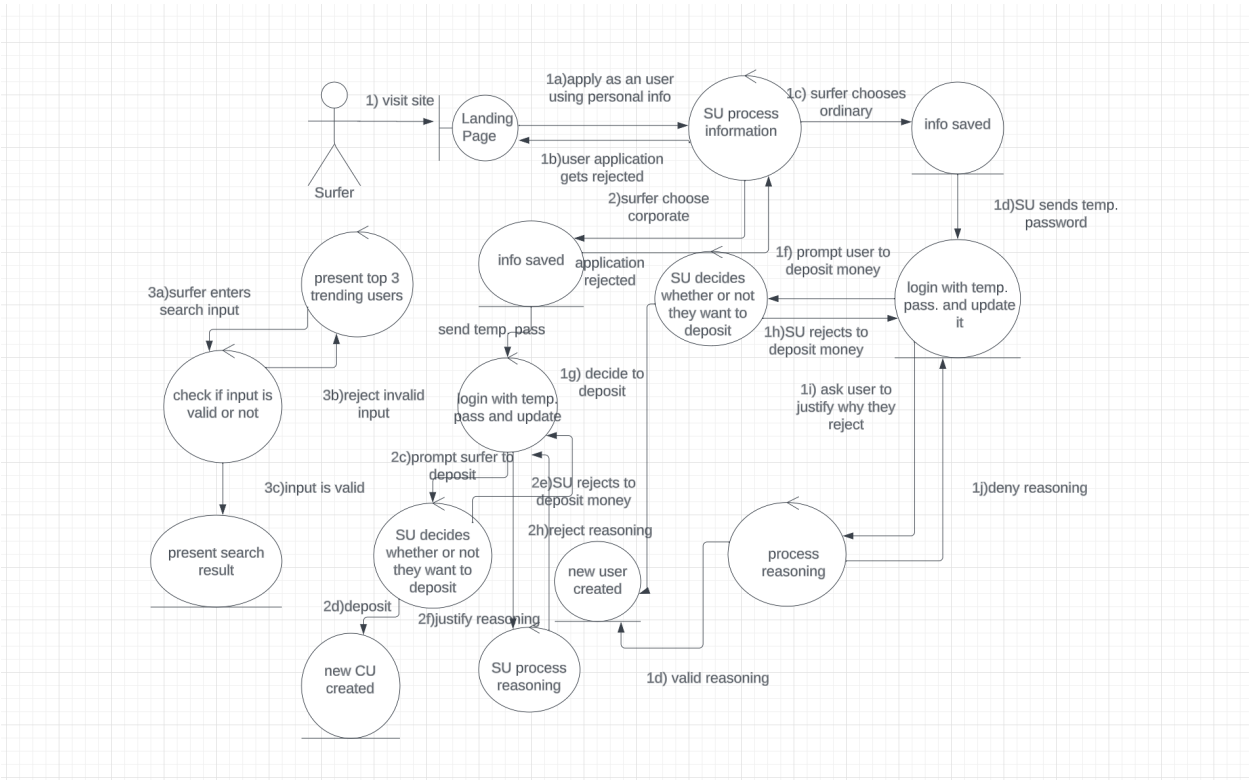
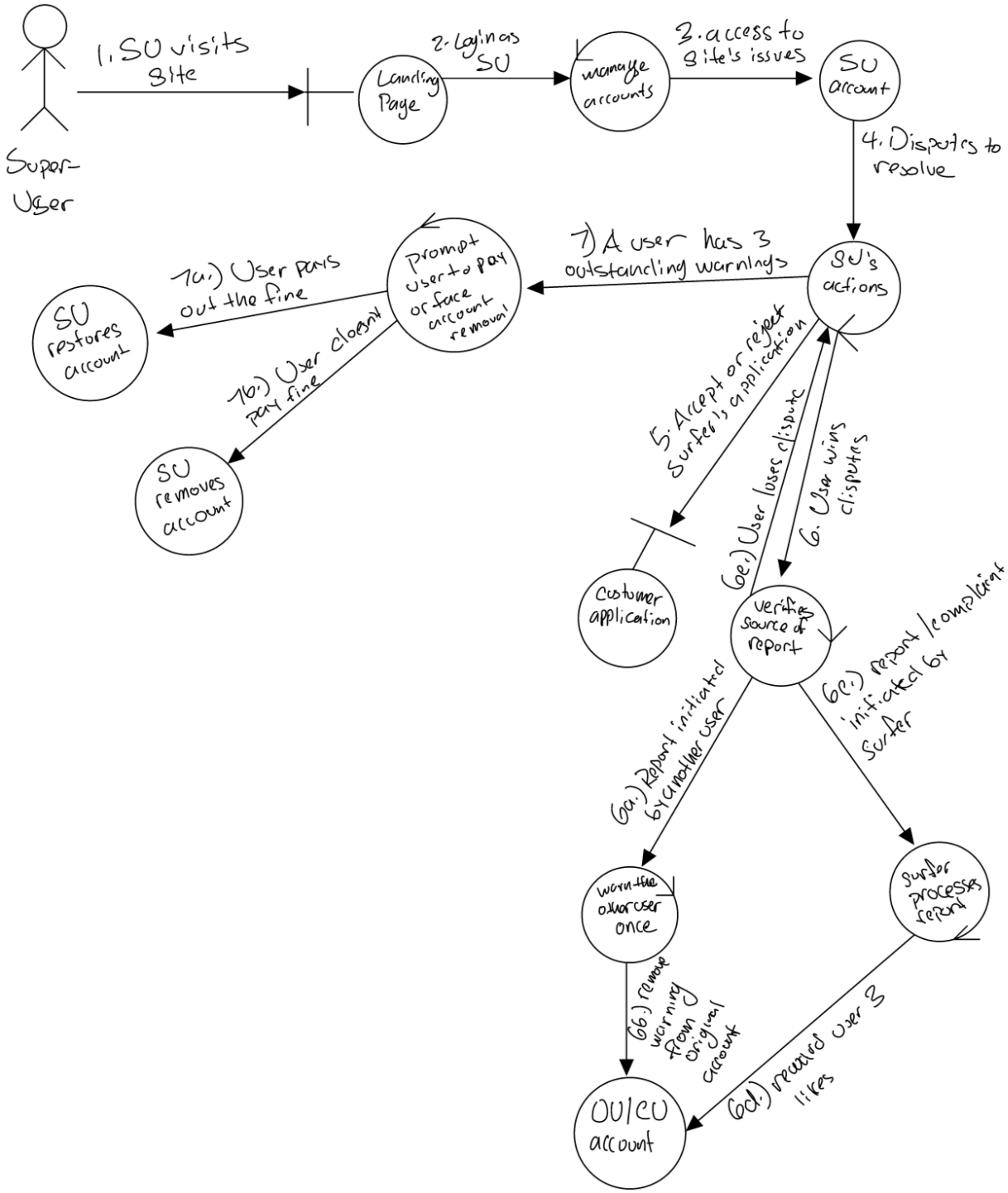
Phase II: Design Report Outline

