



---

**CS 30700**  
**Design Document**  
**02.07.24**

**Team 28:**

- Aaryan Srivastava
- Aditya Patel
- Atharva Gupta
- Rishabh Pandey
- Sanjhee Gupta

# Index

<b>Purpose.....</b>	<b>3</b>
1.0 Non-Functional Requirements.....	3
1.1 - Front End.....	3
1.2 - Back End.....	4
2.0 Functional Requirements.....	4
2.1 Users - Virtual Closet.....	4
2.2 User Settings.....	5
2.3 Users Social Media.....	5
2.4 User's Calendar.....	5
2.5 Marketplace.....	5
<b>Design Outline.....</b>	<b>7</b>
1.0 SwiftUI Client.....	7
2.0 Python-Firebase Server.....	8
3.0 Firebase Database.....	8
<b>Design Details - TODO.....</b>	<b>9</b>
1.0 Class Design.....	9
1.1 Descriptions of Classes and Interaction between Classes.....	10
2.0 Sequence Diagram.....	13
2.1 User Signup/Login Sequence.....	14
2.2 Social Post Edit Sequence.....	15
2.3 Clothing/Outfit Edit Sequence.....	16
3.0 Navigation Flow Map.....	17
4.0 UI Mockup -.....	18
5.0 API Routes.....	25
<b>Design Issues.....</b>	<b>26</b>
1.0 Functional Issues.....	26
1.1 - Database Issues.....	26
1.2 - Authentication Issues.....	26
1.3 - Intersystem Communication Issues.....	27
2.0 Non functional issues.....	28
2.1 Performance.....	28
2.2 Client Security.....	28
2.3 Server/Firebase Security.....	28
2.4 Scalability and Maintainability.....	29

# Purpose

Efficiently managing a wardrobe, crafting the perfect outfit, and discovering new fashion pieces can be challenging and time-consuming for many individuals. Traditional methods of wardrobe organization and outfit planning often lack objectivity and struggle to keep pace with expanding personal collections. Current Smart Closet applications on the market, like [GetWardrobe Outfit Planner](#), [OpenWardrobe](#), and [Acloset](#), have a similar management system when displaying a user's clothes, but they often require links to brands to upload the clothes onto the smart closet. Our aim is to simplify this process by simply adding a scanner feature. Additionally, to stimulate activity and traffic, we will add a social media aspect of our app for users to share their inspired fits with friends and family.

Our project aims to revolutionize these processes by developing a sophisticated management system for clothing. This system will offer innovative outfit suggestions and elevate personal fashion styles, taking into account modern trends, price considerations, and the unique preferences of each user, as reflected in their existing wardrobe.

Our goal is to transform closet management and outfit planning into a streamlined, personalized experience. By integrating user-specific data, such as their current wardrobe items, we will provide tailored fashion recommendations, enabling users to effortlessly explore and integrate new trends and outfits. Our solution is designed to alleviate the overwhelming nature of a growing wardrobe, making fashion-forward choices more accessible and enjoyable for everyone.

## 1.0 Non-Functional Requirements

### 1.1 - Front End

As a developer, I would like to:

- Keep an organized file system with easy access and proper naming convention (ModelCards are a separate directory from Screens).
- Have in-built IOS Simulators and an iPhone for testing the app as development continues.
- Have the app efficiently communicate with the server without too many unnecessary requests.
- Have a clean, easy-to-use, and appealing UI for our users to use.
- Implement an AppCheck Security check to ensure our server only accepts and responds to verified requests from our app.
- Use notifications frequently to foster activity with the app.
- Have the server handle up to 200-500 users without lag or delay

- Have an easy-to-understand network layer that translates information between the server and the front-end
- Test the app on several IOS versions to ensure functionality for users who haven't updated the app
- Use testflight for official tests using the Firebase database and test concurrency/server activity.

## 1.2 - Back End

As a developer, I would like to:

- Keep server requests as limited as possible to avoid redundancy.
- Implement an MVVM approach where the back-end and front-end are as separated as possible.
- Store documents in Firebase efficiently with all necessary data only.
- Utilize Firebase's several services for IOS development such as metrics, image labeling/scanning APIs, and notification setups that we can use to better facilitate the creation of our app.
- Implement proper cybersecurity by making user data stored on OOTD servers encrypted using AES Encryption. Server side security will include OAuth 2.0 flows for all RESTful API Calls, and hidden env variables.
- Use DigitalOcean for hosting, as it provides automatic HTTPS and custom domain deployment free of cost, and 4 GiB of memory and 2 vCPUs for 6 months for hosting our server.

## 2.0 Functional Requirements

### 2.1 Users - Virtual Closet

As a user, I would like to:

- Take pictures of my clothes and upload them to the app, where I can view items inputted.
- Optimize and standardize all clothing pictures on the app to have a white background.
- Include an experimental outfit planner, where the user can scroll through different clothes you own to find a desirable combination.
- Be able to categorize my clothes based on tops, shorts, jackets, pants.
- View an outfit of different clothes I've selected and see how they fit together.
- Sort clothing items by type, size, and color
- Allow users to favorite clothes.

- Be able to save outfits that I like.

## 2.2 User Settings

- Create an account that I can sign in and out of while including the option of a biometric login (Face ID/TouchID)
- Have secured entry with login and password.
- Input clothing data (size, preferences) for creating preferences that the app can standardize to.
- Have the option to change my account's credentials.
- Get photo album/camera authorization.

## 2.3 Users Social Media

As a user, I would like to:

- Incorporate a feed into the app which shows posts from your friends and yourself.
- Send friend requests / accept friend requests
- Pin my favorite outfits onto my profile
- Share a picture of my outfit once a day, having the clock to post reset at midnight.
- Create a recap that replays all previously saved outfit
- See my past outfits and what day I wore them.
- Be able to block users from seeing my profile
- Be able to remove users from my profile
- Allow users to react to one another's posts.
- See what my friends are wearing by encouraging interaction through posting.

## 2.4 User's Calendar

As a user, I would like to:

- Create a calendar feature which allows users to plan/save outfits for certain days/events for more organized planning.
- See what outfits I've worn previously in the calendar to avoid redundancy or to see patterns.
- Get a reminder at the beginning of the day about my planned outfits.

## 2.5 Marketplace

As a user, I would like to:

