RMIT UNIVERSITY

COSC2659 – iOS Development
Assignment 3

Lecturer
Tom Huynh

Sanghwa Jung – s3768999
Jaeheon Jeong – s3821004
Khanh Nguyen – s3927238
Hoang Duc Anh – s3847506
Hanjun Lee – s3732878

25th Sep 2023

School of Science, Engineering, and Technology

Table of Contents

1. Introduction

In the contemporary landscape, dating apps have emerged as a ubiquitous and highly sought-after means of forging connections among young people, with platforms like Tinder, Badoo, and Bumble celebrated for their ability to transcend geographical boundaries and match individuals based on their romantic preferences. In this fast-paced digital age, these apps serve as a lifeline for a spectrum of individuals, whether experiencing prolonged singledom, grappling with communication challenges, or naturally gravitating toward introversion. The demands of contemporary workplaces, marked by relentless schedules and competing priorities, further compound the challenge of finding a compatible partner. Dating apps, like beacons of hope, provide a straightforward and accessible solution for many navigating the intricacies of modern romance.

[Figure 1] The logo of finder

Finder, a dating application inspired by the renowned platform Tinder, seeks to redefine the modern dating experience. With its origins rooted in the success of Tinder, Finder aspires to offer a unique approach to connecting people, fostering meaningful relationships, and facilitating romantic encounters. The name "Finder" pays homage to its inspiration while embodying its core mission of helping individuals discover and connect with potential dating partners.

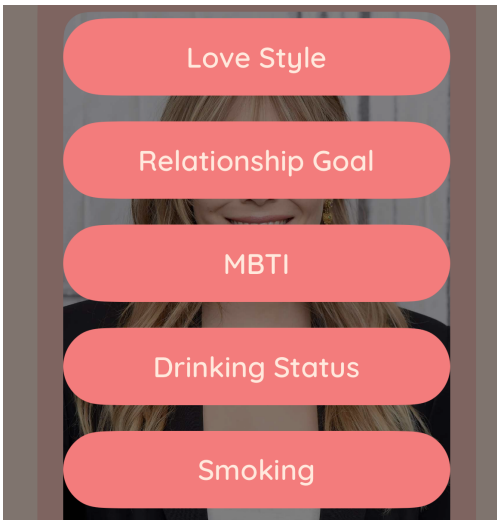
[Figure 2] The application icon

2. Project Description

The Finder application places a strong emphasis on simplicity, effective filtering, and precise matching. It's designed with user-friendliness in mind and ensures a seamless experience, starting with a readily available signup function. This feature is crucial in fostering a robust and diverse user community. During the signup process, users are prompted to provide their preferences, creating a personalized profile that serves as a foundation for the matching algorithm.

Once users log in, they are greeted by an intuitive interface. The initial view showcases profiles of potential matches from the opposite gender, each accompanied by images. Users can express their interest by liking a profile or indicate disinterest by swiping in the opposite direction. The "info" button offers a deeper dive into a user's details, providing valuable insights.

However, what truly sets Finder apart is its sophisticated filtering capabilities. Positioned prominently above the displayed profiles, users have access to two powerful filters. These filters allow users to fine-tune their search criteria, homing in on individuals who align with their specific preferences. This personalized approach significantly elevates the quality of potential matches, making the matchmaking process more tailored and efficient.



[Figure 3] Filtering options

As an added feature, Finder enables two-way communication through a built-in chat function. This functionality becomes accessible when both users have expressed mutual interest in each other. It provides a platform for meaningful conversations, allowing individuals to explore compatibility and build connections.

Furthermore, Finder understands that self-presentation is crucial in the world of online dating. To this end, users have the option to edit their profiles, enhancing their appeal and increasing their chances of attracting compatible matches. This feature empowers users to put their best foot forward and make a lasting impression.

3. Technical Implementation Details

3.1. Main features and advance Features

a. Main Features

i. User Management

When the user first enters the Finder application, it will show 'WelcomeView'(Appendix[1]) so that the user can log in or sign up to enter the app. For the user who has no account, it can go to 'SignInView' to create an account. There are 3 views for creating the account like the appendix[2][3][4]. Therefore, when the user goes to the last view to register the account, it should store the data that the user has made. It will use 'StateObject' when the view goes next, and '.environmentObject' is used so that it can pass the struct data and receive it. For the registration, the user is mandatory to put the nickname, email, password, gender, and photos (maximum 3 photos). Additionally, it can select the preferences or leave it as a blank.

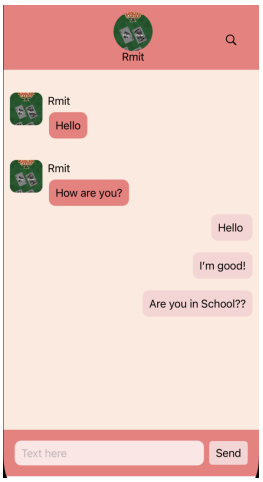
During the registration, it will check the format that the user inputs the right data and email duplication by 'checkAuth' and 'checkEmail' as Appendix[5] shows. The photos are checked by 'isEmpty' so that it cannot move on next when the user does not select the images.

After the user fills out every data, it will upload the data to Firebase, for the email and the password, it uploads to the authentication area so that it can be used when the user login. The data for the user is stored in the Firebase database by a collection called 'users. Each document ID is named a user email. Images cannot upload in the Firebase database, hence, it will upload in storage 'images' file. the name of the image format is 'useremail_index'.