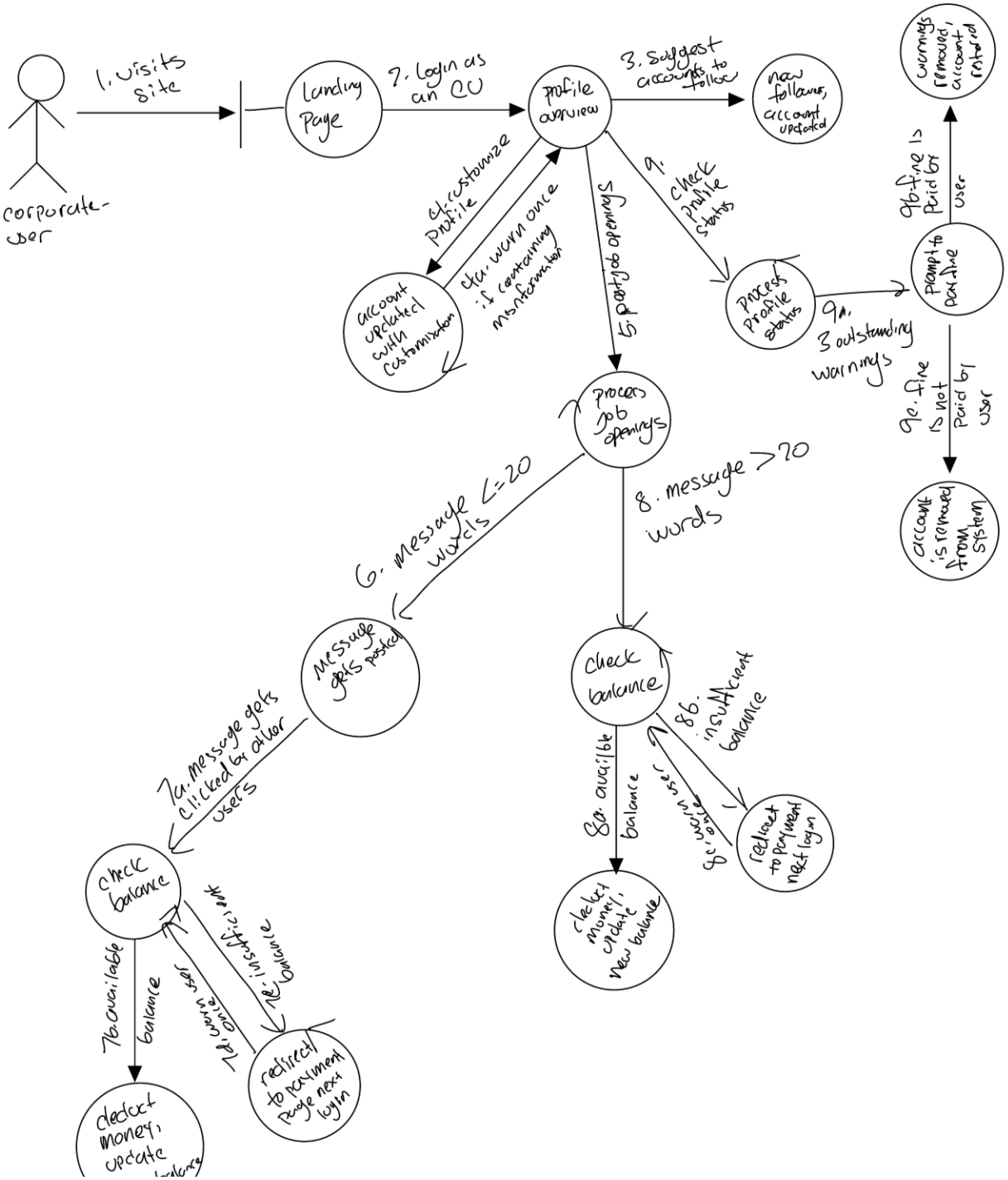
1. **Introduction**
    - Surfer



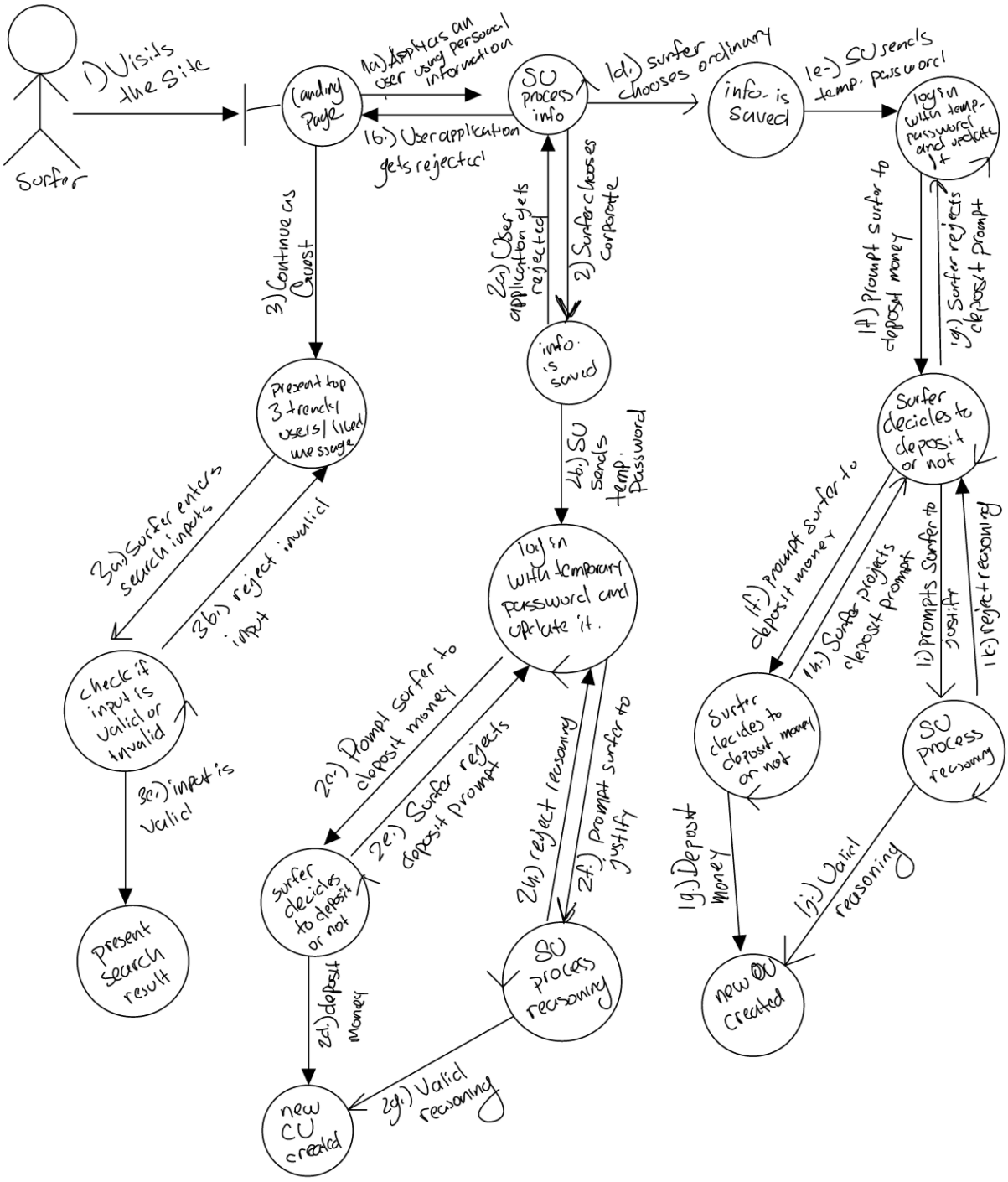
- Super User


Super-User
1. SU visits site
Landing Page
2. Log in as SU
manage accounts
3. access to site's issues
SU account
4. Disputes to resolve
SU's actions
7) A user has 3 outstanding warnings
prompt user to pay or face account removal
7a.) User pays out the fine
SU restores account
7b.) User doesn't pay fine
SU removes account
5. Accept or reject surfer's application
Customer application
6. User wins disputes
6e.) User loses dispute
verifies source of report
6a.) Report initiated by another user
6e.) report/complaint initiated by surfer
warn the other user once
surfer processes report
6b.) remove warning from original account
6d.) received user 3 likes
OU/CU account

