Individual Assignment

Module Code	CT112-3-3-MPMA
Intake Code	APD3F2402IT(MBT)
Lecturer Name	Mr. Amad Arshad
Hand-out Date	12 August 2024
Hand-in Date	1 November 2024

Student Name	Yip Zi Xian (TP059963)

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

Findrobe is a proposed digital solution designed to promote clothing management, community engagement, and sustainable fashion practices among individuals. It is developed with a user-centred approach where it enables individuals to catalogue and organize their personal wardrobe while introducing a community of shared lifestyles, clothing pairing ideas, and sustainable living practices. According to Fernandes (2023), he mentioned that nowsadays with the convenience of online shopping application like Shopee and Lazada, indiviudals tend to purchase more clothes compared to previous era of going to shopping malls. As a result, it can cause individuals to have unplanned shopping and messy wardrobe. Martine (2023) and Miles (2024) said that having wardrobe management applications can allow users to easily find their desired clothes without going through multiple drawers. Furthermore, when users browsing through these applications, it can indirectly reduce users' tendency to shop for more when they have a lot of not-aged clothes recorded.

This documentation provides an insight into the proposed application's features, technical specifications, and implementation demostration. It obliquely align with the United Nations Sustainable Development Goals (SDGs) number 3: Good Health and Well-Being as the proposed application contributes to personal well-being by simplifying daily routines of dressing and shopping habits while reducing decision fatigue and developing individuals' self-confidence and self-discipline in overpurchasing. Secondly, the proposed solution also align itself with SDG number 12: Responsible Consumption and Production where it encourages individuals' to track and record their wardrobe to avoid buying additional clothes which will only be worn a few times and left aside.

2. Application Functionalities

The application requires users to login to their registered account and it will logically identify the role of the account, either client or admin. If the user has not registered to the application, they are required to create a new client account. On the other hand, admin will only have access to admin view when their role of the account is changed manually through Firestore to "admin" status. Both of which will navigate users to client view or admin view. Below are the different functionalities of both client and admin:

2.1. Client

- **Manage posts:** Perform create, read, and delete operation for their own posts.
- **Manage comments:** Perform create, read, and delete operation for their own comments on posts.
- **Manage likes:** Perform create, read, and delete operation for their own likes on posts.
- **Manage clothing:** Perform create, read, update, and delete operation for their own clothing information.
- **Manage profile:** Perform read and update operation for their own profile information.
- **Manage followers:** Perform read and update operation for their followers.

2.2. Admin

- **Moderate posts:** Perform read and delete operation for clients' posts.
- **Moderate comments:** Perform read and delete operation for clients' comments on posts.
- **View data analytics:** Perform read operation via graphs in terms of total number and monthly basis for posts, comments, clients, likes, and clothing recorded in Firestore.

3. System Design

3.1. System Architecture Design

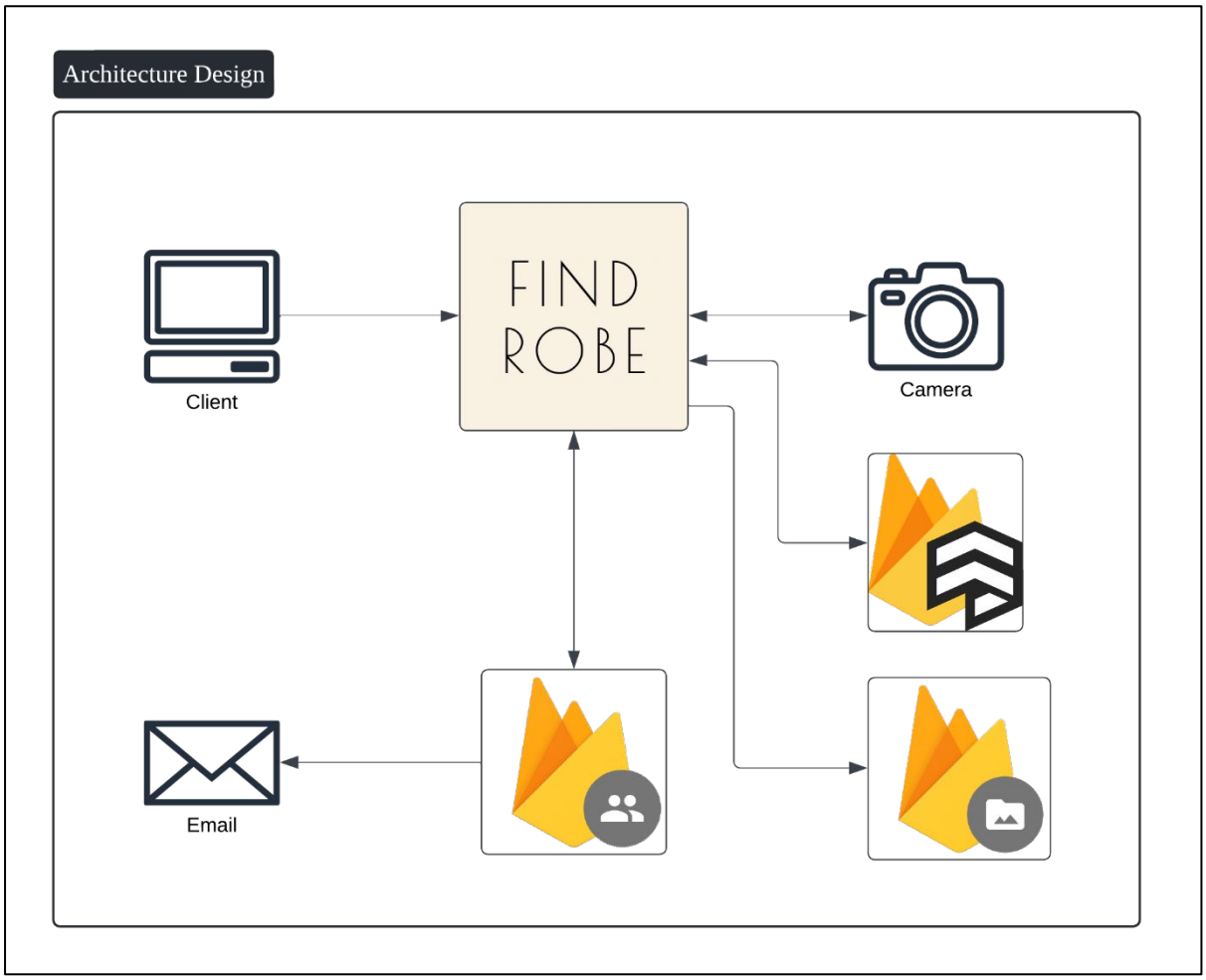


Figure 1: Findrobe Architecture Design

The diagram above illustrated the overall architecture design of Findrobe where it starts with client interaction with the application. The application will authenticate the clients whether they are new users, previously registered, or logged in and currently having a session via Firebase Authentication. After clients logged in, they can manage their posts, comments, likes, followers, adding clothes to different categories via phone camera or gallery. These images will save to Firebase Storage and the URLs are extracted to save into Cloud Firestore along with the clothes details.

