NUS Orbital 2023 – Milestone 3

Proposed Level of Achievement:

Apollo 11

Motivation

As university students, we often feel **stress** from learning adulting to managing our relationship and studies and that often takes a toll on us. While there are friends who seek help openly with therapists and psychiatrists, many of the people around us often **remain quiet about their emotions and feelings**. This sort of emotional gatekeeping often leads to mental health illness. It is proven through research that sharing your thoughts and feelings to close friends and families can help drastically with mental health. However, not everyone has the privilege of calling their loved ones daily. By **improving the ease** and **accessibility** of which individuals are **able to share close and personal thoughts with close friends and family**, we hope that this can encourage healthy sharing of one's emotions and feelings to loved ones and help to take a step towards improved mental health in our society.

Currently, while there are diary apps that are present in the app store, we felt that they do not **go beyond being offline**. We believe that additional features that could help bring friends and families closer would be great in **amplifying the effects** that a **diary app** is able to provide in the **realm of emotional needs.**

Aim

The aim of our project is to provide a **diary with additional interaction features,** allowing a more active approach towards sharing one's feelings.

While we plan to keep the **traditional diary aspect**, we hope to provide an online system such that users are able to share their personal thoughts and worries easily, be it by **snippets** of their diary or posts in general. Through additional features of **personalization and styling**, as well as well thought out UI-UX processes, we hope to create a **comfortable and relaxing environment** so that users can feel at ease to share their **inner thoughts.**

The focus of our project is to **extend** on the **offline diary apps**, and add a more **social feature** to allow close friends and families to **interact via our app**. By creating features such as a friend-list **system, home-page feed, posting and quick reactions**, we will allow users to **garner quick feedback** from their posts as well as allow them to **notice** the people that are struggling around them. Of course, if the problem is personal, they can always choose to not post the diary and keep it private.

User Stories

1. As a user, I want to record down my daily thoughts and worries in my diary.
2. As a user, I want to be able to express my thoughts and worries with my closest friends easily, be it snippets of my diary or posts while remaining in control of what I share.
3. As a user, I want to notice that my friends may not be coping well emotionally or mentally so that I am able to talk to them or provide encouragement.
4. As a user, I want to personalize my profile and display information about myself with things like fun questions and trivia to lighten the mood.
5. As a user, I want to be able to react to the posts by people in my friend list and react to their own posts through an inbox.
6. As a user, I want to have a notification system to notify me if my friends are in trouble.

Features (Planned)

1. Diary feature
2. Login / Signups (With email)
3. Profiling & friend list system
4. HomePage feed
5. Posting (expire after 24 hour)
6. Notifications (scrapped due to need of payment for server-side notifications_

Features (Implemented as per Milestone 3)

1. Diary Feature
2. Login / Signups (With email)
3. Profiling & Friend List System
4. Home Page feed
5. Follow permissions
6. Posting (expire after 24 hours) (including backend overhaul)
7. Post history
8. Diary Snippets

Elaboration on features

The **Diary** feature simply encompasses the standard features of being able to write in text.

For the **Login and Signup** feature, we view it as rather routine given the online nature of interaction. Thus there is a need to create accounts in-order to store account specific information and personalization for individual users.

Profiling and friend list systems act as a precursor for the main Homepage feed function. The idea of profiling is to allow users to edit and create a biography that serves as an introduction as well as a profile picture used in the dashboard. This helps to reinforce the idea of sharing information. Whereas, the friend list/ following system allows users to have a list of friends in which they can selectively choose who to share.

The **homepage feed** will then feature the posts from the user's list of accounts that he or she follows. These posted diaries are presented in cards, making it easy to preview the diary. Users can then click on these cards to view more of the posted diary.

Lastly, there should be a **notification feature** to alert users that a friend requests for help or companionship. This can be activated or deactivated.