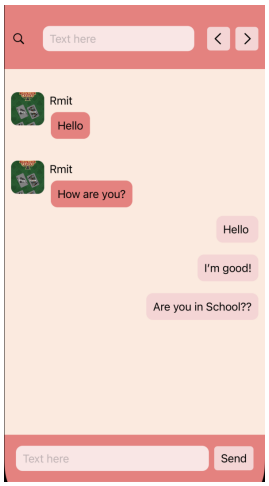
After finishing the registration, it will return to 'WelcomeView'. When the user clicks the log-in button after filling out the email and password, it verifies from the authentication of the firebase that the email and the password are corresponding by the function 'login' from appendix[0].

Log out is occurred when the user click the log out button in the bottom of 'ProfileView'.

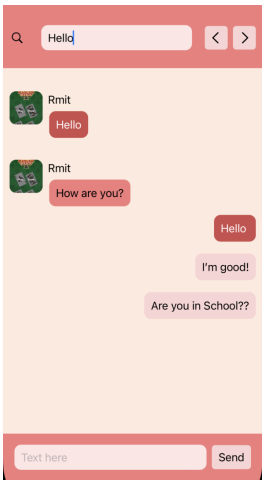
ii. Search



[Figure 4] Searching 1



[Figure 5] Searching 2



[Figure 6] Searching 3

For user convenience, the searching message function has been added to search for previously sent messages to chat rooms. Press the Magnifier button to search for the message content user wants to find in the text field. You can also browse and view all chats, including the searched text, using the arrow buttons on the right.

```
func findMessage(text: String){
    foundTexts = []
    currentFoundText = 0
    currentTextIndex = 0
    if text != ""{
        for index in 0..<(chatRoom.messageList ?? [[:]]).count {
            let message = Array((chatRoom.messageList ?? [[:]])[index].values)
            if let value = message.first{
                if value.lowercased().contains(text.lowercased()){
                    foundTexts.append(index)
                }
            }
        }
    }
    if foundTexts.count != 0 {
        currentFoundText = foundTexts[0]
        scrollPosition = currentFoundText
    }
}
```

[Figure 7] findMessage() Logic

When the user enters a string in the text field and presses the magnifier button, findMessage() is activated. However, in order to avoid unnecessary work, if there is no value in the input field, do not do the following. When the user's input value is not "", for loop traverses the chatRoom.messageList, which stores all chats in the chat room in the form of a key:value array, and stores index values in foundTexts containing the user-entered string in the form of an integer array.

```
ForEach(messageList.indices, id: \.self){ index in
    let message = messageList[index]
    let key = Array(message.keys)
    let value = Array(message.values)
    if key.first == myName {
        if foundTexts.contains(index){
            SpeechBubbleMy(name: myName, text: (value.first ?? ""), isSelected: true)
                .id(index)
        }
        else{
            SpeechBubbleMy(name: myName, text: (value.first ?? ""), isSelected: false)
                .id(index)
        }
    } else {
        if foundTexts.contains(index){
            SpeechBubble(name: partnerName, text: value.first ?? "", partnerImage:
                partnerImage, isSelected: true)
                .id(index)
        }
        else{
            SpeechBubble(name: partnerName, text: value.first ?? "", partnerImage:
                partnerImage, isSelected: false)
                .id(index)
        }

    }
}
```

[Figure 8] ForEach() to print speech bubbels

Since foundTexts is also a @State variable, whenever the above findMessage() is executed and the foundTexts value changes, the chatteringRoom is regenerated. Using this, the user enters a value in the text field, and each time a button is pressed. True or false is assigned to SpeechBubbleMy() and SpeechBubble(), determining the color of the text window.

iii. Filter

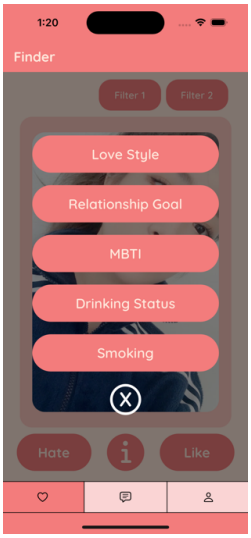


Figure 9: Filter options show when clicking filter button

To prioritize the user's interest, two filters are implemented which are based on the user's preferences. When the user selects 1 of the preferences from the filter option when the screen shows like Figure 9 it will display a random user who has the same preference that the user has. For

example, if the user's love style is 'Thoughtful gesture' and selects the love style filter button, the next random user who has 'Thoughtful gesture' will be shown, and so on.

When the user selects the filter preference, it will execute the 'filterOneUsers' function. It will compare the given text with preferences one to five and if one of the preferences is matched with the text, it will add to the 'filteredOneUsers' array and replace to the 'users' array after finishing the filtering as the appendix[7] shows. The 'filterTwoUsers' will have the same logic.

iv. CRUD operations

1. Create

The code for creating an operation is implemented in the registration part. When the user finishes the signup, it uses 'addUser' which is in the 'userViewModel' class to create a new user struct in the Firebase database. From Appendix[8], it will convert the User object to a dictionary and set the data to the document ID which is called 'userEmail'.

2. Read

Read operations are made for getting random users, information for profiles, and images. Each is implemented as 'getAllUsers', 'getUserByEmail", and 'getImagesByEmail'.