- Corporate User

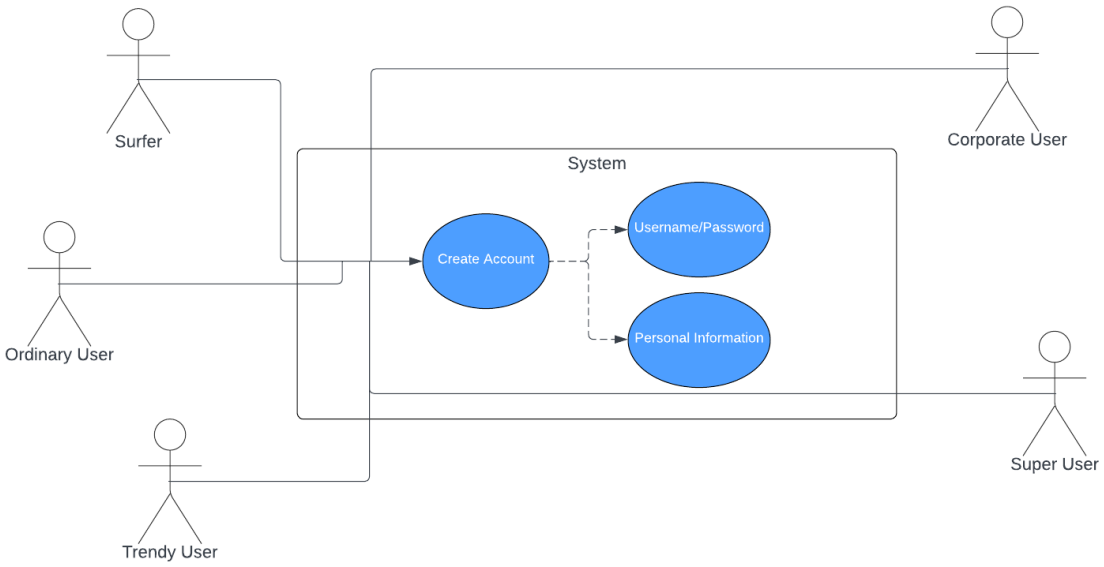


- Surfer

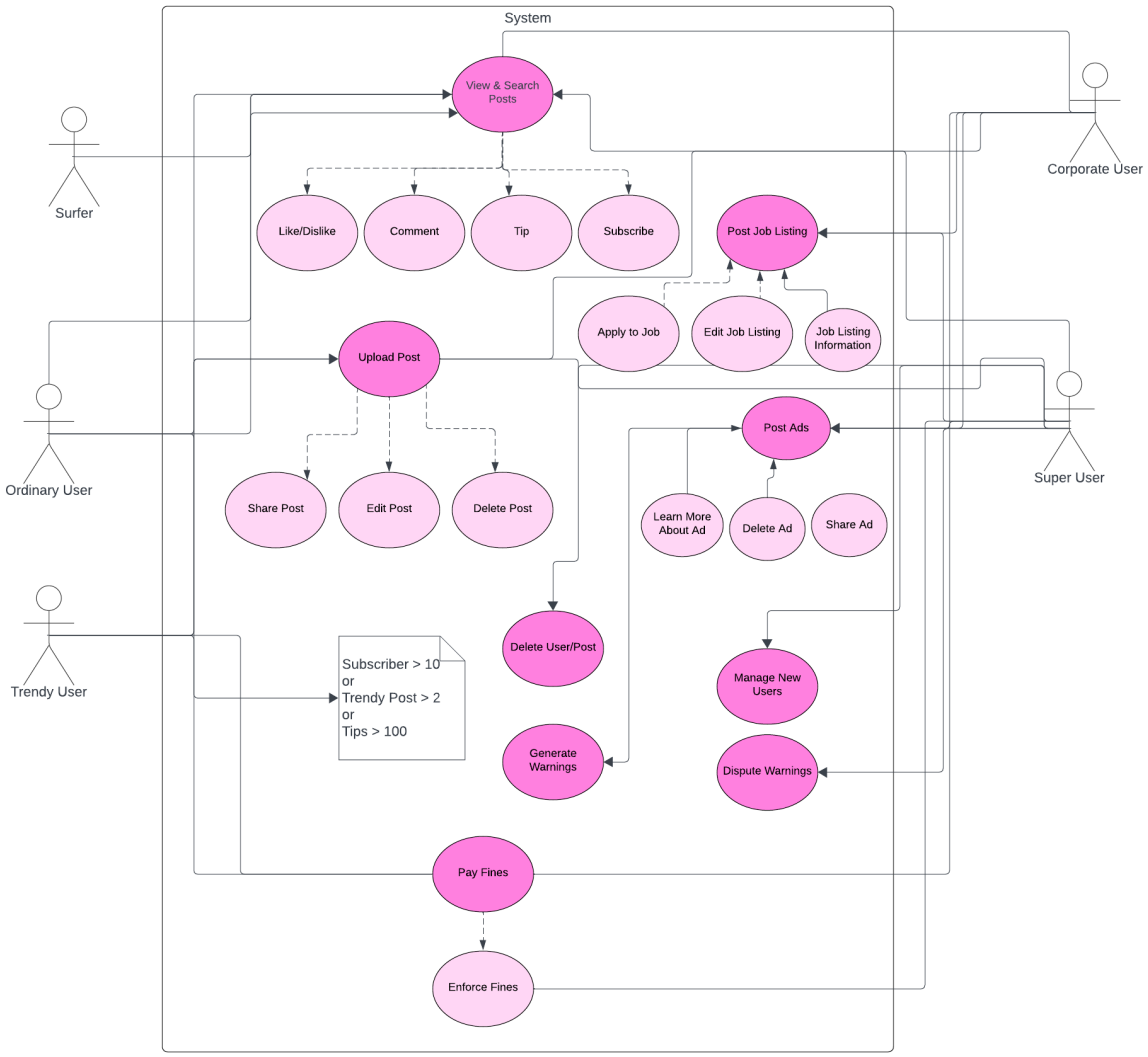


-

2. **All Use Cases**
    - <u>Scenarios for each use case: normal and exceptional scenarios.</u>
    - <u>Collaboration or sequence class diagram for each use case.</u>



- i. **Users**: There are four types of actors or users interacting with the system:
    1. Surfer
    2. Ordinary User
    3. Corporate User
    4. Super User
    5. Trendy User
- ii. **System**: Represents the software system or platform with which the users interact.
- iii. **Create Account**: This is the use case that all users except Trendy User can perform with the system.
- iv. **Username/Password**: This is a data store or a requirement when creating an account, indicating that users need to provide this information to the system.
- v. **Personal Information**: This is a data store or a requirement for the account creation process, indicating that users must provide personal details to the system.