Explanation of Implementation of features

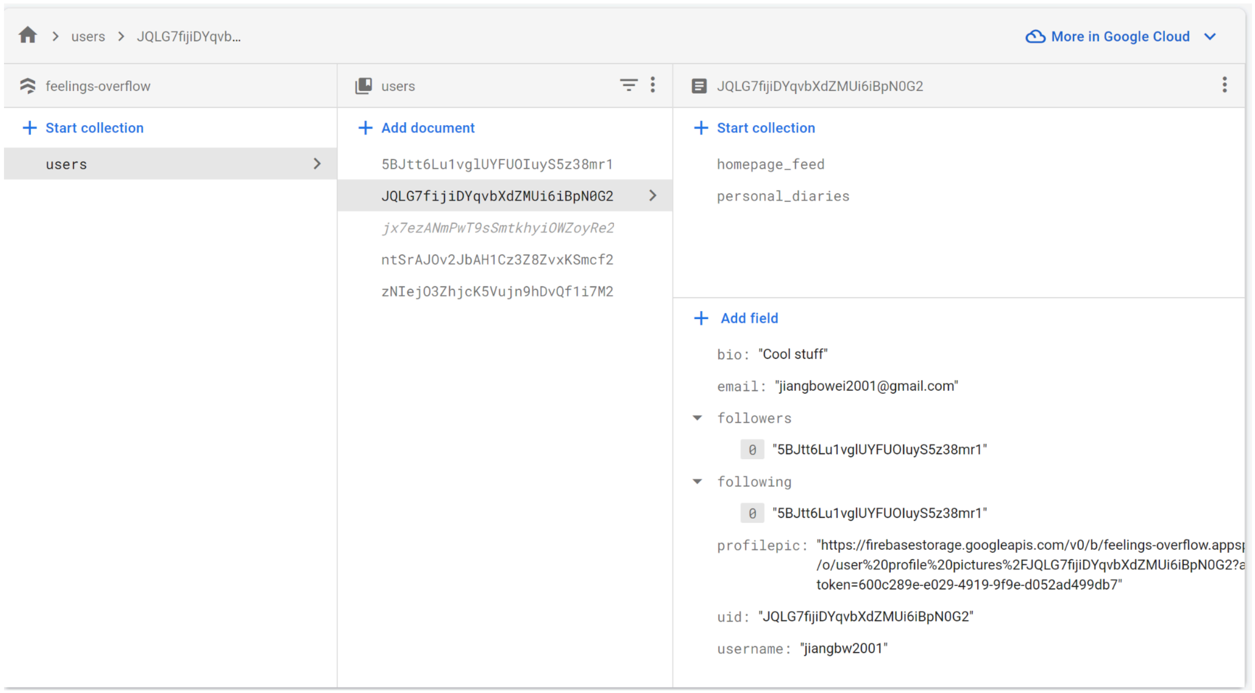
Sign ups and login:

After filling up the email, password and username fields when signing up, upon pressing the 'sign up' button, the system will create a new user on FirebaseAuth with the email and password. The user on FirebaseAuth will then have a UID (unique ID) associated with the account. Then, the system will create a new document in the users collection in Firestore and name it using the UID, and store other relevant details of the account as fields in the document, such as username, the list of followers and following, profile picture url etc.

Personal Diaries:

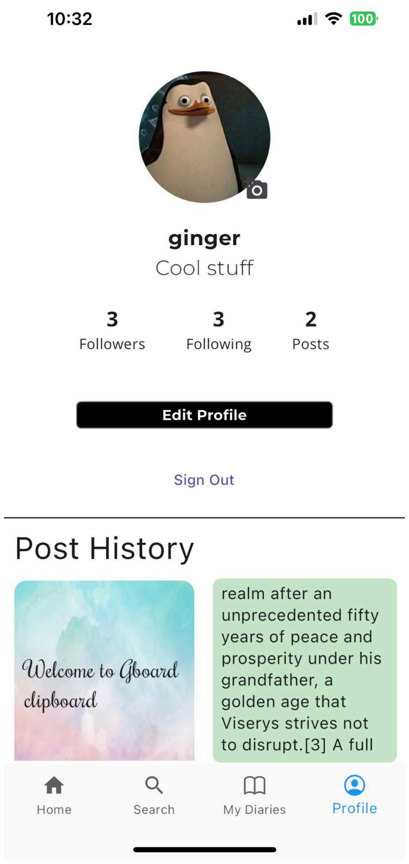
Each user has their own document under the users collection in the Firestore database, and there is a collection under that document that is used to store their personal diaries. Upon saving a diary, a new document with the relevant information about the diary such as diary title, date, diary content is then created and stored under the personal_diaries collection under the user's document in the database. When viewing his or her own personal diaries in MyDiariesTab, the app will read the data from the users' personal_diaries collection.