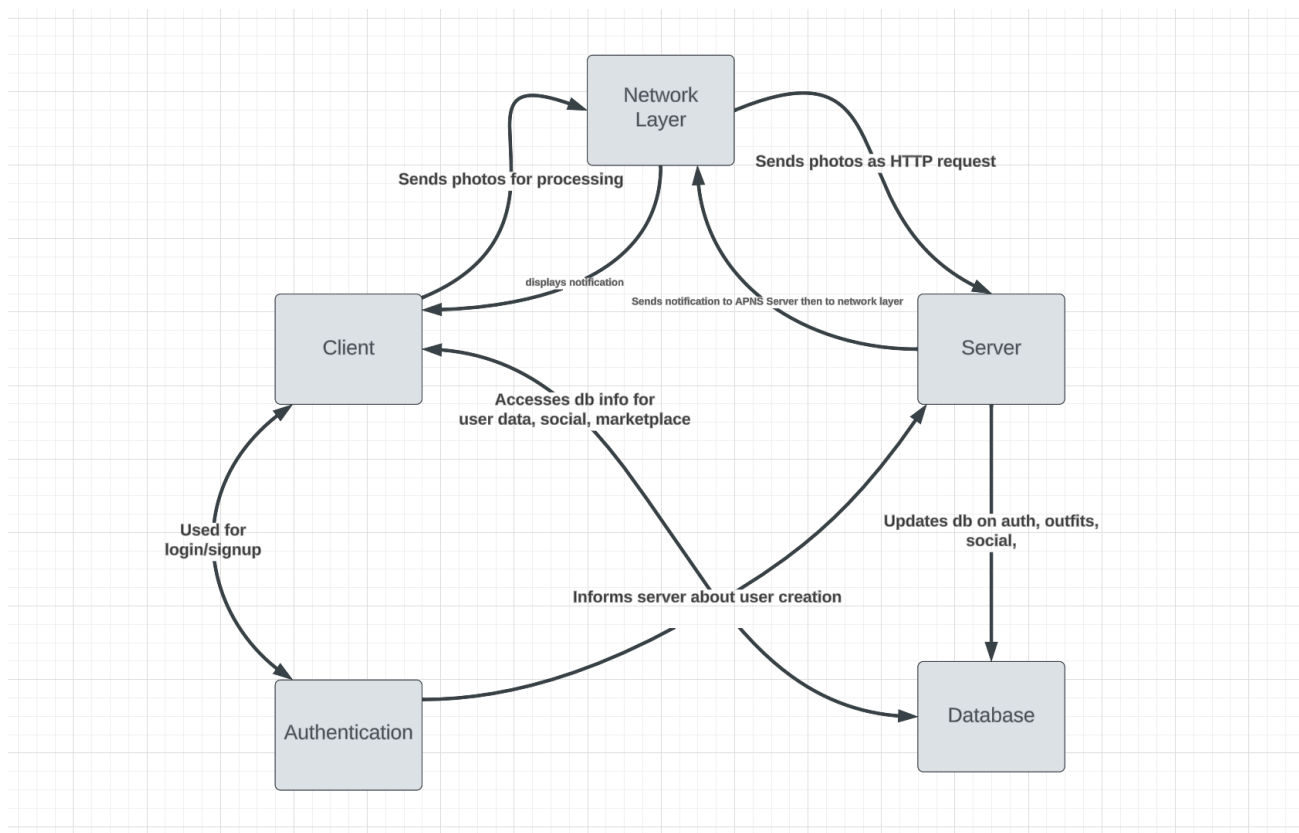
- Get shopping recommendations based on user size, style, and preferences
- Add a marketplace which imports clothing items from stores.
- Include a filtering option with a marketplace that allows you to search for specific brands, clothing types, color, and price range.
- Be able to view fits on a virtual mannequin

- Wishlist feature for the future fits through the shopping page
- Marketplace features brands and uses affiliate links that users can click on to navigate to the retailer's site to raise money.
- Add a review system for the marketplace that will allow users to leave reviews on items in the marketplace.

# Design Outline

## High level Overview

This project will be a SwiftUI iOS mobile app that allows users to upload, select, and post clothing. This mobile application will use a client-server model where the client is based on SwiftUI and our server will be built on Python and Firebase. There will also be a network layer that is connected to the client and the server which is responsible for sending photos as HTTP requests.



## 1.0 SwiftUI Client

Our Client will be based entirely on SwiftUI. Every user will have a Client in the form of an app. We will use HTTP requests to send to our server and respective APIs, using a JSON format. The data received will be parsed in the Network Layer, which will give a valid output to the front-end for display.

## 2.0 Python-Firebase Server

Our Backend will be built on Python and Firebase. The server will handle all traffic between all the client servers and the main backend will handle all the documents with updates and retrieval. On top of that, the server will send relevant information to the requesting client in the form of a JSON payload. The JSON payload will go through the Network layer before going to the client.

## 3.0 Firebase Database

Our database will be hosted by Google Firebase. It is a NoSQL data structure based on a tree. It will store documents storing user information and store item documents. User information will include data from the social media branch as well. Any queries will be handled by firebase and sent to the server.

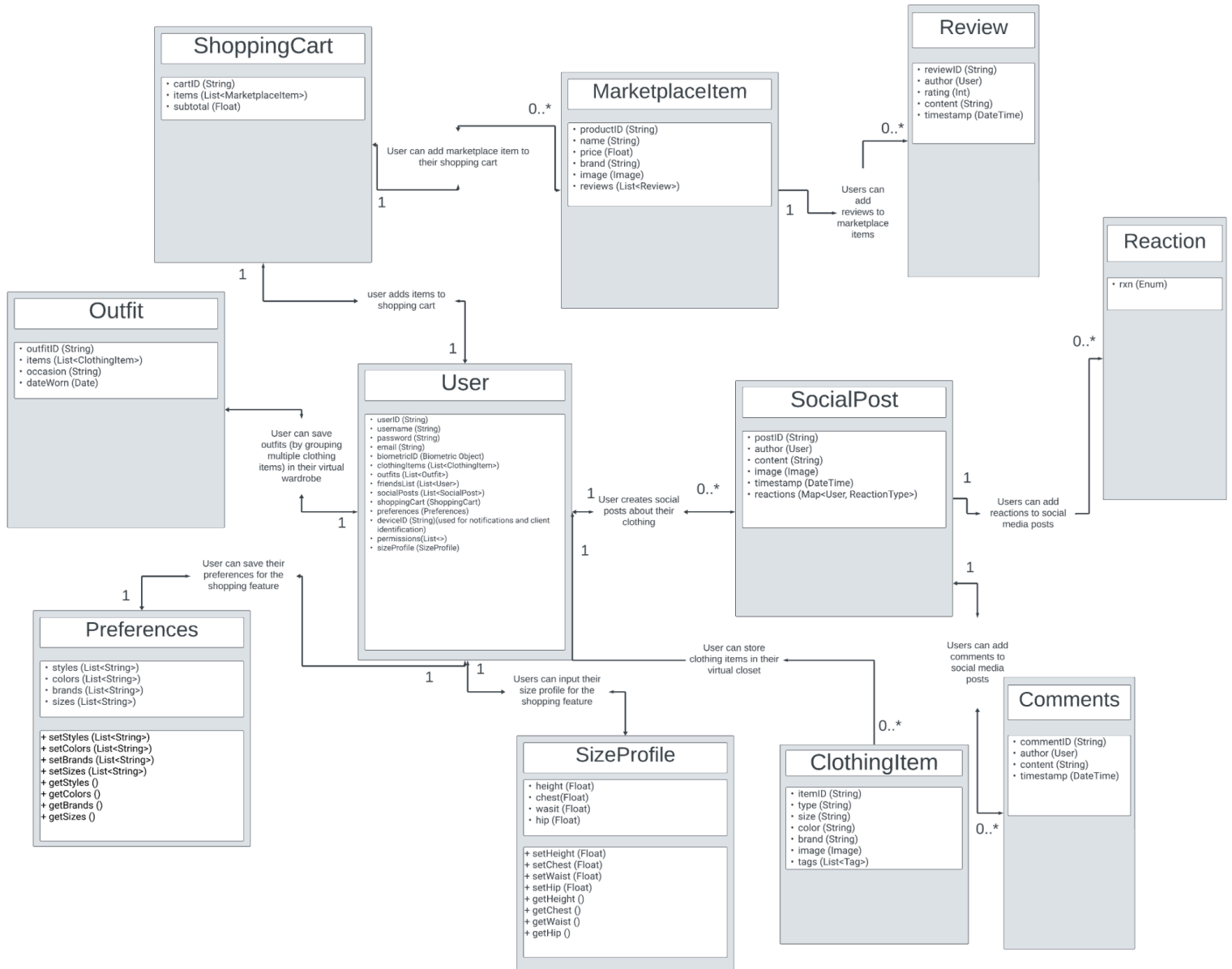
## 4.0 Network Layer

The Network Layer is an intermediary between our front-end and back-end. It will receive information from the backend in the form of a JSON payload and parse it appropriately before sending it to the client. Additionally, any information sent via an HTTP request will go through this layer as well before going to the main backend server.