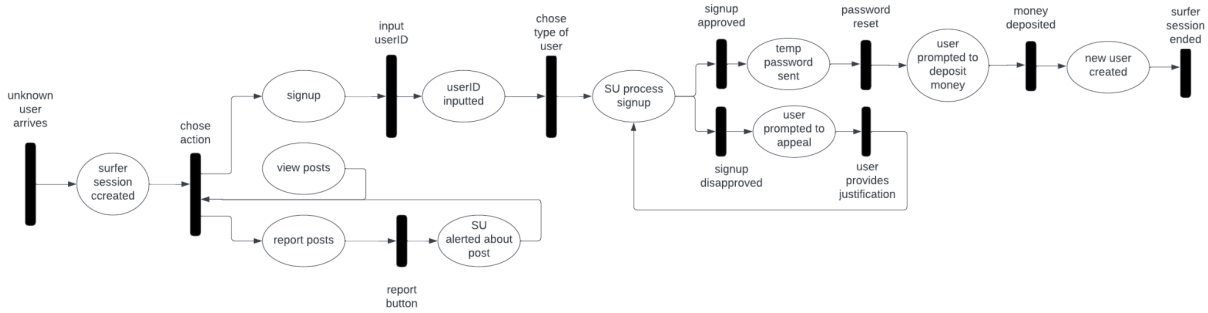
-
  i. **Surfer**:
     1. Can view and search posts within the system.
  ii. **Ordinary User** inherits Surfer's use cases and also can:
     1. Like or dislike posts.
     2. Comment on posts.
     3. Tip content creators.
     4. Subscribe to other users.
     5. Upload posts.
     6. Share posts.
     7. Edit their posts.
     8. Delete their posts.
  iii. **Trendy User** inherits Ordinary User's use cases and is defined by either having:

1. More than 10 subscribers.
2. More than 2 trendy posts.
3. Tips totaling over 100.

iv. **Corporate User** inherits Surfer's use cases and also can:
1. Post job listings.
2. Edit job listing information.
3. Apply to jobs posted by others.

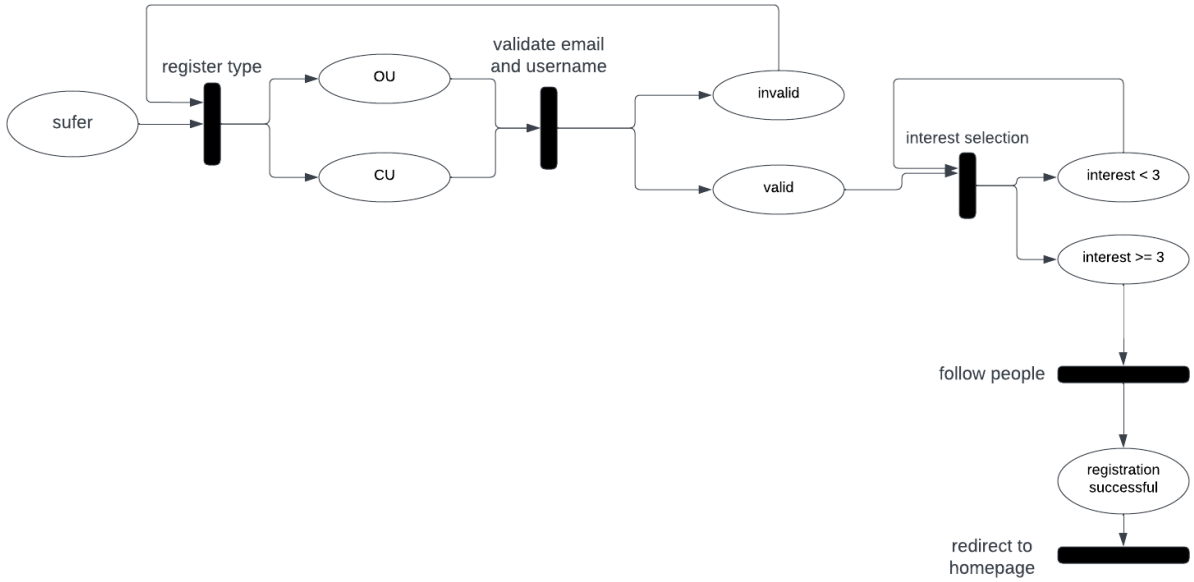
v. **Super User** inherits from Corporate User and also can:
1. Post advertisements.
2. Learn more about advertisements.
3. Share advertisements.
4. Delete advertisements.
5. Manage new users.
6. Generate warnings for users.
7. Delete user posts.
8. Dispute warnings.
9. Pay fines.
10. Enforce fines.

- *Petri-Net* for surfer



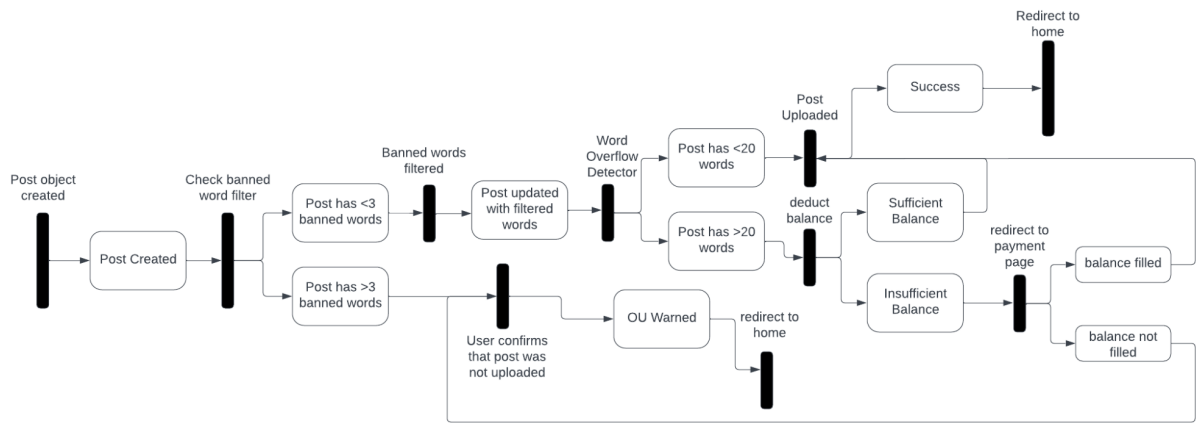
- An unknown user arrives at the platform,
- Initiating a surfer session.
- The user chooses an action:
  - To sign up.
  - To view posts.
  - To report posts.
- If the user chooses to sign up:
  - They input a userID.
  - They choose the type of user they want to register as.
  - The Super User (SU) processes the signup.
    - If the signup is approved:

            - A temporary password is sent to the user.
            - The user is prompted to deposit money.
            - Once the money is deposited, a new user account is created.
            - The surfer session is ended.
        - If the signup is disapproved:
            - The user is prompted to appeal.
            - The user provides justification for the appeal.
            - If the appeal leads to a password reset, the user may deposit money and proceed to account creation as above.
    - If the user chooses to view posts, they may continue browsing without further actions described.
    - If the user chooses to report posts:
        - The SU is alerted about the post via a report button.
- *Petri-Net* for registration



i. A surfer begins the registration process.
ii. The surfer selects a registration type:
    1. OU (Ordinary User)
    2. CU (Corporate User)
iii. The platform validates the email and username provided by the surfer.

        1. If either is invalid, the process stops or loops back to correct the information.
        2. If both are valid, the process continues.
    iv. The surfer selects interests.
        1. If fewer than 3 interests are selected, the process may loop back to select more.
        2. If 3 or more interests are selected, the process moves forward.
    v. The surfer is prompted to follow people (likely based on selected interests).
    vi. Upon following people, the registration is marked as successful.
    vii. Finally, the new user is redirected to the homepage of the platform.
- *Petri-Net* for post-submission