Profile:
As mentioned above, relevant details about the account will be stored as fields under the document in Firestore titled the users' UID. The users' bio is initially stored as an empty string ('') during account sign up and the user can click on the edit profile button to write a new bio that will overwrite the previous one in the database. The user's profile picture url in the database is initially set as a link to a default profile picture on the internet during signup. The user can choose another picture from their phone's photo gallery. The app will then upload the image to Firebase Storage and get the URL of the image and save it to Firestore and overwrite the previous one.



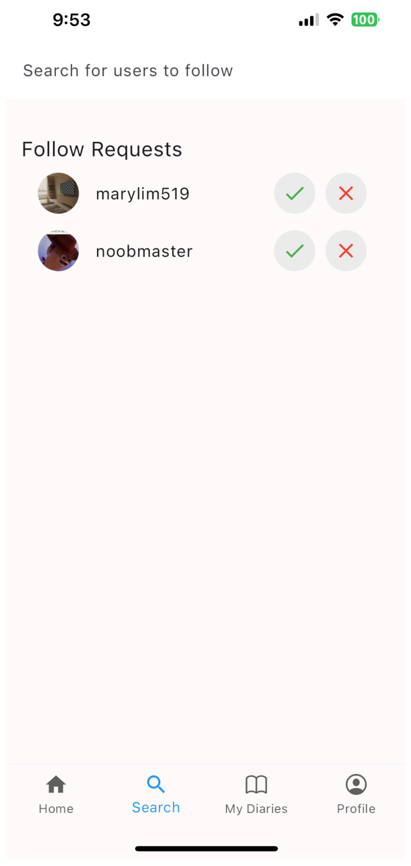
```
users
5BJtt6Lu1vglUYFUOIuyS5z38mr1
JQLG7fijiDYqvbXdZMUi6iBpN0G2
jx7ezANmPwT9sSmtkhyiOWZoyRe2
ntSrAJOv2JbAH1Cz3Z8ZvxKSmcf2
zNIej03ZhjcK5Vujn9hDvQf1i7M2

homepage_feed
personal_diaries

bio: "Cool stuff"
email: "jiangbowei2001@gmail.com"
followers
    0 "5BJtt6Lu1vglUYFUOIuyS5z38mr1"
following
    0 "5BJtt6Lu1vglUYFUOIuyS5z38mr1"
profilepic: "https://firebasestorage.googleapis.com/v0/b/feelings-overflow.apps
            /o/user%20profile%20pictures%2FJQLG7fijiDYqvbXdZMUi6iBpN0G2?a
            token=600c289e-e029-4919-9f9e-d052ad499db7"
uid: "JQLG7fijiDYqvbXdZMUi6iBpN0G2"
username: "jiangbw2001"
```

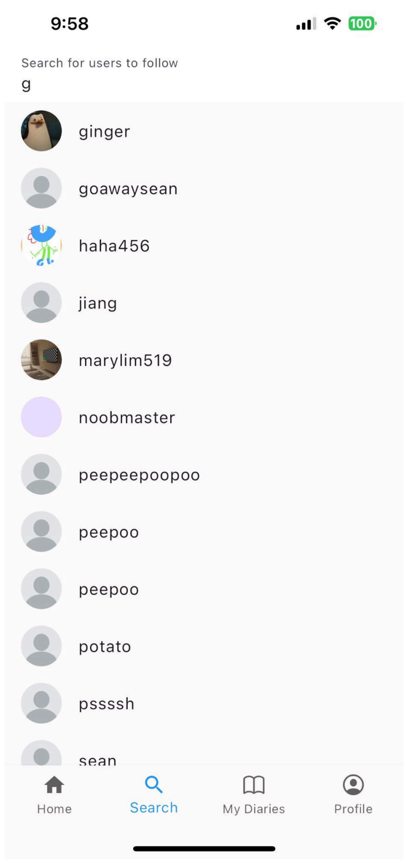
The user's post history, together with other relevant information, will be displayed in the profile page.