3.2. Use Case Diagram

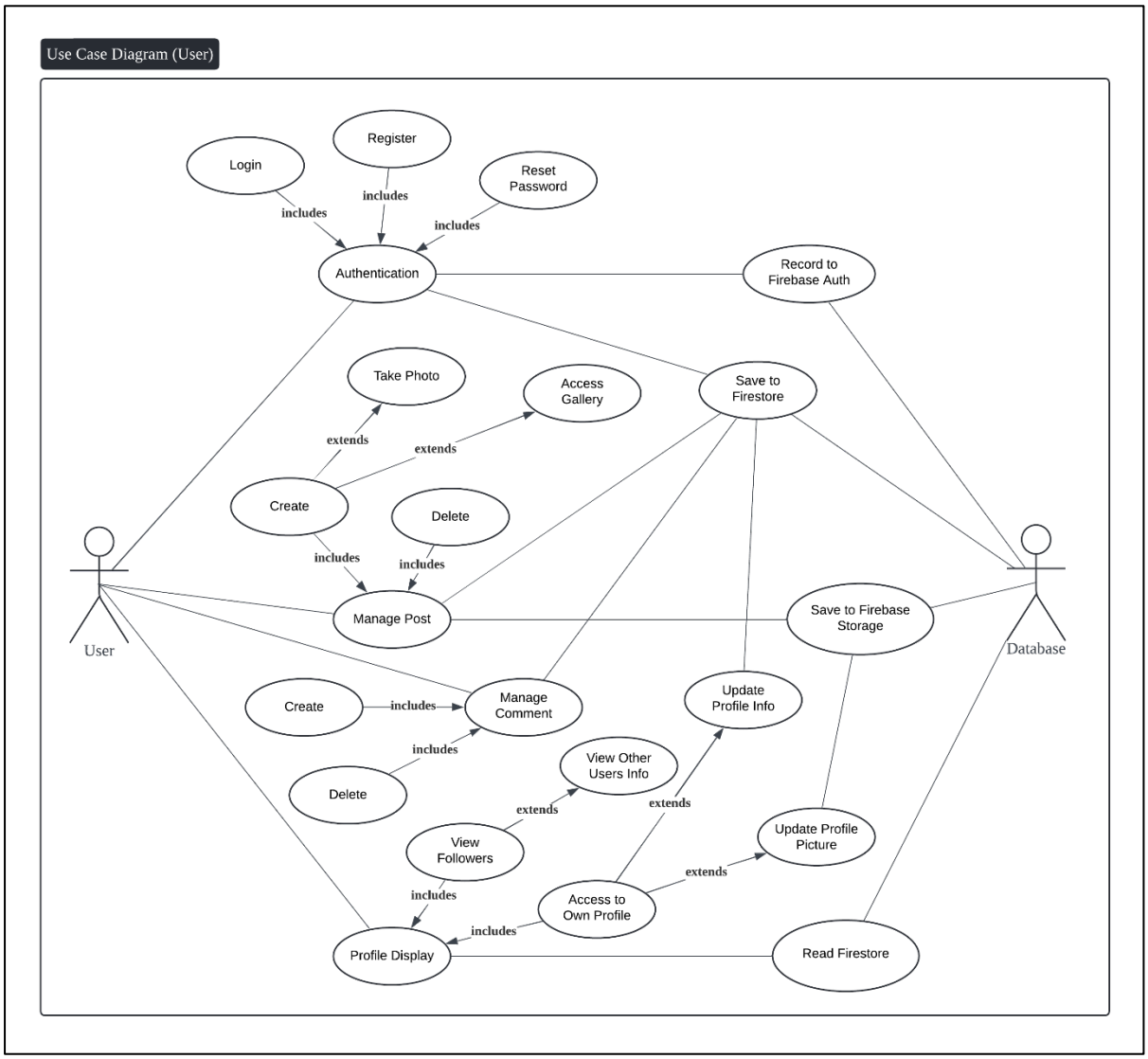


Figure 2: Client Use Case Diagram

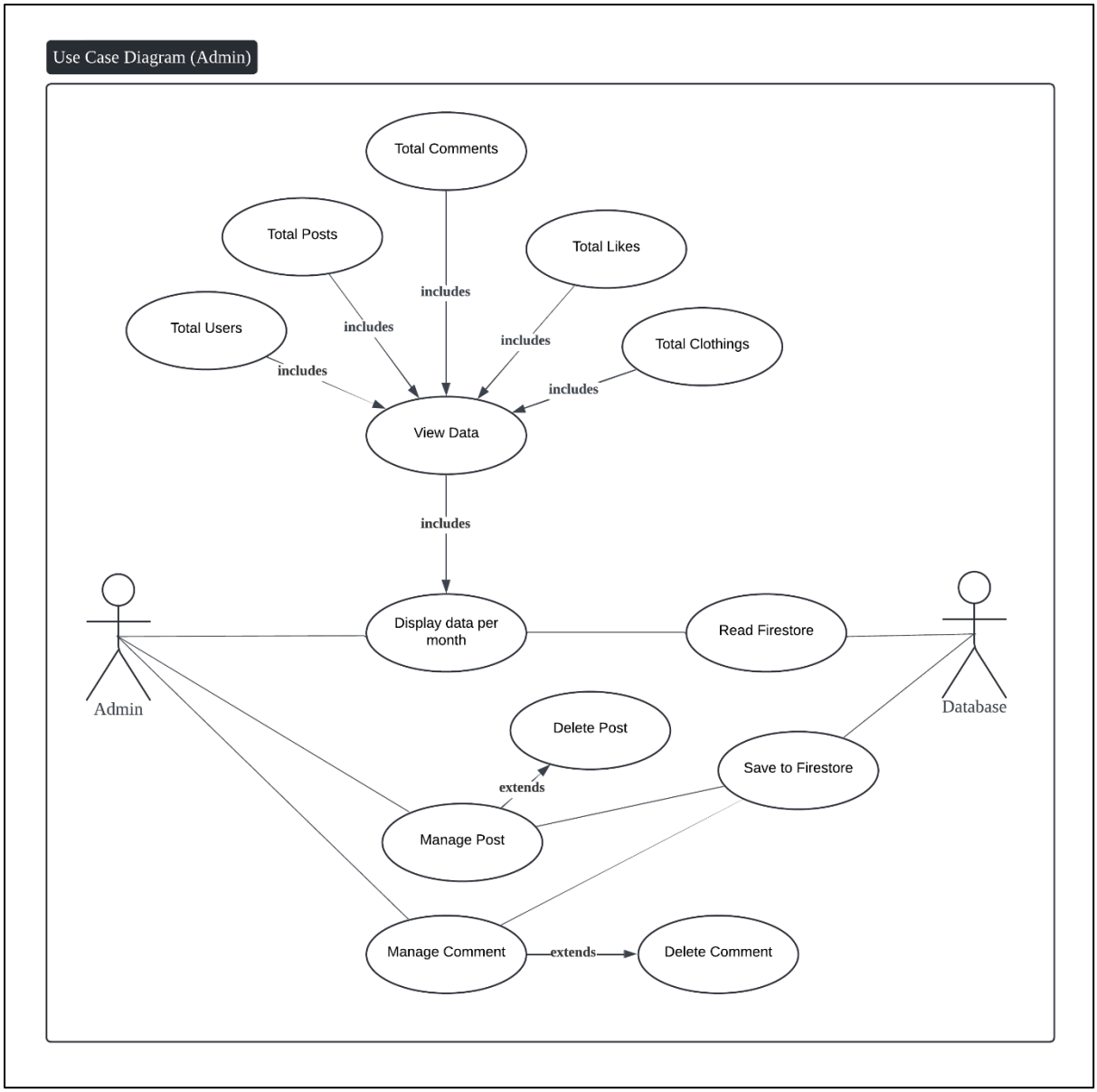


Figure 3: Admin Use Case Diagram

3.3. Entity Relationship Diagram

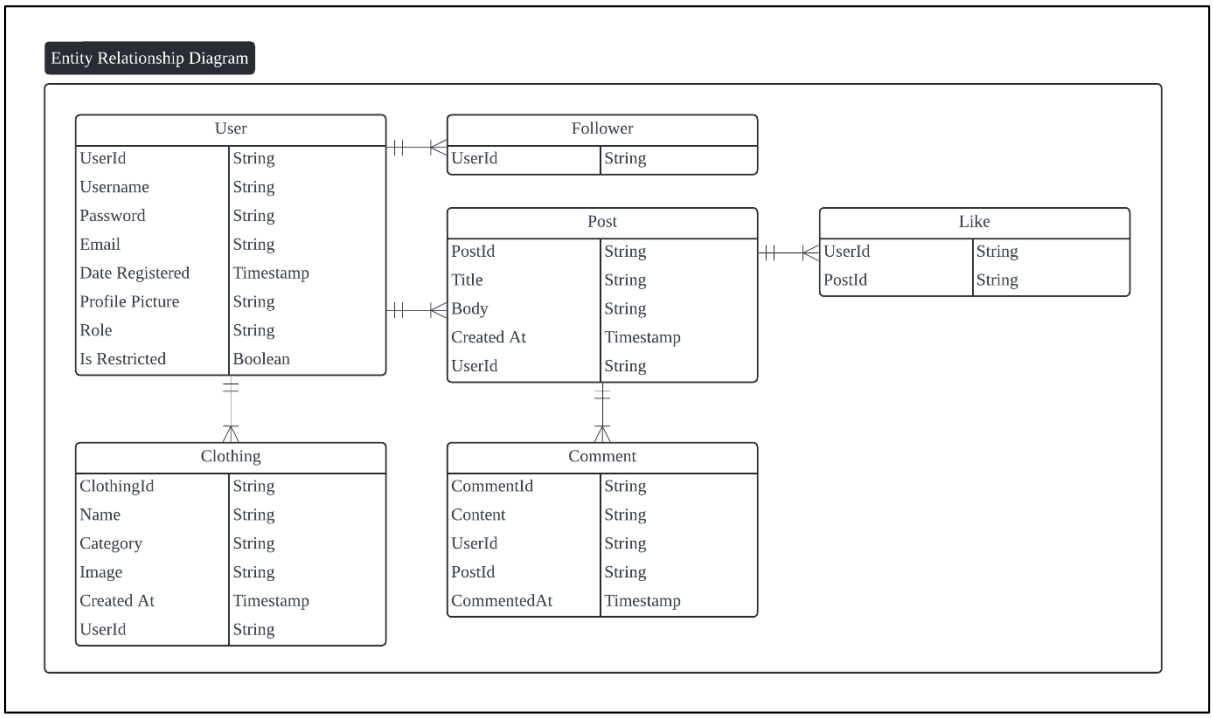


Figure 4: Findrobe Entity Relationship Diagram

Findrobe application consist of six entities, user, clothing, follower, post, comment, and like. Some entities do not have a unique ID as they will be saved as a sub-collection in Cloud Firestore. For users' profile pictures and clothing images, they will be saved in Firebase storage and URLs are extracted to save in Cloud Firestore. User, post, comment, clothing entities contain timestamp for data analytics to track the application usage which can only be viewed by admin.