The 'getAllUsers'(appendix 9) and 'getUserByEmail'(appendix 10) will retrieve user data from Firestore and provide it to the caller through the completion closure. The 'getImagesByEmail'(appendix 11) is to fetch multiple images associated with a user's email and provide them to the caller in an array of UIImage objects through the completion closure.

```
userService.getAllUsers { users in
    if let users = users {
        // Handle the retrieved user data
        self.users = users
        filterDifferentGender(gender: fetchedUser.gender ?? "")
        getRandomUser()

        let randomEmail = randomUser?.email ?? "DefaultEmail"
        print(randomEmail)


        userService.getImagesByEmail(email: randomEmail) {
            fetchedImages in
            if let fetchedImages = fetchedImages {
                // Populate the images array with fetchedImages
                randomUserImages = fetchedImages
                print("success")
            } else {
                print("No images found for user: \(email)")
            }
        }
```

Figure 10: Process of reading data and displaying into UI

Read operations are used inside of the '.onAppear' part and update to the retrieved data. Multiple functions can be used in the one .onAppear'.

3. Update

To customize the user's own profile, an update operation is required. In the 'ProfileView' and ProfileEditView', the user can change the user info, preferences, and images.

```
func updateUser(email: String, newData: [String: Any], completion: @escaping (Error?) -> Void) {
    // Reference to the "users" collection
    let usersCollection = db.collection("users")

    // Reference to the document with the provided email
    let userDocument = usersCollection.document(email)

    // Update the document with the new data
    userDocument.updateData(newData) { error in
        if let error = error {
            print("Error updating user data: \(error.localizedDescription)")
            completion(error)
        } else {
            print("User data updated successfully!")
            completion(nil)
        }
    }
}
```

Figure 11: 'updateUser' for customize the database

The preference, age, and phone number can be modified by the 'updateUser'. For the preference, it will directly update the database when the user changes the preference (appendix 12). For the other data except email and password, it will update the data by clicking the 'done' button (appendix 13). 'updateUser' provides a convenient way to update user data in Firestore by specifying the email of the user and the new data to be applied. 'updateAuthentication' will follow the same logic.

```
for (index, selectedImage) in selectedImages.enumerated() {
    // Convert the UIImage to Data
    guard let imageData = selectedImage.jpegData(compressionQuality: 0.5) else {
        let error = NSError(domain: "", code: 0, userInfo: [NSLocalizedDescriptionKey: "Failed to convert
            image to data"])
        completion(error)
        return
    }

    // Create a unique filename for each image based on user's email and index
    let fileName = "\(userEmail)_\(index).jpg"

    // Create a reference to Firebase Storage where you want to store the image
    let imageReference = storageReference.child("images").child(fileName)

    // Upload the image data to Firebase Storage
    let metadata = StorageMetadata()
    metadata.contentType = "image/jpeg"

    imageReference.putData(imageData, metadata: metadata) { metadata, error in
        if let error = error {
            print("Error updating image \(index): \(error.localizedDescription)")
            completion(error)
        } else {
            print("Image \(index) updated successfully.")
            // You can perform additional actions here, such as saving the download URL or updating your UI.
        }
    }
}
```

Figure 12: Process of updating the images in 'updateImages'

When the user clicks the image in the ‘ProfileEditView’, it will show the image folder. after selecting the images (maximum 3 images) and pressing the ‘done’ button, it will update a user's images in Firebase Storage like the Figure 12 shows. It will delete previous images and replace them to the selected images.

4. Delete

```
func removeChatMessage(documentID: String) {
    let db = Firestore.firestore()
    let chatRef = db.collection("chats").document(documentID)

    chatRef.delete { error in
        if let error = error {
            print("Error removing chat message: \(error)")
        } else {
            print("Chat message successfully removed!")
        }
    }
}
```

Figure 13: Function for deleting the chat room

In ‘ChattingLIst’, each of the listed chat rooms has the delete function that can directly delete the chat room in the screen and the database when the user swipes left of the chat room (appendix 00). ‘.onDelete’ which is applied to ‘List’ is used for handling the row deletion. To remove the chat room, it should find the document ID first which is based on the two user’s emails from the following chat. It uses ‘removeChatMessage’ so that it coordinates Firestore deletions and local data removal at once when a user deletes a partner's chat room.

b. Advance features

i. **Biometric Authentication**

In the ‘WelcomeView’ (Appendix [1]), it has the face ID icon in the middle of the login button and the sign-up button. When the user finishes the sign-up, it will update the email and the password to the ‘@AppStorage’ wrapper according to ‘bioEmail’ and ‘bioPassword’.

