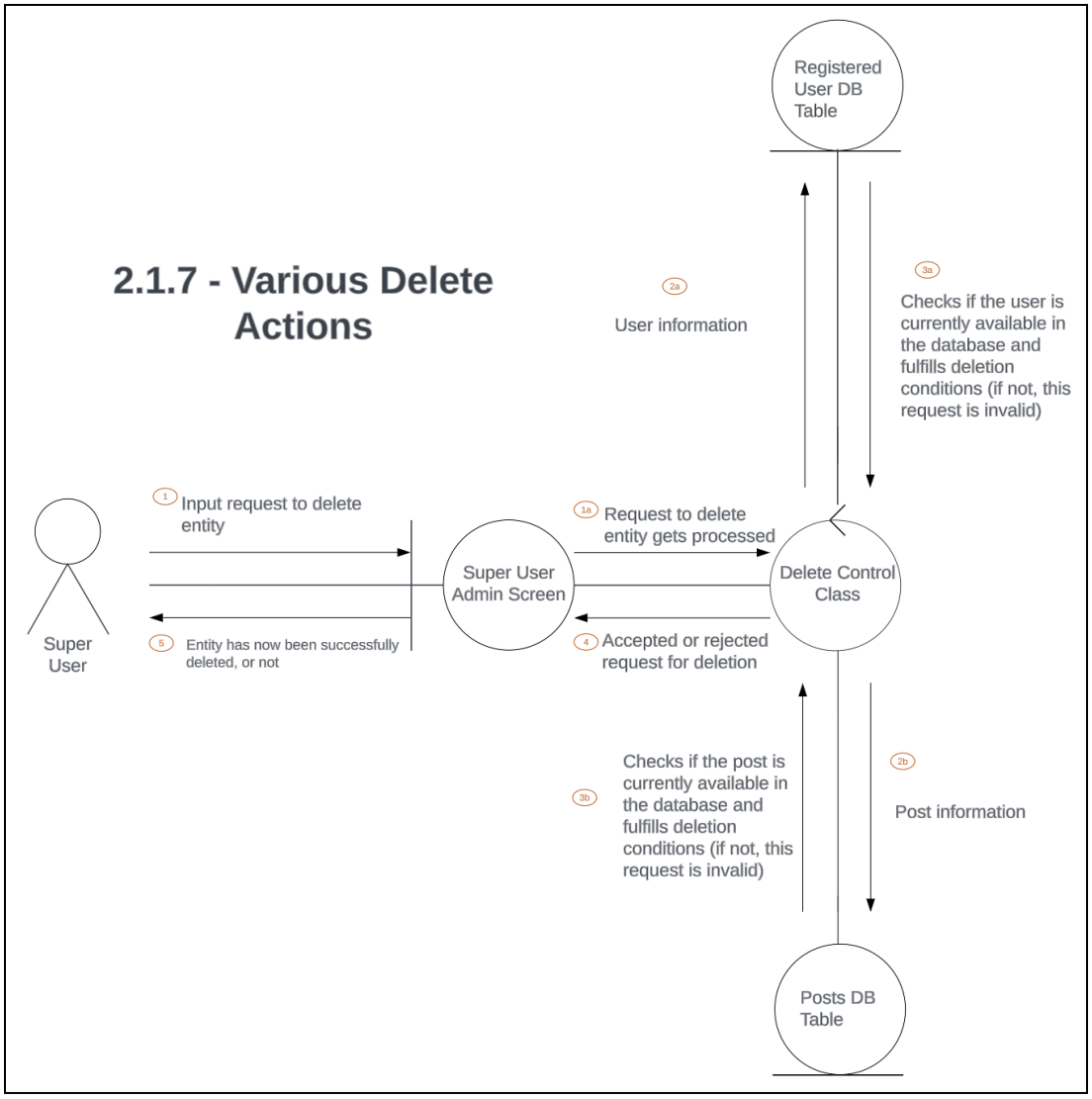
- Normal: They can view the messages they have received. This includes messages between other users, warnings, and disputing warnings. Warnings are issued if a registered user's post contains misinformation and/or if a registered user does not have enough funds to make a post and/or if a registered user has their message blocked due to it containing more than 2 taboo words.



2.1.7 Various Delete Actions (Super user)

Description: The super user can delete various entities.

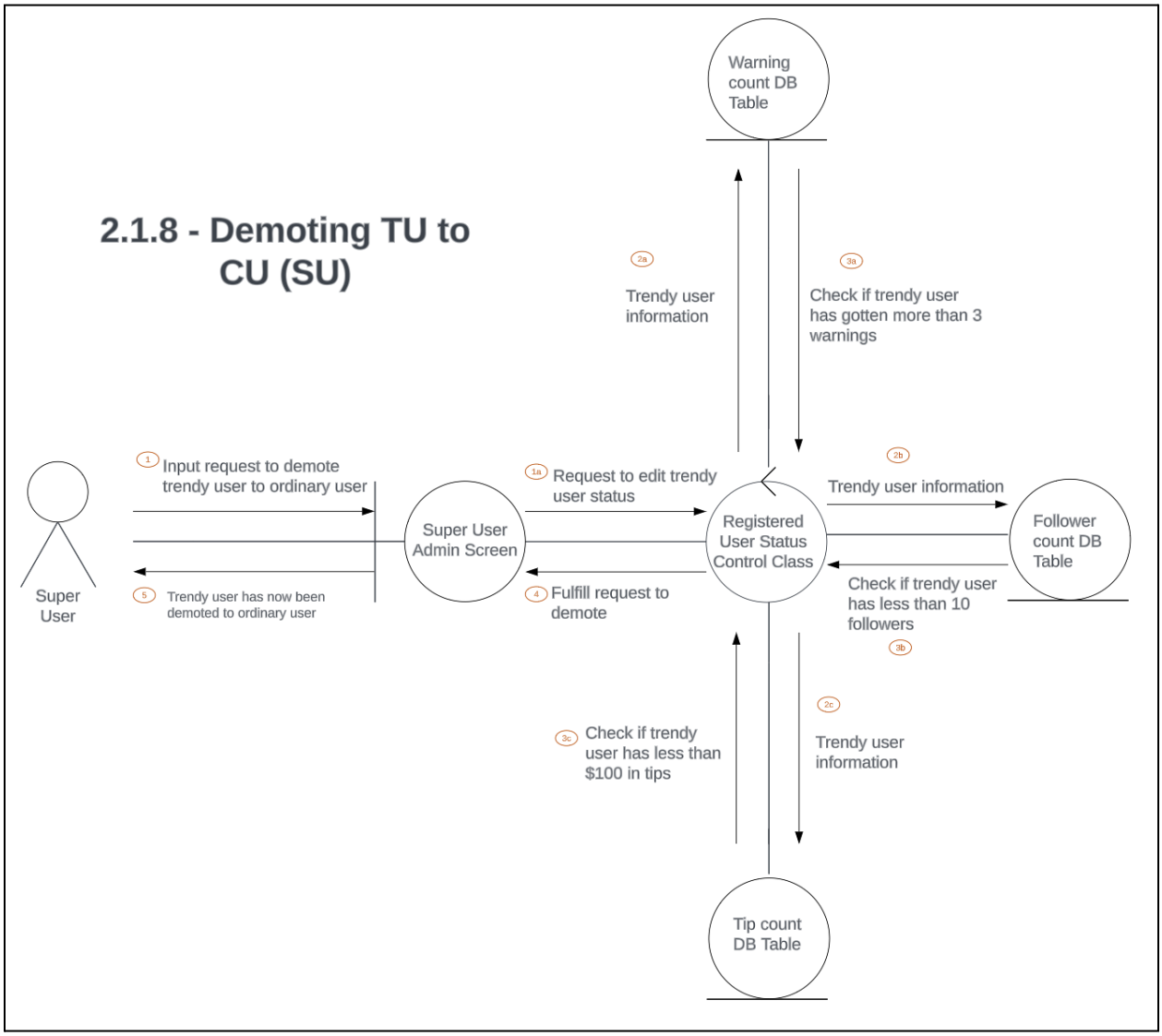
- Normal: The super user can:
    - Delete a user, in which case the username would now be free to use for new registration.
    - Delete a user's post if it contains misinformation.
- Exceptional:
    - The exceptional case is that these users and posts are not available in the database (i.e. if the super user tries to input a username that does not exist in the database, this delete request will not go through.)



2.1.8 Demoting TU to OU (Super user)

Description: The super user can demote a trendy user back to an ordinary user based on certain conditions.

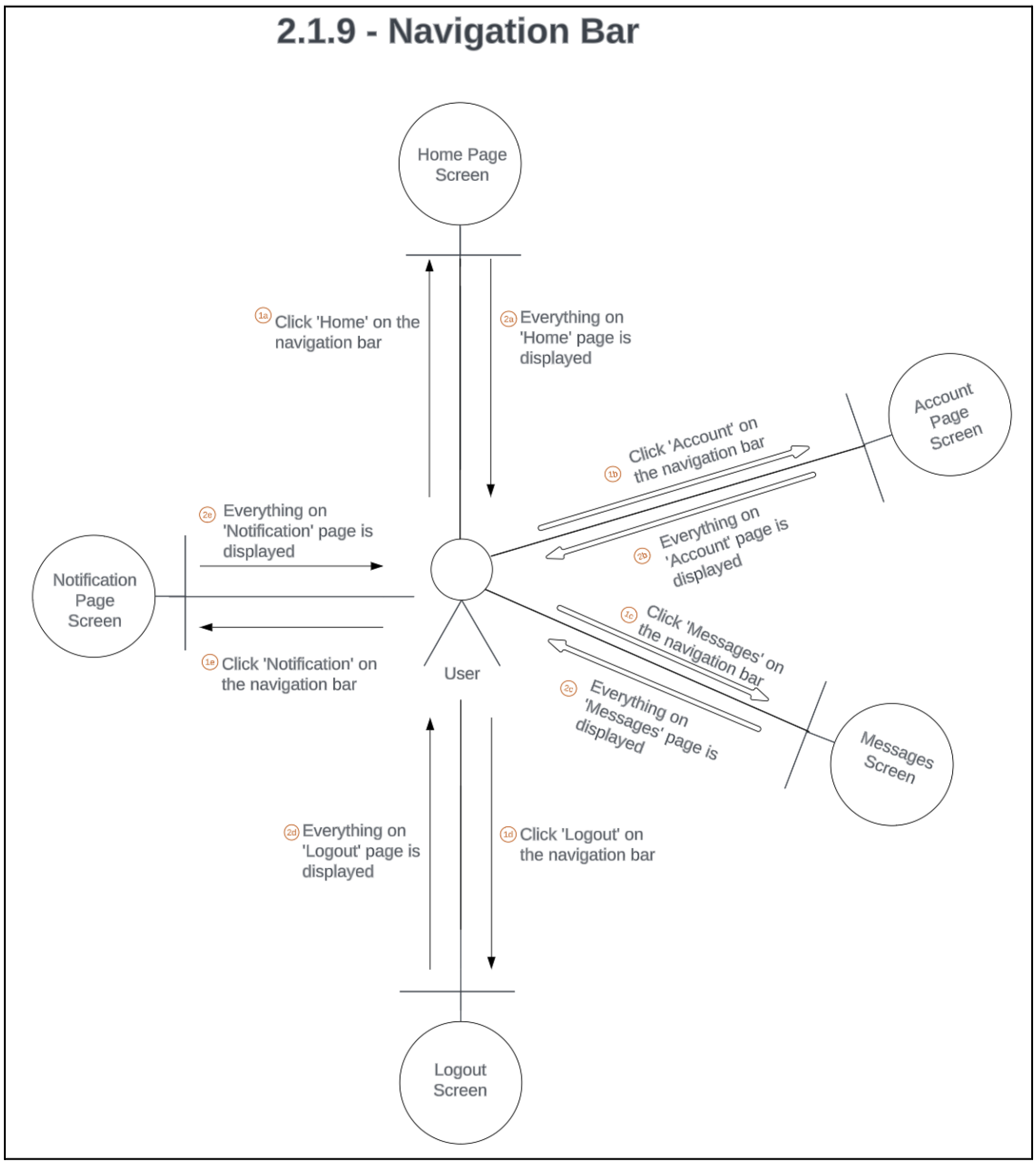
- Normal: The trendy user gets demoted if:
  - They have more than 3 warnings.
  - They have less than 10 followers.
  - They have less than $100 in tips.



2.1.9 Navigation Bar

Description: The navigation bar will display on the side of the users home page in which they will be able to choose from the following options:

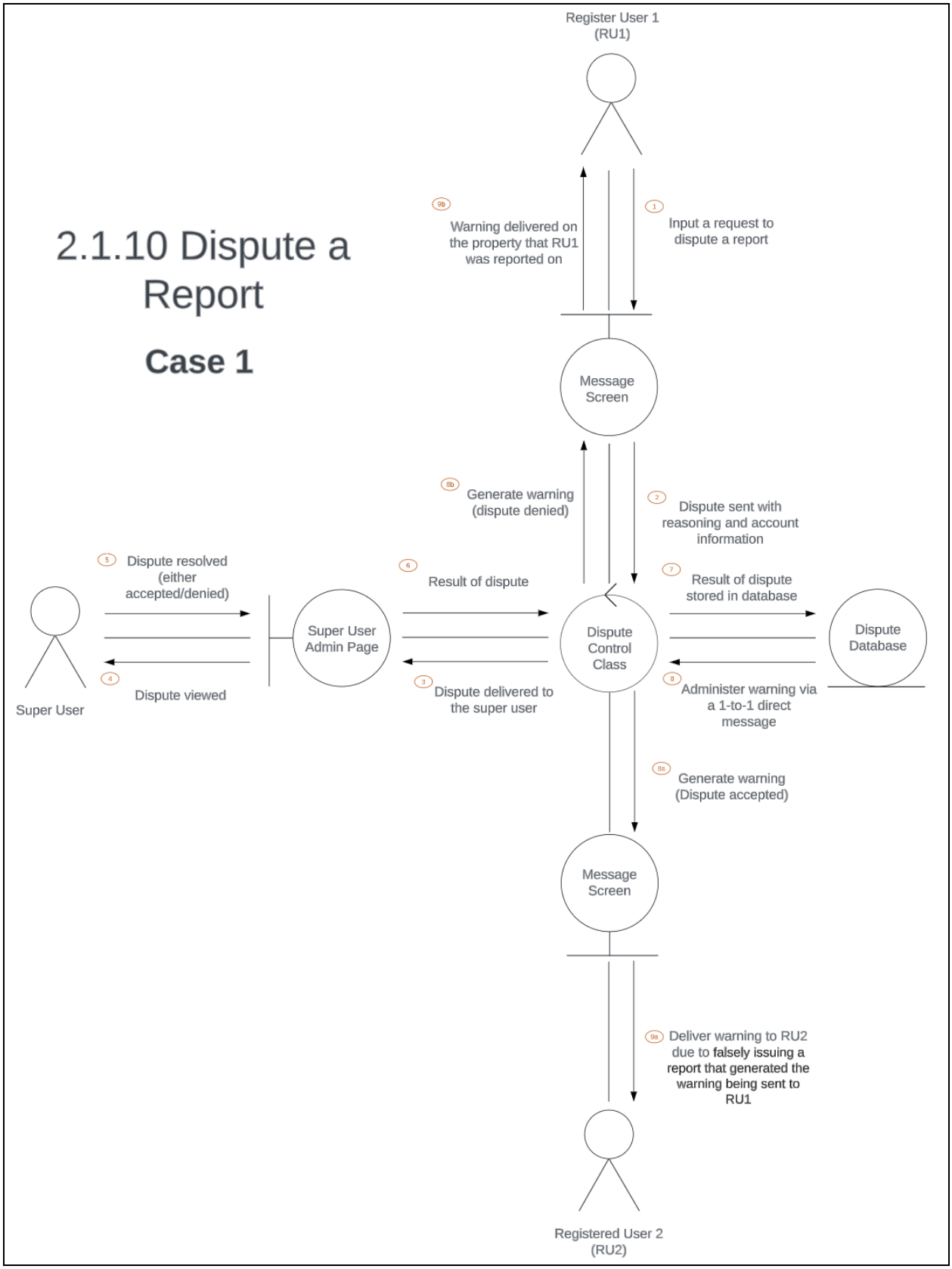
- Normal:
  - Home
  - Messages
  - Notifications
  - Account
  - Logout