3.4. User Interface Diagram

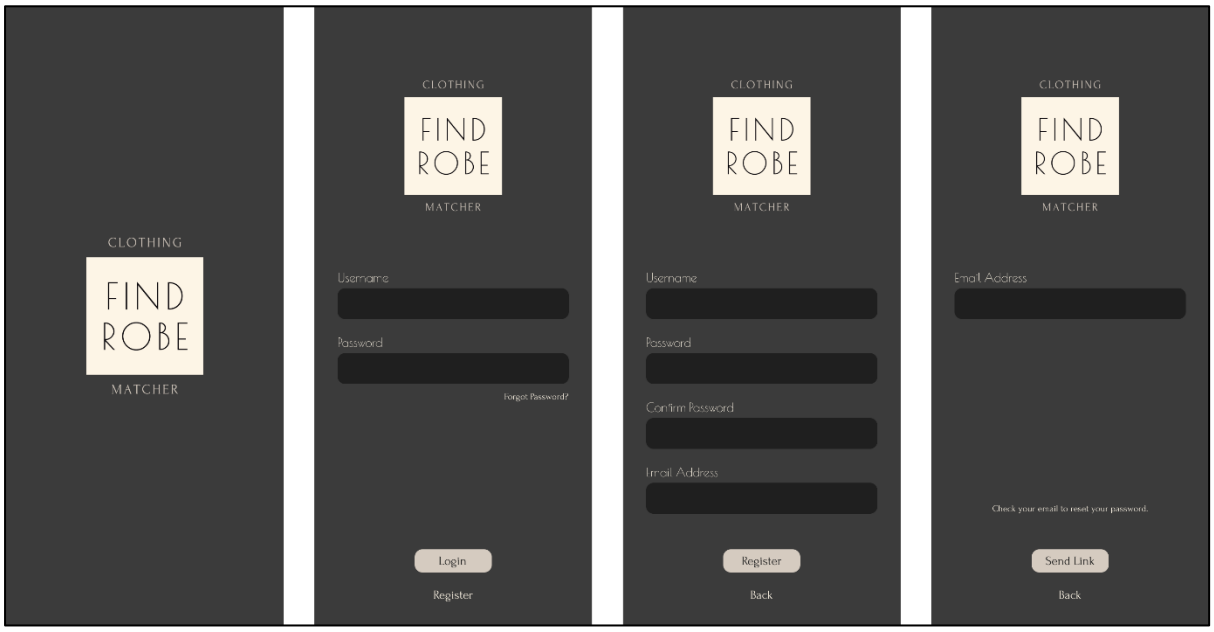


Figure 5: Findrobe Authentication Section

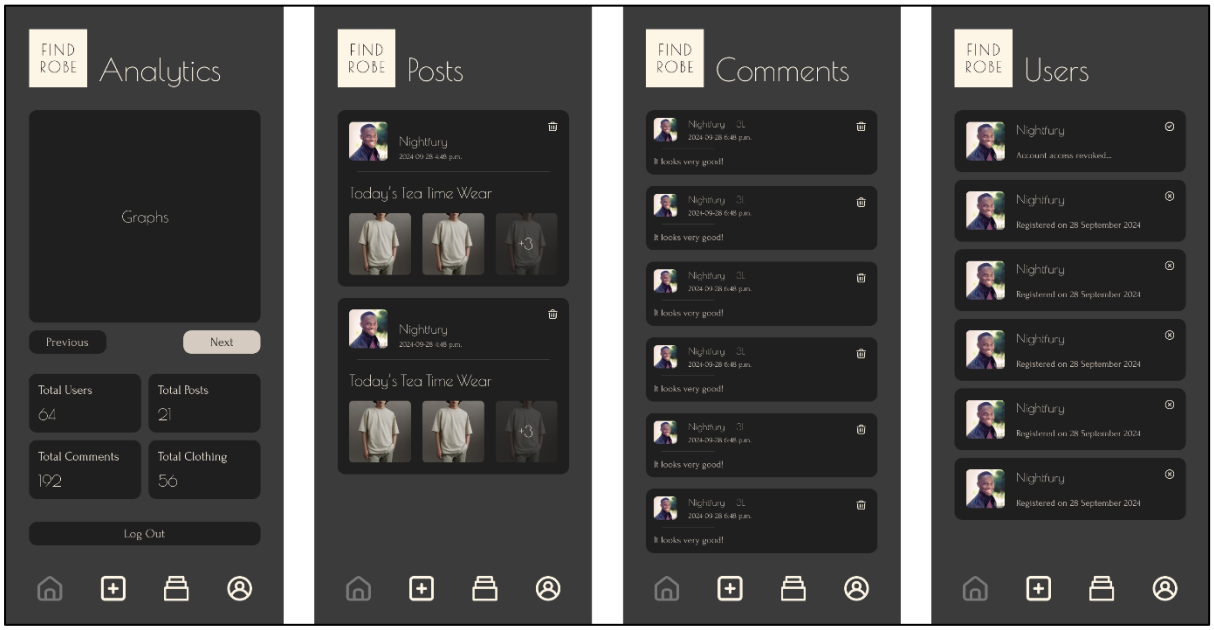


Figure 6: Findrobe Admin View

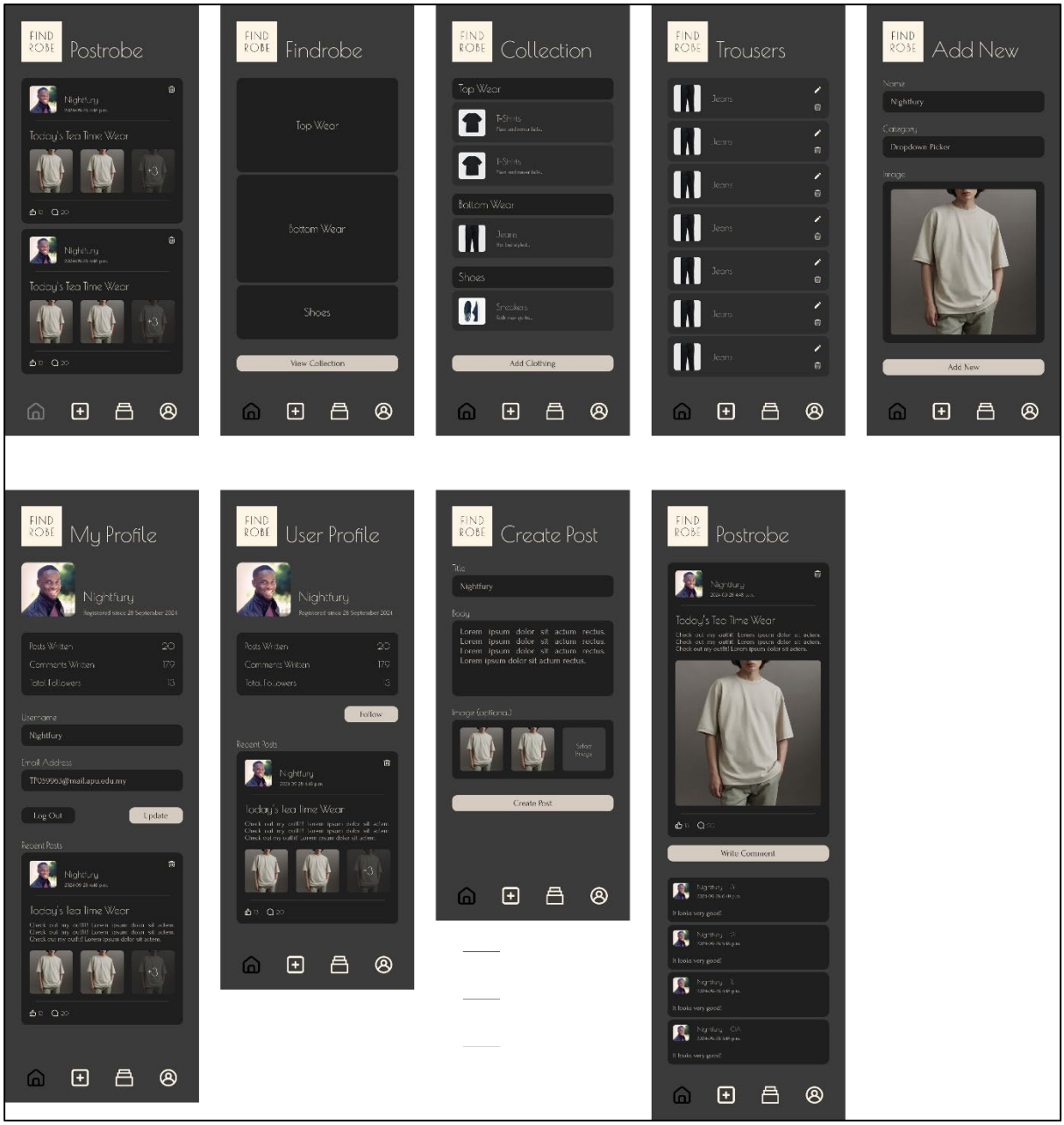


Figure 7: Findrobe Client View

4. Data Modelling Design

4.1. Data Management

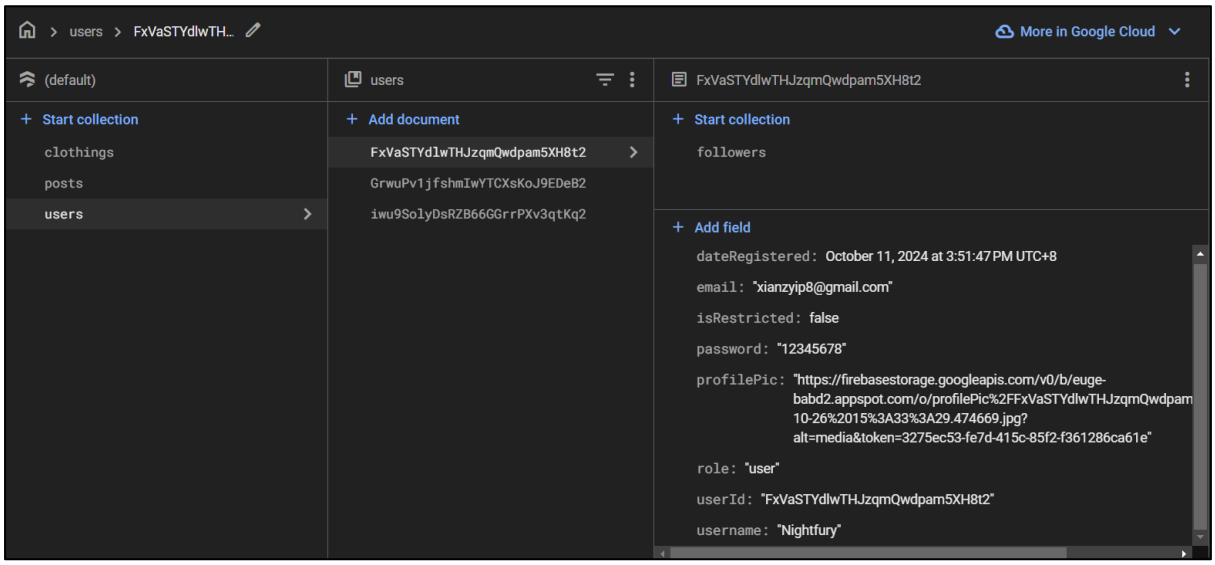


Figure 8: Users Collection with followers sub-collection in Cloud Firestore

In the user collection, each document ID is equal to the unique user ID. Within each user document, it contains the information of each user and a sub-collection for the user's follower.

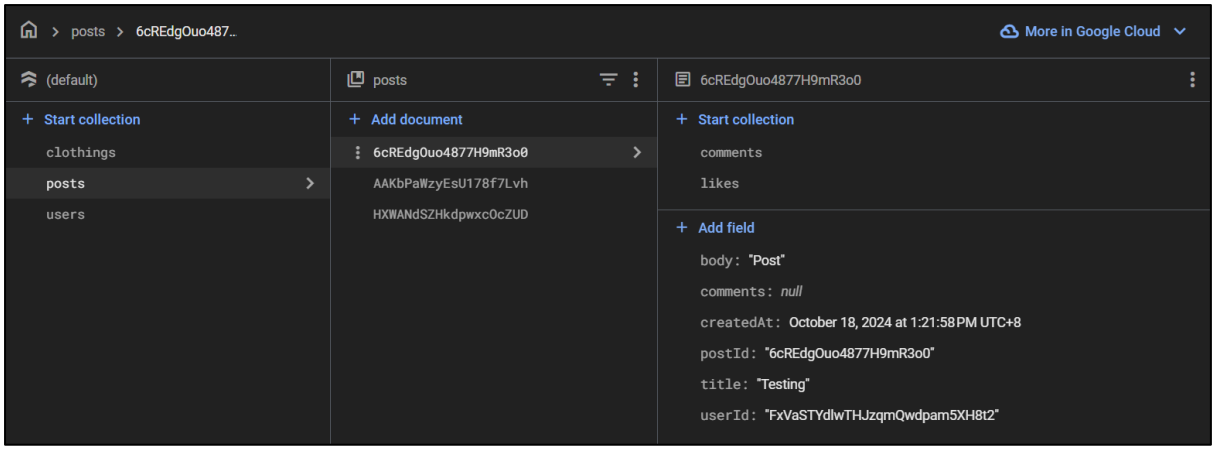


Figure 9: Posts Collection with comments and likes sub-collection in Cloud Firestore

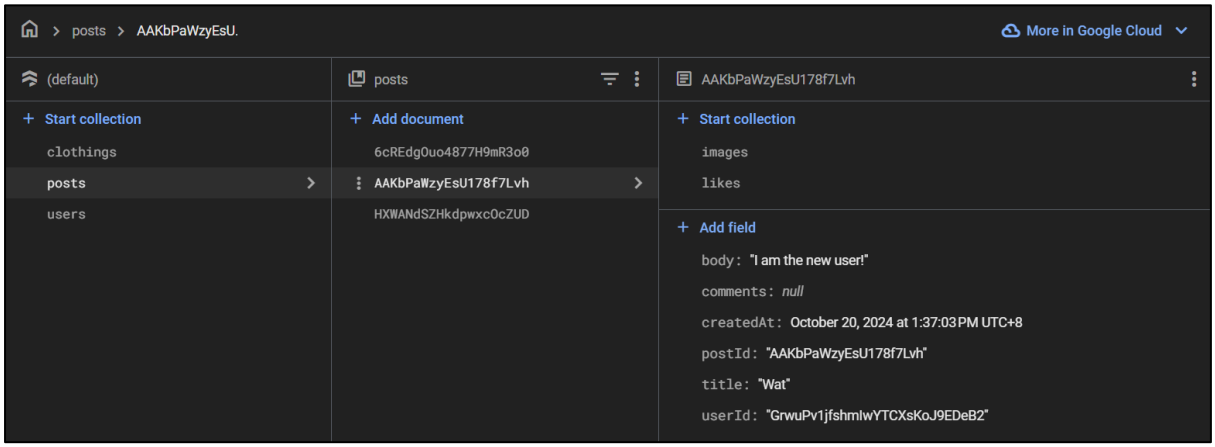


Figure 10: Posts Collection with images and likes sub-collection in Cloud Firestore

In the post collection, each document ID is equal to the unique post ID. Within each post document, it contains the information of each post and three sub-collections, images, comments, and likes. The image sub-collections will only be included when a post is created with image(s) attached.

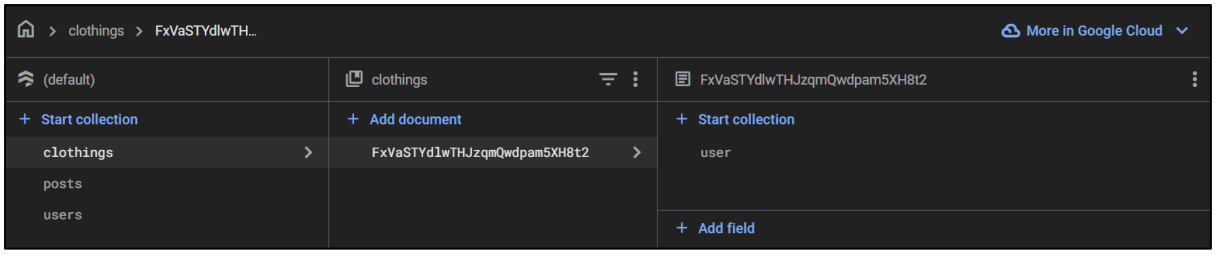


Figure 11: Clothings collection with user sub-collection in Cloud Firestore

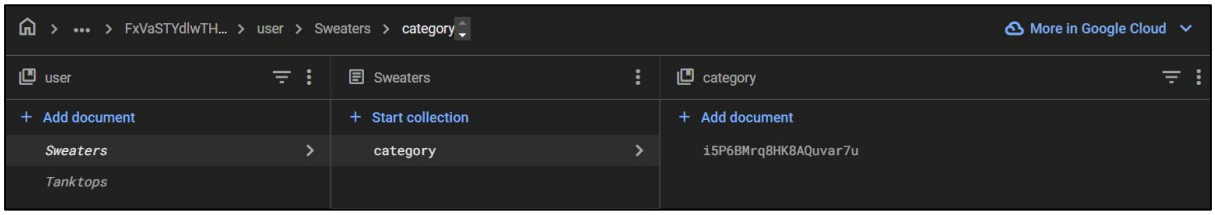


Figure 12: User sub-collection with category sub-collection in Cloud Firestore

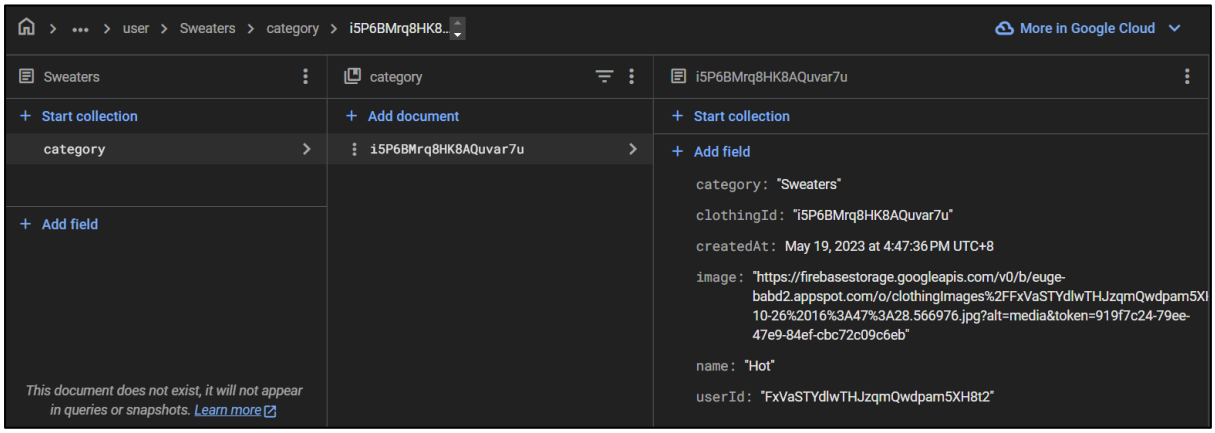


Figure 13: Category sub-collection in Cloud Firestore

In the clothing collection, the first level document ID is equal to the unique user ID with a sub-collection to identify the user. The second level document ID is based on the hardcoded clothes category from the proposed application with a sub-collection to store each clothing assigned to the category. Lastly, the third level document ID is based on the unique clothing ID. In the third level document, it contains the information of each clothing recorded by the user to the selected clothing category.