10:32
100
ginger
Cool stuff
3
Followers
3
Following
2
Posts
Edit Profile
Sign Out
Post History
Welcome to Gboard clipboard
realm after an unprecedented fifty years of peace and prosperity under his grandfather, a golden age that Viserys strives not to disrupt.[3] A full
Home
Search
My Diaries
Profile

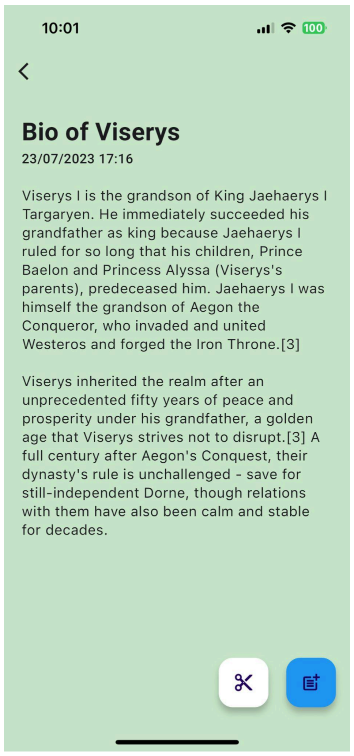
Friend list/ following system and follow permissions:
As mentioned above, the lists of following, followers and requests are stored as fields under the document in Firestore titled the users' UID. These lists are of type List<String> that contains the following, followers and requests' account UID. To follow an account, go to the search page by pressing the search icon in the navigation bar, search for users to follow > select the account and press the follow button. Upon pressing the follow button, the current user's UID is added to the target account's requests list. The account owner can login to their account and also go to the search page to look at the list of follow requests and choose to approve or reject them. This ensures that the user can choose who to share their sensitive information with.

If the target account holder chooses to approve the follow request, the system will add the target account's UID to the current user's following list, and also add the current user's UID to the target account's follower list. Vice versa for unfollowing. The following screen and followers screen thus read from the two lists of strings respectively and map each UID to that account's profile picture and username to display to the user.





Posting:
To post a diary, press the plus button on the top right of the home page and select a diary to post. Then, the user can choose to either post the whole diary or just a snippet of the diary. If the user wants to post the whole diary, he or she can just tap on the post button directly and the diary will be added to the posts collection under the user document. If the user wants to post only a snippet instead, he or she will need to select / highlight a short text in the diary (tap and hold on android, double tap on iPhone) first, and tap on the cut button. The user will be brought to another page to set the font and background image and preview the diary snippet before posting.



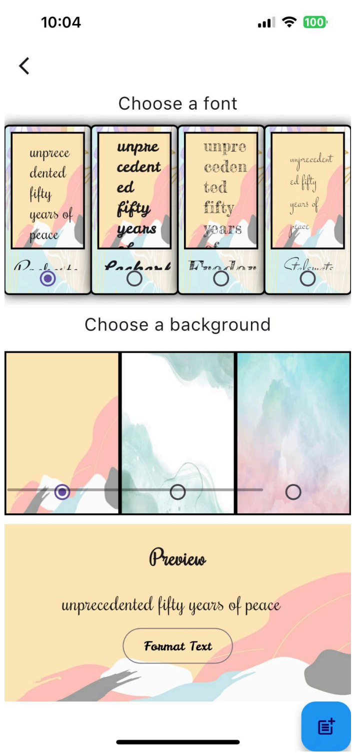
10:01

Bio of Viserys

23/07/2023 17:16

Viserys I is the grandson of King Jaehaerys I Targaryen. He immediately succeeded his grandfather as king because Jaehaerys I ruled for so long that his children, Prince Baelon and Princess Alyssa (Viserys's parents), predeceased him. Jaehaerys I was himself the grandson of Aegon the Conqueror, who invaded and united Westeros and forged the Iron Throne.[3]