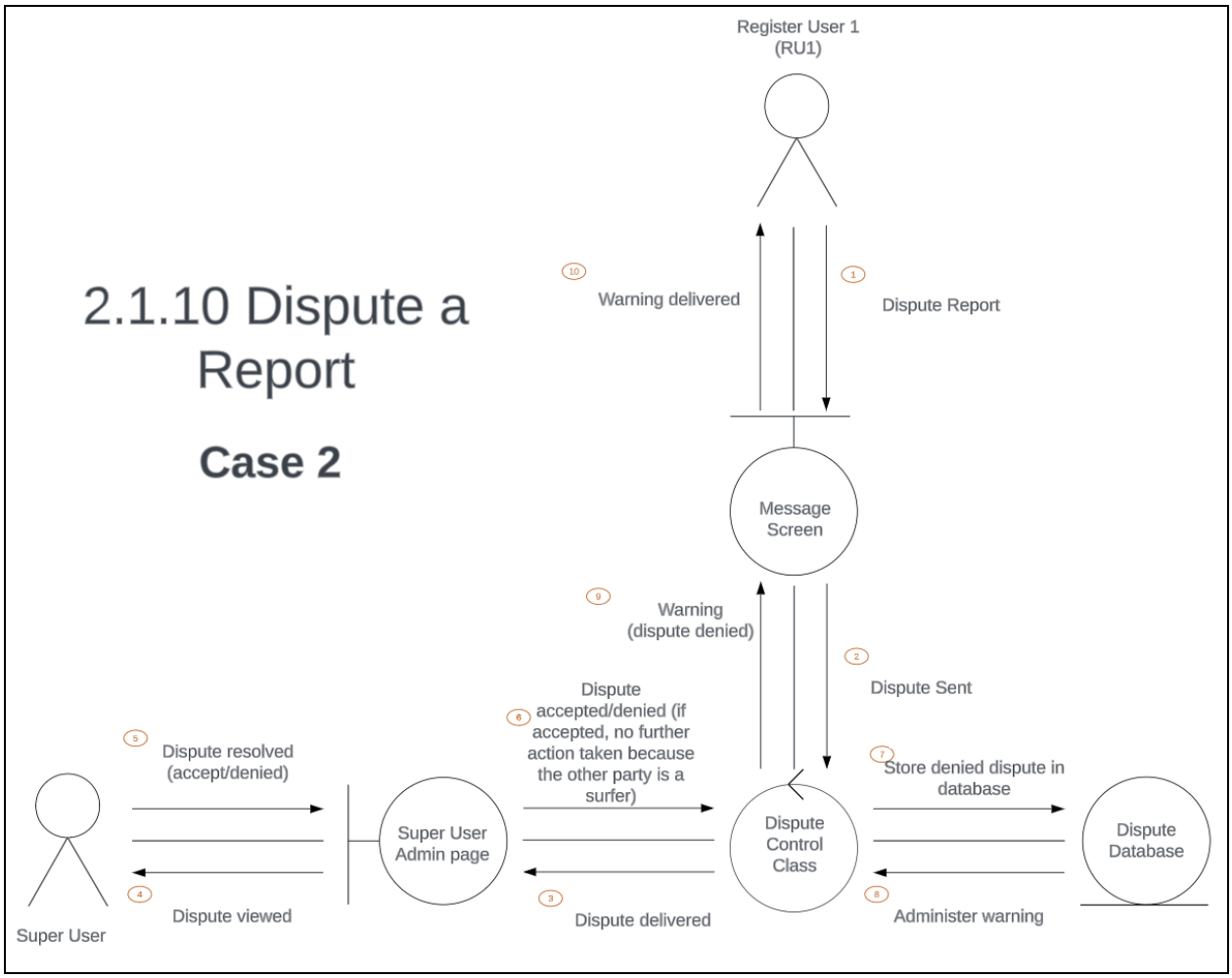


2.1.10 Dispute a Report

Description: A super user (SU) oversees a dispute that happens between two parties. This is a function accessible to a Super User to determine whether a dispute issued by registered users (ordinary users, trendy users, and corporate users) and unregistered users (surfer users) alike is valid or not.

- Normal:
    - Case 1 - registered user (RU1) vs. registered user (RU2): If RU1 files a dispute to RU2, the SU will determine who wins or not. If RU1 wins the dispute, then RU2 will receive a warning of the property they have been reported on from RU1's issued dispute. If RU1 loses the dispute, then RU1 will receive a warning for falsely issuing a dispute, and no further action will be taken on RU2.
    - Case 2 - surfer vs. registered user (RU1): If the surfer files a dispute on RUI and the surfer wins, then RU1 will receive a warning of the property they have been reported on from the surfer's issued dispute. If the surfer loses the dispute, then no further action will be taken on both the surfer and RU1.


2.1.10 Dispute a Report
Case 1
Register User 1 (RU1)
9b Warning delivered on the property that RU1 was reported on
1 Input a request to dispute a report
Message Screen
8b Generate warning (dispute denied)
2 Dispute sent with reasoning and account information
5 Dispute resolved (either accepted/denied)
6 Result of dispute
7 Result of dispute stored in database
Super User
Super User Admin Page
Dispute Control Class
Dispute Database
4 Dispute viewed
3 Dispute delivered to the super user
8 Administer warning via a 1-to-1 direct message
8a Generate warning (Dispute accepted)
Message Screen
9a Deliver warning to RU2 due to falsely issuing a report that generated the warning being sent to RU1
Registered User 2 (RU2)

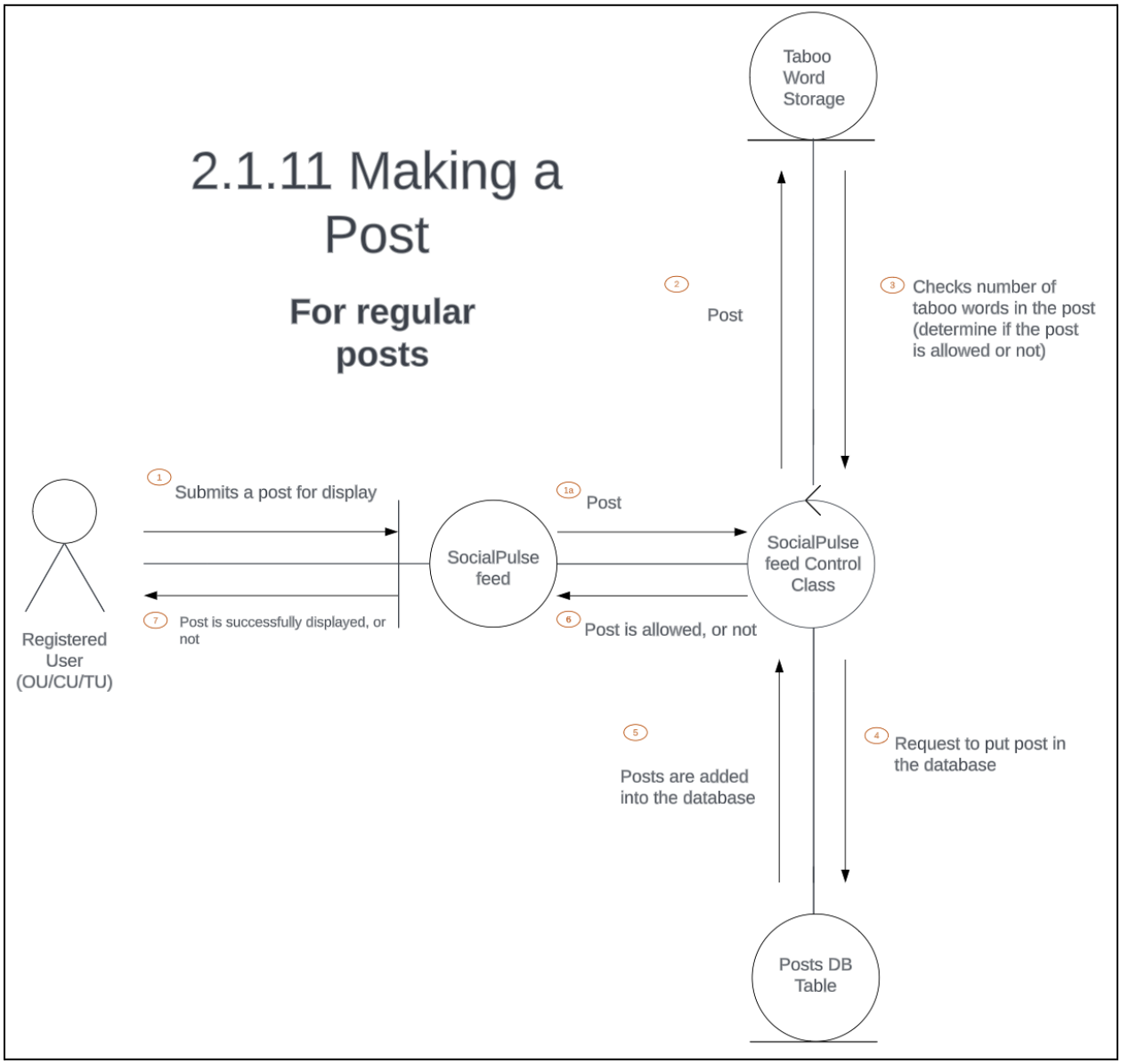


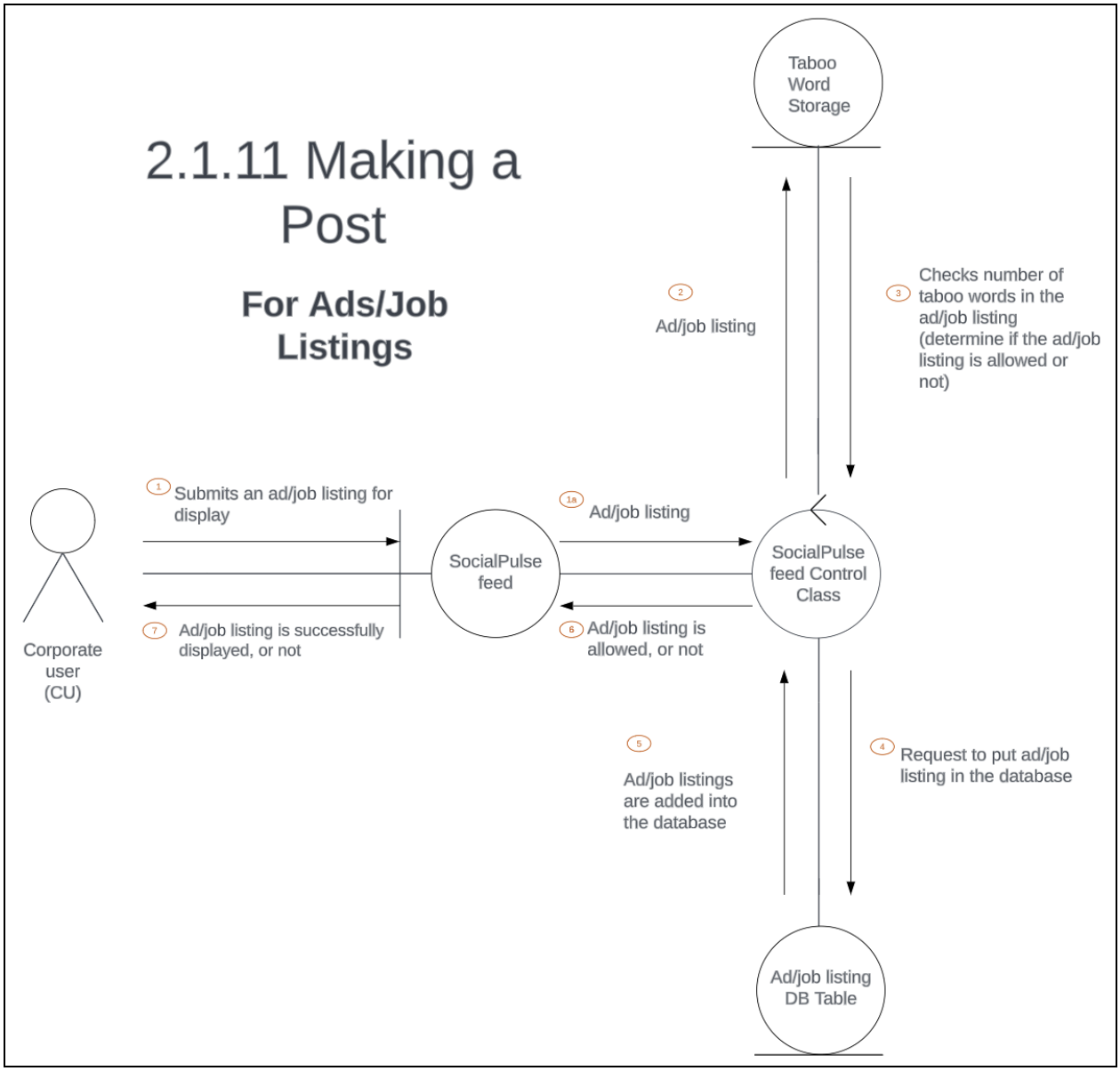
2.1.11 Making a Post

Description: A registered user can make posts that will be shown up on other users' feeds.

- Normal:
  - A registered user (OU/CU/TU) can make a valid post.
    - If the post contains one or two tabooed words from the list managed by the super user, the post will still be considered valid, but the word(s) will be changed to asterisks.
  - A corporate user (CU) has the additional ability of posting ads/job listings.
- Exceptional:
  - A registered user cannot make a valid post if:

- There are more than two taboo words in the post; if this is the case, the author of the post will be blocked from making that post, and is issued a warning.



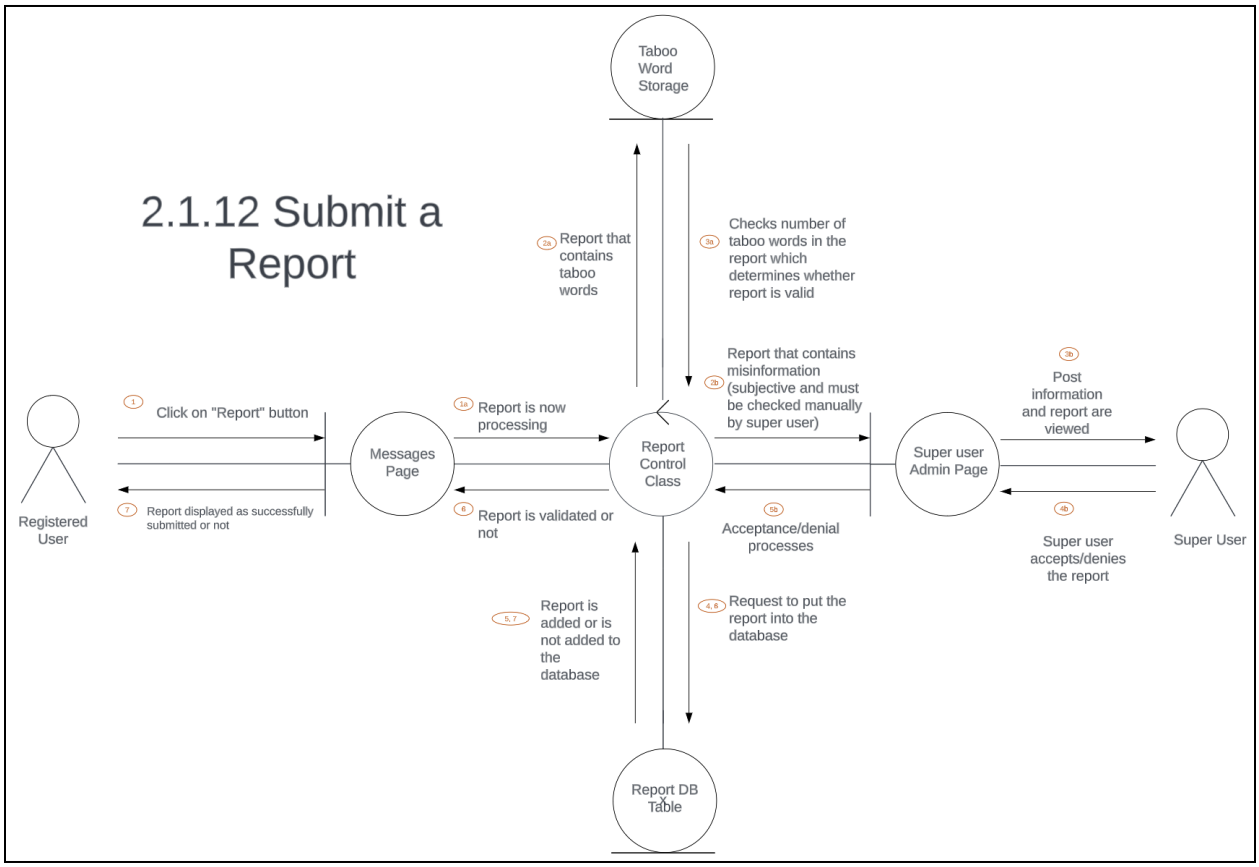


2.1.12 Submitting a Report

Description: Surfers and all registered users can file a report on posts.