4.2. Query Management

```
Future<List<FindrobeClothing>> fetchClothing(String userId, String category) async {
  try {
    QuerySnapshot clothingDoc = await clothingCollection
        .doc(userId)
        .collection(clothingsByUserCollection)
        .doc(category)
        .collection(categoryInClothingCollection)
        .get();

    List<FindrobeClothing> clothings = clothingDoc.docs.map((clothingDoc) {
      return FindrobeClothing.fromMap(clothingDoc);
    }).toList();

    return clothings;
  } catch (e) {
    print("Failed to fetch clothing: $e");
    return [];
  }
}
```

Figure 14: Querying clothing from Specific Category of Specific User

The code above is to query and return a list of clothing based on the selected category of the current logged in user.

```
Future<List<FindrobeUser>> fetchAllUsers() async {
  try {
    QuerySnapshot querySnapshot = await usersCollection
        .where("role", isEqualTo: "user")
        .get();

    List<FindrobeUser> users = await Future.wait(querySnapshot.docs.map((doc) async {
      return FindrobeUser.fromMap(doc);
    }).toList());

    return users;
  } catch (e) {
    print("Failed to fetch all users: $e");
    return [];
  }
}
```

Figure 15: Querying only user role

The code above is to query and return a list of users with the role of “users” as both user and admin role account are saved together.

```dart
Future<FindrobePost?> fetchSinglePost(String postId) async {
  try {
    DocumentReference postRef = postsCollection.doc(postId);
    QuerySnapshot commentsSnapshot = await postRef.collection(commentsInPostCollection).get();
    QuerySnapshot likesSnapshot = await postRef.collection(likedInPostCollection).get();

    DocumentSnapshot postSnapshot = await postRef.get();
    FindrobePost thePost = FindrobePost.fromMap(postSnapshot);

    List<PostrobeComment> comments = [];
    for (var commentDoc in commentsSnapshot.docs) {
      String userId = commentDoc["userId"];
      DocumentSnapshot userDoc = await usersCollection.doc(userId).get();

      PostrobeComment comment = PostrobeComment.fromMap(commentDoc);
      comment.user = FindrobeUser.fromMap(userDoc);

      comments.add(comment);
    }
    comments.sort((a, b) => b.commentedAt.compareTo(a.commentedAt));

    List<PostrobeLike> likes = likesSnapshot.docs.map((likeDoc) {
      return PostrobeLike.fromMap(likeDoc);
    }).toList();

    thePost.comments = comments;
    thePost.likes = likes;

    return thePost;
  } catch (e) {
    print("Failed to fetch single post: $e");
    return null;
  }
}
```

Figure 16: Querying a single post and tabulate with user, like and comment data

The code above is to query and return a post along with its respective user, like and comment data which belongs to the unique post ID.

```dart
Future<int> fetchAllClothings() async {
  try {
    int totalClothings = 0;

    QuerySnapshot clothingSnapshot = await _firestore.collectionGroup(categoryInClothingCollection).get();

    totalClothings = clothingSnapshot.size;

    return totalClothings;
  } catch (e) {
    print("Failed to fetch all users: $e");
    return 0;
  }
}
```

Figure 17: Querying the total number of clothing recorded

The code above is to query and return the total number of clothing recorded in Cloud Firestore by all users.

```dart
Future<Map<String, int>> fetchAllClothingsByMonth() async {
  Map<String, int> clothings = {};

  try {
    QuerySnapshot querySnapshot = await _firestore.collectionGroup(categoryInClothingCollection).get();

    for (var doc in querySnapshot.docs) {
      Timestamp createdAt = doc["createdAt"];
      DateTime date = createdAt.toDate();

      String monthKey = "${date.year}-${date.month.toString().padLeft(2, '0')}";
      clothings.update(monthKey, (index) => index + 1, ifAbsent: () => 1);
    }

    return clothings;
  } catch (e) {
    print("Failed to fetch clothings by month: $e");
    return {};
  }
}
```

Figure 18: Querying the clothing recorded monthly

The code above is to query and return a map of string and integer of clothing recorded each month where the string will be the month and year, and the integer will be the total number of clothing recorded.

```dart
Future<void> fetchFollowers(String userId) async {
  QuerySnapshot followersSnapshot = await usersCollection
      .doc(userId)
      .collection(followersInUserCollection)
      .get();

  List<String> followers = followersSnapshot.docs.map((followDoc) {
    return followDoc.id;
  }).toList();

  List<FindrobeUser> followersDetail = [];
  for (String follower in followers) {
    DocumentSnapshot followerDoc = await usersCollection.doc(follower).get();

    if (followerDoc.exists) {
      followersDetail.add(FindrobeUser.fromMap(followerDoc));
    }
  }

  state = state.copyWith(
    followers: followersDetail,
    followersCount: followersDetail.length
  );
}
```

Figure 19: Querying the followers' detail of a user

The code above is to query and return the details of followed accounts along with the total number of followed accounts by the user.

5. System Implementation

5.1. Libraries Used

5.1.1. Cupertino Icons

```
Row(
  children: [
    const Icon(
      CupertinoIcons.person_3_fill,
      color: AppColors.beige,
      size: 28.0,
    ), // Icon
    const SizedBox(width: 20.0),
    Text(
      "${analytics.allUsers.length}",
      style: AppFonts.poiret24,
    ) // Text
  ]
) // Row
```

Figure 20: Cupertino Icons Usage

The code above utilizes Cupertino Icons library to have access to additional icon designs to improve the user interface overall looks and feels.

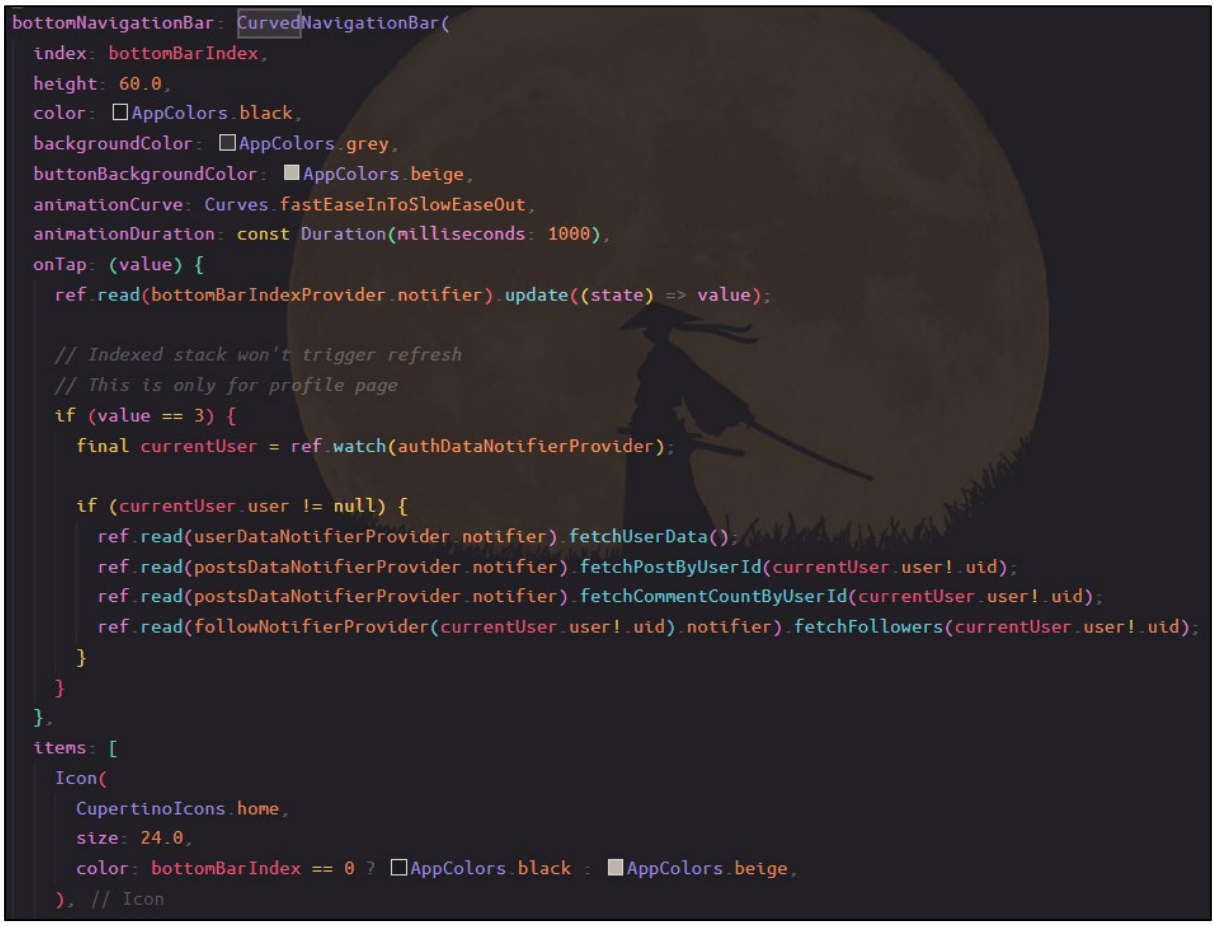
5.1.2. Crypto

```dart
// Helper function to calculate hash
Future<String> _calculateHash(File file) async {
  final bytes = await file.readAsBytes();
  return md5.convert(bytes).toString();
}

Future<void> _pickImages() async {
  final List<XFile>? pickedImages = await picker.pickMultiImage();

  if (pickedImages != null) {
    for (XFile image in pickedImages) {
      File imageFile = File(image.path);
      String imageHash = await _calculateHash(imageFile);

      if (!imageHashes.contains(imageHash)) {
        setState(() {
          imageFiles.add(File(image.path));
          imageHashes.add(imageHash);
        });
      }
    }
  }
}
```

Figure 21: Crypto Usage

The code above utilizes Crypto library to calculate the image hash and distinguish which images are duplicated or repeated when users are selecting images for their posts. If the images are duplicated, it will not be added to the main list.

5.1.3. Curved Navigation Bar

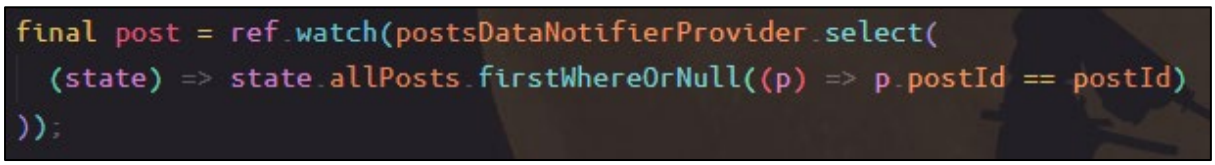


```
bottomNavigationBar: CurvedNavigationBar(
  index: bottomBarIndex,
  height: 60.0,
  color: AppColors.black,
  backgroundColor: AppColors.grey,
  buttonBackgroundColor: AppColors.beige,
  animationCurve: Curves.fastEaseInToSlowEaseOut,
  animationDuration: const Duration(milliseconds: 1000),
  onTap: (value) {
    ref.read(bottomBarIndexProvider.notifier).update((state) => value);

    // Indexed stack won't trigger refresh
    // This is only for profile page
    if (value == 3) {
      final currentUser = ref.watch(authDataNotifierProvider);

      if (currentUser.user != null) {
        ref.read(userDataNotifierProvider.notifier).fetchUserData();
        ref.read(postsDataNotifierProvider.notifier).fetchPostByUserId(currentUser.user!.uid);
        ref.read(postsDataNotifierProvider.notifier).fetchCommentCountByUserId(currentUser.user!.uid);
        ref.read(followNotifierProvider(currentUser.user!.uid).notifier).fetchFollowers(currentUser.user!.uid);
      }
    }
  },
  items: [
    Icon(
      CupertinoIcons.home,
      size: 24.0,
      color: bottomBarIndex == 0 ? AppColors.black : AppColors.beige,
    ), // Icon
```

Figure 22: Curved Navigation Bar Usage

The code above utilizes Curved Navigation Bar library to create a dynamic and animated bottom navigation bar with customizable design and multiple tabs.

5.1.4. Collection



```
final post = ref.watch(postsDataNotifierProvider.select(
  (state) => state.allPosts.firstWhereOrNull((p) => p.postId == postId)
));
```

Figure 23: Collection Usage

The code above utilizes Collection library to extend the list filtering functionality like .firstWhereOrNull() which allows the first element satisfying the query, or null if there are none.