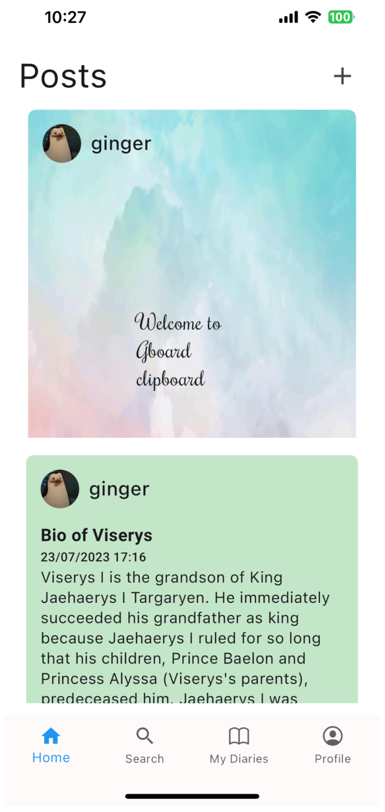
Viserys inherited the realm after an unprecedented fifty years of peace and prosperity under his grandfather, a golden age that Viserys strives not to disrupt.[3] A full century after Aegon's Conquest, their dynasty's rule is unchallenged - save for still-independent Dorne, though relations with them have also been calm and stable for decades.



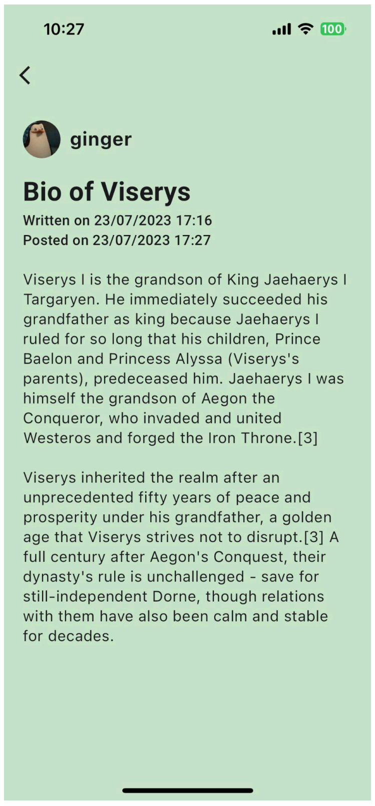
Homepage Feed + 24hr post expiry:
When the user opens up their home page, the app will run a getPosts method. The method will look through the list of followed accounts and retrieve each of their respective posts collection. Then, it will append all of them into a single stream. This stream is then finally run through a filter that checks if the posts are younger than 24 hours, before it is finally displayed to the user.

Each individual post will be presented as a card. The post document in Firebase only holds the poster's UID. When the post is loaded, the app will look on Firebase for the poster's username and profile picture. This ensures that the posts are all real time.

For full diary posts, users can click on the cards to expand them and view the full contents of the diary since the diary can be very long. However, for snippets, the text is very short. Hence, the cards are not interactable.



Bio of Viserys
23/07/2023 17:16
Viserys I is the grandson of King Jaehaerys I Targaryen. He immediately succeeded his grandfather as king because Jaehaerys I ruled for so long that his children, Prince Baelon and Princess Alyssa (Viserys's parents), predeceased him. Jaehaerys I was



Bio of Viserys
Written on 23/07/2023 17:16
Posted on 23/07/2023 17:27

Viserys I is the grandson of King Jaehaerys I Targaryen. He immediately succeeded his grandfather as king because Jaehaerys I ruled for so long that his children, Prince Baelon and Princess Alyssa (Viserys's parents), predeceased him. Jaehaerys I was himself the grandson of Aegon the Conqueror, who invaded and united Westeros and forged the Iron Throne.[3]

Viserys inherited the realm after an unprecedented fifty years of peace and prosperity under his grandfather, a golden age that Viserys strives not to disrupt.[3] A full century after Aegon's Conquest, their dynasty's rule is unchallenged - save for still-independent Dorne, though relations with them have also been calm and stable for decades.

Tech Stack
1. Flutter (Open-Source UI SDK, Compatible for IOS / Android)
2. Dart (Programming Language used in Flutter)
3. FireBase (Backend database handling)
4. Github (Version control and team work)

Development Plan:
4th week of June: Testing and debugging
1st week of July: Re-implement posting
2nd week of July: Add in notifications.
3rd week of July: Testing and debugging

Challenges faced during development of product
- Bad point is rewriting data base numbers, high interaction for measuring follower / following count if used on a large scale
- UI generally is hard to make customisable

- Error cases for the authentication had to be manually written (No implementation provided by FirebaseAuth for printing errors)
- Profile pictures are stored as URLs, and loading the profile pictures are done using the NetworkImage widget in flutter. However, depending on the user's internet connection, loading the network image may take some time and the user will see a blank picture during loading.
- Firestore does not have a TTL(Time to live) concept for documents. The only way to delete is either using code or through accessing the database on firebase website. Hence, we are unable to clean up old posts. Hence, most of the data that are loaded at the homepage feed will be filtered out (older than 24hrs), leading to inefficiency.
- We originally wanted to implement a notification system that sends notifications to followers when the user posts a diary. However, sending push notifications to followers would mean using Firebase Cloud Messaging and Firebase Build API in order to get the server to trigger a notification to relevant users. Unfortunately, Firebase Build API requires a premium Firebase plan. Hence, we chose not to implement notifications.