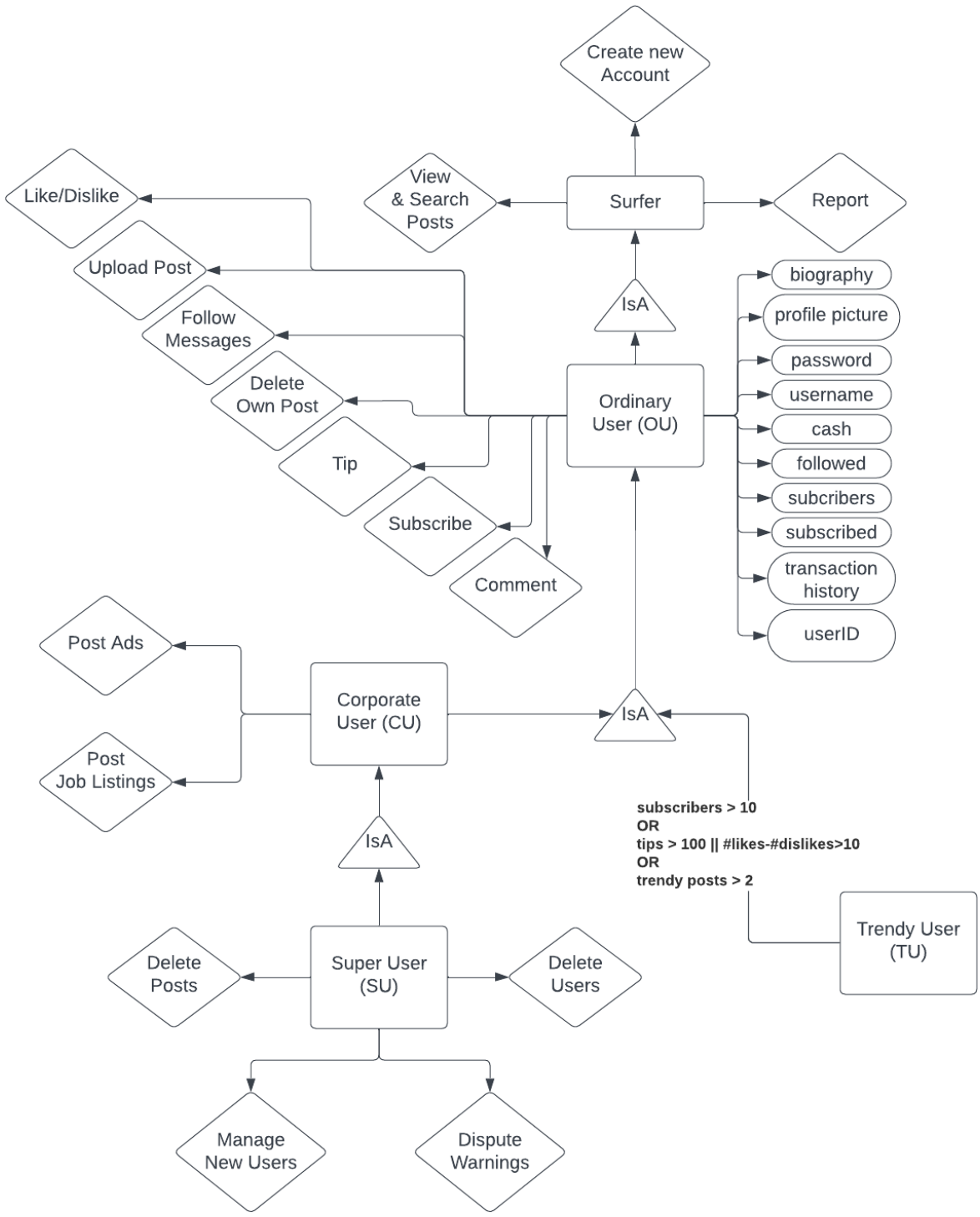


    i. A post object is created.
    ii. The post is then checked against a banned word filter.
    iii. If the post has fewer than 3 banned words:
        1. The banned words are filtered out.
        2. The post is updated with the filtered words.
    iv. If the post has 3 or more banned words:
        1. The Ordinary User (OU) is warned.
        2. The user confirms that the post was not uploaded.
        3. The user is redirected to the home page.
    v. Concurrently, a word count is checked.
        1. If the post has 20 words or fewer, it proceeds to the next step.

        2. If the post has more than 20 words, a Word Overflow Detector is triggered.
    vi. If the post passes the word count and banned word filter, the balance is checked.
        1. If there is sufficient balance, the post is uploaded successfully, and the user is redirected to the home page.
        2. If there is insufficient balance:
            a. The user is redirected to the payment page.
            b. If the balance is filled, the process ends successfully.
            c. If the balance is not filled, the process ends without the post being uploaded.

3. **E-R Diagram for the Entire System**

- Users



i. **Surfer**:
   1. Can create a new account.
   2. Can view and search posts.

    3. Can report other users or content.

ii. **Ordinary User (OU)** inherits from Surfer:
    1. Can like or dislike content.
    2. Can upload posts.
    3. Can follow messages from other users.
    4. Can delete their own posts.
    5. Can tip other users.
    6. Can subscribe to other users.

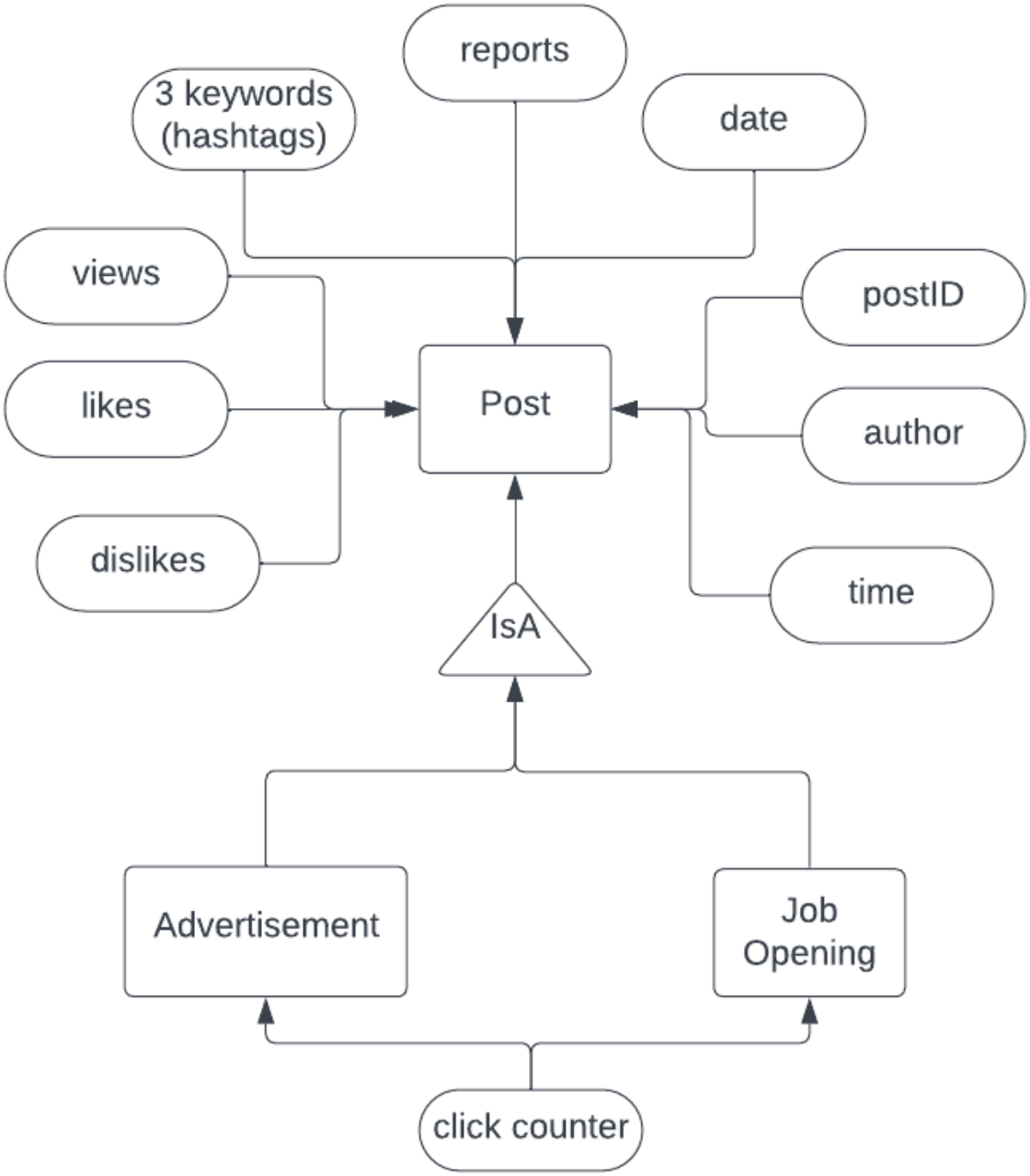
iii. **Corporate User (CU)** inherits from Ordinary User:
    1. Can post advertisements.