5.1.5. Firebase Package Libraries

```dart
final FirebaseFirestore _firestore = FirebaseFirestore.instance;

final CollectionReference usersCollection = _firestore.collection("users");
final CollectionReference postsCollection = _firestore.collection("posts");
final CollectionReference clothingCollection = _firestore.collection("clothings");

const clothingsByUserCollection = "user";
const categoryInClothingCollection = "category";
const imagesInPostCollection = "images";
const likedInPostCollection = "likes";
const commentsInPostCollection = "comments";
const followersInUserCollection = "followers";      You, 3 weeks ago • Done login
```

Figure 24: Firebase Firestore Usage

The code above declares a fix name for all collections and sub-collections which are used in saving data from the application to Cloud Firestore.

```dart
Future<User?> signInWithEmailPassword(String email, String password) async {
  try {
    UserCredential userCredential = await _firebaseAuth.signInWithEmailAndPassword(
      email: email,
      password: password
    );

    User? user = userCredential.user;

    return user;
  } on FirebaseAuthException catch (e) {
    print("Firebase sign in error: ${e.message}");
  } catch (e) {
    print("Error signing in: $e");
  }

  return null;
}
```

Figure 25: Firebase Auth Usage

The code above is to check and sign in an existing user using email address and password. If the credentials match with Firebase Authentication, it will navigate to the client view. If the credentials do not match, it will prompt a message for the user.

```dart
Future<String> uploadImage(File profilePic, String userId) async {
  try {
    final storageRef = _storage.ref().child("profilePic/$userId/image_${DateTime.now()}.jpg");
    final uploadTask = await storageRef.putFile(profilePic);
    final profilePicUrl = await uploadTask.ref.getDownloadURL();

    return profilePicUrl;
  } catch (e) {
    print("Failed to upload image: $e");
    return "";
  }
}
```

Figure 26: Firebase Storage Usage

The code above is to save an image to Firebase Storage while creating and returning the URL to be accessed as a network image.

5.1.6. FL Charts

```
child: BarChart(
  swapAnimationCurve: Curves.fastEaseInToSlowEaseOut,
  swapAnimationDuration: const Duration(milliseconds: 1000),
  BarChartData(
    alignment: BarChartAlignment.spaceEvenly,
    minY: 0,
    maxY: maxY + 5.0,
    barGroups: barGroups,
    titlesData: FlTitlesData(
      topTitles: const AxisTitles(
        sideTitles: SideTitles(
          showTitles: false,
        ) // SideTitles
      ), // AxisTitles
      rightTitles: const AxisTitles(
        sideTitles: SideTitles(
          showTitles: false,
        ) // SideTitles
      ), // AxisTitles
      leftTitles: AxisTitles(
        sideTitles: SideTitles(
          showTitles: true,
          reservedSize: 40,
          getTitlesWidget: (double value, TitleMeta meta) {
            return Padding(
              padding: const EdgeInsets.only(left: 20.0),
              child: Text(
                "${value.toInt()}",
                style: AppFonts.forum12
              ) // Text
            ); // Padding
```

Figure 27: FL Chart Usage

The code above is to generate a bar chart by tabulating it with a dynamic data source and customized design to fit the user interface.

5.1.7. Flutter Native Splash

```yaml
flutter_native_splash:
  android: true
  color: "#3B3B3B"
  image: "assets/logo.png"

  android_12:
    color: "#3B3B3B"
    image: "assets/logo.png"
```

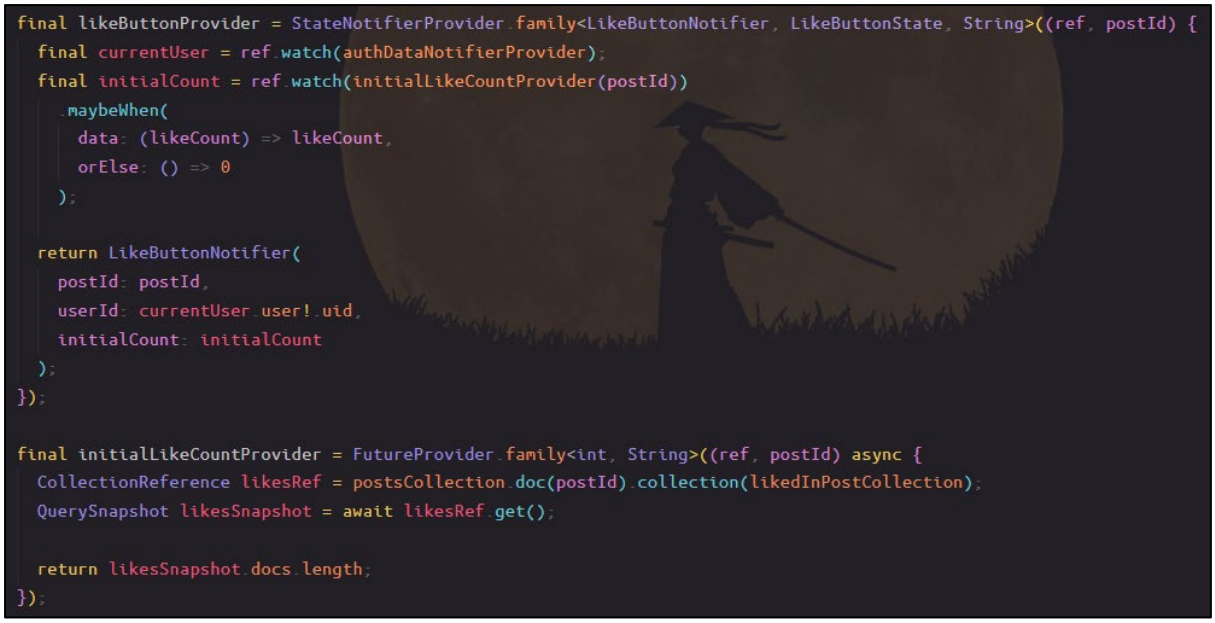
Figure 28: Flutter Native Splash Usage

The code above is to create a native splash screen with the hardcoded image and background colour when the application starts.

5.1.8. Flutter Riverpod

```dart
class LikeButtonNotifier extends StateNotifier<LikeButtonState> {
  final String postId;
  final String userId;

  LikeButtonNotifier({
    required this.postId,
    required this.userId,
    required int initialCount,
  }) : super(
    LikeButtonState(
      isLiked: false,
      likeCount: initialCount
    )
  ) {
    _checkIfLiked();
  }

  Future<void> _checkIfLiked() async {
    DocumentSnapshot likeDoc = await postsCollection
        .doc(postId)
        .collection(likedInPostCollection)
        .doc(userId)
        .get();

    if (likeDoc.exists) {
      state = LikeButtonState(isLiked: true, likeCount: state.likeCount);
    }
  }
```

Figure 29: Flutter Riverpod Usage Part 1



```dart
final likeButtonProvider = StateNotifierProvider.family<LikeButtonNotifier, LikeButtonState, String>((ref, postId) {
  final currentUser = ref.watch(authDataNotifierProvider);
  final initialCount = ref.watch(initialLikeCountProvider(postId))
    .maybeWhen(
      data: (likeCount) => likeCount,
      orElse: () => 0
    );

  return LikeButtonNotifier(
    postId: postId,
    userId: currentUser.user!.uid,
    initialCount: initialCount
  );
});

final initialLikeCountProvider = FutureProvider.family<int, String>((ref, postId) async {
  CollectionReference likesRef = postsCollection.doc(postId).collection(likedInPostCollection);
  QuerySnapshot likesSnapshot = await likesRef.get();

  return likesSnapshot.docs.length;
});
```

Figure 30: Flutter Riverpod Usage Part 2

The code above is to create a StateNotifier of a specific data type to observe the changes from the backend and display it to the user interface. It also provides function to update the state when it is called from the user interface. This will encourage separation of concerns between the front-end and back-end of the application.

5.1.9. Flutter Spinkit

```
class LoadingOverlay extends StatelessWidget {
  const LoadingOverlay({super.key});

  @override
  Widget build(BuildContext context) {
    return const Stack(
      children: [
        ModalBarrier(
          color: AppColors.grey,
          dismissible: false
        ), // ModalBarrier
        Center(
          child: Column(
            mainAxisSize: MainAxisSize.min,
            children: [
              SpinKitFadingCube(
                color: AppColors.beige,
                size: 50.0,
              ) // SpinKitFadingCube
            ],
          ) // Column
        ), // Center
      ]
    ); // Stack
  }
}
```

Figure 31: Flutter Spinkit Usage

The code above is to create a loading overlay when CRUD operations are executing to notify users that they should wait for a while when the state is updating.

5.1.10. Google Fonts

```
class AppFonts {
  // Poiret One
  static TextStyle poiret40 = GoogleFonts.poiretOne(
    fontSize: 40.0,
    fontWeight: FontWeight.normal,
    color: AppColors.white
  );

  static TextStyle poiret32 = GoogleFonts.poiretOne(
    fontSize: 32.0,
    fontWeight: FontWeight.normal,
    color: AppColors.white
  );

  static TextStyle poiret24 = GoogleFonts.poiretOne(
    fontSize: 24.0,
    fontWeight: FontWeight.normal,
    color: AppColors.white
  );

  static TextStyle poiret20 = GoogleFonts.poiretOne(
    fontSize: 20.0,
    fontWeight: FontWeight.normal,
    color: AppColors.white
  );
```

Figure 32: Google Fonts Usage

The code above is to declare a set of pre-defined Google Fonts with specific font size, font weight, and colour for consistency throughout the application.

5.1.11. Image Picker

```dart
Future<void> _pickImage(WidgetRef ref, ImageSource source, ImagePicker picker) async {
  final pickedFile = await picker.pickImage(source: source);

  if (pickedFile != null) {
    ref.read(addImageProvider.notifier).state = File(pickedFile.path);
  }
}

void _showImageSourceDialog(BuildContext context, WidgetRef ref, ImagePicker picker) {
  showModalBottomSheet(
    backgroundColor: AppColors.white,
    context: context,
    shape: const RoundedRectangleBorder(
      borderRadius: BorderRadius.only(
        topLeft: Radius.circular(10.0),
        topRight: Radius.circular(10.0)
      ) // BorderRadius.only
    ), // RoundedRectangleBorder
```

Figure 33: Image Picker Usage

The code above is to create an image picker for users to either select photos directly from phone gallery or take a new photo using the phone camera.

5.1.12. Intl

```dart
String formatDate({required DateTime dateTime}) {
  return DateFormat("d MMMM y - h:mm a").format(dateTime);
}

String formatTimestamp({required Timestamp timestamp}) {
  DateTime dateTime = timestamp.toDate();

  String formattedDate = DateFormat("d MMMM y - h:mm a").format(dateTime);

  return formattedDate;
}      You, 4 weeks ago • Done outer navigation screens
```

Figure 34: Intl Usage

The code above is to create functions to format how the date time should be displayed and convert timestamp to datetime type.

5.1.13. Multi Image Picker View

```dart
Future<void> _pickImages() async {
  final List<XFile>? pickedImages = await picker.pickMultiImage();

  if (pickedImages != null) {
    for (XFile image in pickedImages) {
      File imageFile = File(image.path);
      String imageHash = await _calculateHash(imageFile);

      if (!imageHashes.contains(imageHash)) {
        setState(() {
          imageFiles.add(File(image.path));
          imageHashes.add(imageHash);
        });
      }
    }
  }
}
```

Figure 35: Multi Image Picker View Usage

The code above is to allow users to select multiple images from their phone gallery and append it to a list to be viewed.

5.1.14. Photo View

```dart
void showImageDialog(BuildContext context, String imageUrl) {
  showDialog(
    context: context,
    builder: (context) {
      return Dialog(
        child: Container(
          decoration: BoxDecoration(
            color: AppColors.grey,
            borderRadius: BorderRadius.circular(10.0)
          ), // BoxDecoration
          width: MediaQuery.of(context).size.width,
          height: MediaQuery.of(context).size.height * 0.5,
          padding: const EdgeInsets.all(15.0),
          child: ClipRRect(
            borderRadius: BorderRadius.circular(5.0),
            child: PhotoView(
              imageProvider: NetworkImage(
                imageUrl
              ), // NetworkImage
              backgroundDecoration: const BoxDecoration(
                color: AppColors.grey
              ), // BoxDecoration
              customSize: MediaQuery.of(context).size,
              minScale: PhotoViewComputedScale.contained * 0.8,
              maxScale: PhotoViewComputedScale.covered * 2.0,
              enablePanAlways: true,
            )  // PhotoView
          ) // ClipRRect
        ), // Container
      ); // Dialog
    }
  );
}
```

Figure 36: Photo View Usage

The code above is to create a pop-up modal to display an enlarge version of the selected image based on the dimension of devices which can be zoomed in and out by users.