Software Development Process

Requirement Gathering

As part of the software development process, upon deciding the tech stack as well as the feature we wished to see in our app to tackle the problems, we also gathered the technology that we have to learn in order to make it happen.

Therefore, we embarked on a learning journey via udemy in order to get a better understanding of Dart and Flutter along with the usage of the IDE Android Studio.

In essence, we have gathered essential knowledge about Dart and its syntax along with Flutter UI management.

Dart generally behaves like Java syntax wise, has generics but no wild cards but has the standard coding syntax such as lambda functions.

In terms of Flutter UI, we learnt about common widgets such as Scaffold, Appbar, Containers, Padding, Margins, along with other more complex widget like ListView, GridView

Furthermore, we also went on to learn how to use Firebase as a backend means to our app. We came up with a way to manage our user data and learnt how to process data transfer and also to manage errors generated from the server.

https://docs.google.com/document/d/1FsN4NDB40uXvuJ1o7eLObudPV4fA1f-p0PtbqwQnMtU/edit?usp=sharing
Is the process of our learning journey and requirement gathering as per milestone 1

Planning

We separated tasks into separate parts and allocated them to each member, with an estimated time of completion for the task. Code is usually pushed by then. A Google document / tracker was used to keep track of a list of features yet to be implemented, known bugs, and personal to do lists. This helps us stay on schedule and ensures that features are implemented on time.

Design

We decided to have a my diaries section where the user can create personal diaries that are stored in a database, which means that they can access it from another device provided they are logged in. This also means that they can also choose to not post the diary if they feel that it is very personal.

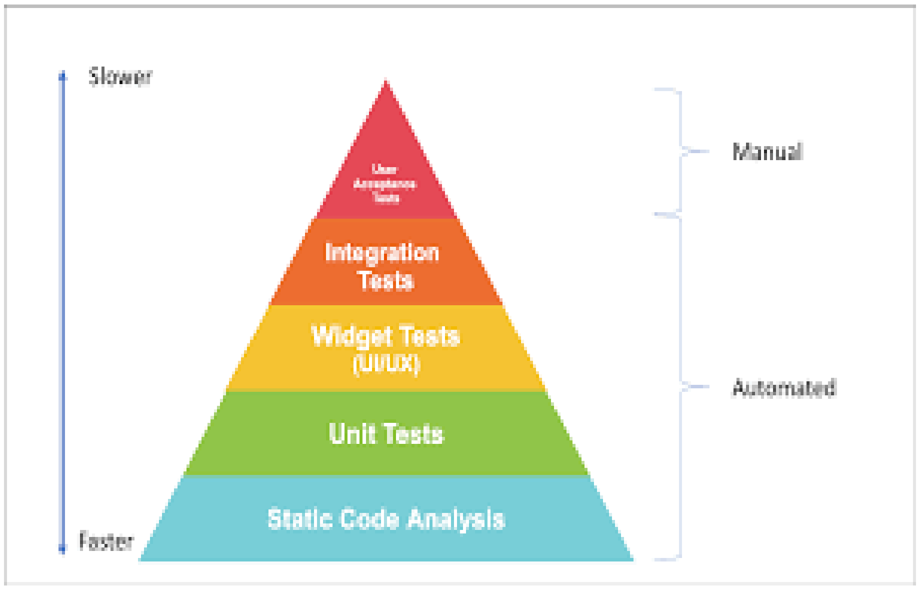
We created diary cards that are used in homepage feed and MyDiaries to provide a sneak peek to contents of a diary. This can help users more easily see the diary contents and other relevant information and help to make navigating through diaries easier.

A minimalist design theme was decided in order to keep the app easy to use and learn.