- Normal:
    - Surfers and registered users can file a report on posts if they either:
        - Contain taboo words (if there are more than 3 taboo words, the report is considered valid).
        - Contain misinformation - a subjective case, as this requires the super user to manually check the post and then see if this report is valid or not.

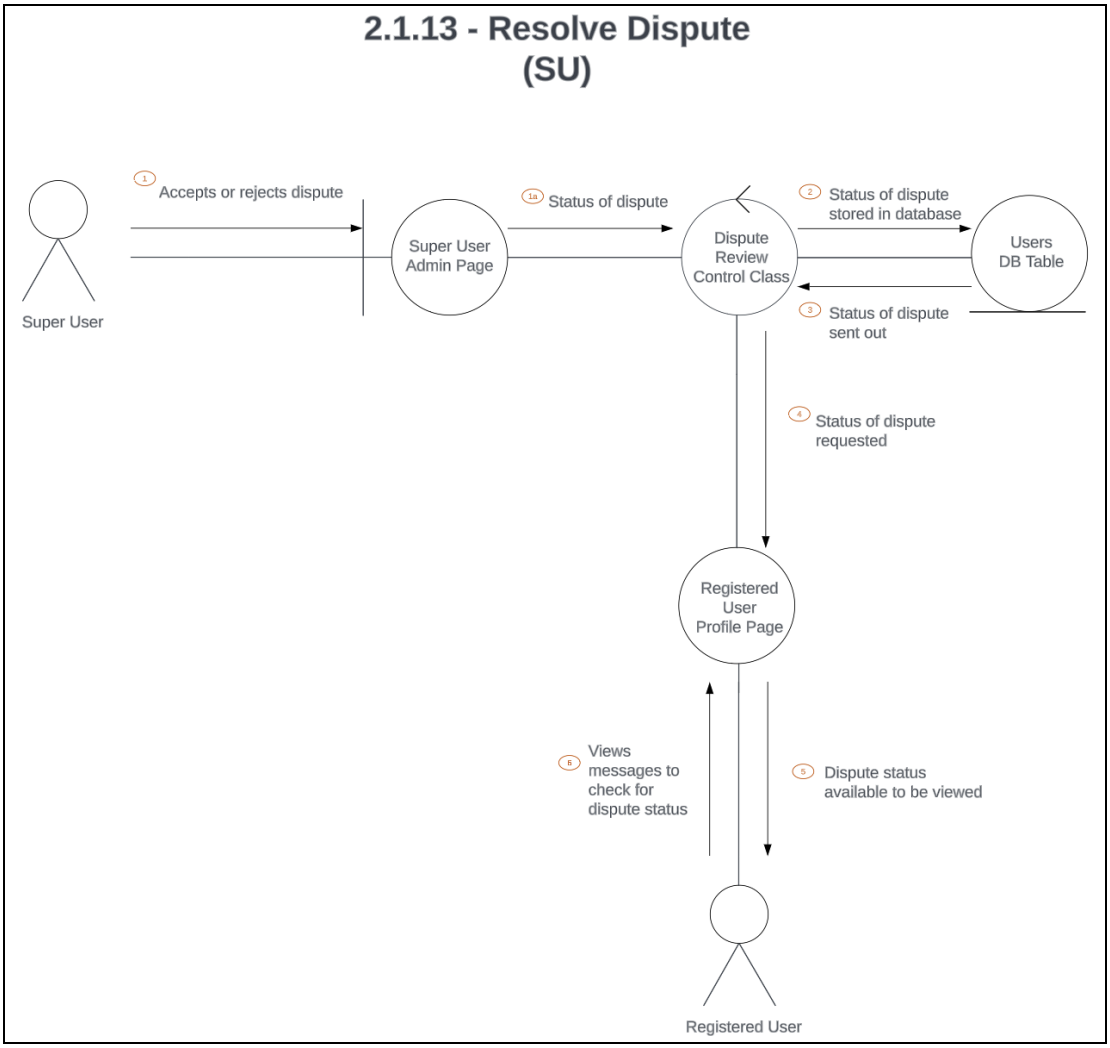
- Exceptional:
    - An exceptional case is if a post has been already reported. If this happens then the super user will be notified of additional reports, which may influence their decision to leave it up or not.



2.1.13 Resolve a Dispute

Description: This is a function accessible to a Super User to determine whether a dispute issued by registered users (ordinary users, trendy users, and corporate users) is valid or not.

- Normal:
    - The Super User either accepts or denies a dispute that is filed by registered users alike Once the result has been processed, the status of the dispute will be displayed within a registered users' messages inbox.

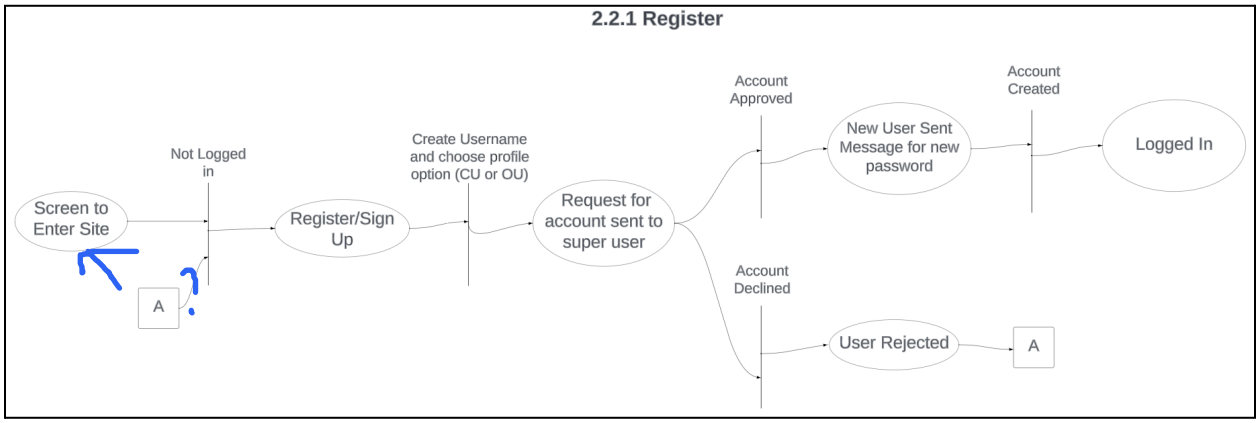

2.1.13 - Resolve Dispute (SU)
Super User
1
Accepts or rejects dispute
Super User Admin Page
1a
Status of dispute
Dispute Review Control Class
2
Status of dispute stored in database
Users DB Table
3
Status of dispute sent out
4
Status of dispute requested
Registered User Profile Page
5
Dispute status available to be viewed
6
Views messages to check for dispute status
Registered User

2.2 Petri-Net Diagram

This section will show various petri-net diagrams that showcase all the state changes and transitions within the system.

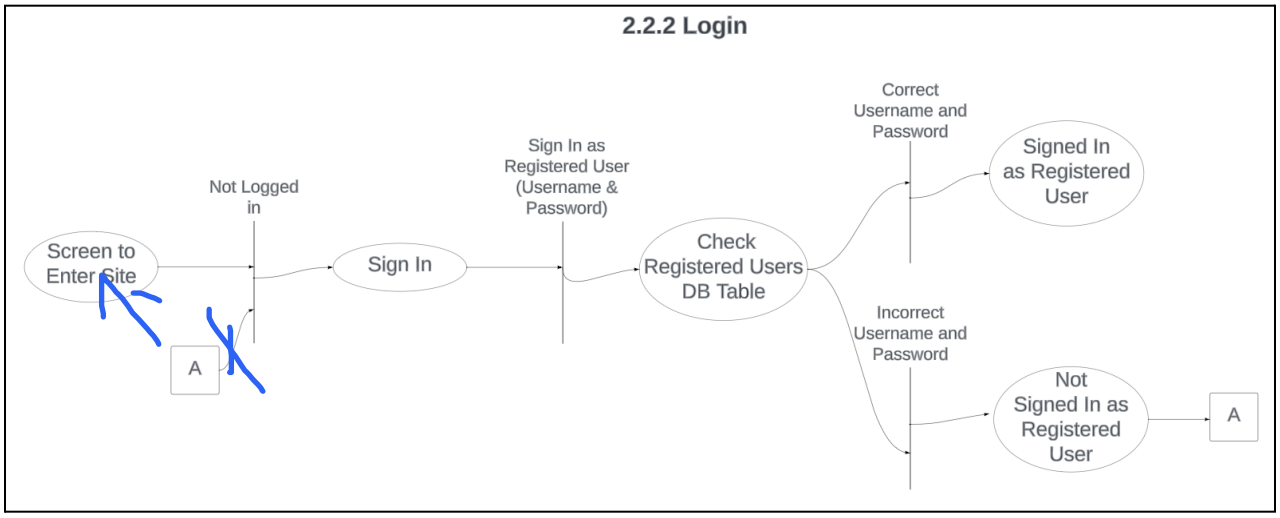
2.2.1 Register

(Use Case: 2.1.1)



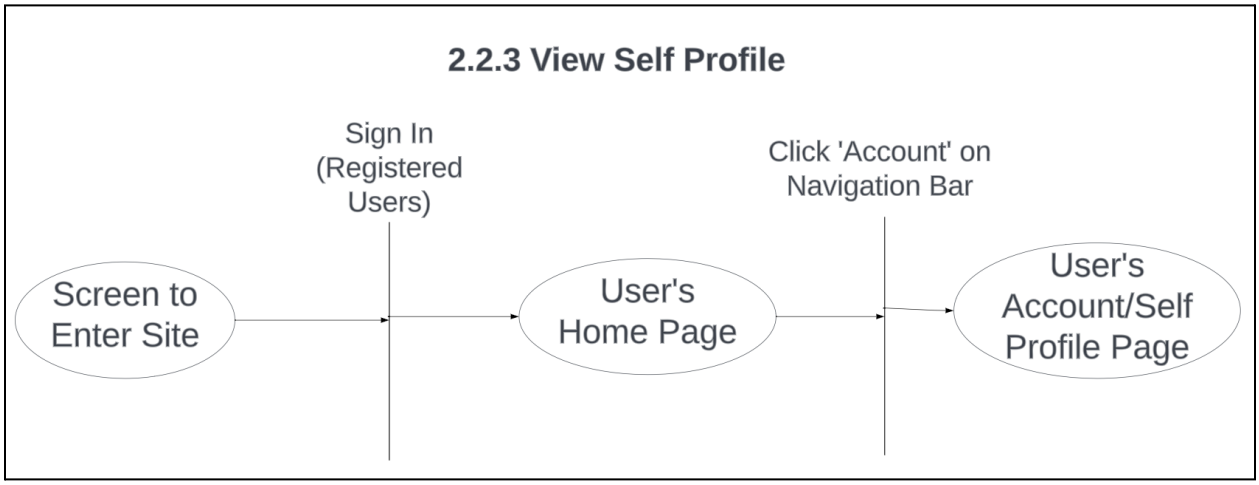
2.2.2 Login

(Use Case: 2.1.2)



2.2.3 View Self Profile

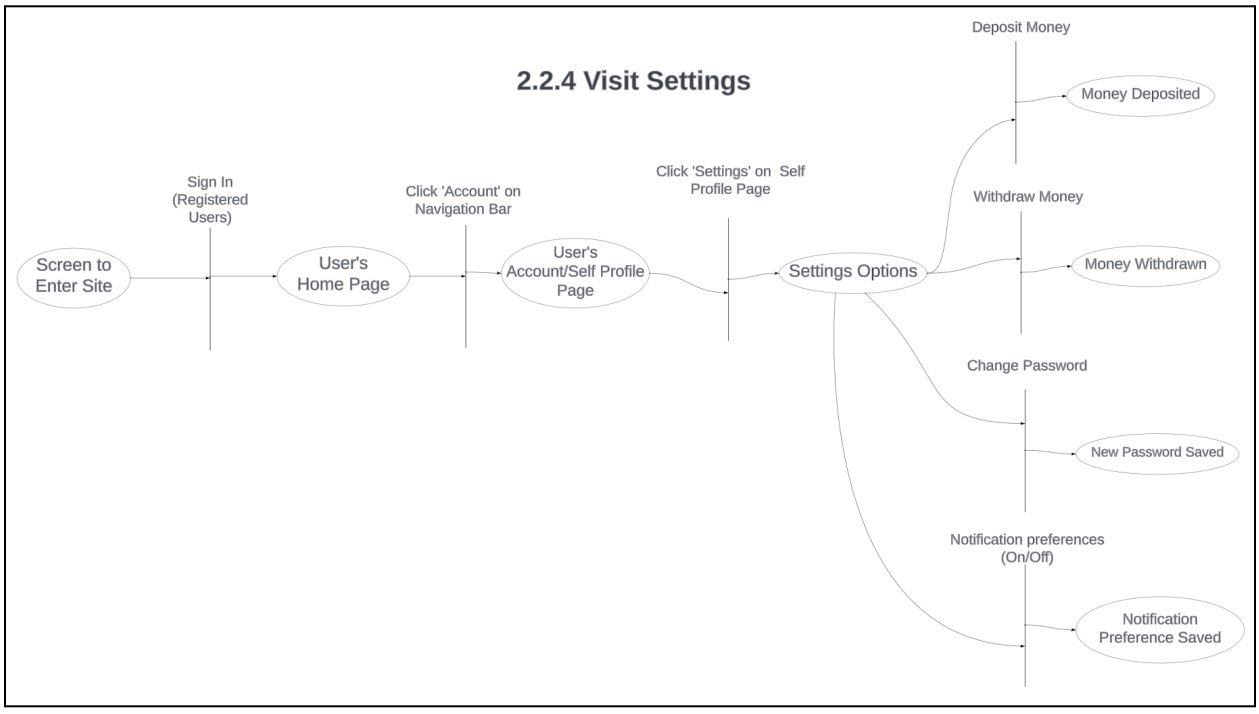
(Use Case: 2.1.3)



this type of petri-net is not useful for its simplicity

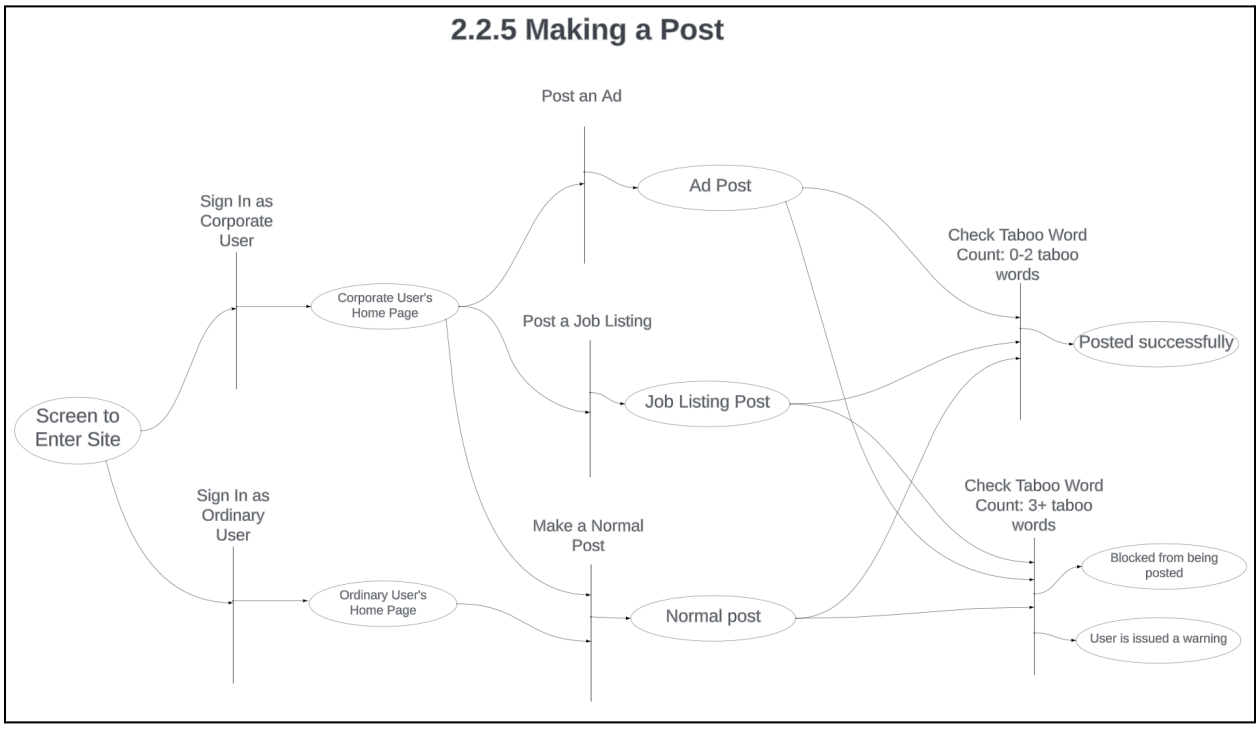
2.2.4 Visit Settings

(Use Case: 2.1.4)