By using ‘authenticate’ when the face ID icon is clicked, it will authenticate(Appendix [15]) a user with biometric data. If the authentication is successful, it will replace the email and the password with ‘bioEmail’ and ‘bioPassword’ and do the process of the ‘login’ function so that it will be directly logged in.

c. Extra Views and Features.

i. Show Random user

When the user successfully logs in, it will move to ‘MainView’ which shows random users with a different gender so that the user can choose ‘like’ or ‘hate’. By clicking the like/hate button, it will directly update the like/hate array of the user database and show the other random user. Users can see the random user info by clicking the info icon. It can see the images by swiping left and right, and see other information such as preferences, age, and phone number as appendix[16] shows. On the bottom, it has a tab bar to move another view.

ii. Remain log in

```
@AppStorage("userEmail") private var userEmail: String?

init () {
    FirebaseApp.configure()
}

var body: some Scene {
    WindowGroup {

        if let userEmail = userEmail, !userEmail.isEmpty {
            MainView(email: userEmail)
        } else {
            WelcomeView()
        }
    }
}
```

Figure 14: check the ‘userEmail’ is not empty and move the ‘MainView’

When the user first logs in, it will save data in the AppStorage called ‘userEmail’. It will keep getting the data until the user clicks the logout button. Therefore, if the user shuts down the app and restarts it and ‘userEmail’ has data, it will go to ‘MainView’directly by passing the ‘userEmail’ data.

iii. Firebase and UserDefaults

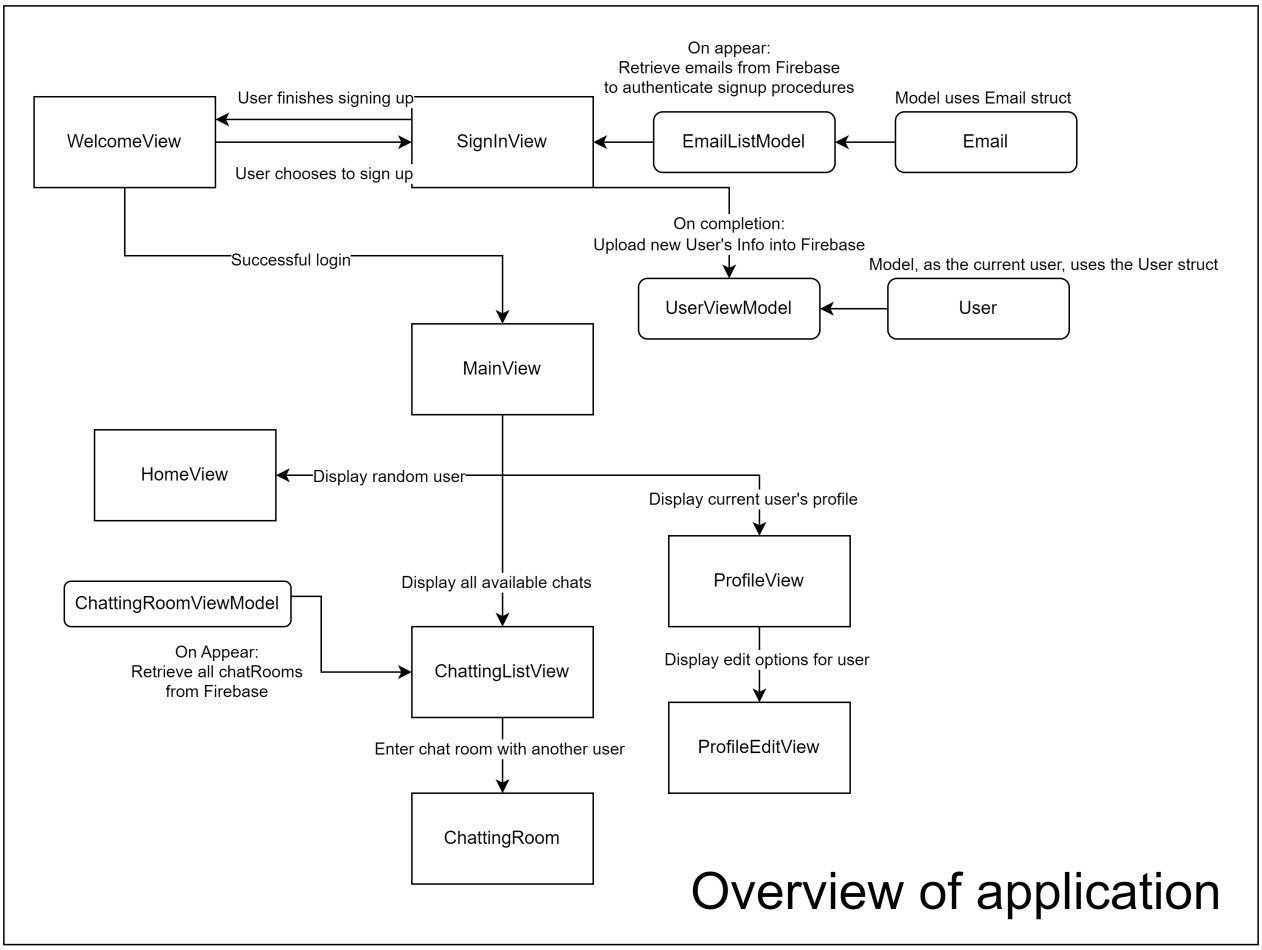
```
@AppStorage("userEmail") private var userEmail: String?
@AppStorage("bioEmail") private var bioEmail: String?
@AppStorage("bioPassword") private var bioPassword: String?
```

Figure 15: Store and get data by a specified key

To store a small amount of data, it is better to store the data in the device not upload it to the database. In this project, the dark mode, login remaining, and face ID can be considered as small data and