All backend methods and functions are kept in either Firebase Methods file or ImageFunction file, so that the files that make up the screens are not too cluttered with backend code.

Testing

As part of the software engineering process, it is crucial for our application to undergo rigorous and extensive testing in order to ensure that user interaction is desired and that there are no bugs when using our application as intended.



Credit:
https://bensonthew.medium.com/understand-the-core-concept-of-test-in-flutter-to-write-a-fewer-errors-of-flutter-app-e1fcdde7f4df
In app development there are different levels of testing in a hierarchical order as seen above. These include manual and automated tests. For our app, we have decided to

embark on 4 forms of testing in order to test the **UI/UX, Back-end functionality**, as well as the **app as a whole**

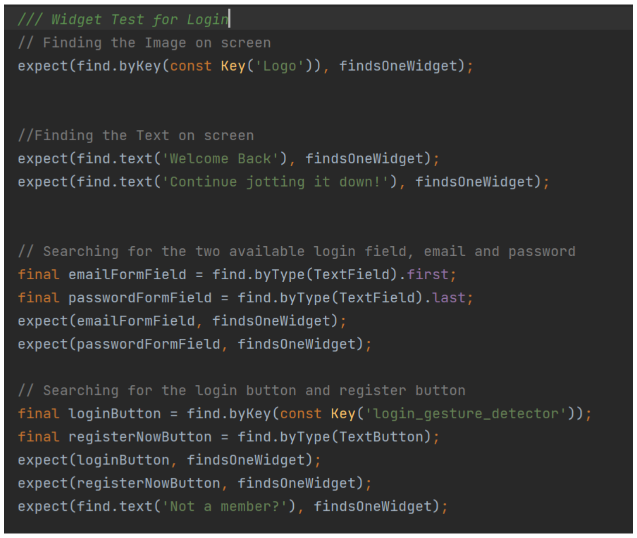
1) Widget Testing
2) Unit Testing
3) Integration Testing
4) User Acceptance Test

In our approach, we will be using **Widget Testing, Unit Testing** while doing our integration test.

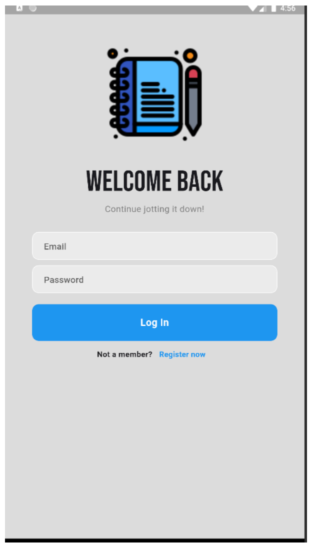
Widget Test

Widget Testing comes as a form of testing for the User Interface (UI). The design of each web screen comes with unique design elements named **widgets**. This could include Text widgets, Image widgets, TextField Widgets, Button widgets and more. By doing widget testing, it helps to ensure the existence of known widgets meant to be in the screens.

Login Screen



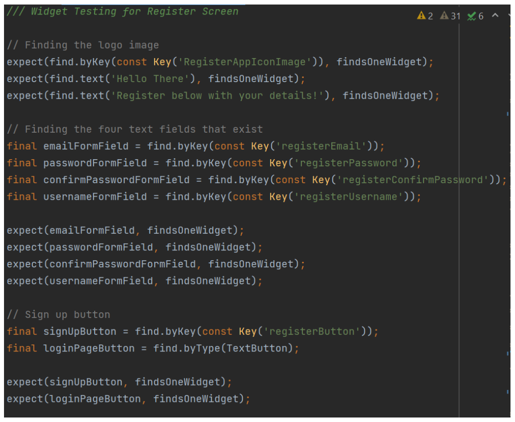
```
/// Widget Test for Login
// Finding the Image on screen
expect(find.byKey(const Key('Logo')), findsOneWidget);


//Finding the Text on screen
expect(find.text('Welcome Back'), findsOneWidget);
expect(find.text('Continue jotting it down!'), findsOneWidget);


// Searching for the two available login field, email and password
final emailFormField = find.byType(TextField).first;
final passwordFormField = find.byType(TextField).last;
expect(emailFormField, findsOneWidget);
expect(passwordFormField, findsOneWidget);

// Searching for the login button and register button
final loginButton = find.byKey(const Key('login_gesture_detector'));
final registerNowButton = find.byType(TextButton);
expect(loginButton, findsOneWidget);
expect(registerNowButton, findsOneWidget);
expect(find.text('Not a member?'), findsOneWidget);
```