# Design Details - TODO

## 1.0 Class Design



## 1.1 Descriptions of Classes and Interaction between Classes

Our data class design is based on the objects within our application and how the attributes within each class are necessary for the interaction between classes. Within each class, there is a list of attributes that are the characteristics of the object.

- **User**
  - User object is created when a user creates an account.
  - Each user will be assigned a unique userID.
  - Each user can customize their username, password, and email for login/account settings purposes
  - Each user can also edit their username, password, and email when necessary in the account settings.
  - Each user will also have a unique biometricID for logging in.
  - Each user will also have a list of their clothing items, outfits, and a list of permissions.
  - Each user will also have access to their friends list, shopping cart, social posts, and preferences based on the features of the app
  - Each user will also have a deviceID which will be used for notifications and client identifications.
- **ShoppingCart**
  - When a user creates an account, they will have access to their own shopping cart.
  - Shopping Cart object is created when a user inputs an item from a clothing store.
  - Each shopping cart will be assigned a unique shopping cart ID.
  - Each shopping cart will also have a list of items that the user adds from clothing stores.
  - Each shopping cart will also have a subtotal variable for the total items in the cart.
- **MarketplaceItem**
  - MarketplaceItem object is created when a shopping cart item is created
    - User selects an item from a clothing store.
  - Each MarketplaceItem object has a unique productID.
  - Each MarketplaceItem object has the name, price, brand, image, and a list of reviews for the item
- **Review**
  - Review object created when a Marketplaceitem object is created
    - User selects to review an item from the marketplace
  - Review object will contain a unique reviewID that is assigned to each individual review that is posted by the user

- It contains author, rating, content, and a timestamp for the review object
- This object will be creating reviews for items in the marketplace
- **Outfit**
  - Outfit object is created when a user object is created
  - Each object will contain a list of clothing items (shirt, trousers, shoes, hat, glasses)
  - Each object will contain an attribute that notes down the occasion
  - Each object will also have the date when the outfit was worn
- **SocialPost**
  - SocialPost object is created when a user object is created
  - It contains a unique postID that is assigned to each individual social post
  - It contains the author, content, image, and a timestamp
  - Each social post will also have a list of reactions which allows the friends of users to react to the social post
- **Comments**
  - Comment object is created when the Social Post is created
    - The friend of a user adds a comment to the post
  - It contains a unique commentID that is assigned to each individual post comment for the user
  - It contains the author, content, and timestamp for the comment that is listed under the socialPost
- **Reaction**
  - Reaction object is created when a a SocialPost object is created
    - The friend of a user reacts to the social post of the user
  - The picture of the object will be a red heart which will be used for reacting to a post
- **ClothingItem**
  - ClothingItem object is created when a user adds the item from the marketplace or scans the closet
  - It contains a unique itemID that is assigned to each individual clothing item for the user
  - It contains the type, size, color, brand, and the image for the clothing item
  - Each clothing item object will also have a list of tags
- **SizeProfile**
  - SizeProfile object is created at the initial stage when a user object is created
  - It contains the attributes of height, chest, waist, and the hip size of the user

- With the height, chest, wait, and the hip size of the user, we can create a preference object for the user
- **Preferences**
  - Preferences object will be created when the SizeProfile object is created
  - It contains the styles, colors, brands, and sizes from the user's inputs
  - Each preferences object will allow the marketplace to be filtered and allow the app to standardize to these customizations

## 2.0 Sequence Diagram

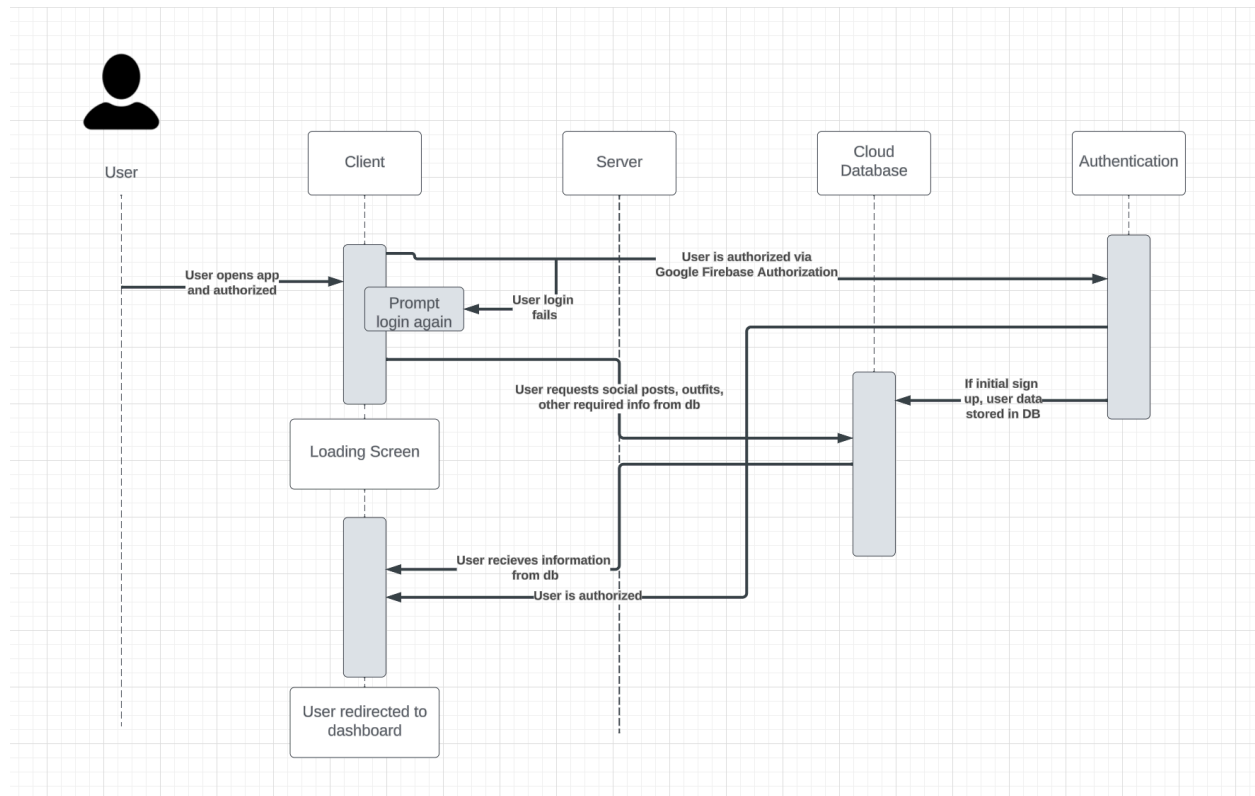
The following sequence diagrams are high-level views into complex sequences. Each of the sequences below have far more interactions and connections than visible in the flows, but to simplify them we have chosen to only display the most important and abstract sections in each sequence. We also chose not to delve directly into the classes and functions, and instead depict our major systems interactions with each other. The four major systems involved in these sequences are:

1. Client: This refers to the iOS app and will deal with the UI interactions with the user.
2. Server: This refers to the cloud-hosted web server, which interacts with various APIs, notifications, alerts, and other various technologies.
3. Cloud Database: This refers to our main database, where long-term storage of User info and stored data will be, along with the social posts and data. The database can be accessed via Firebase, from either the server or the Client.
4. Authentication: This refers to Firebase Authentication. Mostly dealing with user login on the client side, this system is heavily involved with many small interactions. Confirming the user is logged in securely is very important. Authentication doesn't have many listed interactions outside of signup/login because the user is assumed to be authenticated for the rest of the sequences.