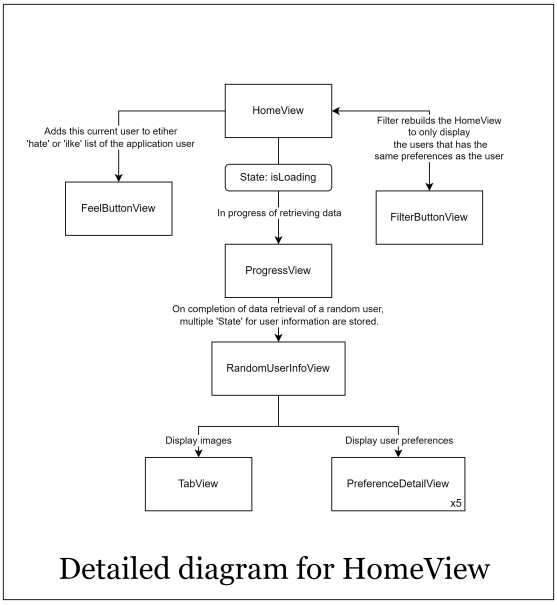
each device will have different data. Therefore, 'AppStorage' is used to store the data and it can easily get the data by a specified key as Figure 00 shows.

3.2. Application flow



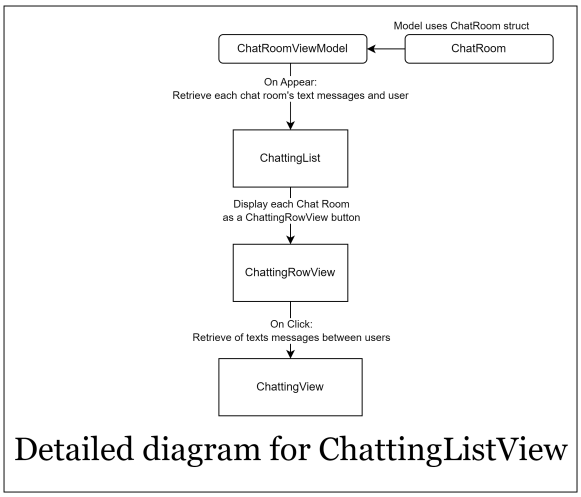
[Figure 16] Application Flow

In figure __, the diagram displays multiple aspects of the application's architecture. The diagram starts with WelcomeView as landing view of the application, it leads to other views with different purposes as indicated by each view's name. If the user were to choose to sign up to use the application, the system would retrieve a list of email to check for email duplications, and upon the completion of the sign up, upload a UserViewModel to Firebase, which contains the current user's information. The hierarchy is designed such that the two most front views are parallel to each other, meaning that users can either login or signup at their first interaction with the application. After successfully logging in, the user is greeted with the MainView that hosts three other views using a tab bar. Here the user can traverse between three views without any order. MainView's purpose is to support the ease of accessibility for users without having to navigate through NavigationView or NavigationStack, users can access all important features of the application at all times.



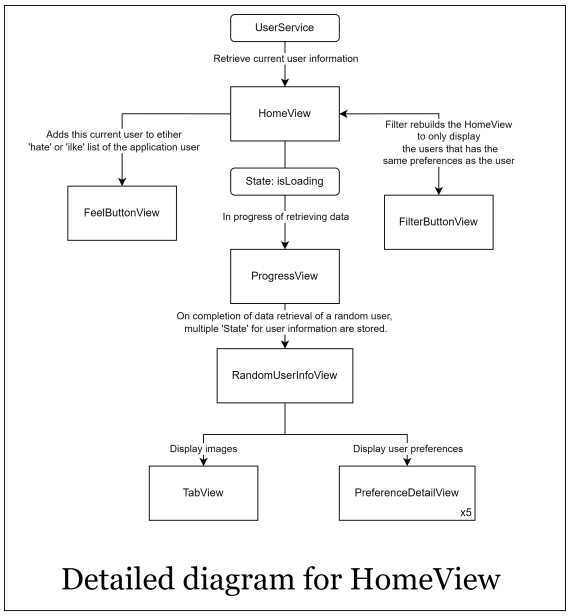
[Figure 17] Application flow for HomeView

In HomeView's display, the view takes in the current user email to retrieve all its necessary information using UserService. During this retrieval process, a loading screen is programmed to indicate the retrieval process. Once retrieval is done, HomeView will display random users of the opposite gender, which the user can make the display filter out those displayed users by matching it with their preferences. In this HomeView, the user can choose to 'like' or 'hate' this person, which would add the displayed user to the current user's list of 'liked' and 'hated' array. Moreover, the current user can view more details upon the displayed user by tapping on the information icon; which displays the image, and all basic information of that user.



[Figure 18] Application flow for ChattingListView

In the ChattingListView, it shows all the user's current chat rooms by retrieving data from the ChatRoomViewModel, which retrieves its information from Firebase. The flow of this function is linear, where the user can click on the chat room with another user, which they can engage in a conversation.



[Figure 19] Application flow for HomeView

In HomeView, information of the current user is retrieved from the UserService in combination with AppStorage. The information is then displayed to the current user in the same manner it would display to other users. However, the users are able to change the preferences within the ProfileView as it supports concurrent change of human nature; whereas to change more sophisticated information, users will have to tap on "Edit Profile" which leads them to the ProfileEditview using the State editProfile, to change the ProfileView into ProfileEditView. The information is once again retrieved from ProfileView, as ProfileEditView is its children, then is displayed as a TextField to allow edits of that information. Finalizing the changes, the user can choose to tap "Done" which uploads all the information given into Firebase using UserViewModel.