6. Application Screenshots

6.1. User

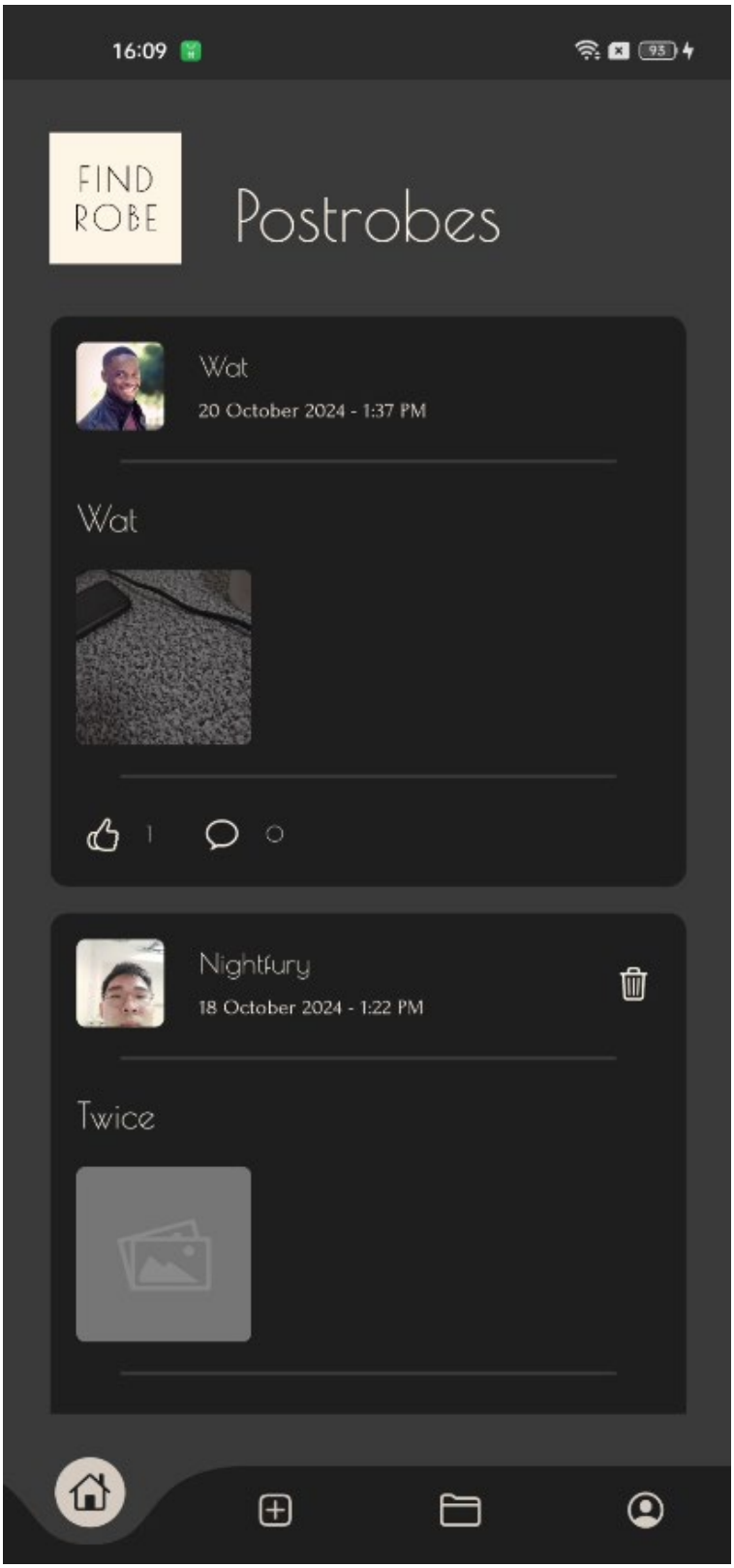


Figure 37: All Post Page

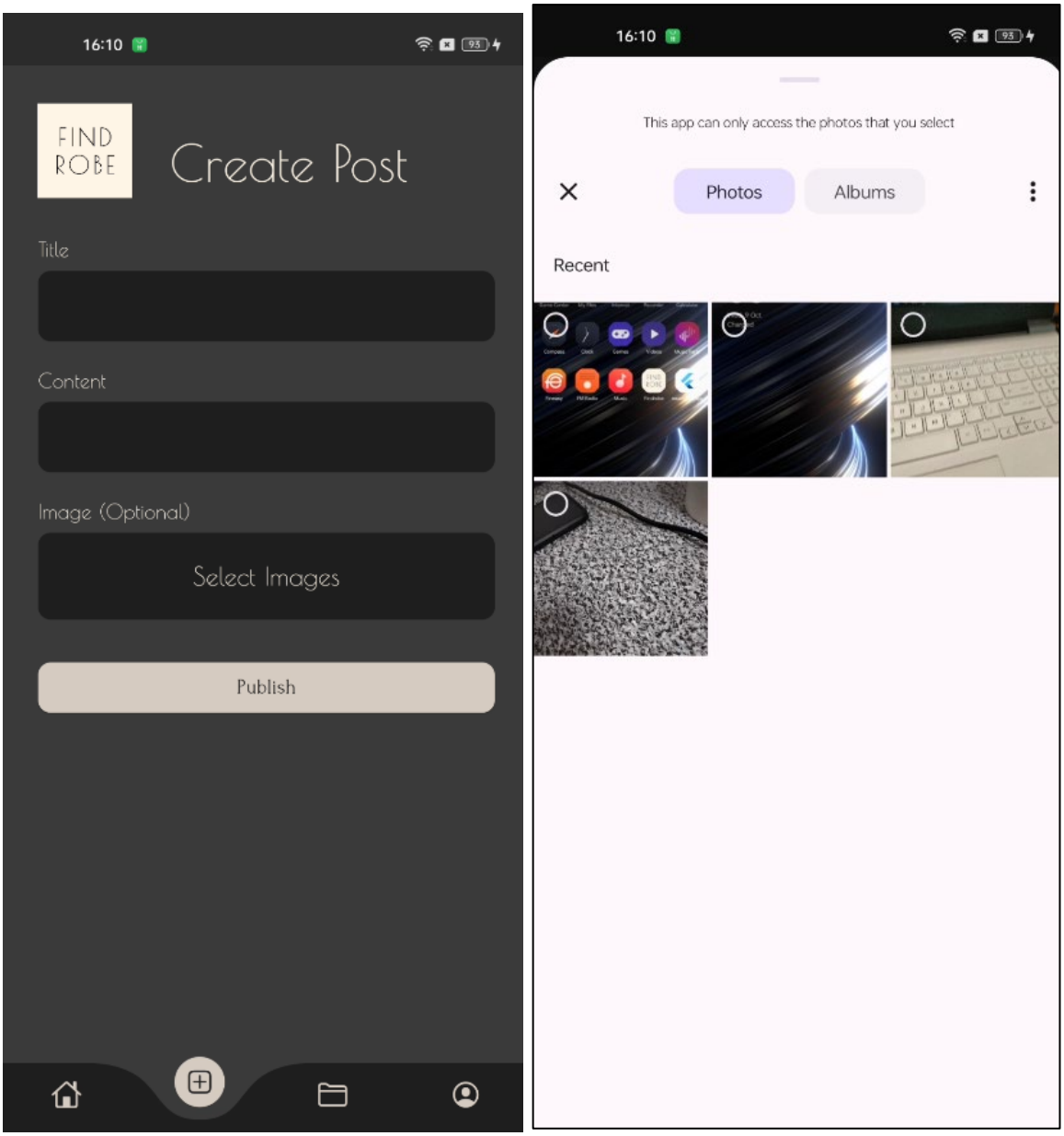


Figure 38: Create Post Page

Figure 39: Multiple Image Picker

Figure 40: Findrobe Mix and Match Page

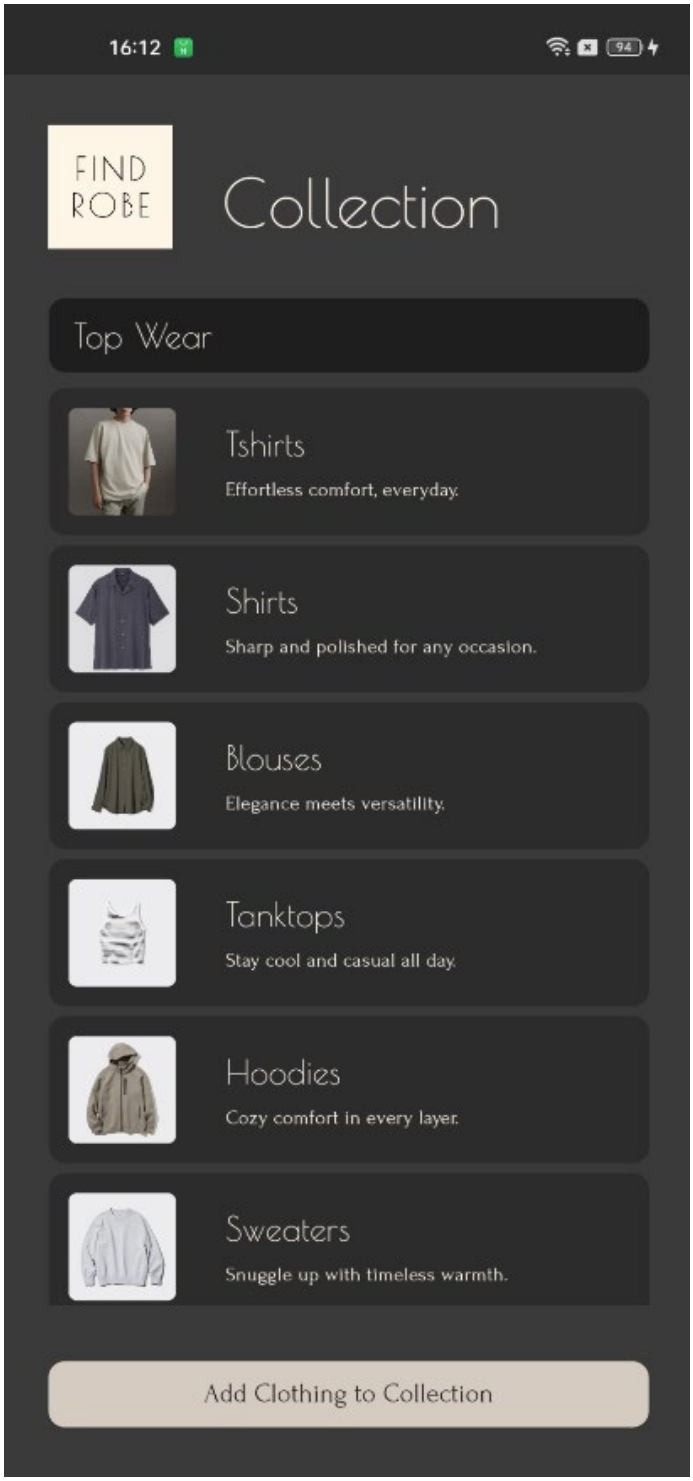


Figure 41: Collection Page

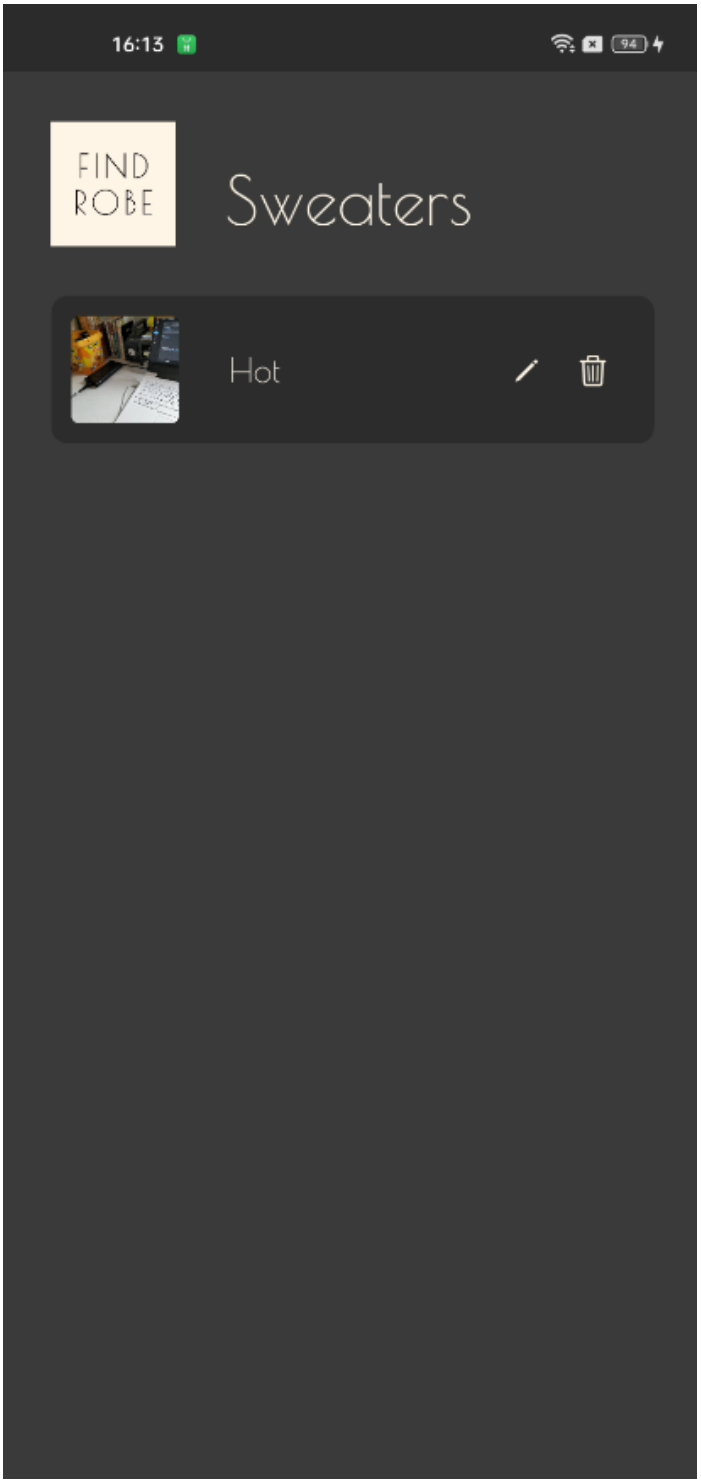


Figure 42: Category Page

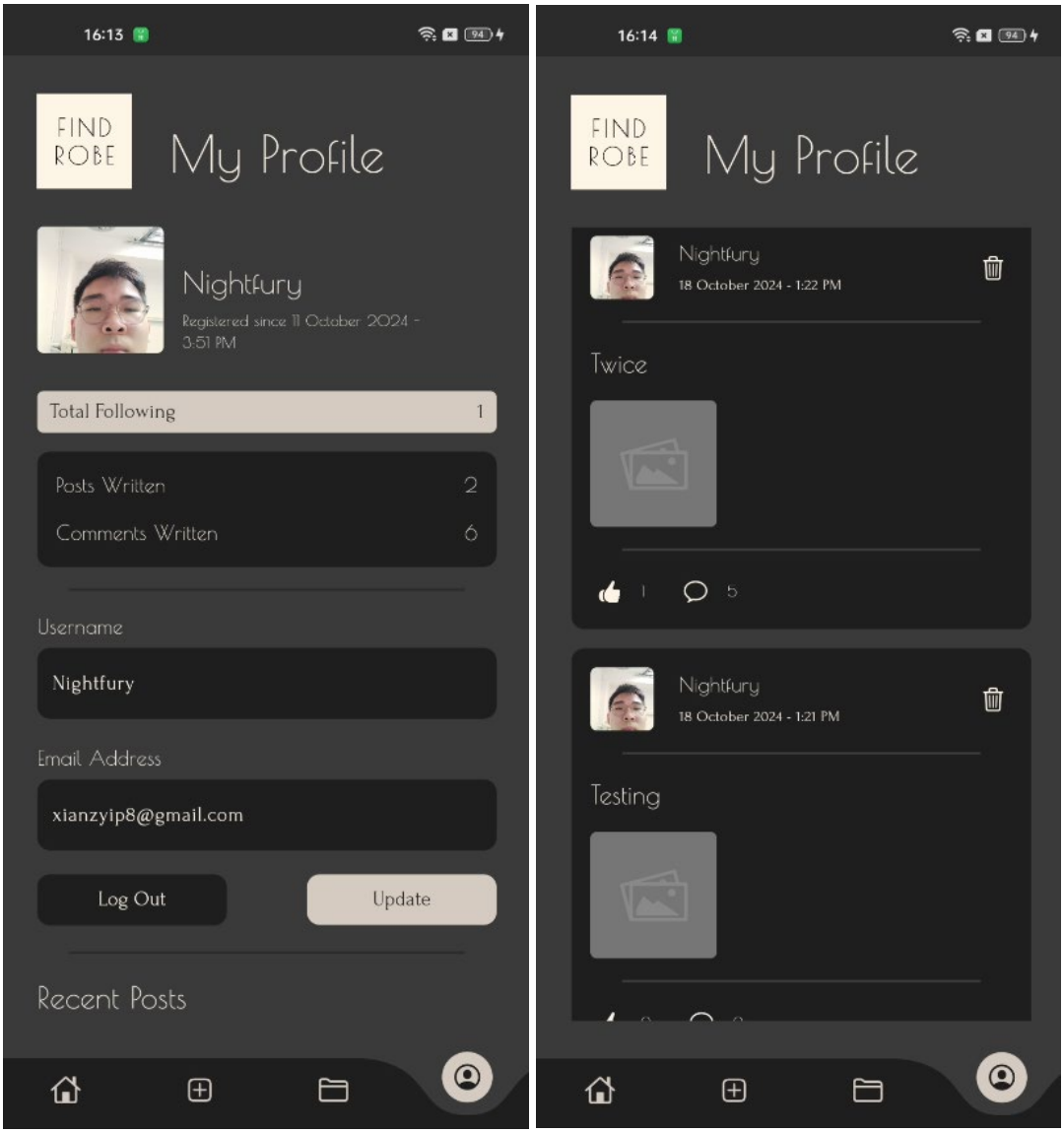


Figure 43 - 44: Profile Page

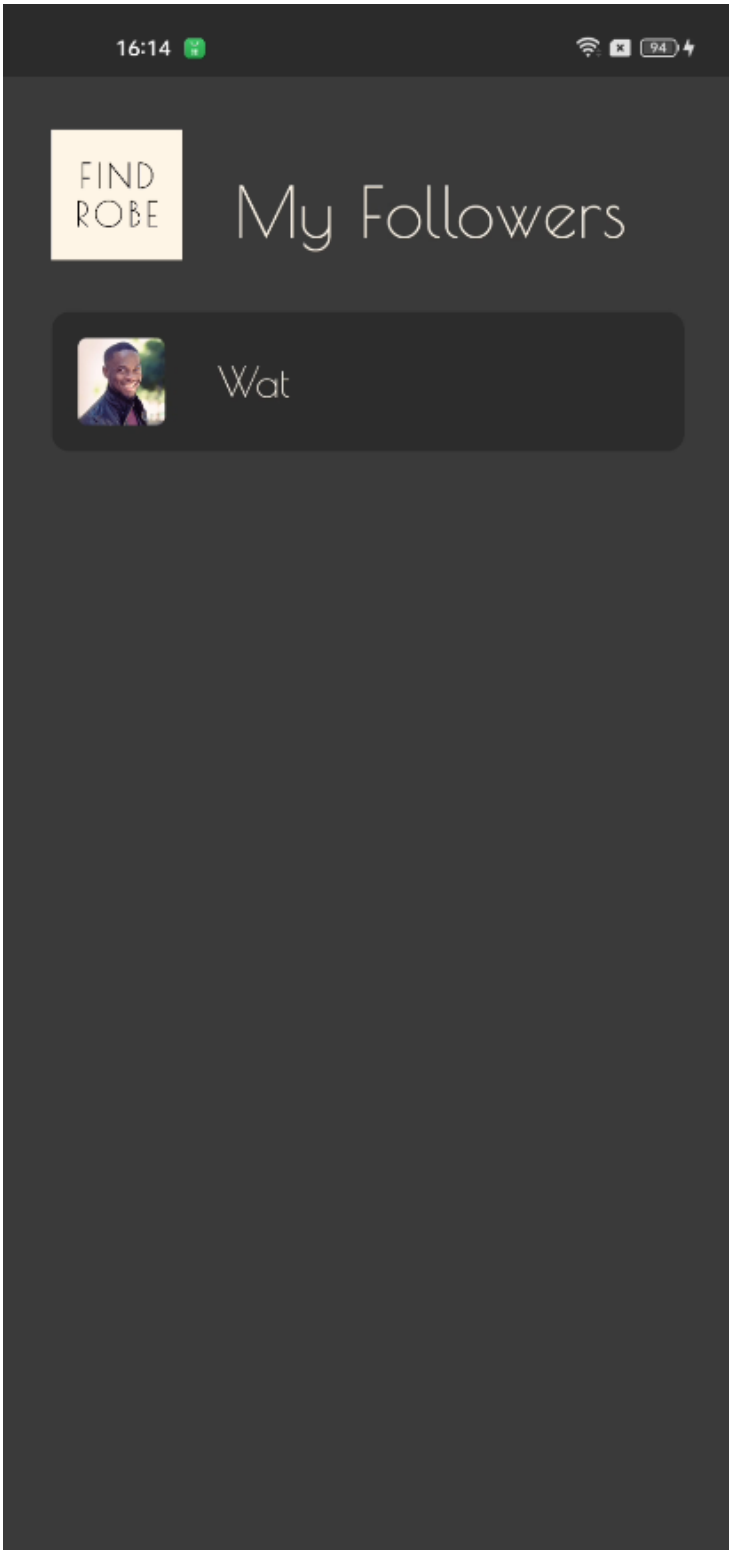


Figure 45: Followers Page

6.2. Admin

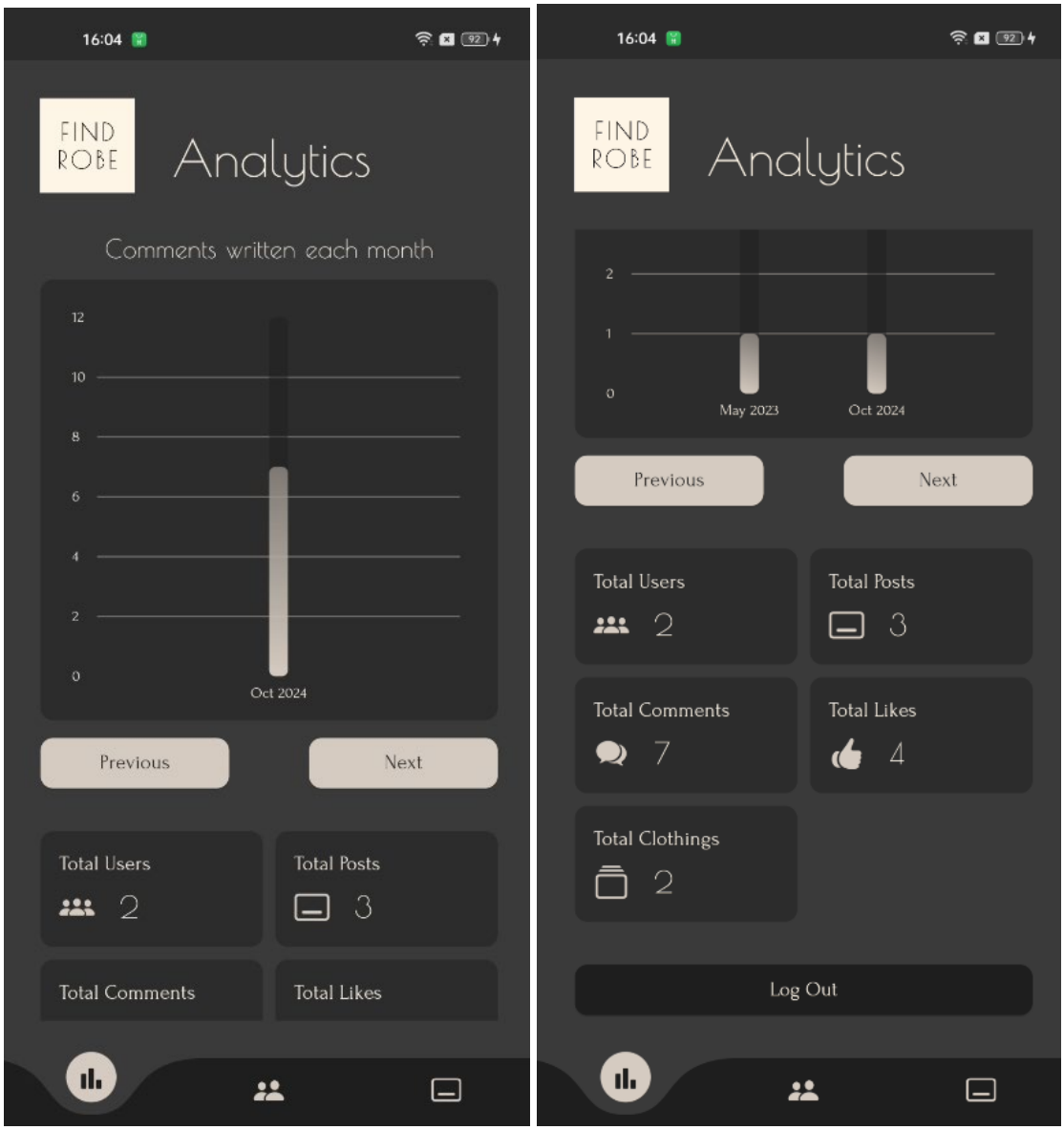


Figure 46 - 47: Analytics Page

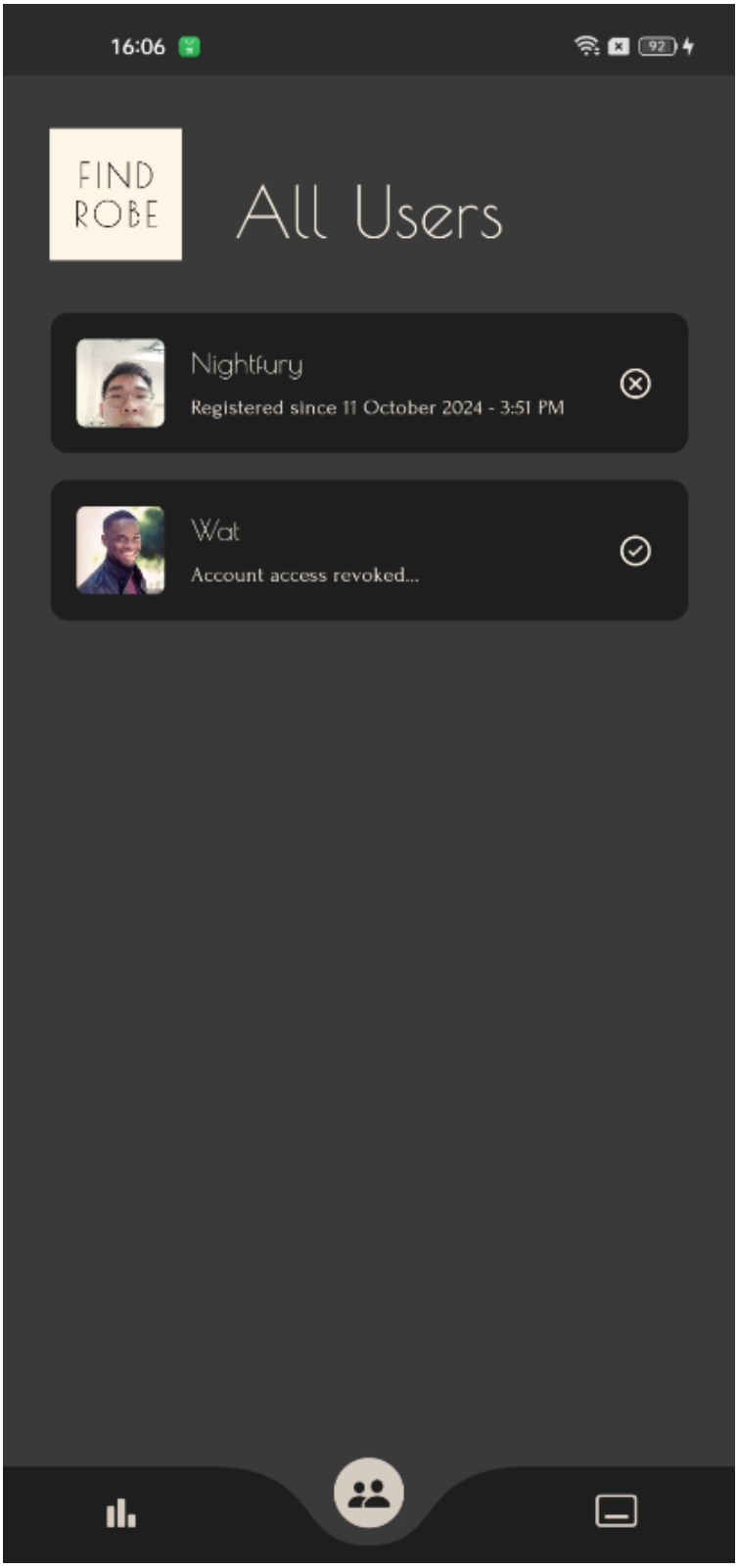


Figure 48: All Users Page

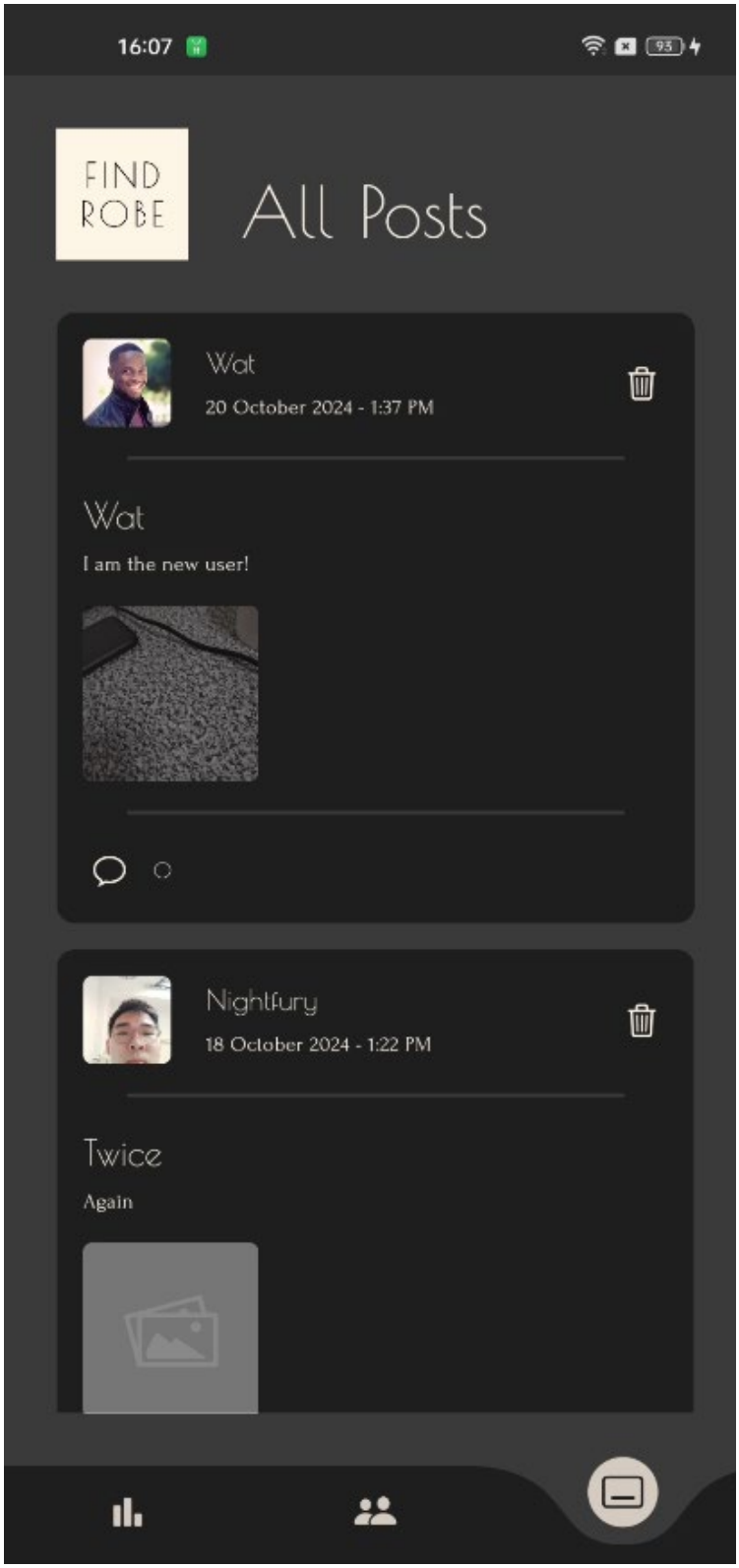


Figure 49: All Posts Page

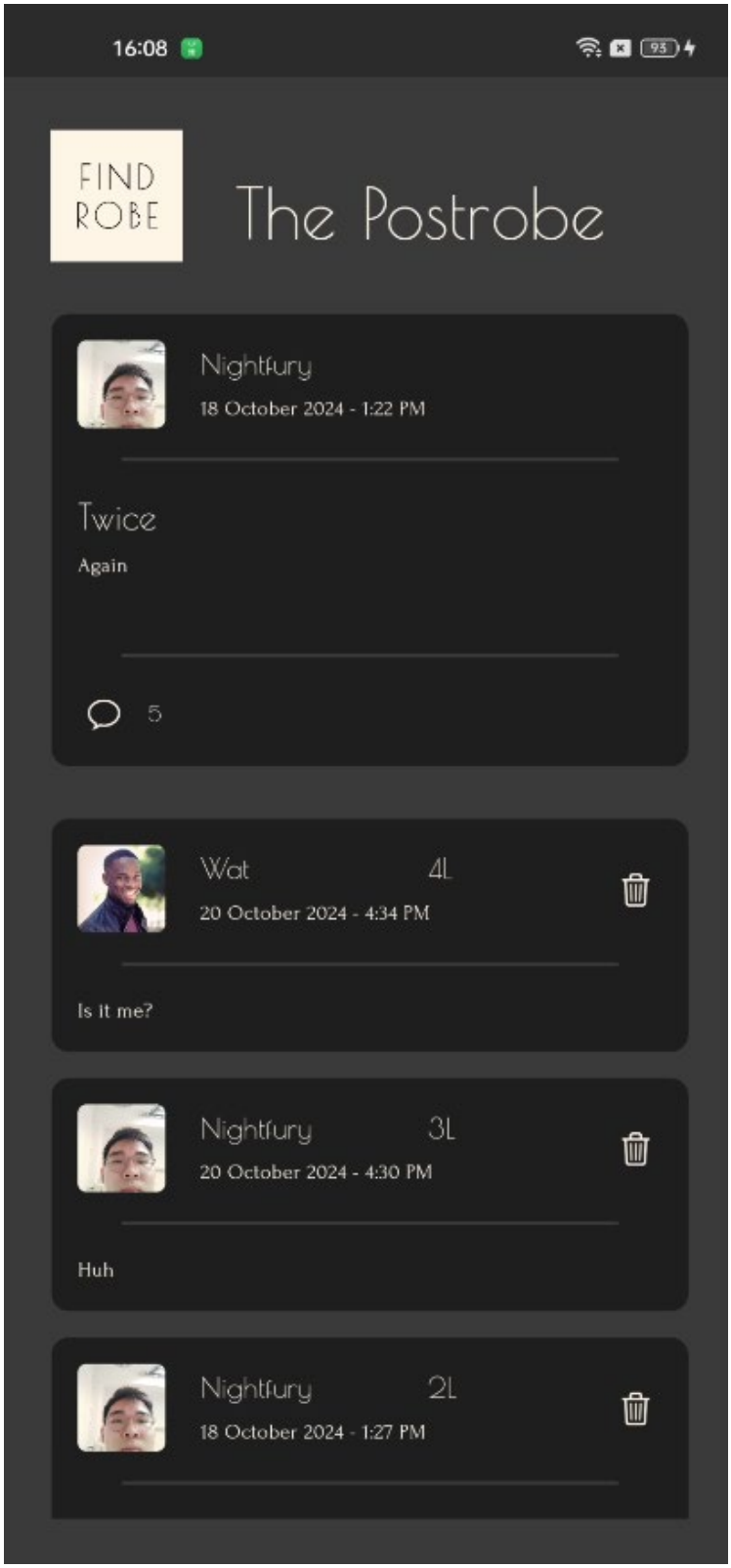