The user will access OOTD via interactions with the client. The iOS client's backend will communicate more with the other 3 systems seamlessly. A major objective for our app will be having interactions between systems be asynchronous, creating a seamless and smooth front-end experience for the user.

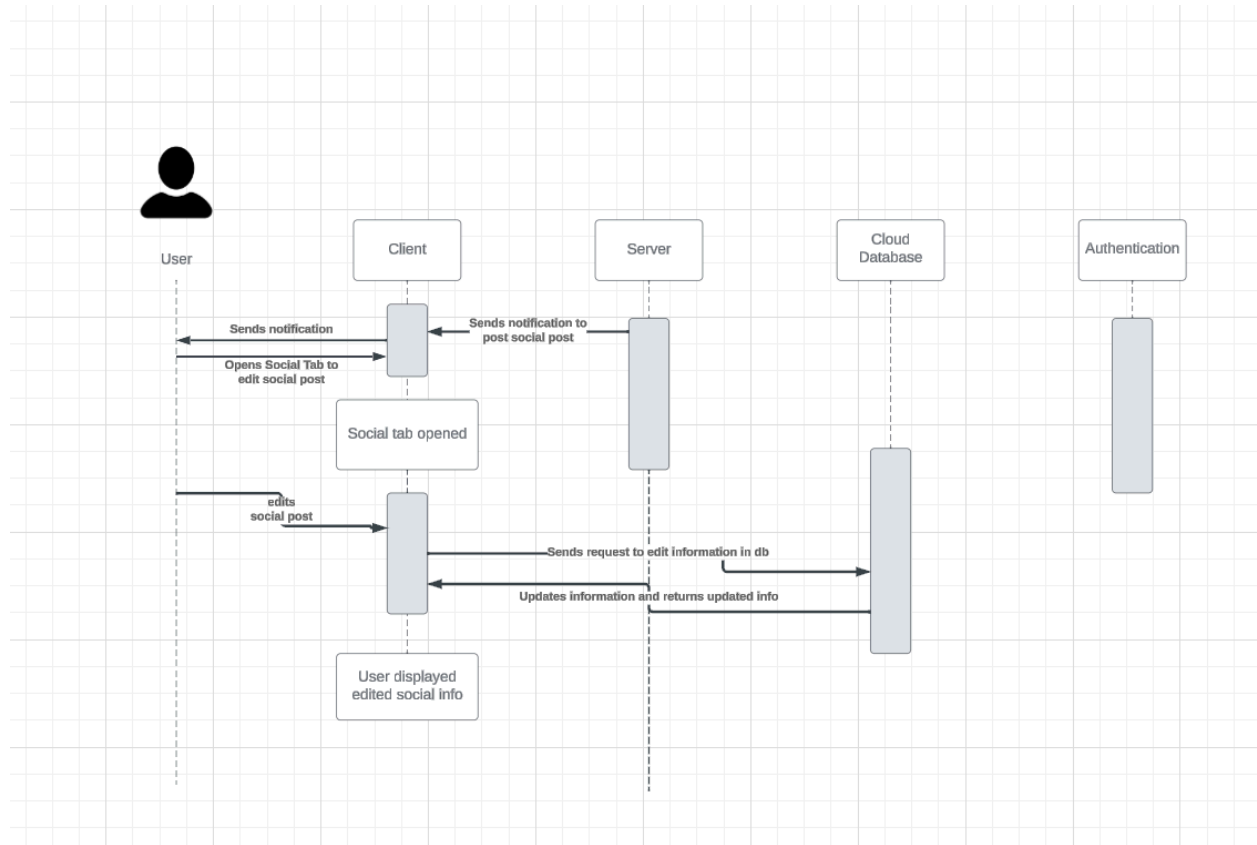
## 2.1 User Signup/Login Sequence

The User Login Sequence deals mainly with Google Firebase Authentication. Google Firebase Authentication can use either an Email/Password or “Login with Google” method. Both of these methods don’t require server use because of a direct link with Google Firebase from the iOS app. This greatly simplifies our authentication process and User Login Sequence. However, we do need to send some information about the user to our database on Sign Up.



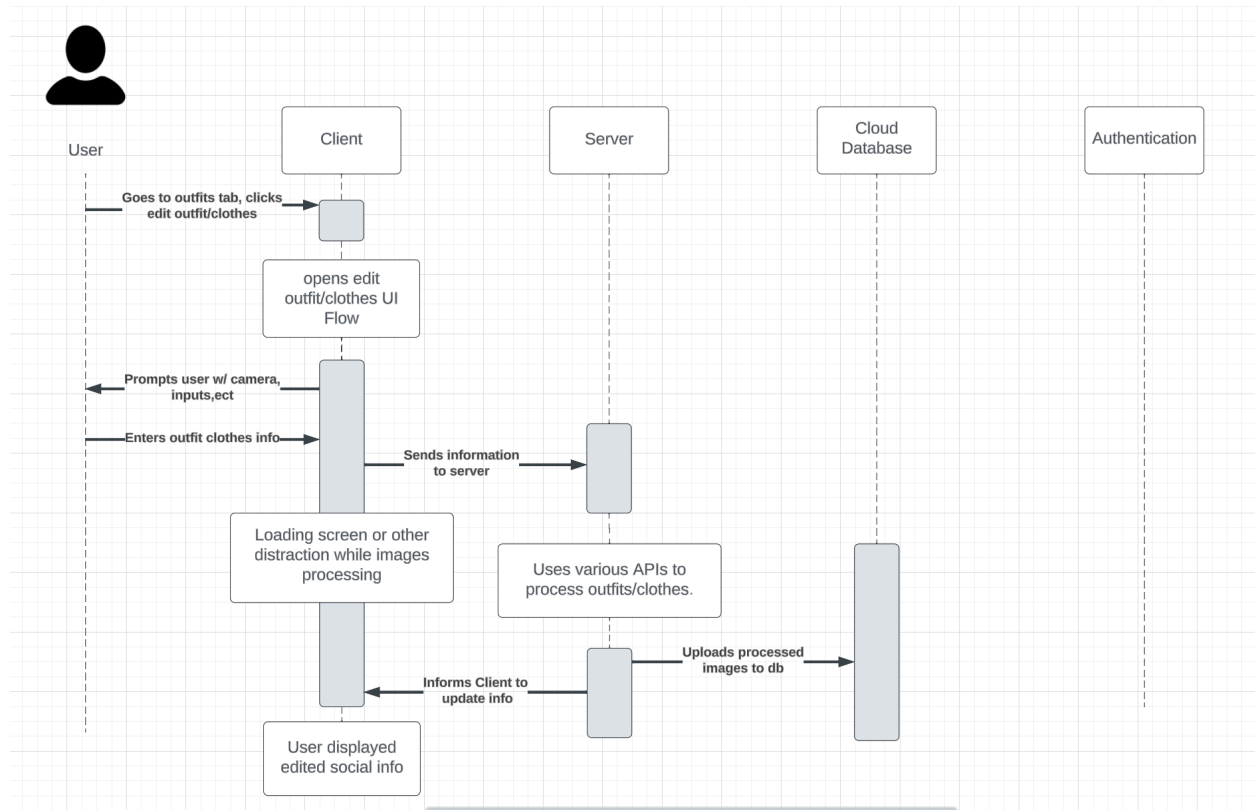
## 2.2 Social Post Edit Sequence

The social post edit sequence describes the processes that occur when a user creates a social post, edits it, or reacts to other social posts. All these processes are similar in their backend communication.