3.3. Technologies & Known Bugs

3.3.1. Technologies:

- Swift Components:
  - UIKit: UIKit is leveraged for some of the foundational elements of the application, such as view controllers and navigation.
- SwiftUI Components:
  - View: SwiftUI's core building block for creating the user interface.

- Text: Used to display text and labels in various parts of the app.
- Image: For presenting images, including user profile pictures.
- Button: Utilized for interactive elements like liking or disliking profiles.
- NavigationLink: Enables seamless navigation between different parts of the app, such as profile details and chat.
- List: Used for displaying lists of profiles and matches.
- TextField: Allows users to input and edit information in their profiles.

- External Libraries:
  - Firebase: We've integrated Firebase for user authentication, real-time chat functionality, and cloud storage for user data.

3.3.2. Known Bugs/Problems:

The bug we've encountered but haven't resolved yet is that when a user logs in, for example, if the ID is "Mina12@gmail.com" with a password "Mina12," and the user attempts to log in with a different capitalization of the ID, such as "mina12@gmail.com" with a password "Mina12", they end up logging in with an entirely new ID, even if that ID doesn't exist in the database. One interesting point is that this behavior doesn't apply to passwords. For instance, if a user's ID is "Mina12@gmail.com," changing only the capitalization of the password, from uppercase to lowercase or vice versa, would result in a failed login.

4. Conclusion

4.1. Conclusion

Finder is the approach that this team has undertaken to approach the topic of love discovery. This application allows user to find others who are interested in finding partners with similar interests, relationship goals and standards (smoking, drinking, etc…).

In detail, Finder encompasses a login user system which allows each user to alter their profile, which is displayed to other users. The main feature with regards to love discovery is the random discovery of other users, which can be filtered with two categories per user's choice of preference. Intricately, upon being in each other's 'like' list, a chat is created to encourage communication and progress on their relationship. Moreover, as people change, the application supports the change of users' profile to update their current preferences and interests.

Through this application's development, challenges were identified and procedurally completed. The greatest challenge of this project is the integration of the backend and frontend. There were

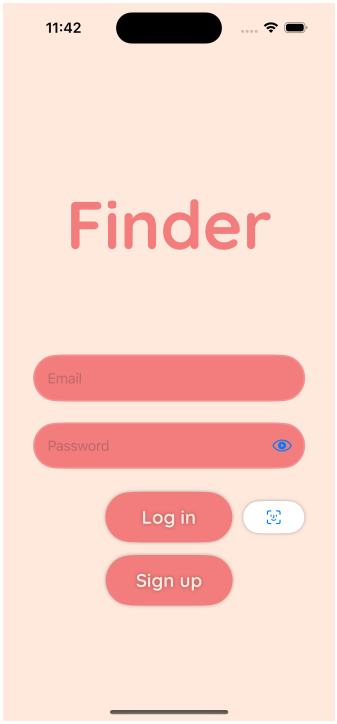
difficulties in debugging on errors that caused information to not display properly. In addition, the application's architecture complete conformation with the Model View View-Model architecture proved to be difficult. Engaging with these challenges, the team collaborated with regular offline meetings to discuss and inquire about the application architecture to establish a firm connection between front and backend.

Ultimately, Finder as an assignment reveals the environment of a short-term project which is an invaluable experience for everyone to learn before entering any workforce. The mock hackathon demonstrates how any simple application, like Tinder, Bumble, at face value must require strict architectural design to enhance the user experience.

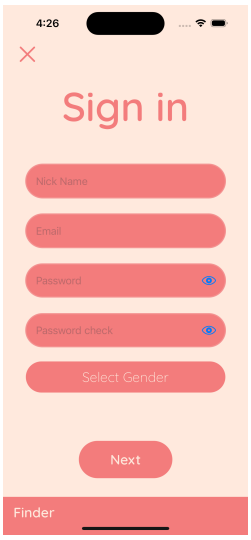
4.2. Project Responsibilities

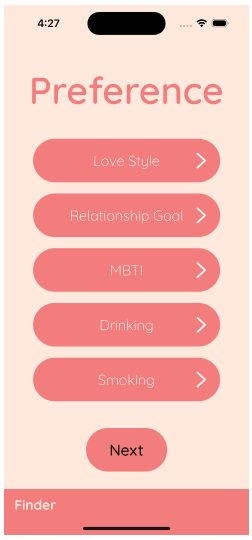
Name	Sanghwa Jung	Jaeheon Jeong	Khanh Nguyen	Hoang Duc Anh	Hanjun Lee
Student Id	s3768999	s3821004	s3927238	s3847506	s3732878
Responsibility	Project Manager	Technical Lead	Team member	Team member	Team member
Contribution	21%	21%	21%	16%	21%

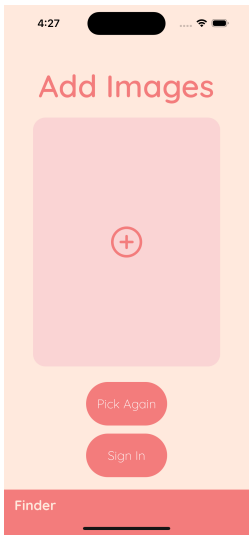
5. Appendix



Appendix [1]: 'WelcomeView' shows when the user first enters the Finder







Appendix [2][3][4]: Process for registration

```
//MARK: -- Validation check
func checkAuth() {

    if nickname.isEmpty {// empty check
        warning = "plz enter nick name"
    } else if !email.contains("@") {// email format
        warning = "wrong format for email"
    } else if password.count < 6 {// password count more than 6
        warning = "pw should longer than 5"
    } else if password != passwordCheck{// password match
        warning = "password is not matched"
    } else if gender.isEmpty {
        warning = "plz check the gender"
    } else {
        isNext.toggle()
    }

}

// Function to check the email duplication
func checkEmail(){
    print("work")
    for element in userEmailList.emails {
        if (element.email ?? "") == email{
            warning = "This is a duplicated email!"
            break
        }
        else{
            warning = "       "
        }
    }
}
```