iv. **Trendy User (TU)** inherits from Ordinary User:
    1. Identified by having subscribers greater than 10.
    2. Alternatively, could be identified by having a tipping amount equal to or greater than 100, or having a like-to-dislike ratio greater than 10.
    3. Could also be identified by having more than 2 trendy posts.

v. **Super User (SU)** inherits from Corporate User:
    1. Can delete any post.
    2. Can delete any users.
    3. Can manage new users.
    4. Can dispute warnings.

vi. **Attributes** of users:
    1. Biography.
    2. Profile picture.
    3. Password.
    4. Username.
    5. Cash
    6. Followed (users they follow).
    7. Subscribers (users who follow them).
    8. Subscribed (users they are subscribed to).
    9. Transaction history.

- Post



i. **Post**:
   1. Can be identified by a unique postID.
   2. Has an author.
   3. Has a timestamp (time).
   4. Can be associated with 3 keywords (hashtags).

5. Can accrue views, likes, and dislikes.
6. Can be reported, with each report having a date.

ii. **Advertisement** and **Job Opening** are both specialized types of Post (indicated by the "IsA" relationship), meaning they inherit all attributes and relationships of Post:
   1. Advertisement has a click counter to track engagement.
   2. Job Opening has a click counter to track engagement..

4. **Detailed Design**
   - Save New Post

```
export async function SaveNewPost(uniquePost) {
  const BANNEDWORDS = getBannedWords();
  // fitler
  function filterText(text) {
    let counter = 0;
    const filteredText = text.replace(/\b\w+\b/g, (match) => {
      if (BANNEDWORDS.includes(match.toLowerCase())) {
        counter++;
        return '*'.repeat(match.length);
      }
      return match;
    });
    return { counter, filteredText };
  }
  if(filterText(uniquePost.bodyText).counter >2){
    console.log("Too many banned words!");
    return false;
  } else{
    uniquePost.bodyText = filterText(uniquePost.bodyText).filteredText;
  }
  try {
    const client = new MongoClient(MONGOURI);
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(POSTS);

    await collection.insertOne(uniquePost);

    await client.close();
```

```
  } catch (err) {
      console.error('Error saving new post:', err);
  }
}
```

- Handle Like/Dislike/Report/View

```
export async function UpdatePostCounter(postId, type) {
  try {
      const client = new MongoClient(MONGOURI);
      await client.connect();
      const db = client.db(DBNAME);
      const collection = db.collection(POSTS);
      const postObject = await collection.findOne({ '_id': new ObjectId(postId) });
      if (!postObject) {
          console.error('Post not found!');
          return;
      }
      switch (type) {
          case 'like':
          postObject.likes++;
          break;
          case 'dislike':
          postObject.dislikes++;
          break;
          case 'report':
          postObject.reports++;
          break;
          case 'view':
          postObject.views++;
          break;
          default:
          console.error('Invalid Update Command Type!');
      }
      await collection.updateOne({ '_id': new ObjectId(postId) }, { $set:
postObject });
      await client.close();
  } catch (err) {
      console.error('Error updating post counter:', err);
  }
}
```

- Fetch Posts

```
export async function FetchPosts(){
  try {
    const client = new MongoClient(MONGOURI);
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(POSTS);

    const posts = await collection.find({}).toArray();

    if (!posts) {
      console.error('Post not found!');
      return;
    }

    await client.close();
    return posts;
  } catch (err) {
    console.error('Error fetching posts:', err);
  }
}
```

- Delete Posts

```
export async function DeletePost(postId){
  try {
    const client = new MongoClient(MONGOURI);
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(POSTS);

    const query = { '_id': new ObjectId(postId) };
    const post = await collection.findOne(query);

    if (!post) {
      console.error('Post not found!');
      await client.close();
      return false;
    }

    // delete logic
    await collection.deleteOne(query);
```

```
    await client.close();
    return;
  }catch(err){
    console.error('Error deleting post' + err);
    return false;
  }
}
```

- Search for Posts

```
export async function Search(hashtag){ // takes array of hashtags as input
  try {
    const client = new MongoClient(MONGOURI);
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(POSTS);
    const query = {
      hashTags: { $all: hashTags }
    };
    const posts = await collection.find(query).toArray();

    if (!posts) {
      console.error('Post not found!');
      return;
    }

    await client.close();
    return posts;
  } catch (err) {
    console.error('Error fetching posts:', err);
  }

}
```

- Fetch Trending Posts

```
export async function FetchTrending(){
  try {
    const client = new MongoClient(MONGOURI);
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(POSTS);
```

```
    const query = {
      $expr: { $gt: [{ $subtract: ["$likes", "$dislikes"] }, 3] }
    };
    const posts = await collection.find(query).toArray();

    if (!posts) {
      console.error('Post not found!');
      return;
    }

    await client.close();
    return posts;
  } catch (err) {
    console.error('Error fetching posts:', err);
  }
}
```

- Create New User

```
export async function CreateUser(newUser){
// assuming proper filtering for unique name has been performed
  if(newUser.admin === false & newUser.corpo === false & newUser.trendy ===
false & newUser.normal === false){
    console.error('Invalid User Type: they have to have some type of
permission!');
    return;
  }
  if(newUser.userName == null){
    console.error('Invalid UserName: user has no name!');
    return;
  }
  try {
    const client = new MongoClient(MONGOURI, { useUnifiedTopology: true });
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(USERS);

    await collection.insertOne(newUser);

    await client.close();
  } catch (err) {
    console.error('Error saving new user:', err);
  }
```

```
  console.log("User Registered!");
  return null;

}
```

- Get User

```
export async function GetUser(userName){
  try {
     const client = new MongoClient(MONGOURI, { useUnifiedTopology: true });
     await client.connect();

     const db = client.db(DBNAME);
     const collection = db.collection(USERS);

     const query = { userName: userName };
     const user = await collection.findOne(query);

     if (!user) {
        console.error('User not found!');
        await client.close();
        return;
     }

     await client.close();
     return user;
  }catch(err){
     console.error('Error fetching user ${userName}' + err);
     return;
  }
}
```

- Get User Posts

```
export async function GetUserPosts(userName){
  try {
     const client = new MongoClient(MONGOURI, { useUnifiedTopology: true });
     await client.connect();

     // first get the user's unique ID
     const db = client.db(DBNAME);
     let collection = db.collection(USERS);

     let query = { userName: userName };
```

```
    const user = await collection.findOne(query);

    if (!user) {
      console.error('User not found!');
      await client.close();
      return;
    }

    const userId = user._id;

    collection = db.collection(POSTS);
    query = { userId: userId};
    const cursor = collection.find(query);
    const posts = await cursor.toArray();

    if (posts.length > 0) {
      console.log('Posts found');
    } else {
      console.log('No posts found for this user');
    }


    await client.close();
    return posts;
  }catch(err){
    console.error('Error fetching user ${userName}' + err);
    return;
  }
}
```