## 2.3 Clothing/Outfit Edit Sequence

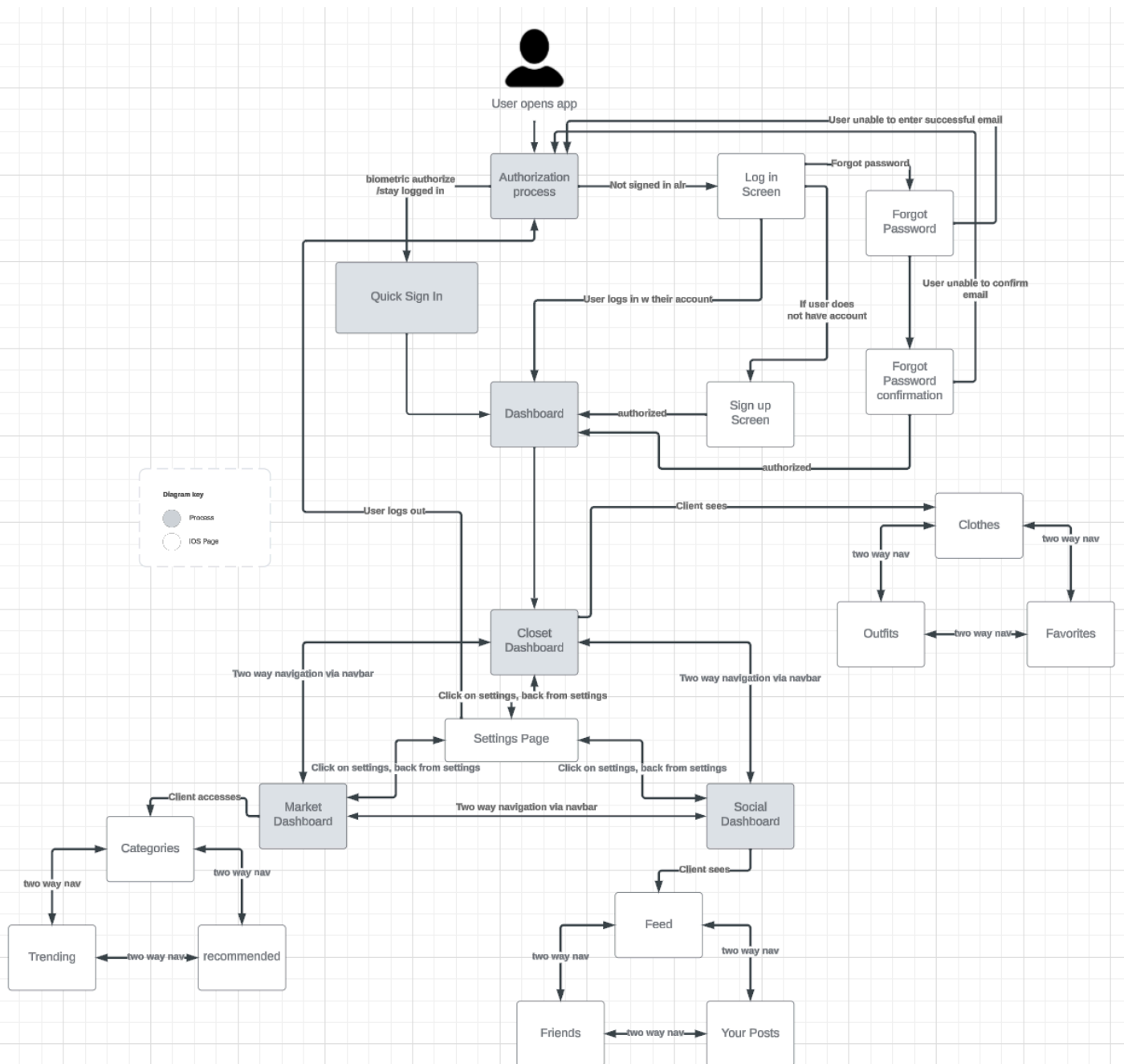
Clothing/Outfit edit sequence is similar to the social post edit sequence in that it mostly deals with the cloud database. However, this sequence has far more interactions between the client and user than between any other systems.





### 3.0 Navigation Flow Map

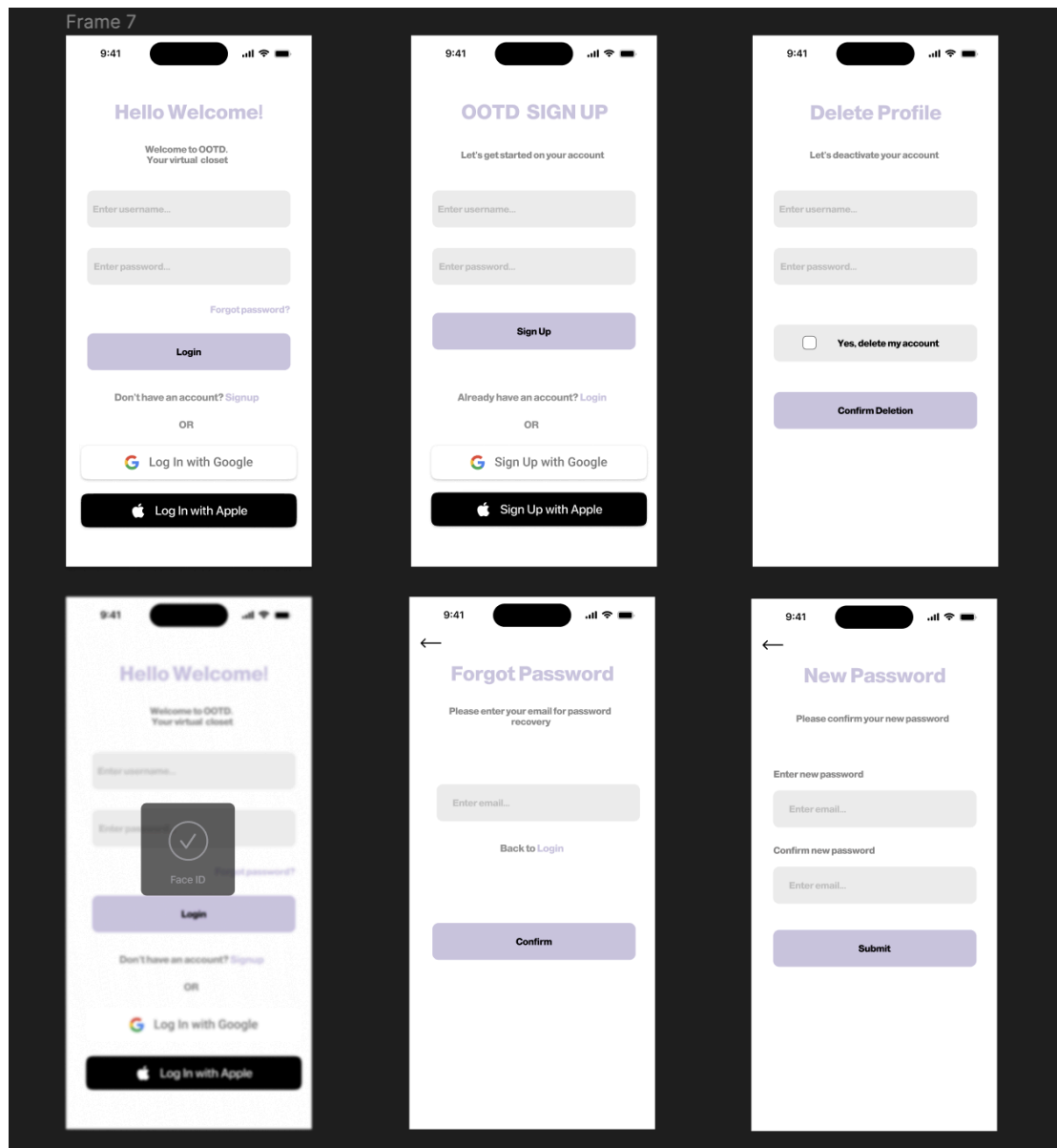
This simple navigation flow map displays the overarching flow of our application. It displays all major display screens and navigation flows for the user. A display key is provided.