Appendix [5]: Process for checking the sign in format and duplication of the email

```
func login() {
    Auth.auth().signIn(withEmail: email, password: password) {
        (result, error) in
        if error != nil {
            print(error?.localizedDescription ?? "")
            isSigned = false
        } else {
            print("success")
            isSigned = true
        }
    }
}
```

Appendix [6]: Verifying the user email and password is correct from Firebase

```
func filterOneUsers(text: String) {
    // Assuming you have an array of users
    let allUsers = users

    filteredOneUsers = []

    // Iterate through all users
    for user in allUsers {
        // Check if the specified text exists in any of the preferences
        if user.preferenceOne?.lowercased() == text.lowercased()
            || user.preferenceTwo?.lowercased() == text.lowercased()
            || user.preferenceThree?.lowercased() == text.lowercased()
            || user.preferenceFour?.lowercased() == text.lowercased()
            || user.preferenceFive?.lowercased() == text.lowercased() {
            // If a match is found, add the user to the filtered list
            filteredOneUsers.append(user)
        }
    }

    users = filteredOneUsers
}
```

Appendix [7]: filtering the same preference with the text

```
func addUser() {
    // Reference to Firestore
    let db = Firestore.firestore()

    // Get the user's email
    guard let userEmail = user.email else {
        return
    }

    // Reference to the "users" collection
    let usersCollection = db.collection("users")

    // Create a document with the user's email as the document ID
    let userDocument = usersCollection.document(userEmail)

    // Convert User object to a dictionary
    let userDictionary: [String: Any] = [
        "id": user.id,
        "nickname": user.nickname ?? "",
        "email": user.email ?? "",
        "password": user.password ?? "",
        "gender": user.gender ?? "",
        "LoveStyle": user.preferenceOne ?? "",
        "RelationshipGoal": user.preferenceTwo ?? "",
        "MBTI": user.preferenceThree ?? "",
        "DrinkingStatus": user.preferenceFour ?? "",
        "Smoking": user.preferenceFive ?? "",
        "phoneNumber": user.phoneNumber ?? "",
        "age": user.age ?? "",
        "like": user.likeArray, // Update "like" array with user.like
        "hate": user.hateArray  // Update "hate" array with user.hate
    ]

    // Set the data for the user's document
    userDocument.setData(userDictionary) { error in
        if let error = error {
            print("Error saving user data: \(error.localizedDescription)")
        } else {
            print("User data saved successfully!")
        }
    }
}
```

Appendix [8]: Create User struct data to Firebase

```swift
func getAllUsers(completion: @escaping ([User]?) -> Void) {
    // Reference to Firestore
    let db = Firestore.firestore()

    // Reference to the "users" collection
    let usersCollection = db.collection("users")

    // Define a closure to handle the query result
    let queryCompletion: (QuerySnapshot?, Error?) -> Void = { (querySnapshot, error) in
        if let error = error {
            print("Error getting all users: \(error.localizedDescription)")
            completion(nil)
            return
        }

        guard let documents = querySnapshot?.documents else {
            print("No documents found")
            completion(nil)
            return
        }

        // Process documents and map to User objects
        let users = self.processUserDocuments(documents)
        completion(users)
    }

    // Fetch documents
    usersCollection.getDocuments(completion: queryCompletion)
}
```

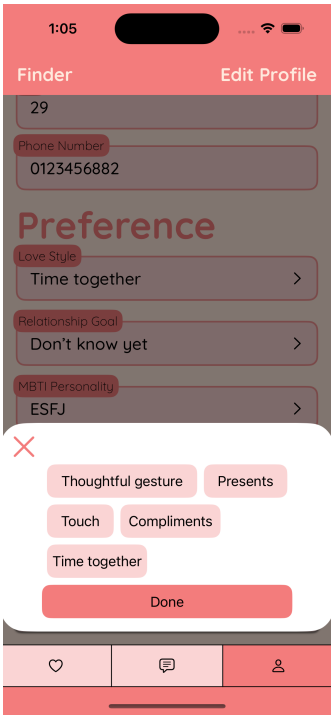
Appendix [9]: Process to get all users from Firebase

```swift
func getUserByEmail(email: String, completion: @escaping (User?) -> Void) {
    db.collection("users")
        .whereField("email", isEqualTo: email)
        .getDocuments { (querySnapshot, error) in
            if let error = error {
                print("Error getting user by email: \(error.localizedDescription)")
                completion(nil)
                return
            }

            guard let documents = querySnapshot?.documents else {
                print("No documents found")
                completion(nil)
                return
            }

            if let userData = documents.first?.data() {
                var user = User()
                user.id = userData["id"] as? String ?? ""
                user.nickname = userData["nickname"] as? String ?? ""
                user.documentID = userData["documentID"] as? String ?? ""
                user.email = userData["email"] as? String ?? ""
                user.password = userData["password"] as? String ?? ""
                user.gender = userData["gender"] as? String ?? ""
                user.preferenceOne = userData["LoveStyle"] as? String ?? ""
                user.preferenceTwo = userData["RelationshipGoal"] as? String ?? ""
                user.preferenceThree = userData["MBTI"] as? String ?? ""
                user.preferenceFour = userData["DrinkingStatus"] as? String ?? ""
                user.preferenceFive = userData["Smoking"] as? String ?? ""
                user.phoneNumber = userData["phoneNumber"] as? String ?? ""
                user.age = userData["age"] as? String ?? ""
                user.likeArray = userData["like"] as? [String] ?? []
                user.hateArray = userData["hate"] as? [String] ?? []

                completion(user)
            } else {
                print("User not found")
                completion(nil)
            }
        }
}
```