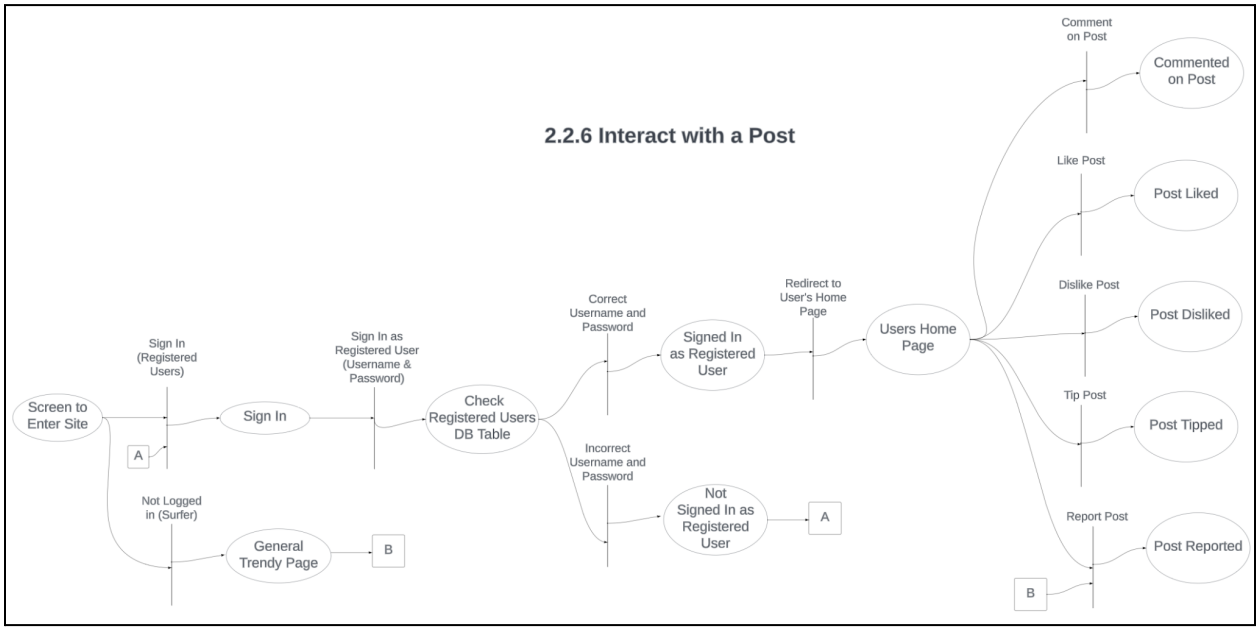


2.2.5 Making a Post

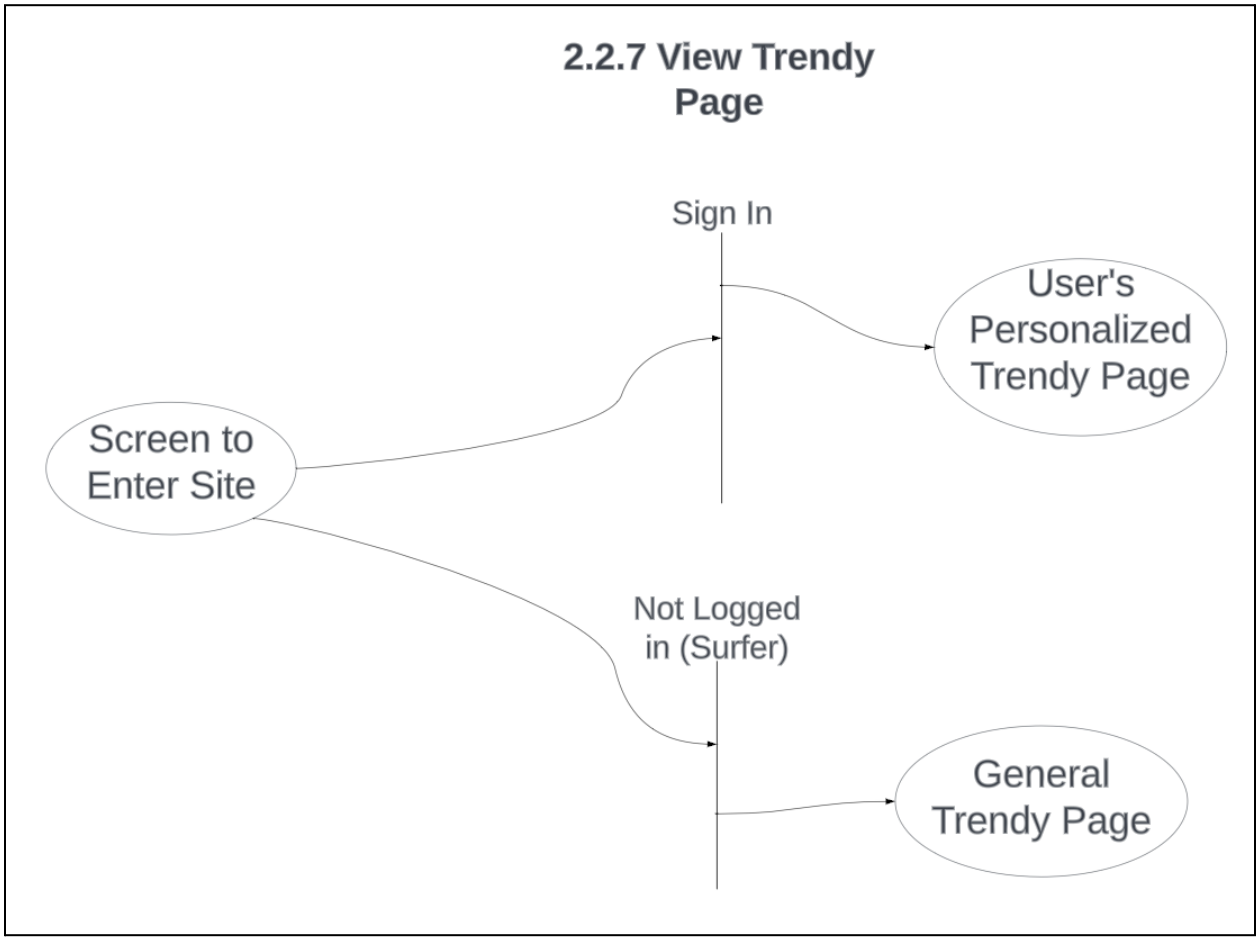
(Use Case: 2.1.11)



2.2.6 Interacting with a Post



2.2.7 View Trendy Page



3. E-R Diagram

The E-R diagram shown showcases all functionalities in the system by displaying its entities and their relationships and attributes.

4. Detailed Design

4.1 Pseudo Code

FRONT-END
REACT + NEXT.JS

Directory: components/Chats/

File: Chat.jsx

Function: Chat

Inputs:

1. chat
2. connectedUsers
3. deleteChat

State: None

Return: JSX element representing the Chat component which shows all previous one to one direct messages.

Actions:

1. Check if the chat partner is online.
2. Render a selectable list item with user information.
3. Set the item as active if the message query matches the current chat.
4. Handle click event to navigate to the messages page for the selected chat.
5. Render a comment with user information.
6. Display the user's avatar in the chat display.
7. Display the user's name and online status. Online status is indicated by a green circle.
8. Display metadata including the last message time.
9. Render a delete icon with a click event to delete a chat history.
10. Display the last message with an ellipsis if it's too long.
11. Render a divider after each chat item.

File: ChatListSearch.jsx

Function: ChatListSearch

Inputs:

1. chats
2. setChats

States:

1. text = ""
2. loading = False
3. results= []
4. router = useRouter()

Effects:

1. Perform a search when the text input changes.
2. Update the loading state during the search.
3. Handle the result selection to add a new chat or navigate to an existing one.

Return: JSX element representing the ChatListSearch component.

Directory: components/Common/

File: CommonInputs.jsx

Function: CommonInputs

Inputs:

1. User information:
    a. Bio
    b. Facebook link
    c. Instagram link
    d. Youtube link
    e. Twitter(X) link
2. handleChange
3. showSocialLinks
4. setShowSocialLinks

State: None

Return: JSX element representing the CommonInputs component.

Actions:

1. Render a Form.Field component for the user's bio.
2. Render a Button component to toggle the visibility of social media links.
3. If showSocialLinks is True, render a Divider component and Form.Input components for each social link.
4. Render a Message component to inform the user that social media links are optional.

File: ImageDropDiv.jsx

Function: CommonInputs

Inputs:

1. highlighted
2. setHighlighted
3. inputRef

4. handleChange
5. mediaPreview
6. setMediaPreview
7. setMedia
8. profilePicUrl

State: signupRoute = router.pathname === "/signup"

Effects:

1. If signupRoute is True, return a Header component with an icon and a text message for the user to click and upload an image.
2. If signupRoute is False, return a span with an Image component and a text message for the user to click and upload an image.

Actions:

1. If the user drops a file, the dropped file is set as the media and a preview of the media is created.
2. If the user is on the signup page, an icon is displayed for the user to click and upload an image.
3. If the user is not on the signup page, an image is displayed with a click handler to upload an image.

Return: JSX element representing the ImageDropDiv component.

File: WelcomeMessage.jsx

Function: HeaderMessage

Input: None.

State: signupRoute = router.pathname === "/signup"

Effects:

1. If signupRoute is True, the header message is "Get Followed", the icon is "user plus", and the content is "Create New Account".
2. If signupRoute is False, the header message is "Glad that you are back!", the icon is "privacy", and the content is "Login with Email and Password".

Return: JSX element representing the HeaderMessage component.

Function: FooterMessage

Input: None.

State: signupRoute = router.pathname === "/signup"

Effects:

1. If signupRoute is True, a warning message is displayed with a help icon and a link to the login page. A divider is also displayed.
2. If signupRoute is False, two messages are displayed:
    a. A red message with a lock icon and a link to the password reset page.

b. A warning message with a help icon and a link to the signup page.

Return: JSX element representing the FooterMessagecomponent.

Directory: components/Home/

File: MessageNotificationModal.jsx

Function: MessageNotificationModal

Inputs:

1. socket
2. showNewMessageModal
3. newMessageModal
4. newMessageReceived
5. user

States:

1. text = ""
2. loading = False

Actions:

1. Render a Modal component with a header, content, and a form for sending a new message.
2. Inside the content of the modal, render a message bubble and a form for sending a new message.
3. Inside the form, render a Form.Input component for the new message text.
4. Below the form, render a link to view all messages and the Instructions component.
5. When the modal is closed, showNewMessageModal is called with False as an argument.
6. When the message is submitted, a message is emitted to the server with the user's id, the id of the user the message is being sent to, and the message text.

Return: JSX element representing the MessageNotificationModal component.

Function: Instructions

Input: username

Action:

1. Render a List component with instructions on how to disable the new message popup.

Return: JSX element representing the Instructions component.

Directory: components/Layout/

File: HeadTags.jsx

Function: HeadTags

Input: None

Actions:

1. Render a Head component with several meta tags and links to CSS stylesheets, logo, and favicon.
2. The title of the webpage is also defined in this function.

Return: JSX element representing the HeadTagscomponent

File: Layout.jsx

Function: Layout

Inputs:

1. children
2. user

States:

1. contextRef = createRef()
2. router = useRouter()
3. messagesRoute = router.pathname === "/messages"

Effects:

1. When a route change starts, nprogress.start() is called.
2. When a route change completes or encounters an error, nprogress.done() is called.

Actions:

1. Render the HeadTags component.
2. If a user is logged in, render a MediaContextProvider component with several nested components and elements.
3. If a user is not logged in, render a Navbar component and a Container component with children.

Return: JSX element representing the Layout component.

File: MobileHeader.jsx

Function: MobileHeader

Input:

1. user

State:

1. router = useRouter()

Actions:

1. Render a Menu component with several Menu.Item components.
2. Each Menu.Item is a link to different routes ("/", "/messages", "/notifications", "/search").

3. The active prop of each Menu.Item is determined by the isActive function, which checks if the current route matches the route of the Menu.Item.
4. If there are unread messages or notifications, the icon of the corresponding Menu.Item is set to "hand point right" indicating unread messages.
5. Render a Dropdown component with several Dropdown.Item components.
6. Each Dropdown.Item is a link to different routes ("/", "/search") or has an onClick event that calls the logoutUser function.

Return: JSX element representing the MobileHeader component.

File: Navbar.jsx

Function: Navbar

Input: None

State:

1. router = useRouter()

Actions:

1. Render a Menu component with two Menu.Item components.
2. Each Menu.Item is a link to different routes ("/login", "/signup").
3. The active prop of each Menu.Item is determined by the isActive variable, which checks if the current route matches the route of the Menu.Item. const isActive = (route) => router.pathname === route;

Return: JSX element representing the Navbar component.

File: NoData.jsx

Function: NoProfilePosts

Input: None

Actions:

1. Render a Message component with an icon, header "Sorry", and content "User has not posted anything yet!".
2. Render a Button component with an icon, content, and href attribute to go to the previous page where the user came from.

Return: JSX element representing the NoProfilePosts component.

Function: NoFollowData

Input:

1. followersComponent
2. followingComponent

Actions:

1. If followersComponent is True, render a Message component with an icon and content "User does not have followers!"

2. If followingComponent is True, render a Message component with an icon and content "User does not follow any user!"

Return: JSX element representing the NoFollowData component.

Function: NoMessages

Input: None

Action:

1. Render a Message component with an icon, header "No Messages?", and content "You have not messaged anyone yet. Search above to message someone!"

Return: JSX element representing the NoMessages component.

Function: NoPosts

Input: None

Action:

1. Render a Message component with an icon, header "Hey!", and content "No Posts. Make sure you have followed someone."

Return: JSX element representing the NoMessages component.

Function: NoProfile

Input: None

Action:

1. Render a Message component with an icon, header "Hey!", and content "No Profile Found."

Return: JSX element representing the NoMessages component.

Function: NoNotifications

Input: None

Action:

1. Render a Message component with an icon and content "No Notifications!"

Return: JSX element representing the NoMessages component.

File: PlaceHolderGroup.jsx

Function: PlaceHolderPosts

Input: None

Actions:

1. Use the lodash range function to generate an array of numbers from 1 to 3.
2. Map over the array and for each number, render a div with a key of the number.
3. Inside the div, render a Placeholder component with a fluid prop.
4. Inside the Placeholder component, render a Placeholder.Header component with an image prop.

5. Inside the Placeholder.Header component, render two Placeholder.Line components.
6. Inside the Placeholder component, render a Placeholder.Paragraph component.
7. Inside the Placeholder.Paragraph component, render four Placeholder.Line components.
8. After the div, render a Divider component with a hidden prop.

Return: JSX element representing the PlaceHolderPosts component.