Figure 50: Single Post Page with Comments

7. Conclusion

In a nutshell, the proposed solution highlights its role in addressing wardrobe management by promoting organization behaviour in individuals while reducing impulsive clothing purchases and encouraging sustainable fashion practices. It also aims to enhance user experience by allowing individuals to manage their wardrobes effectively and connect with community sharing sustainable living values. The proposed solution supports responsible consumption and motivate individuals to have better well-being, aligned with Sustainable Development Goals (SDGs).

As for future updates, the proposed application could implement AI-driven recommendations for outfits from users' wardrobe based on user preferences, weather data etc, offering personalized styling suggestions. Aside from that, it could have an automatic notification about the freshness of clothing to determine whether the clothes are brand-new or aged as this could align with sustainable fashion practices.

8. References

Fernandes, B. (21 November, 2023). *Wardrobe Management 101- How Wardrobe Apps Can Help You Stay Organized?* Retrieved from Medium: https://medium.com/technology-insider/wardrobe-management-101-how-wardrobe-apps-can-help-you-stay-organized-2e9a667726ea

Martine, C. (25 October, 2023). *Will a digital wardrobe app help me streamline getting dressed?* Retrieved from Fashion Journal: https://fashionjournal.com.au/fashion/digital-wardrobe-dressed/

Miles, A. (26 June, 2024). *Wardrobe Organising Apps Are Worth the Initial Effort: Here's Why*. Retrieved from Good on You: https://goodonyou.eco/wardrobe-organising-apps/