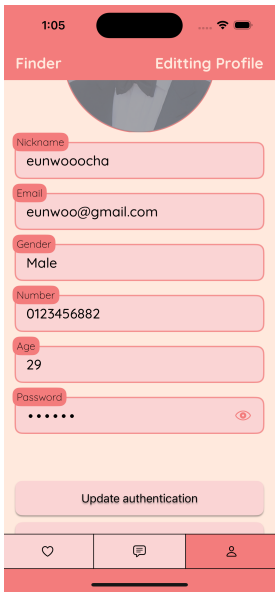
Appendix [10]: Database Schema for User Information

```swift
func getImagesByEmail(email: String, completion: @escaping ([UIImage]?) -> Void) {
    // Fetch user data by email
    getUserByEmail(email: email) { user in
        guard let user = user else {
            print("User not found for email: \(email)")
            completion(nil)
            return
        }
        var images: [UIImage] = []

        // Loop through user's images and download them from Firebase Storage
        if let userEmail = user.email {
            let storageReference = self.storage.reference().child("images") // Use 'self' here
            for index in 0..<5 { // Assuming you have up to 5 images
                let fileName = "\(userEmail)_\(index).jpg"
                let imageReference = storageReference.child(fileName)

                imageReference.getData(maxSize: 5 * 1024 * 1024) { data, error in
                    if let error = error {
                        print("Error downloading image \(index): \(error.localizedDescription)")
                    } else if let data = data, let image = UIImage(data: data) {
                        images.append(image)
                        |
                        // Print the image count for debugging
                        print("Downloaded \(images.count) images")

                        if images.count == 5 { // Check if all images have been fetched
                            completion(images)
                        }
                    } else {
                        print("Failed to convert data to UIImage for image \(index)")
                    }
                    completion(images)
                }
            }
        } else {
            print("User email is nil.")
            completion(nil)
        }
    }
}
```

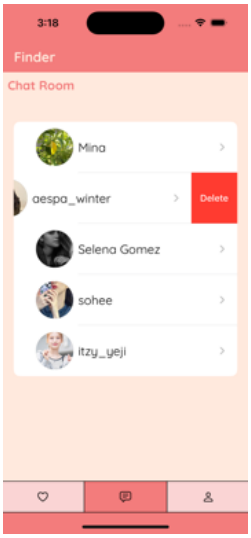
Appendix [11]: retrieve images from the Storage



Appendix [12]: Changing preference in the 'ProfileView'



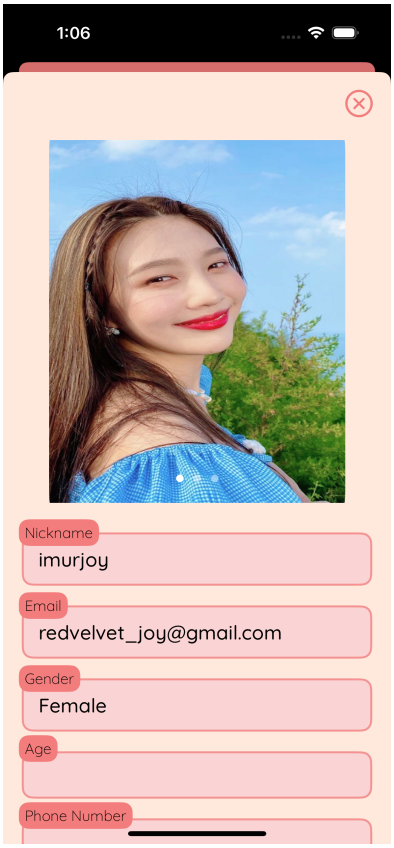
Appendix [13]: Changing user information in ‘ProfileEditView’



Appendix [14]: ‘ChattingList’ shows delete button when swiping left

```
func authenticate(completion: @escaping (Bool, Error?) ->
    Void) {
    let context = LAContext()
    var error: NSError?

    if
        context.canEvaluatePolicy(
        .deviceOwnerAuthenticationWithBiometrics, error:
        &error) {
        context.evaluatePolicy(
            .deviceOwnerAuthenticationWithBiometrics,
            localizedReason: "this is for security reasons")
            { success, authenticationError in
            if success {
                email = bioEmail!
                password = bioPassword!
                login()
                print("Log In with Face ID Successful")
            } else {
                print("Face ID not found. Please try again")
            }
        }
    } else {
        print("Device does not have biometrics !")
    }
}
```

Appendix [15]: processing to verify the face ID



Appendix [16]: Show random user info when clicking info icon