Function: PlaceHolderSuggestions

Input: None

Actions:

1. Render a List.Item component.
2. Inside the List.Item component, render a Card component with a color prop set to "red".
3. Inside the Card component, render a Placeholder component.
4. Inside the Placeholder component, render a Placeholder.Image component with a square prop.
5. Inside the Card component, render a Card.Content component.
6. Inside the Card.Content component, render a Placeholder component.
7. Inside the Placeholder component, render a Placeholder.Header component.
8. Inside the Placeholder.Header component, render a Placeholder.Line component with a length prop set to "medium".
9. Inside the Card.Content component, render another Card.Content component with an extra prop.
10. Inside the Card.Content component, render a Button component with several props.

Return: JSX element representing the PlaceHolderSuggestions component.

Function: PlaceHolderNotifications

Input: None

Actions:

1. Use the range and map function to generate an array of numbers from 1 to 10.
2. Map over the array and for each number, render a Placeholder component with a key of the number.
3. Inside the Placeholder component, render a Placeholder.Header component with an image prop.

4. Inside the Placeholder.Header component, render a Placeholder.Line component.
5. After the Placeholder component, render a Divider component with a hidden prop.

Return: JSX element representing the PlaceHolderNotifications component.

Function: EndMessage

Input: None

Actions:

1. Render a Container component with a textAlign prop set to "center".
2. Inside the Container component, render an Icon component with a name prop set to "hand point up" and a size prop set to "large".
3. After the Icon component, render a Divider component with a hidden prop.

Return: JSX element representing the EndMessage component.

Function: LikesPlaceHolder

Input: None

Actions:

1. Use the lodash range function to generate an array of numbers from 1 to 6.
2. Map over the array and for each number, render a Placeholder component with a key of the number and a style prop set to { minWidth: "200px" }.
3. Inside the Placeholder component, render a Placeholder.Header component with an image prop.
4. Inside the Placeholder.Header component, render a Placeholder.Line component with a length prop set to "full".

Return: JSX element representing the LikesPlaceHolder component.

File: Search.jsx

Function: ResultRenderer

Inputs:

1. title
2. image

Actions:

1. Render a List component with a List.Item component.
2. Inside the List.Item component, render an Image component with src={image} and a List.Content component with header={title}.

Return: JSX element representing the ResultRenderer component.

Function: SearchComponent

Input: None

States:

1. text = ""
2. loading = false
3. results = []
4. searchTimer = null

Effect:

1. When the text state changes, if the length of the text is 0 and loading is true, set loading to false.

Actions:

1. Define a handleChange function that makes an axios GET request to the search API with the current text as a parameter.
2. If the length of the response data is 0, set results to an empty array and set loading to false.
3. Map over the response data and set results to the mapped results.
4. Define an onBlur function that sets loading to false and text to an empty string.
5. Define an onSearchChange function that sets loading to true, clears the searchTimer, sets text to the current value, and sets searchTimer to a new timeout that calls handleChange after 2 seconds.
6. Define an onResultSelect function that navigates to the profile of the selected user.

Return: JSX element representing the SearchComponent component.

File: SideMenu.jsx

Function: SideMenu

Inputs:

1. user object = { unreadNotification, email, unreadMessage, username }
2. pc = True (boolean flag to either display side menu or not)

State:

1. router = useRouter()

Actions:

1. Render a List component with several List.Item components.
2. Each List.Item is a link to different routes ("/", "/messages", "/notifications", /${username}).
3. The active prop of each List.Item is determined by the isActive function, which checks if the current route matches the route of the List.Item.

4. Inside each List.Item, render an Icon component and a List.Content component.
5. The color of the Icon component is determined by the isActive function and the unreadMessage and unreadNotification props.
6. After the List.Item, render a br element.
7. After the last List.Item, render another List.Item with an onClick function that calls the logoutUser function with the email prop.

Return: JSX element representing the SideMenu component.

Directory: components/Messages/

File: Banner.jsx

Function: Banner

Input:

1. bannerData

Actions:

1. Destructure the bannerData object to get the name and profilePicUrl properties.
2. Render a Segment component with a color prop set to "green" and an attached prop set to "top".
3. Inside the Segment component, render a Grid component.
4. Inside the Grid component, render a Grid.Column component with a floated prop set to "left" and a width prop set to 14.
5. Inside the Grid.Column component, render a h4 element.
6. Inside the h4 element, render an Image component with an avatar prop and a src prop set to the profilePicUrl property.
7. After the Image component, render the name property.

Return: JSX element representing the Banner component.

File: Message.jsx

Function: Message

Inputs:

1. message
2. user
3. deleteMsg
4. bannerProfilePic
5. divRef

State:

1. deleteIcon = false

Actions:

1. Render a div that depends on whether the message sender is the current user.
2. Inside the div, render an img element with src that depends on whether the message sender is the current user.
3. After the img element, render another div with a className that depends on whether the message sender is the current user.
4. Inside the div, render the message text.
5. If deleteIcon is True, render a Popup component with a trigger that is an Icon component with a name prop set to "trash" for a trash icon and a color prop set to "red" for the icon's color, and an onClick function that calls deleteMsg with the message id.
6. Finally, after the div, render a span with a className that depends on whether the message sender is the current user.
7. Inside the span, render the time of the message.

Return: JSX element representing the Message component.

File: MessageInputField.jsx

Function: MessageInputField

Input:

1. sendMsg

State:

1. test = ""
2. loading = False

Effects:

1. Calls sendMsg function when the form is submitted, passing the current text as an argument.
2. Resets the text state to an empty string after sending the message.

Action:

1. Renders a sticky message input field at the bottom of the screen

Return: JSX element representing the MessageInputField component.

Directory: components/Notifications/

File: CommentNotification.jsx

Function: CommentNotification

Input:

1. notification

Action:

1. Renders a notification component for comments on a user's post.

Return: JSX element representing the CommentNotification component.

File: FollowerNotification.jsx

Function: FollowerNotification

Inputs:

1. notification
2. loggedUserFollowStats
3. setUserFollowStats

State:

1. Disabled = False

Effect:

1. Handles the follow/unfollow action on button click, updating follow stats accordingly.

Action:

1. Renders a notification component for follower-related activities.

Return: JSX element representing the FollowerNotification component.

File: LikeNotification.jsx

Function: LikeNotification

Input:

1. notification

Action:

1. Renders a notification component for likes on a user's post.

Return: JSX element representing the LikeNotification component.

Directory: components/Post/

File: CardPost.jsx

Function: CardPost

Inputs:

1. post
2. user
3. setPosts
4. setShowToastr

States:

1. likes
2. isLiked
3. comments
4. error
5. showModal

Effects:

1. Handles like and delete post actions.
2. Manages a modal for displaying post images.

Action:

1. Renders a card component for displaying a post, including user information, post content, like and comment functionality, and additional features for the post owner or users with special roles.

Return: JSX element representing the CardPost component.

File: CommentInputField.jsx

Function: CommentInputField

Inputs:

1. postId
2. user
3. setComments

States:

1. text = ""
2. loading = False

Effect:

1. Handles comment submission and updates the comment state.

Action:

1. Renders a comment input field component for adding comments to a post.

Return: JSX element representing the CommentInputField component.

File: CreatePost.jsx

Function: CreatePost

Inputs:

1. user
2. setPosts

States:

1. newPost = { text: "", location: "" }
2. loading = false
3. inputRef
4. error = null
5. highlighted = False
6. media = null
7. mediaPreview = null
8. showModal = False

Effect:

1. Handles the image upload, cropping, and post submission.

Action:

1. Renders a form component for creating a new post, including text, location, and image upload functionality.

Return: JSX element representing the CreatePost component.

File: CropImageModal.jsx

Function: CropImageModal

Inputs:

1. mediaPreview
2. setMedia
3. showModal
4. setShowModal

State:

1. cropper

Effects:

1. Listens for key events to control cropper actions.
2. Destroys the cropper instance when the modal is closed.

Action:

1. Renders a modal component for cropping an image before upload.

Return: JSX element representing the CropImageModal component.

File: ImageModal.jsx

Function: ImageModal

Inputs:

1. post
2. user
3. setLikes
4. likes
5. isLiked
6. comments
7. setComments

Effects:

1. Handles like actions on the post.
2. Displays the post's image, user information, post content, like count, list of likes, and a subset of comments.
3. Provides an option to view more comments in a modal.

Action:

1. Renders a modal component for displaying a post with an image, including user information, post content, like and comment functionality, and additional features.

Return: JSX element representing the ImageModal component.

File: LikesList.jsx

Function: LikesList

Inputs:

1. postId
2. Trigger

States:

1. likesList = []
2. loading = False

Effect:

1. Fetches and updates the list of users who liked the post.

Action:

1. Renders a Popup component displaying a list of users who liked a post when triggered.

Return: JSX element representing the LikesList component.

File: NoImageModal.jsx

Function: NoImageModal

Inputs:

1. post
2. user
3. setLikes
4. likes
5. isLiked
6. comments
7. setComments

Action:

1. Renders a Card component for displaying a post without an image, including user information, post content, like and comment functionality, and additional features.

Return: JSX element representing the NoImageModal component.

File: PostComments.jsx

Function: PostComments

Inputs:

1. comment
2. user
3. setComments
4. postId

State:

1. disabled = False

Effect:

1. Updates the disabled status of the delete button.

Action:

1. Renders a Comment component for a given comment, including user information, comment text, time metadata, and an option to delete the comment for admins.

Return: JSX element representing the PostComments component.

Directory: components/Profile/

File: Followers.jsx

Function: Followers

Inputs:

1. user
2. loggedUserFollowStats
3. setUserFollowStats
4. profileUserId

State:

1. followers = []
2. loading = False
3. followLoading = False

Effect:

1. Fetches followers data on component mount.

Action:

1. Renders a list of followers for a given user, including user information, a button to follow/unfollow each follower, and a loading spinner while data is being fetched.

Return: JSX element representing the Followers component.

File: Following.jsx

Function: Following

Inputs:

1. user
2. loggedUserFollowStats
3. setUserFollowStats
4. profileUserId

State:

1. following = []
2. loading = False
3. followLoading = False

Effect:

1. Fetches following data on component mount.

Action:

1. Renders a list of followers for a given user, including user information, a button to follow/unfollow each follower, and a loading spinner while data is being fetched.

Return: JSX element representing the Following component.

File: ProfileHeader.jsx

Function: ProfileHeader

Inputs:

1. profile
2. ownAccount
3. loggedUserFollowStats
4. setUserFollowStats

State:

1. loading = False

Actions:

1. Renders the header section of a user profile, including the user's name, bio, social media links, and a profile picture.
2. Displays a follow button for logged-in users to follow/unfollow the profile user, with loading indicator during the action.
3. If it's the user's own profile, the follow button is not displayed.

Return: JSX element representing the ProfileHeader component.

File: ProfileMenuTabs.jsx

Function: ProfileMenuTabs

Inputs:

1. activeItem
2. handleItemClick
3. followersLength
4. followingLength
5. ownAccount
6. loggedUserFollowStats

Actions:

1. Renders a menu with tabs for navigating different sections of a user profile.
2. Tabs include "Profile," "Followers", and "Following"..
3. Additional tabs ("Update Profile", "Account Balance", and "Settings") are shown for the user's own profile.
4. Displays the number of followers or following users in the respective tabs.

5. Handles tab switching through the `handleItemClick` function.

Return: JSX element representing the ProfileMenuTabs component.

File: Settings.jsx

Function: Settings

Input:

1. newMessagePopup

States:

1. passwordFields = False
2. newMessageSettings = False
3. popupSetting
4. success = False

Effect:

1. Displays a success message for 3 seconds after successful updates.

Actions:

1. Renders the settings page for the user, allowing password updates and toggling the new message popup.
2. Updates the new message popup setting and displays a success message accordingly.

Return: JSX element representing the Settings component.

Function: UpdatePassword

Input:

1. setSucess
2. showPasswordFields

States:

1. loading= False
2. errorMsg= null
3. userPasswords = { currentPassword: "", newPassword: "" }
4. typed = { field1: false, field2: false }

Effect:

1. Displays an error message for 5 seconds after a failed password update.

Actions:

1. Renders a form for updating the user's password, with the option to show/hide password fields.
2. Updates the password and handles form submission.

Return: JSX element representing the UpdatePassword component.

File: Balance.jsx

Function: Balance

Input:

1. balance

States:

1. setSucess
2. showBalance

Effect:

1. Displays a success message for 3 seconds after successful account balance updates.

Actions:

1. Renders the balance page for the user, allowing balance updates.
2. Updates the balance amount and displays a success message accordingly.

Return: JSX element representing the Balance component.

File: UpdateProfile.jsx

Function: UpdateProfile

Input:

1. Profile

States:

1. profile = { profilePicUrl, bio, facebook, youtube, instagram, twitter }
2. errorMsg = null
3. loading = false
4. showSocialLinks = false
5. highlighted = false
6. inputRef
7. media = null
8. mediaPreview = null

Actions:

1. Renders a form for updating the user's profile information, including bio, social media links, and profile picture.
2. Allows users to upload a new profile picture with image preview.
3. Displays error message in case of form submission failure.

Return: JSX element representing the UpdateProfile component.

Note:

The pseudo code above is to create the components that will be used to create the pages of the application. Components are utilized so pages don’t have to be rewritten multiple times over for small changes. Components make the application more robust and help in the reusability of code.

BACK-END
NODE.JS + EXPRESS.JS

File:
authMiddleware.js
Description:
The file authMiddleware.js is a middleware for Express.js that checks the authorization header in the incoming request, verifies the JWT (JSON Web Token) using a secret key, and extracts the user ID from the token. If the token is valid, it attaches the user ID to the request (req.userId) and allows the request to proceed to the next middleware or route handler. If the token is invalid or missing, it responds with a 401 status code and "Unauthorized!" message.
Pseudocode:

```
const jwt = require("jsonwebtoken");
module.exports = (req, res, next) => {
  try {
    if (!req.headers.authorization) {
      return res.status(401).send("Unauthorized!");
    }
    const { userId } = jwt.verify(
      req.headers.authorization,
      process.env.jwtSecret
    );
    req.userId = userId;
    next();
  } catch (error) {
    console.error(error);
    return res.status(401).send("Unauthorized!");
  }
};
```

File:

authUser.js

Description:

The file authUser.js is for user authentication, registration, and redirection in a Next.js application. It uses Axios for making HTTP requests, Router for client-side navigation, and js-cookie for handling cookies. The registerUser function sends a POST request to register a new user, loginUser function sends a POST request to authenticate a user, and redirectUser function redirects the user either on the server or client side. The setToken function sets the user token in a cookie upon successful authentication, and logoutUser logs out the user by removing the token cookie and redirecting to the login page

Pseudocode:

```
# Function to register a new user
function registerUser(user, profilePicUrl, setError, setLoading):
    try:
        # Send a POST request to the signup API endpoint
        res = axios.post(`${baseUrl}/api/signup`, { user, profilePicUrl })

        # Set the user token upon successful registration
        setToken(res.data)

    catch (error):
        # Handle errors and set error message
        errorMsg = catchErrors(error)
        setError(errorMsg)

    # Set loading to false
    setLoading(false)

# Function to authenticate a user
function loginUser(user, setError, setLoading):
    # Set loading to true
    setLoading(true)

    try:
        # Send a POST request to the authentication API endpoint
        res = axios.post(`${baseUrl}/api/auth`, { user })

        # Set the user token upon successful authentication
        setToken(res.data)
```

```
    catch (error):
        # Handle errors and set error message
        errorMsg = catchErrors(error)
        setError(errorMsg)

    # Set loading to false
    setLoading(false)

# Function to redirect the user
function redirectUser(ctx, location):
    if (ctx.req):
        # Redirect on the server-side
        ctx.res.writeHead(302, { Location: location })
        ctx.res.end()
    else:
        # Redirect on the client-side
        Router.push(location)

# Function to set the user token in a cookie
function setToken(token):
    cookie.set("token", token)
    Router.push("/")

# Function to log out the user
function logoutUser(email):
    cookie.set("userEmail", email)
    cookie.remove("token")
    Router.push("/login")
    Router.reload()
```

File:

connectDB.js

Description:

The file connectDb.js exports a function, connectDb, which utilizes the Mongoose library to establish a connection to a MongoDB database. The function is designed to be asynchronous, handling potential errors during the connection process, and logs a success message to the console upon a successful database connection. It is intended to be used as a module in a Node.js application, typically invoked during the application's initialization to ensure a connection to the MongoDB database.