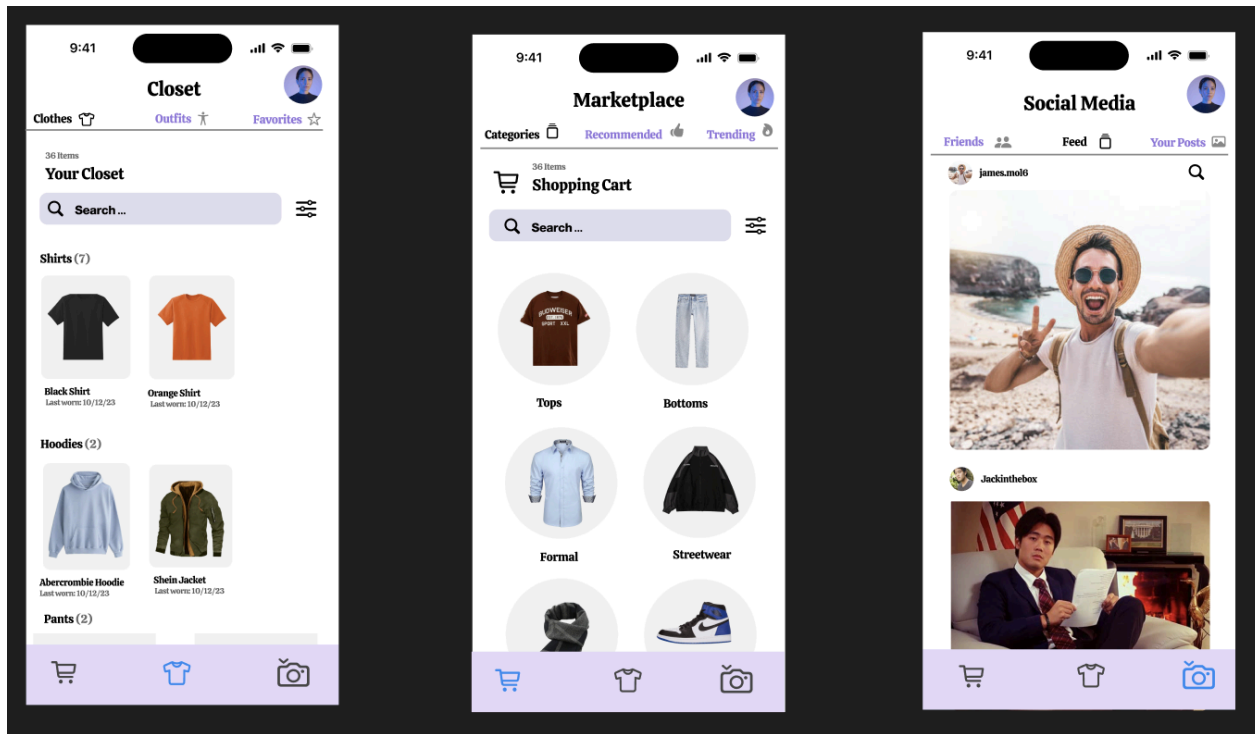
## 4.0 UI Mockup -

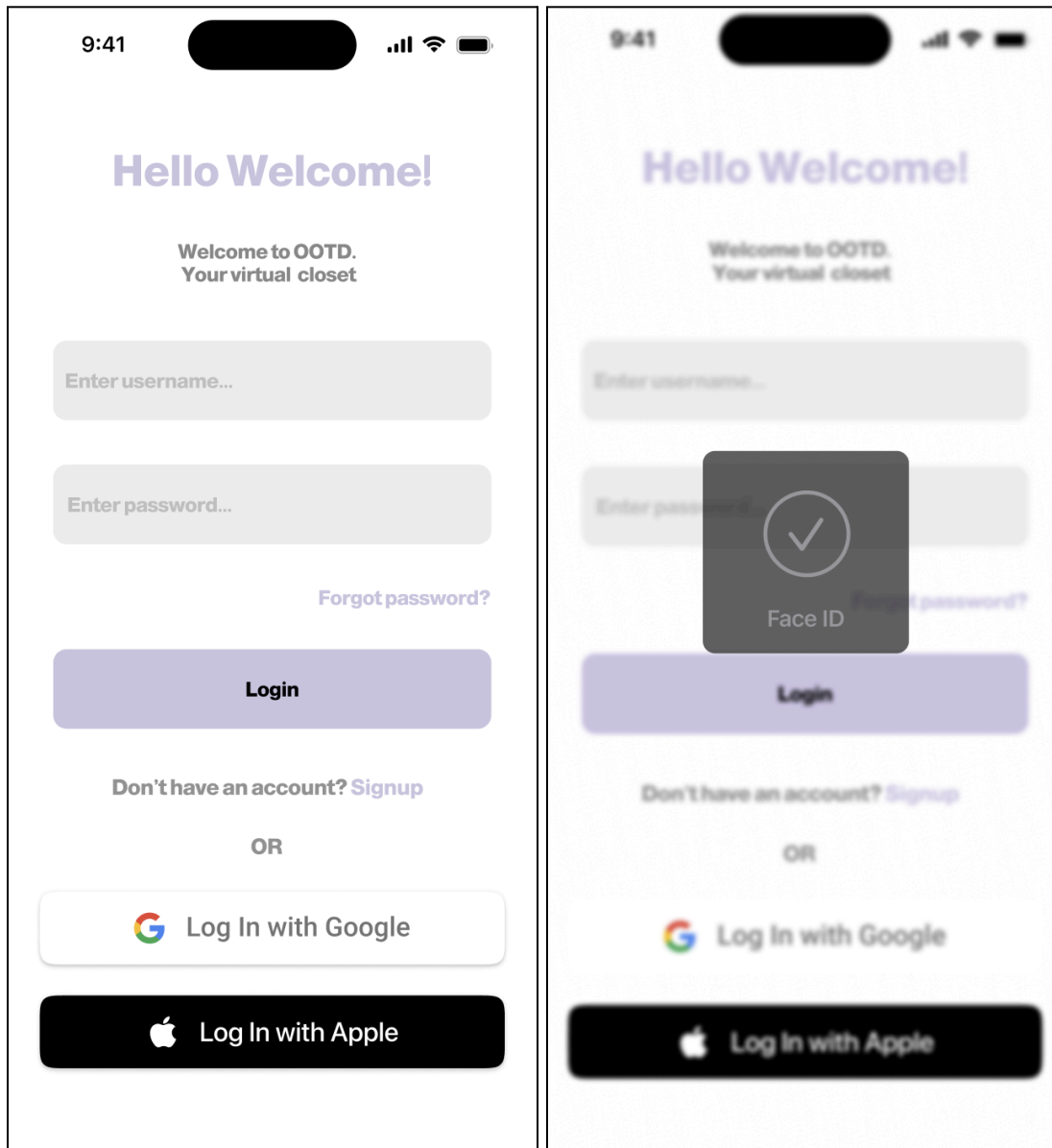
The UI is clean, organized, and concise for the user. All related functions will be on the same pages or sub pages that can be traversed through button presses. The color scheme is calming, with neutral colors creating an environment where the user can focus on the features of the app and not be distracted by the colors.