- Update User

```
export async function UpdateUser(userName, changedUser){
  try {
    const client = new MongoClient(MONGOURI, { useUnifiedTopology: true });
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(USERS);

    const query = { userName: userName };
    const oldUserData = await collection.findOne(query);

    if (!oldUserData) {
```

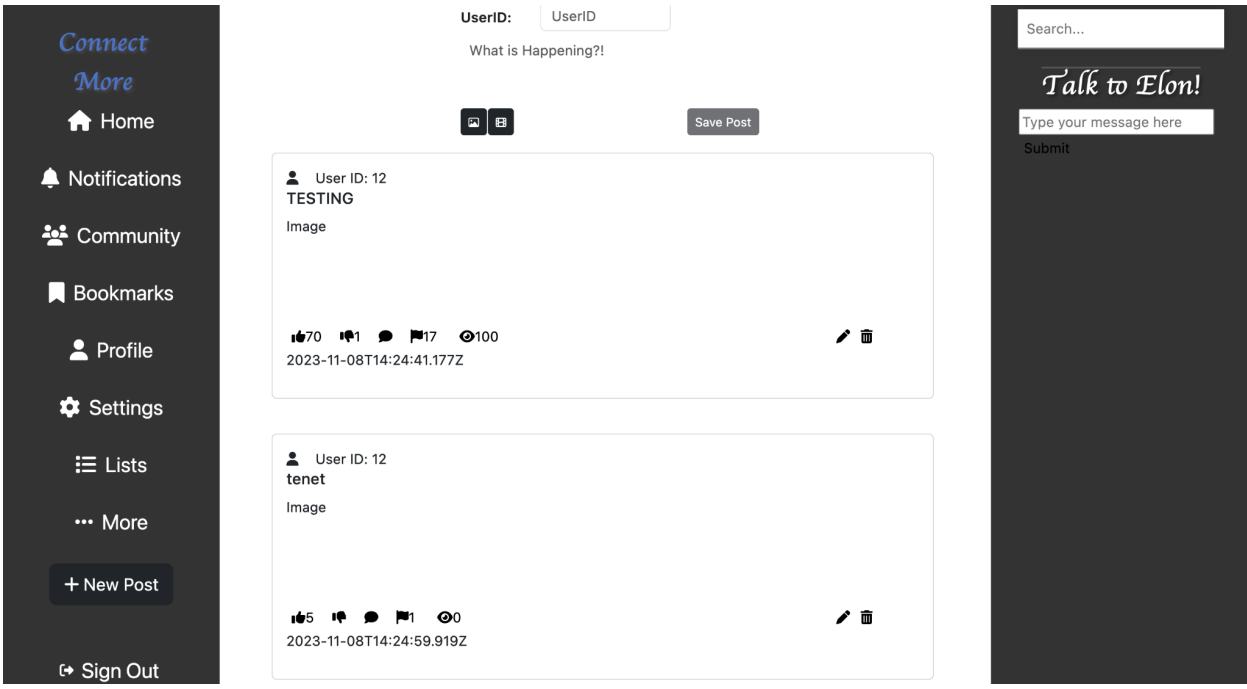
```
        console.error(`User not found for userName: ${userName}`);
        await client.close();
        return;
    }

    const oldUser = new User(
        oldUserData.admin,
        oldUserData.corpo,
        oldUserData.trendy,
        oldUserData.normal,
        oldUserData.userName,
        oldUserData.cash,
        oldUserData.picture,
        oldUserData.bio,
        oldUserData.following,
        oldUserData.interests
    );
    oldUser.updateWith(changedUser);

    await collection.updateOne({ userName }, { $set: oldUser });
    console.log("User Updated!");

    await client.close();
    return oldUser;

  } catch(err){
    console.error('Error fetching user ${userName}' + err);
    return;
  }
}
```

- Delete User

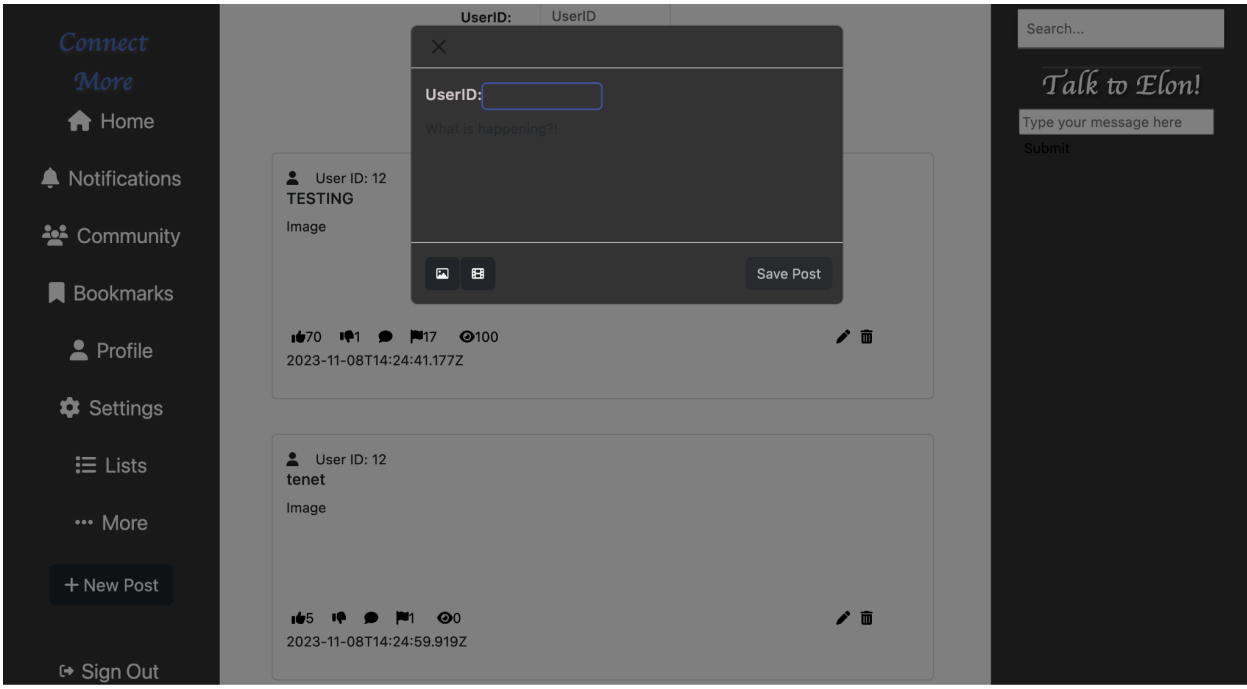
```
export async function DeleteUser(userName){
  try {
    const client = new MongoClient(MONGOURI, { useUnifiedTopology: true });
    await client.connect();

    const db = client.db(DBNAME);
    const collection = db.collection(USERS);

    const query = { userName: userName };
    const user = await collection.findOne(query);
```

```
    if (!user) {
        console.error('User not found!');
        await client.close();
        return;
    }

    // delete logic
    await collection.deleteOne(query);

    await client.close();
    return;
  }catch(err){
    console.error('Error fetching user ${userName}' + err);
    return false;
  }

}
```