Pseudocode:

```
const mongoose = require("mongoose");
async function connectDb() {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      useCreateIndex: true,
      useFindAndModify: false,
    });
    console.log("MongoDB connected.");
  } catch (error) {
    console.log(error);
    process.exit(1);
  }
}
module.exports = connectDb;
```

File:

messageActions.js

Description:

The file messageActions.js defines functions related to managing one to one direct chat messages in a Node.js application using Mongoose. The loadMessages function retrieves chat messages between a specified user and another user, while the sendMsg function sends a new message between two users, updating or creating chat records as needed. Additionally, the file includes functions like setMsgToUnread to mark a user's messages as unread and deleteMsg to delete a user's own messages from a chat. These functions are intended for use in a real-time chat application.

Pseudocode:

```
FUNCTION loadMessages(userId, messagesWith):
  TRY:
    // Find the user's chat information, populating messagesWith details
    user = AWAIT ChatModel.findOne({ user: userId }).populate("chats.messagesWith");

    // Find the specific chat based on messagesWith user ID
    chat = user.chats.find((chat) => chat.messagesWith._id.toString() === messagesWith);

    // If no chat is found, return an error
    IF !chat:
      RETURN { error: "No chat found" };

    // Return the chat
    RETURN { chat };
  CATCH error:
    // Handle errors and return an error object
    LOG_ERROR(error)
    RETURN { error };

FUNCTION sendMsg(userId, msgSendToUserId, msg):
  TRY:
    // Find the sender and receiver users' chat information
    user = AWAIT ChatModel.findOne({ user: userId });
    msgSendToUser = AWAIT ChatModel.findOne({ user: msgSendToUserId });

    // Create a new message object
    newMsg = {
      sender: userId,
      receiver: msgSendToUserId,
```

```
        msg,
        date: Date.now(),
      };

      // Find the previous chat with the receiver user
      previousChat = user.chats.find((chat) => chat.messagesWith.toString() ===
msgSendToUserId);

      // If a previous chat exists, add the new message; otherwise, create a new chat
      IF previousChat:
        previousChat.messages.push(newMsg);
        AWAIT user.save();
      ELSE:
        newChat = { messagesWith: msgSendToUserId, messages: [newMsg] };
        user.chats.unshift(newChat);
        AWAIT user.save();

      // Repeat the process for the receiver user
      previousChatForReceiver = msgSendToUser.chats.find((chat) =>
chat.messagesWith.toString() === userId);

      IF previousChatForReceiver:
        previousChatForReceiver.messages.push(newMsg);
        AWAIT msgSendToUser.save();
      ELSE:
        newChat = { messagesWith: userId, messages: [newMsg] };
        msgSendToUser.chats.unshift(newChat);
        AWAIT msgSendToUser.save();

      // Return the new message
      RETURN { newMsg };
    CATCH error:
      // Handle errors and return an error object
      LOG_ERROR(error)
      RETURN { error };

FUNCTION setMsgToUnread(userId):
    TRY:
      // Find the user by ID
      user = AWAIT UserModel.findById(userId);
```

```
        // If unreadMessage is false, set it to true and save
        IF !user.unreadMessage:
            user.unreadMessage = true;
            AWAIT user.save();

        RETURN;
    CATCH error:
        // Handle errors and log error
        LOG_ERROR(error)

FUNCTION deleteMsg(userId, messagesWith, messageId):
    TRY:
        // Find the user's chat information
        user = AWAIT ChatModel.findOne({ user: userId });

        // Find the specific chat based on messagesWith user ID
        chat = user.chats.find((chat) => chat.messagesWith.toString() === messagesWith);

        // If no chat is found, return
        IF !chat:
            RETURN;

        // Find the message to delete
        messageToDelete = chat.messages.find((message) => message._id.toString() ===
messageId);

        // If no message is found, return
        IF !messageToDelete:
            RETURN;

        // Check if the message sender is the same as the user; if not, return
        IF messageToDelete.sender.toString() !== userId:
            RETURN;

        // Find the index of the message and remove it from the chat
        indexOf = chat.messages.map((message) =>
message._id.toString()).indexOf(messageToDelete._id.toString());
        AWAIT chat.messages.splice(indexOf, 1);
```

```
        // Save the user's chat information
        AWAIT user.save();

        // Return success
        RETURN { success: true };
    CATCH error:
        // Handle errors and log error
        LOG_ERROR(error)

// Export the functions for external use
EXPORT loadMessages, sendMsg, setMsgToUnread, deleteMsg;
```

File:
notificationActions.js
Description:
This file notificationActions.js consists of functions related to managing notifications. The functions interact with a NotificationModel and a UserModel. The setNotificationToUnread function sets a user's unread notification status. The file also includes functions for creating and removing notifications for likes, comments, and followers, each with specific parameters and operations based on the notification type.
Pseudocode:

```
FUNCTION setNotificationToUnread(userId):
    TRY:
        user = FIND_USER_BY_ID(userId)

        IF user.unreadNotification IS FALSE:
            user.unreadNotification = TRUE
            SAVE_USER(user)
    CATCH error:
        LOG_ERROR(error)

FUNCTION newLikeNotification(userId, postId, userToNotifyId):
    TRY:
        userToNotify = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        newNotification = CREATE_NOTIFICATION("newLike", userId, postId, Date.now())

        PUSH_TO_FRONT_OF_ARRAY(userToNotify.notifications, newNotification)
        SAVE_NOTIFICATION_USER(userToNotify)

        CALL setNotificationToUnread(userToNotifyId)
    CATCH error:
        LOG_ERROR(error)

FUNCTION removeLikeNotification(userId, postId, userToNotifyId):
    TRY:
        user = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        notificationToRemove = FIND_NOTIFICATION_TO_REMOVE(user, "newLike", userId,
postId)

        IF notificationToRemove EXISTS:
            indexOf = FIND_INDEX_OF_NOTIFICATION(user.notifications,
notificationToRemove)
```

```
        REMOVE_FROM_ARRAY_AT_INDEX(user.notifications, indexOf)
        SAVE_NOTIFICATION_USER(user)
    CATCH error:
        LOG_ERROR(error)

FUNCTION newCommentNotification(postId, commentId, userId, userToNotifyId, text):
    TRY:
        userToNotify = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        newNotification = CREATE_NOTIFICATION("newComment", userId, postId, commentId,
text, Date.now())

        PUSH_TO_FRONT_OF_ARRAY(userToNotify.notifications, newNotification)
        SAVE_NOTIFICATION_USER(userToNotify)

        CALL setNotificationToUnread(userToNotifyId)
    CATCH error:
        LOG_ERROR(error)

FUNCTION removeCommentNotification(postId, commentId, userId, userToNotifyId):
    TRY:
        user = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        notificationToRemove = FIND_NOTIFICATION_TO_REMOVE(user, "newComment",
userId, postId, commentId)

        IF notificationToRemove EXISTS:
            indexOf = FIND_INDEX_OF_NOTIFICATION(user.notifications,
notificationToRemove)
            REMOVE_FROM_ARRAY_AT_INDEX(user.notifications, indexOf)
            SAVE_NOTIFICATION_USER(user)
    CATCH error:
        LOG_ERROR(error)

FUNCTION newFollowerNotification(userId, userToNotifyId):
    TRY:
        user = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        newNotification = CREATE_NOTIFICATION("newFollower", userId, Date.now())

        PUSH_TO_FRONT_OF_ARRAY(user.notifications, newNotification)
        SAVE_NOTIFICATION_USER(user)
```

```
        CALL setNotificationToUnread(userToNotifyId)
    CATCH error:
        LOG_ERROR(error)

FUNCTION removeFollowerNotification(userId, userToNotifyId):
    TRY:
        user = FIND_NOTIFICATION_USER_BY_ID(userToNotifyId)
        notificationToRemove = FIND_NOTIFICATION_TO_REMOVE(user, "newFollower",
userId)

        IF notificationToRemove EXISTS:
            indexOf = FIND_INDEX_OF_NOTIFICATION(user.notifications,
notificationToRemove)
            REMOVE_FROM_ARRAY_AT_INDEX(user.notifications, indexOf)
            SAVE_NOTIFICATION_USER(user)
    CATCH error:
        LOG_ERROR(error)

EXPORT newLikeNotification, removeLikeNotification, newCommentNotification,
removeCommentNotification, newFollowerNotification, removeFollowerNotification
```

File:
uploadPicToCloudinary.js
Description:
The file uploadPicToCloudinary.js uses the axios library to upload an image file to the Cloudinary cloud storage service. All images need to be stored in the cloud to be able to be retrieved once the application is deployed.

Pseudocode:

```
Function: uploadPic
Input: media (image file)

Begin
   Create a FormData object
   Append "file" with the image file to FormData
   Append "upload_preset" to FormData
   Append "cloud_name" to FormData
   Make a POST request to Cloudinary using axios
      Use the Cloudinary URL from the environment variables
      Pass FormData in the request body
   If the upload is successful (HTTP status 200)
      Return the URL of the uploaded image
   If there is an error during the upload
      Return undefined
End
```

File:
auth.js
Pseudocode:

Method 1 : GET "/"

Input:
req (request object with userId obtained from authMiddleware)
Output:
res (response object)
Actions:
1. Extract userId from req.
2. Try:
a. Fetch user details by userId from UserModel.
b. Fetch user follow statistics by userId from FollowerModel.
c. Respond with user details and follow statistics in the response (res).
3. Catch any errors:
a. Log the error.
b. Respond with a 500 status and a server error message.

Method 2: POST "/"

Input:
req (request object with email and password)
Output:
res (response object)
Action:
1. Extract email and password from req.
2. Validate email format using isEmail function.
3. If email is invalid, respond with a 401 status and "Invalid Email" message.
4. Validate password length (must be at least 6 characters).
5. If password is too short, respond with a 401 status and "Password must be at least 6 characters" message.
6. Try:
a. Find user by email (case-insensitive) and select password field from UserModel.
b. Check if user exists. If not, respond with a 401 status and "Invalid Credentials" message.
c. Compare provided password with stored password hash.
d. If passwords don't match, respond with a 401 status and "Invalid Credentials" message.
e. Check if a chat model exists for the user. If not, create one.
f. Generate a JWT token with userId as payload.

g. Respond with the generated token in the response (res).

7. Catch any errors:
   a. Log the error.
   b. Respond with a 500 status and a server error message.

File:

chats.js

Pseudocode:

METHOD 1: GET MESSAGE:

Input:

Req (request userID and their messages)

Output:

Chatstosend (The chat message, along with any other attributes of the chat such as date, time, sender, receiver)

Action:

1. Try:
   a. Obtain userID from the input Req
   b. If the message being written > 0 words:
      A. Create js map *Chatstosend* including the receivers userID, receivers name, their profile pic, the last message sent to the user, and the date
      B. The map is awaited on by the user to create the chat, so other attributes such as time can be updated
   c. Return the Chatstosend
2. Catch error
   a. Return 500 error with service error

METHOD 2: FIND USER & GET MESSAGE:

Input:

Req (Search for user by userID and their messages)

Output:

User (User to be messaged)

Action

1. Try:
   a. Finduser by userID
   b. If !user (no user grabbed using Finduser)
      A. Throw 404 error "No user found"
   c. return user.name, and user.profile_picture
2. catch error:
   a. Throw 500 server error

METHOD 3: FIND USER & GET MESSAGE:

Input:

Req (obtain userID, the message receiver, and the messages between them)

Output:

Save the deleted message

Action:

1. Try:
   a. Obtain userID and messages with, from req
   b. Find user given the userID
   c. chat to delete is found given the two users doing the messaging
   d. If !chat to delete (not exist)
      A. Return 404 error message (chat NOT FOUND)
   e. Find the location of the message and save into the index of.
   f. Using an index to delete the message.
   g. Return status proclaiming that the message was deleted
2. Catch error
   a. Throw 500 server error message

File:

notifications.js

Pseudocode:

METHOD 1: GET NOTIFICATION

Input:

Req (request userID and their attributes)

Output:

The notification regarding a post or other user

1. Try
   a. Obtain userID from req
   b. Obtain user information from userID
   c. update notification attributes of user regarding a post or user notification
   d. Return the users notifications obtained from c.
2. Catch error
   a. Return error status 500 (Server Error)

METHOD 2: POST UPDATE NOTIFICATION

Input:

Req (request userID and their attributes)

Action:

Try

1. Obtain the user ID from req and assign to userID

2. Using the userID, obtain the users attributes using findById
3. If the users unreadNotification attribute is True (The notification being visited is unread) then mark the notification as read
4. Await the user object to save the notification read to update

Catch
1. Throw error 500 server error

Return:
Res -> user notification is updated

File:
signup.js

Pseudocode:

METHOD 1: Get Username

Input:
username (the username entered by the applicant)

Action:
Try
1. Username = reqs
2. If the username.length < 1
   a. Return error (401) invalid username
3. If (user) (if the username is already taken)
   a. Return error (401) invalid username
4. Return username available

Catch error 500 server error

Return:
Res (The final constructed user, but not officially registered until super user accepts and designates their role)

METHOD 2: Get Username

Input:
req (The applicants inputs such as email, password, bio ect)

Actions:
1. Assign information from req usr, to objects name, email, username, password, bio
2. If email is not a proper email address, return error 401 Invalid Error
3. If password < 8, return error 401 Password must include at least 8 characters

Try
4. If email address is already registered, return error 401 Email already in use
5. Create new user object from email, username, and password
6. Hash password and reassign to user object

7. Assign next available userID to user in the user attribute

8. Utilize send message method to send users information (including bio) to super user

Result:

Message sent to super user

METHOD 2: Superuser Decision

Input:

req (The decision of the superuser, as well as the information regarding qualified user and further user attributes)

States:

Decision = req.decision

For Corp Users:

Company

Job Link

For All:

Username

Bio

Password

Other Social Medias

Actions:

1. Assign assign reqs.user to user and reqs.decision to decision

2. If !decision return message that the users application was not accepted

3. Else the user is defined by decision; if decision is Ordinary User then the user is kept as an ordinary user, if decision is corporate then the user is made a subclass Corporate User.

4. Assign users alternate social media: profilefields.linkedin = linkedin

5. Create Follower model so followers can be assigned to user: followers = [], following = []

6. Create notification model: notifications = []

7. Create chat model: chat = []

8. If the user is a Corporate user, assign Company and Job attributes to the user.

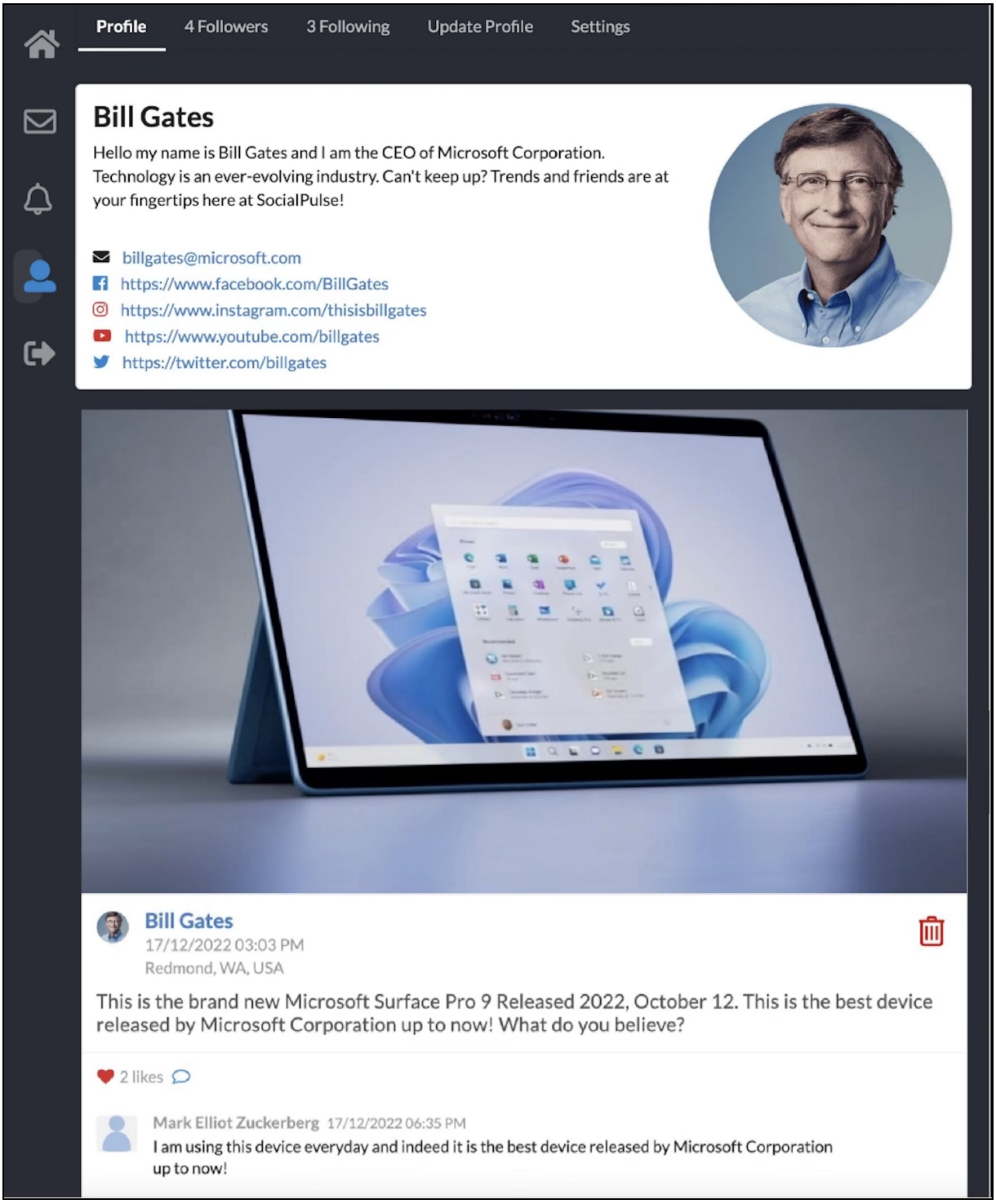
Result:

User substantiated.