### ACCOUNT SETUP LAYOUT:





## MAIN SCREEN LAYOUT:











**LOGIN:**

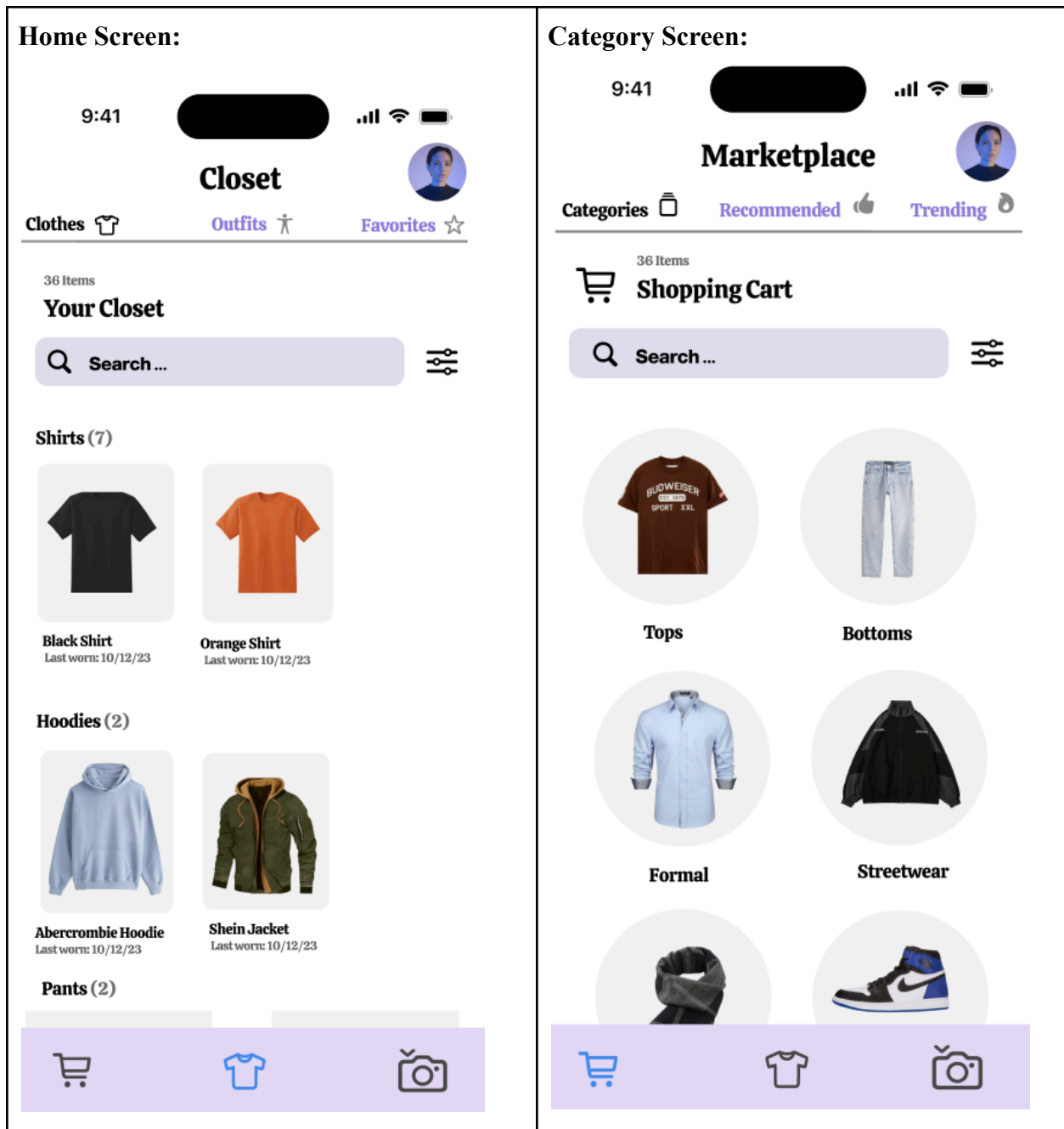
The account setup interface features a minimalistic design with the convenient option of signing up and logging in through Google and Apple. This eliminates the need to create separate credentials for the app. This is the login page. There are three buttons: one for normal login with username and password, one for login with Google, and one login with Apple. There is also a button for signing up in case the user doesn't have an account and a button for password recovery.

Sign up:	Delete Account:
<div>9:41 [Redacted] [Signal] [Wi-Fi] [Battery]</div> <div><h2>OOTD SIGN UP</h2><p>Let's get started on your account</p><div>Enter username...</div><div>Enter password...</div><div>Sign Up</div><p>Already have an account? <a href="#">Login</a></p><p>OR</p><div> Sign Up with Google</div><div> Sign Up with Apple</div></div>	<div>9:41 [Redacted] [Signal] [Wi-Fi] [Battery]</div> <div><h2>Delete Profile</h2><p>Let's deactivate your account</p><div>Enter username...</div><div>Enter password...</div><div><input type="checkbox"/> Yes, delete my account</div><div>Confirm Deletion</div></div>

This is the Sign up page that requires the user to enter the username and password. There are also buttons at the bottom of the screen for the user to sign up through Google and Apple. In the event that the user chooses to discontinue using the app, they can delete their profile as well. The Delete Account page requests the user to enter their username, password, and confirm their deletion by checking the box. This ensures deletion was confirmed by the user and wasn't a mistake.

Forgot Password:	Enter New Password:
<div>9:41    </div> <div>←</div> <div><b>Forgot Password</b></div> <div>Please enter your email for password recovery</div> <div><input type="text" value="Enter email..."/></div> <div><a href="#">Back to Login</a></div> <div><b>Confirm</b></div>	<div>9:41    </div> <div>←</div> <div><b>New Password</b></div> <div>Please confirm your new password</div> <div><b>Enter new password</b></div> <div><input type="text" value="Enter email..."/></div> <div><b>Confirm new password</b></div> <div><input type="text" value="Enter email..."/></div> <div><b>Submit</b></div>

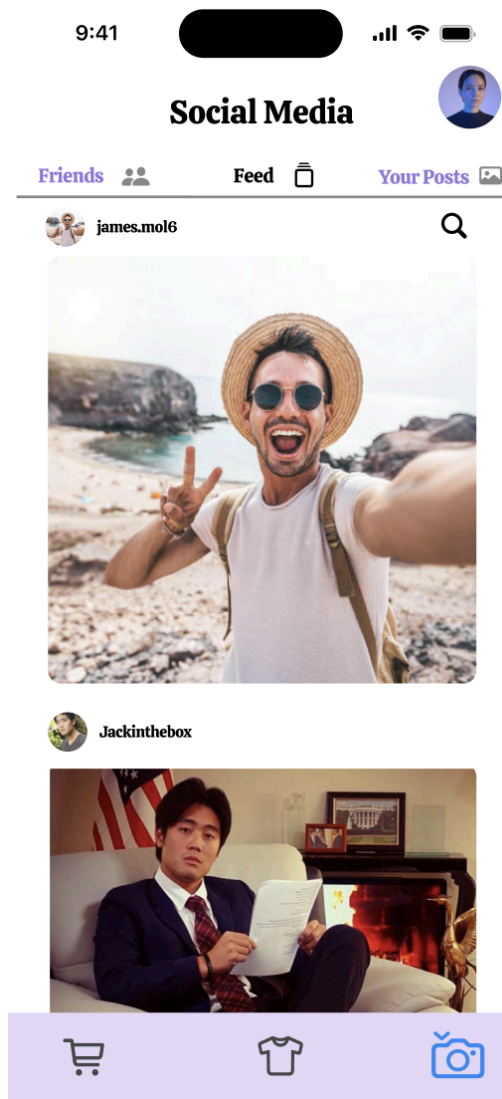
For added flexibility, users have the option to create a new password in case they forget their current one. To initiate this process, the users have to enter the email associated with their account, and then proceed to change their password. Once the user gets the new password, they can enter on the confirmation screen.