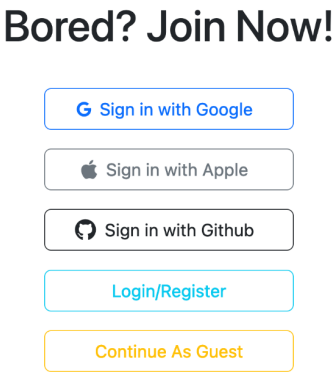
5. **System Screens**
    - *Home Page*



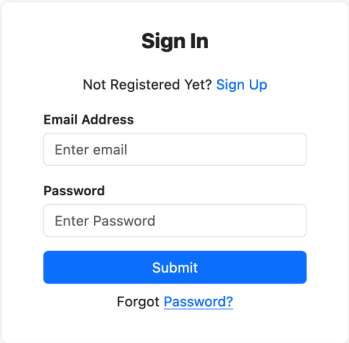
- *Upload Button Clicked*



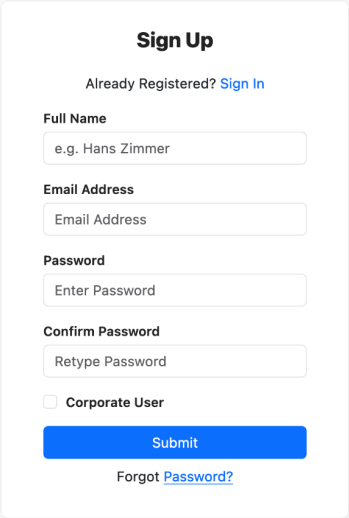
- *Landing Page*



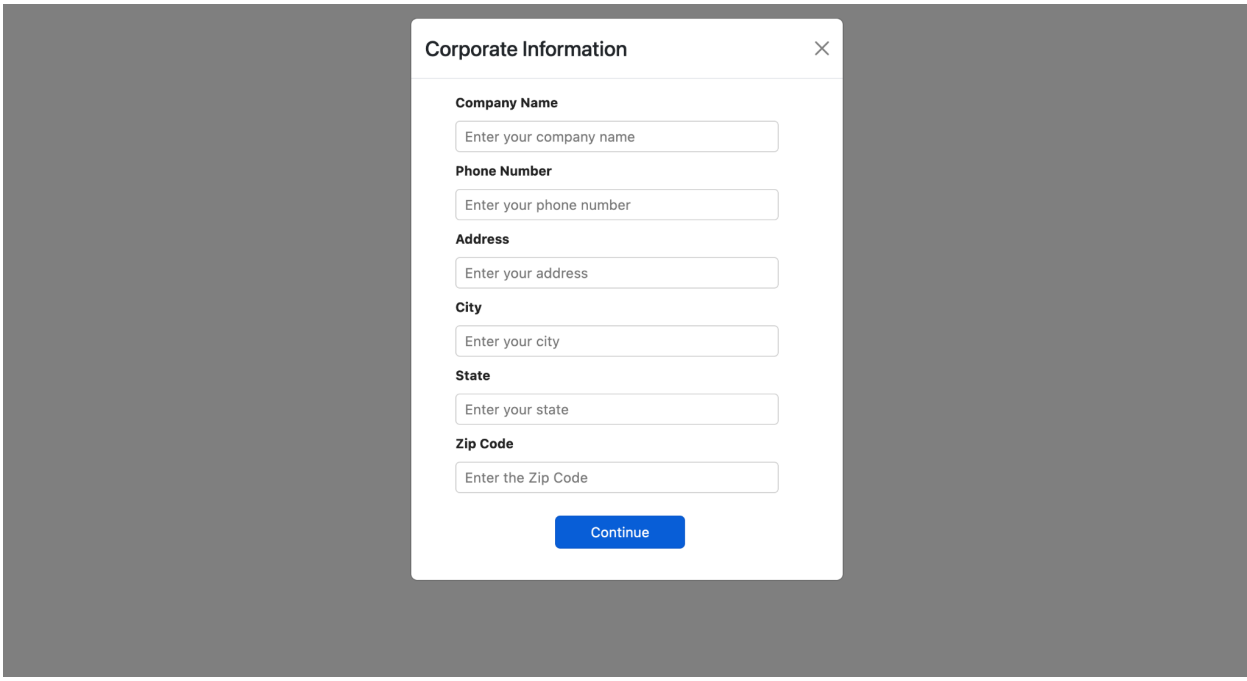
- *Login*



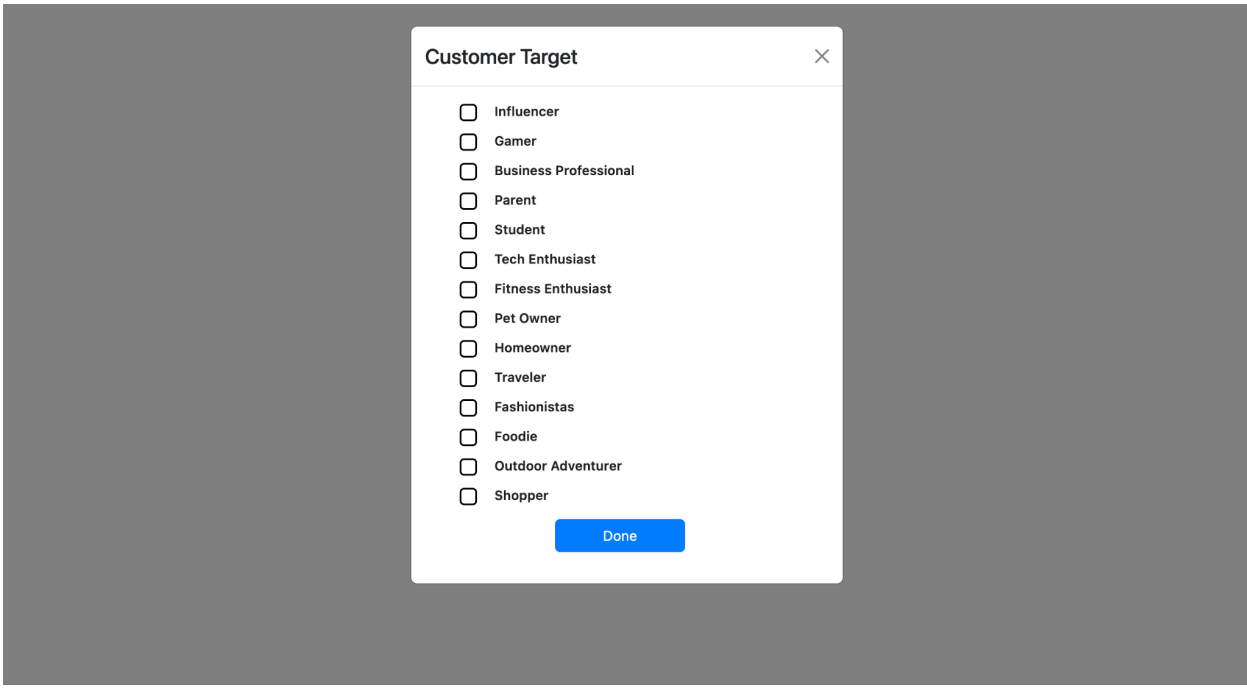
- *Sign Up*



- *Corporate User Sign Up*



- *CU Target*



- [Prototype sample of uploading posts!]
- [Alternative Link]

6. **Memos of Group Meetings and Teamwork Concerns**
    - Group meetings occur on a regular weekly basis - with the rare exception of the occasional week.

- Each member reports on the progress made on their deliverable of the week.
- Every team meeting since the beginning of the project's start has been productive and constructive.
- Lesser experienced team members ask for support and more experienced team members offer their support.
- The two backend engineers update each other on the progress/changes they've made to the software.
- The three front end engineers update the backend engineers on the progress they've made on the frontend.
- At the end of the meeting, next week's deliverables are set with a reasonable expectation.
- The only concerns are of not having enough people running the backend - while having the majority of the team focus on the front end.
    i. We may have to quickly train & get some front-end engineers to work backend to ease the workload.
- Another concern is that the two backend engineers are overwhelmed with other responsibilities, slowing down their progress on deliverables.

7. **Repository Address**
    - Link to Repo
        i. We gotta add THIS report to the repo as well it sounds like