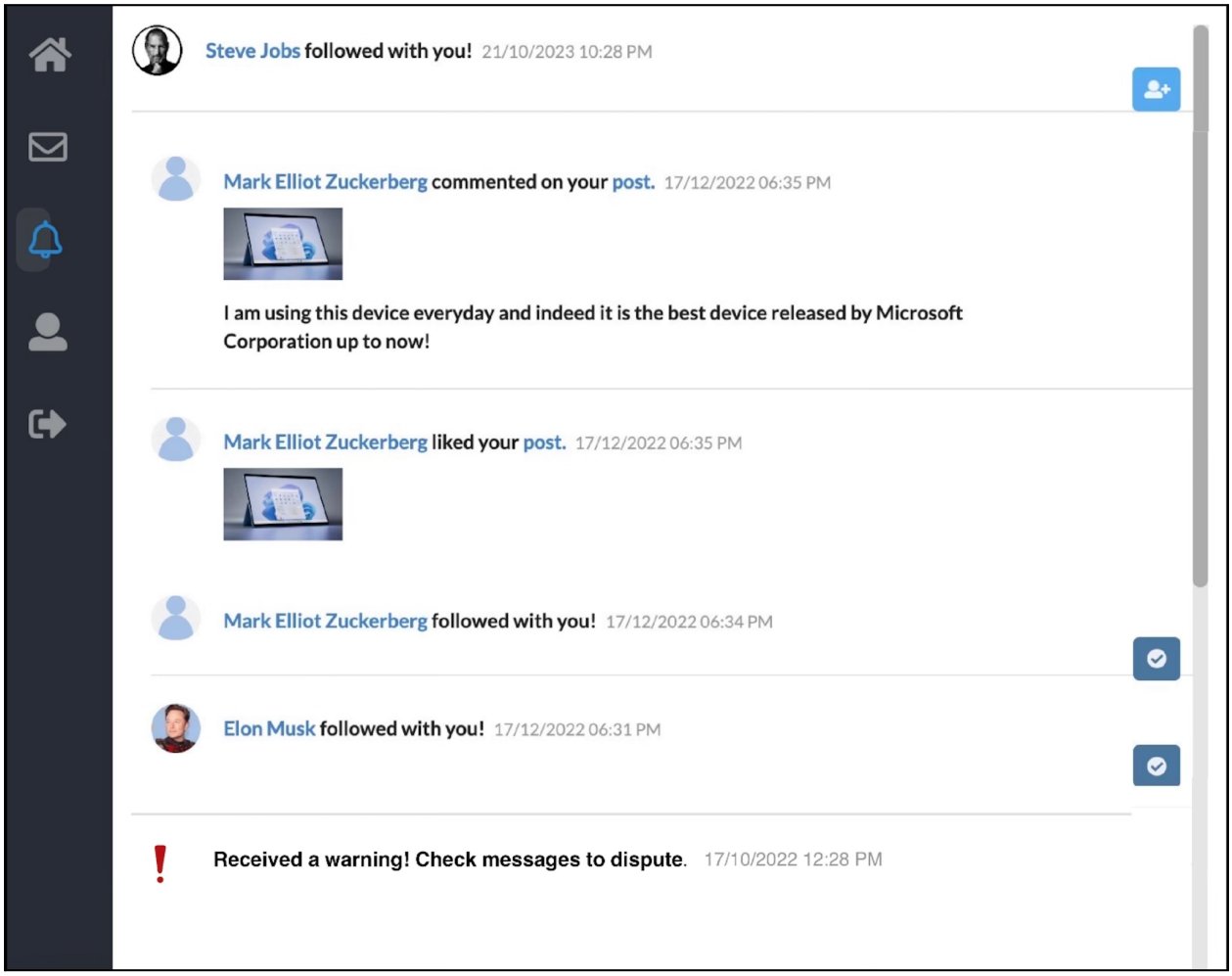
5. System Screens

5.1 GUI Screens

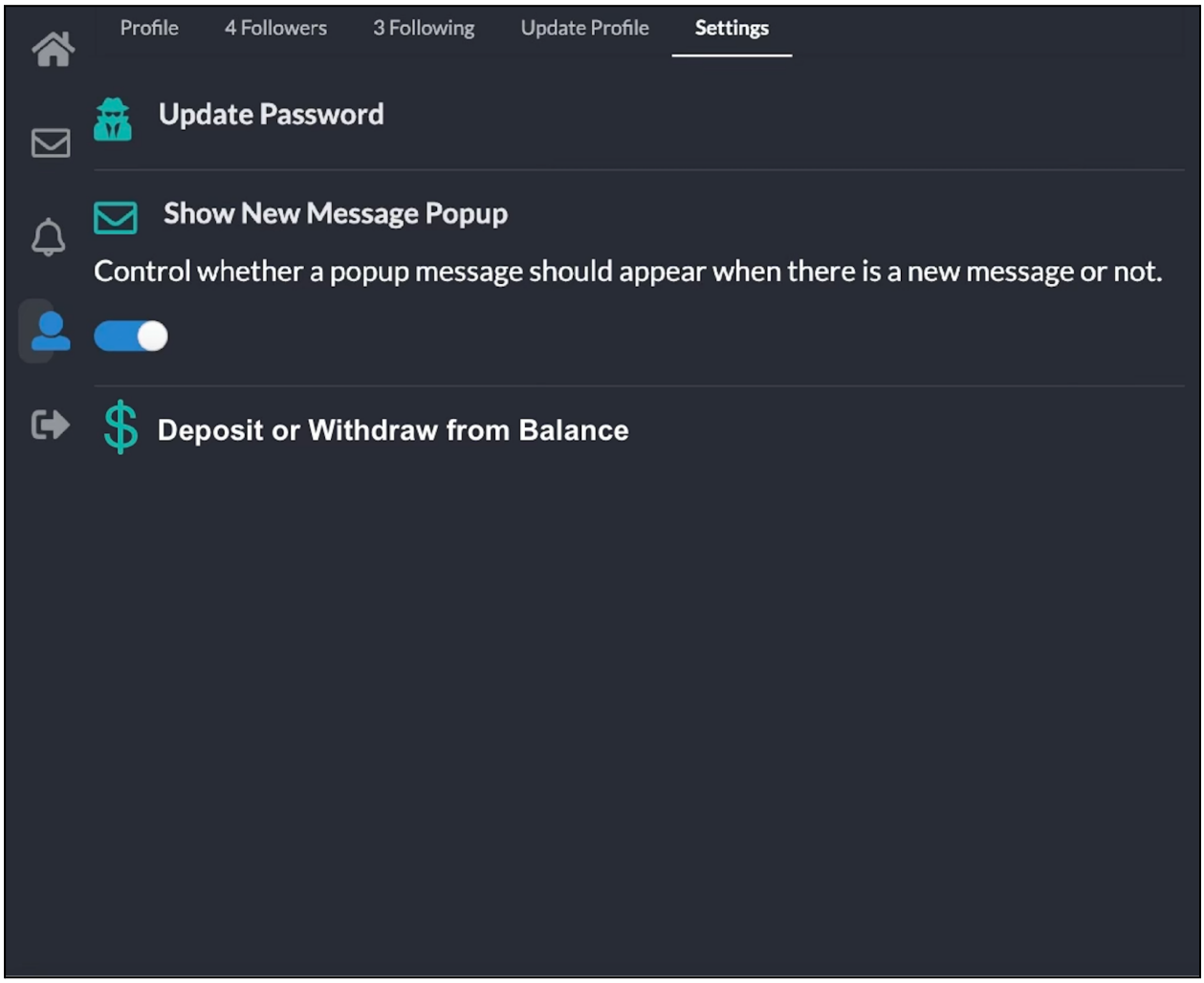
5.1.1 View Self Profile



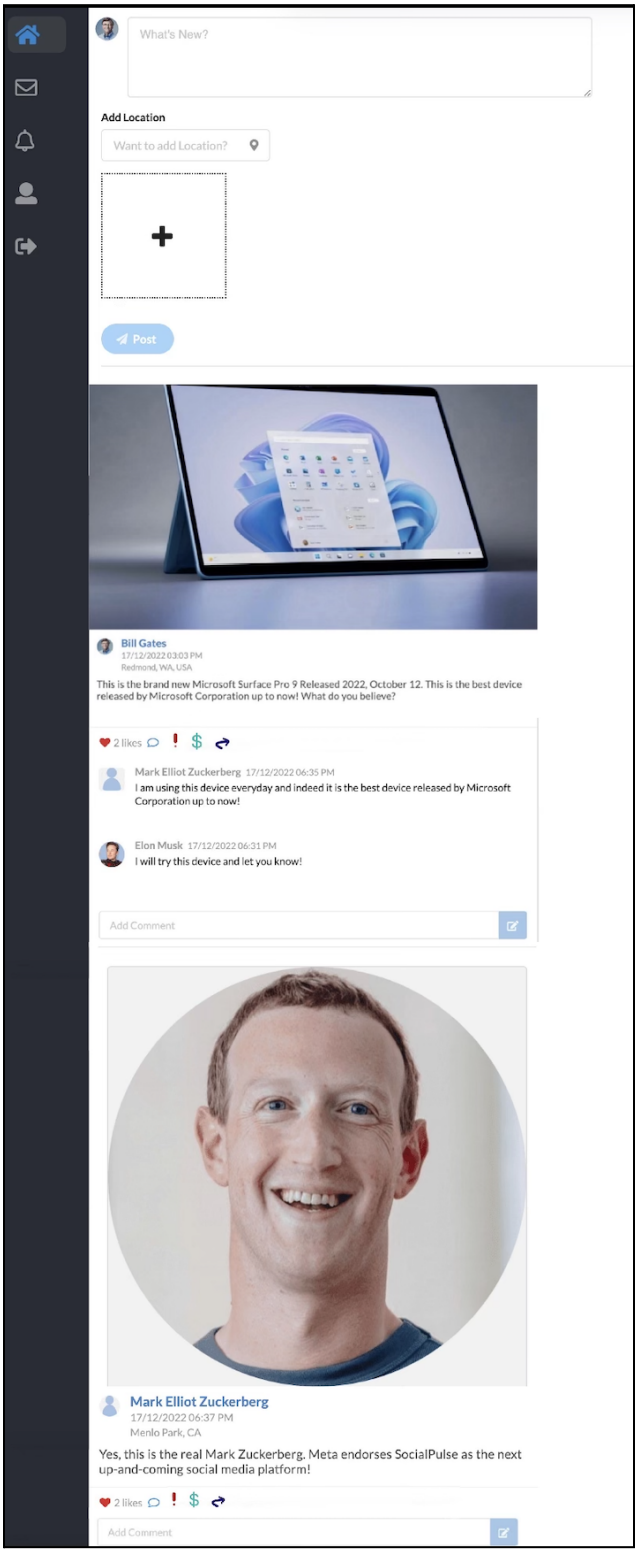
5.1.2 Notifications Page



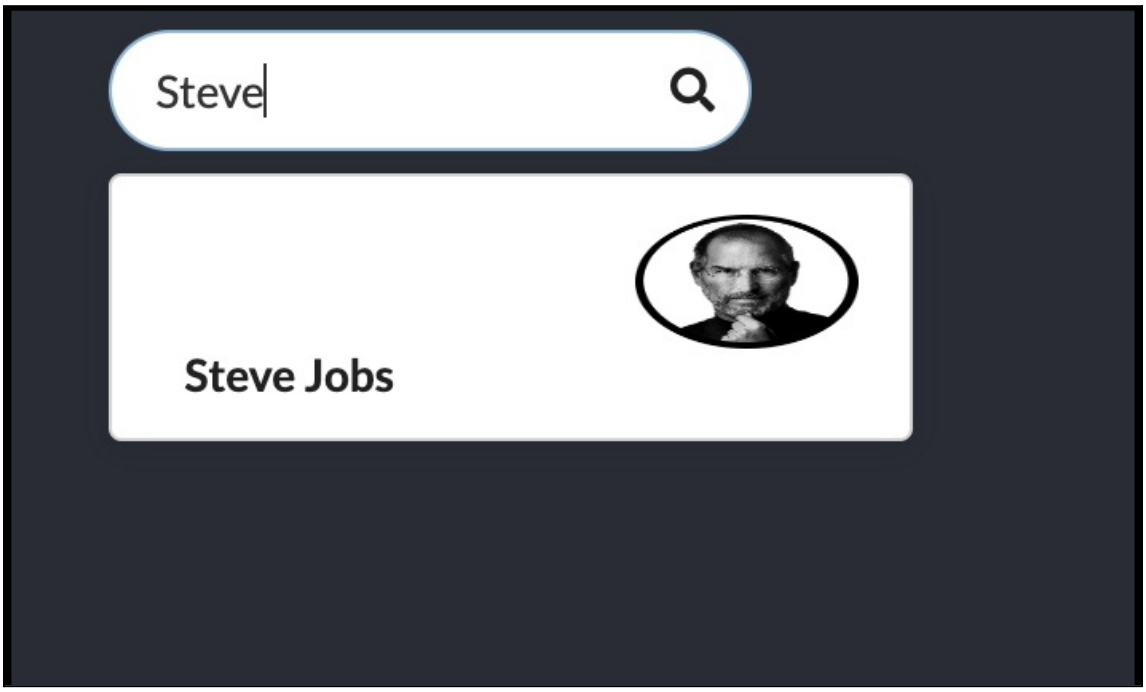
5.1.3 Settings Page



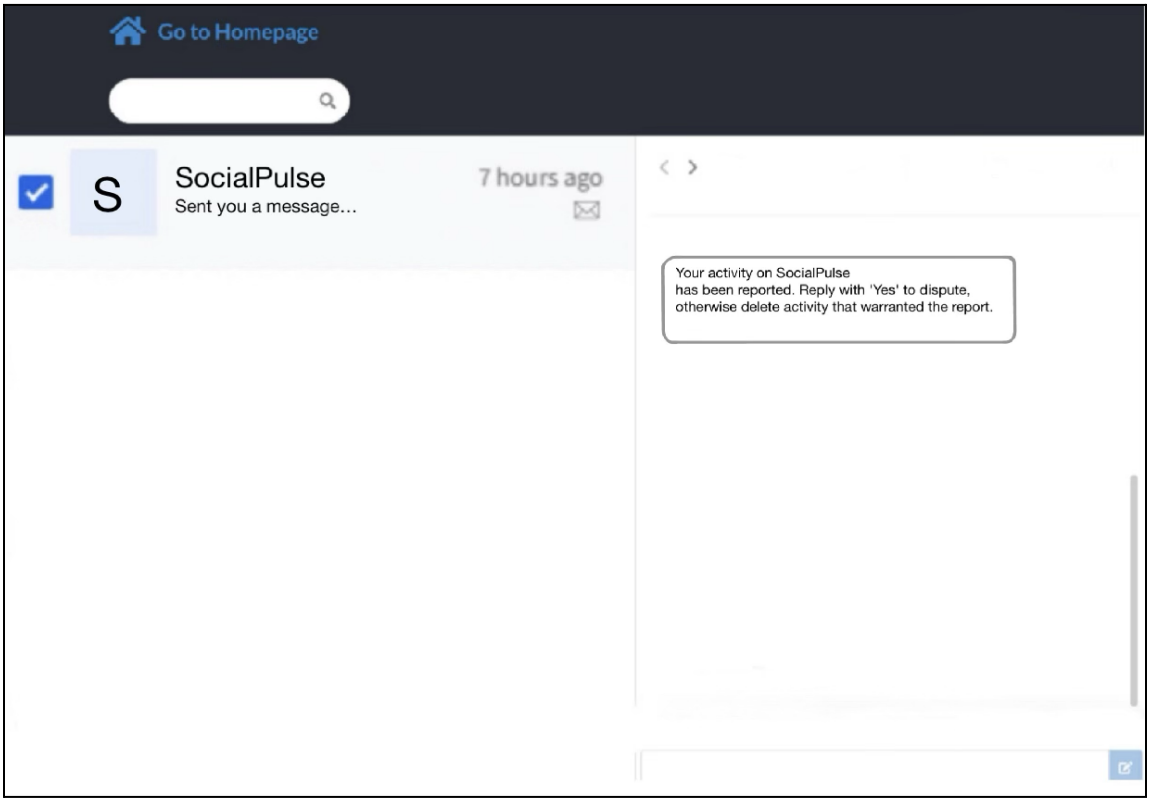
5.1.4 Personalized Trendy Page



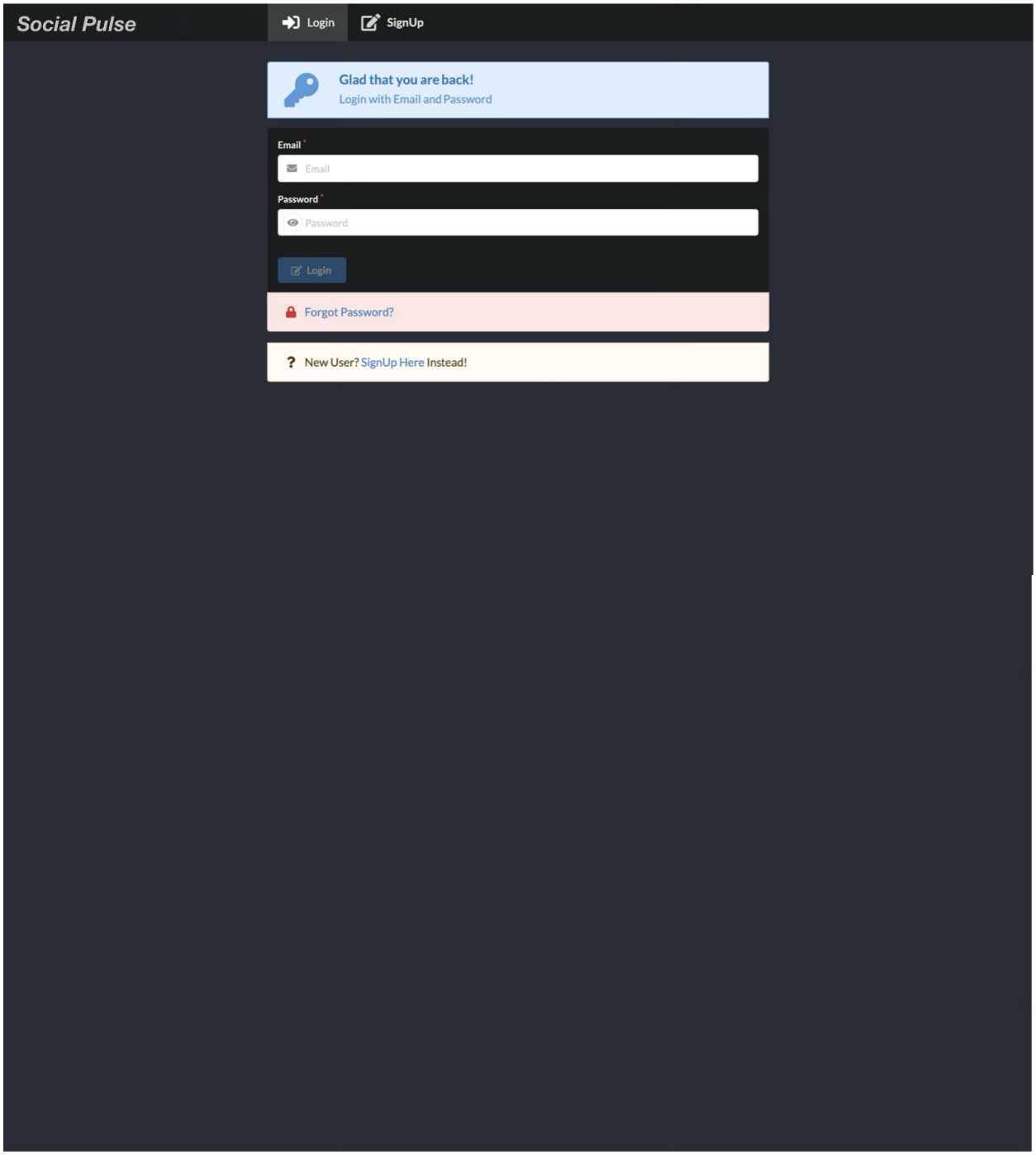
5.1.5 Search Bar Functionality



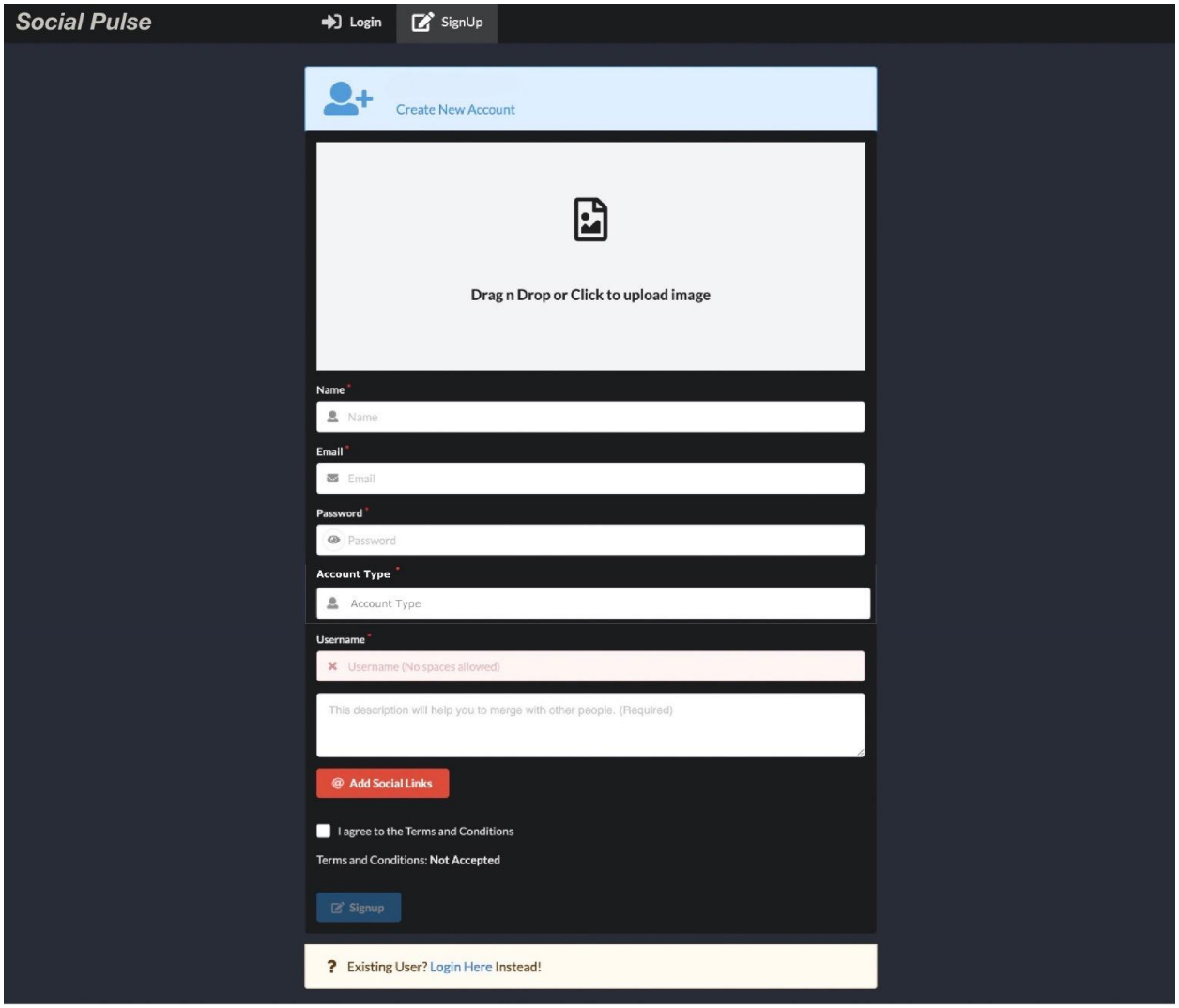
5.1.6 Messages Page



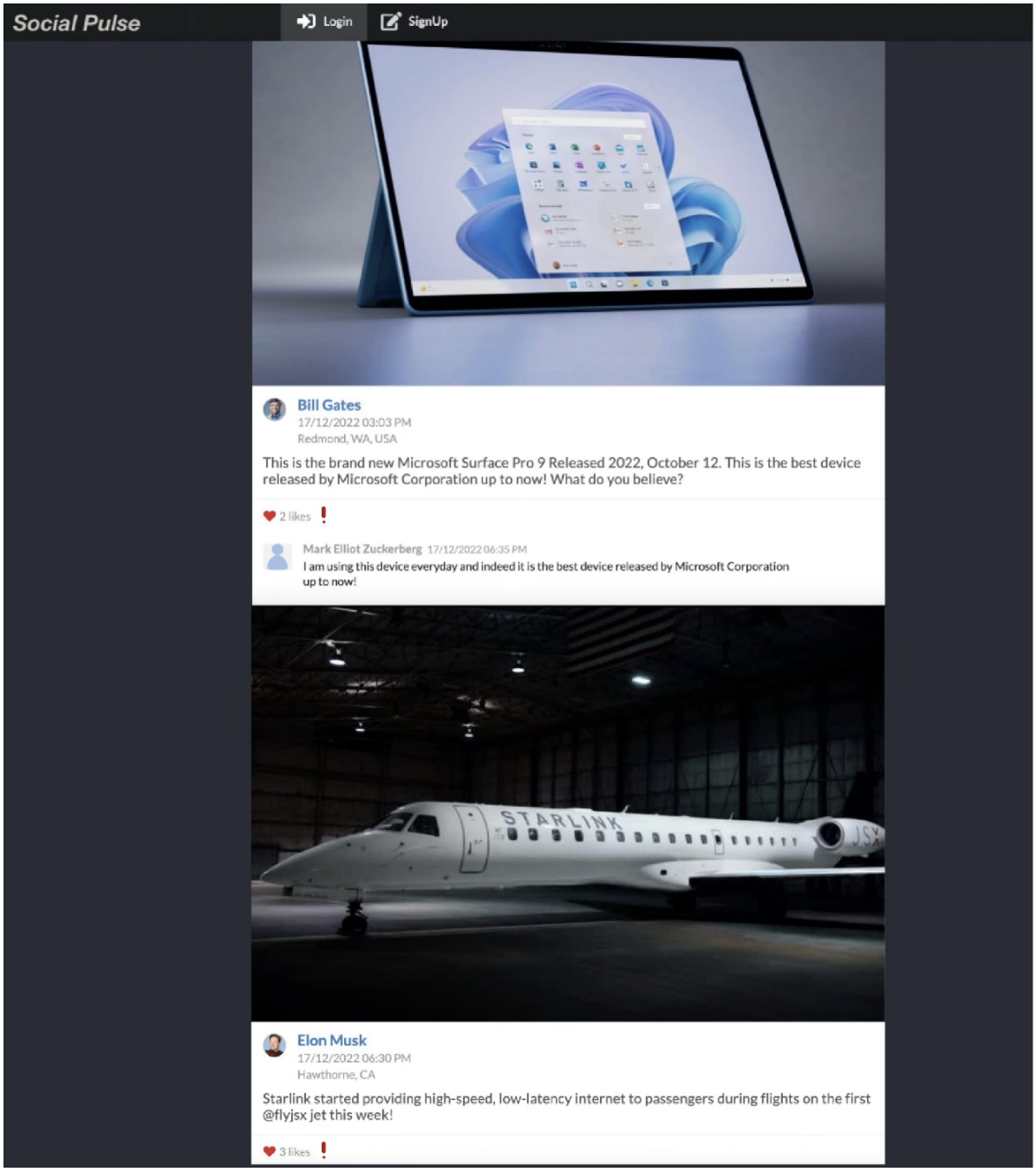
5.1.7 Login Page



5.1.8 Register Page



5.1.9 General Trendy Page



6. Group Meetings

6.1 Meeting Logs

Date	Who attended?	What was discussed?
09/10/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Meeting each other, forming our team, and discussing our team letter
09/14/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Virtual meeting to discuss ideas and each of our coding preferences
10/20/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	First meeting: went over what to do for Phase 1 report
10/23/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Worked on Phase 1 report together on call
10/27/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Finalized Phase 1 Report + Submission
11/4/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Preliminary talk on potential stack options, divvying up who works on front end and back end
11/11/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Went over potential pseudocode for how the application will work
11/18/2023	Jolie Huang, Leon Belegu, Selma Doganata	Met up to get a good idea on how we should approach making diagrams + system screens

11/18/2023	Georgios Ioannou and Christian Rasmussen	Split up pseudocode work
11/19/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Planning how to work on the things until Tuesday and delegating work amongst each other.
11/20/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Discussed current collaboration diagrams and planned what final touches need to be made.