This Home screen features the User's closet, housing all the images they have scanned and uploaded to the app. For a cleaner look, the app automatically removed the background from the pictures. Within their wardrobe, users can curate their own outfits, which are accessible via the "Outfits" tab. The app also grants users the flexibility to delete individual clothing items and outfits as needed. For efficient navigation, users can mark outfits as "Favorite", eliminating the need to scroll through their entire collection. This allows users to quickly locate and view preferred outfits with minimal effort.

The Category screen features the Marketplace, where users can browse different clothing apparel from the latest brands. The clothes have been categorized based on different style aesthetics and clothing type. Under the “Recommended” tab, users can view a set of different clothes based on their current closet. Users can also browse trending items under the “Trending” tab.

### SOCIAL MEDIA SCREEN:



This is the Social Media aspect of the app. The user can see a list of all the friends they have added on the app by clicking on the “Friends” tab. They can also access their profiles which will feature their pinned outfits. The feed shows all the pictures the user’s friends have been posting. The users are motivated to post pictures of their outfits of the day as that is the only way they can see the pictures their friends have been posting. The “Your Posts” tab contains all the pictures the user has posted.



## 5.0 API Routes

The API routes are the URIs that are made available by the API server and they define the communication contract between the client and the server in our client-server architecture. Each route represents a resource or set of resources that can be provided by the server (via GET requests), or created and updated by the client (via PUT, POST, and DELETE requests). A resource is identified by an id, which matches the Primary Key of that resource in our database.

Route	Supported HTTP Methods
/users	GET, POST
/users/{userId}	GET, PUT, DELETE
/users/{userId}/clothing	GET, POST
/users/{userId}/clothing/{itemId}	GET, PUT, DELETE
/users/{userId}/outfits	GET, POST
/users/{userId}/outfits/{outfitId}	GET, PUT, DELETE
/users/{userId}/social/posts	GET, POST
/users/{userId}/social/posts/{postId}	GET, PUT, DELETE
/marketplace/items	GET, POST
/marketplace/items/{itemId}	GET, PUT, DELETE
/users/{userId}/social/friends	GET, POST
/users/{userId}/social/friends/{friendId}	DELETE
/users/{userId}/preferences	GET, PUT
/users/{userId}/sizeProfile	GET, PUT
/users/{userId}/outfits/recommendations	GET
/users/{userId}/clothing/recommendations	GET

# Design Issues

## 1.0 Functional Issues

### 1.1 - Database Issues

The user will store clothes, outfits, social posts, reviews, and more in our database. This complexity will result in our database having many collections, documents, and references. This complexity, especially as our codebase is developed, might become confusing and unnavigable. Our group plans on creating a well documented model, which will allow us to avoid common pitfalls of having such a complex database.

Having multiple versions of the database will add further complexity. As we update the database to have more features compatible with our app, prior versions of the database might be incompatible with the current version. Our group plans on solving this issue by deleting all data every time we update the database. This is possible because we will have no active users until the app is completely developed.

We also have many parties accessing the database, adding further complexity. Many users will be accessing and editing the database from their iOS devices. We must be sure that users don't access or edit information they shouldn't be editing, not only for security and privacy reasons, but because it might confuddle and mix up the information. We can regulate the database parts accessed by users by restricting the documents available to the user from Firebase FireCloud.

### 1.2 - Authentication Issues

- a. How are we going to standardize authentication for users?
  - SMS-based authentication - have users make an account using a unique phone number
  - Email-based authentication - have users make an account using a unique email.
  - OTP - have users log in using a time-limited one-time password.
  - Hardware tokens - use the deviceID for the verification.
  - Biometric login - Face ID/Touch ID

We decided to handle authentication of users by integrating several of these methods. All users will be required to login using their gmail or Apple ID, as these two seem to be the most popular and convenient among applications. After they make an account, they are given the option of a biometric login in order to simplify their onboarding process. Finally, a hardware token does need to be stored to enable notifications, as the APNS

servers that receive a notification payload will only know what device to send it to with the given hardware ID.

- b. How are we going to handle error handling in case the onboarding process is interrupted?
  - Re-prompt the user to complete the entire onboarding process from scratch.
  - Only have the user complete parts of the onboarding that they missed in order to complete their account.
  - Allow the user to continue using the app, and as they access parts of the application that need complete information, we will prompt the user to give that information.
  - Allow users to edit their information in the account settings page

For context, the issue being discussed has to do with the onboarding process specifically. In case a user's device shuts down or they delete the app mid-onboarding, we need to find a way to get the necessary data to complete the user document to then push into Firebase. The best way to standardize this is to check if a user document is incomplete, and if it is, we will re-prompt the user to complete the entire onboarding process from scratch again. This makes things simple and standardizes the process to avoid any extreme edge cases where a user's information is incomplete and could inconvenience their use of the app.

### 1.3 - Intersystem Communication Issues

- a. How are we going to handle communication from the front-end to the back-end and vice versa?
  - Directly code in http requests in the front-end whenever necessary and parse the back-end response into a valid format.
  - Implement a layer between the front-end and back-end that acts a translator between the two platforms.
  - Translate the information directly from the back-end and then send that packet to the front-end.

Since our front-end and back-end are split, ensuring proper communication between the two is paramount to having a functional app. We decided to implement a network layer in the front-end that will receive the json packets from the back-end and properly translate them into the format that the front-end needs. The front-end will then call that method from the network layer.

## 2.0 Non functional issues

### 2.1 Performance

One of the main goals of our project is to have a smooth and easy to use app. The performance of the iOS app will be a huge part of making this a reality. By performance, we refer to a few different issues:

1. Authentication speed
2. Loading screens
3. UI Smoothness
4. Low latency network

SwiftUI is very powerful in creating a smooth UI experience, and handling authentication speeds and Low latency networks won't be too large of an issue as Firebase is scalable. Ensuring that we don't overuse navigation links, and create unnecessary pages, the performance of the app should be stable.

### 2.2 Client Security

How will we guarantee client security?

- Prompt the user for permission when accessing camera, deviceID, and other sensitive information.
- Rather than storing these, only access them whenever the user needs them by sending an OTP for verification

We decided to store the deviceID and enabled permissions to reduce repeats of access permissions and make the experience as smooth as possible. If the user consents to notifications and camera permissions and any other information, we will assume that that will be maintained throughout their use for the app.

### 2.3 Server/Firebase Security

How will we ensure Firebase is secure?

- Add an appcheck verification to every request payload to ensure the request is from our app.
- Incorporate JWT to encode our data requests to the backend.

We decided to go with AppCheck. AppCheck is a simple but effective way of keeping our backend secure by adding a temporarily generated token to our request payload that is unique and identifiable by the backend. This lets the server know that the request is indeed coming from our app and not from third parties.

## 2.4 Scalability and Maintainability

How will we make sure our app is scalable for future updates and traffic?

- Implement modular architecture onto our front-end.
- Use caching to store certain information on the user's disk to ensure smooth access.
- Use Load balancing by sending requests to less busy servers.
- Implement Stateless architecture

We decided to use a modular approach to our front-end while also adding caching to ensure smooth loading of slightly larger data objects like images. Modular approaches will ensure that the updates we give to the app will be minimal in terms of rewrites. Stateless architecture is unfortunately very difficult as SwiftUI bases its screen updates entirely on the state of the screen, so many of our variables will be constantly updating and refreshing the screen as a result.