11/21/2023	Georgios Ioannou, Jolie Huang, Christian Rasmussen, Selma Doganata, and Leon Belegu	Finalizing the Phase 2 report and final touches.

6.2 Work Distribution

1. Software Requirements Specification (Phase 1) Work Distribution:
    - Georgios Ioannou
        - Definitions, Acronyms, and Abbreviations (Section 1.3)
        - Scope (Section 2.1)
    - Jolie Huang
        - Use-case Reports (Section 3.1)
        - Supplementary Requirements (Section 3.2)
        - Use Case Diagram (Section 4)
    - Christian Rasmussen
        - Table of Contents
        - Use Case Reports (Section 3.1)
    - Selma Doganata
        - Purpose (Section 1.1)
    - Leon Belegu
        - Use Case Model Survey (Section 2.1)

    - Use Case Diagram (Section 4)

2. Design Report (Phase 2) Work Distribution:
    - Georgios Ioannou
        - ER-Diagram (Section 3)
        - Pseudocode (Section 4)
        - Group Meetings (Section 6)
    - Jolie Huang
        - System Collaboration Class Diagram (Section 1.2)
        - Use Case Collaboration Diagrams (Section 2.1)
        - Use Case Petri-Net Diagrams (Section 2.2)
        - Group Meetings (Section 6)
    - Christian Rasmussen
        - ER-Diagram (Section 3)
        - Pseudocode (Section 4)
        - Group Meetings (Section 6)
    - Selma Doganata
        - Use Case Collaboration Diagrams (Section 2.1)
        - System Screens (Section 5)
        - Group Meetings (Section 6)
    - Leon Belegu
        - Purpose (Section 1.1)
        - Use Case Collaboration Diagrams (Section 2.1)
        - Use Case Petri-Net Diagrams (Section 2.2)
        - Group Meetings (Section 6)

6.3 Team Concerns

- How can we better manage our time well and stay on track with deadlines?
- Are there better ways to reduce miscommunication? Is everyone contributing their fair share, or are there concerns about uneven workloads?

- Do team members have the necessary resources and support to fulfill their roles?
- Are we adapting to changes and challenges effectively, or are there roadblocks?

7. GitHub Repository

Link to the GitHub Repository: https://github.com/jolie-huang/CSC322-Team-W