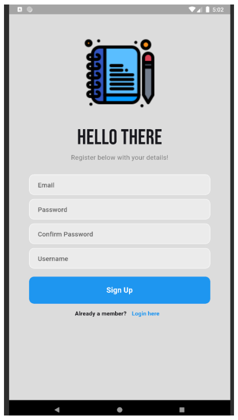
We are looking for **unique UI elements** (Also known as **widgets**) in this case to identify that it is the Login Screen and that it contains the right elements and widgets. In this case, we know our Login screen has a few unique elements, that is the **Logo, Word Text ("Welcome Back", "Continue jotting it down"), Two Text Fields, a Login Button, a "Register now" button, Text("Not a member?")**.

If this test passes, it means that the **UI** for the login page passes the test case.

Register Screen

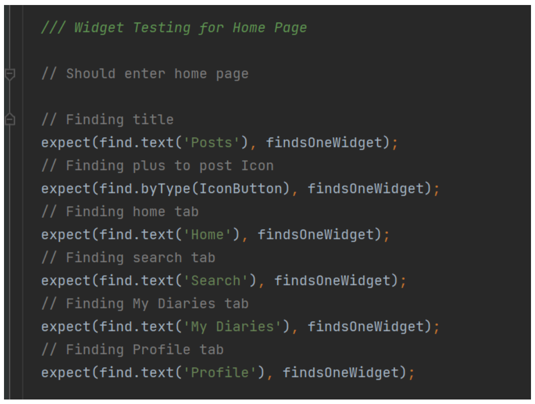


```
/// Widget Testing for Register Screen

// Finding the logo image
expect(find.byKey(const Key('RegisterAppIconImage')), findsOneWidget);
expect(find.text('Hello There'), findsOneWidget);
expect(find.text('Register below with your details!'), findsOneWidget);

// Finding the four text fields that exist
final emailFormField = find.byKey(const Key('registerEmail'));
final passwordFormField = find.byKey(const Key('registerPassword'));
final confirmPasswordFormField = find.byKey(const Key('registerConfirmPassword'));
final usernameFormField = find.byKey(const Key('registerUsername'));

expect(emailFormField, findsOneWidget);
expect(passwordFormField, findsOneWidget);
expect(confirmPasswordFormField, findsOneWidget);
expect(usernameFormField, findsOneWidget);

// Sign up button
final signUpButton = find.byKey(const Key('registerButton'));
final loginPageButton = find.byType(TextButton);

expect(signUpButton, findsOneWidget);
expect(loginPageButton, findsOneWidget);
```

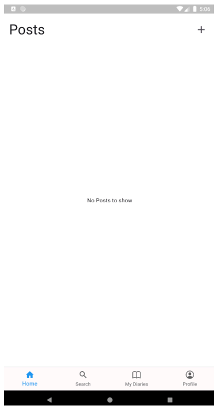

HELLO THERE
Register below with your details!
Email
Password
Confirm Password
Username
Sign Up
Already a member? Login here

Testing for the **Logo, “Hello There” , “Register below with your details”, 4 Text Fields that can be inputted (email, password, confirm password, username), Sign Up button, “Login now” button**

Homepage screen

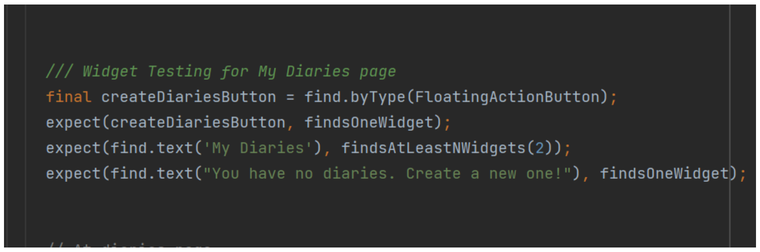


```
/// Widget Testing for Home Page

// Should enter home page

// Finding title
expect(find.text('Posts'), findsOneWidget);
// Finding plus to post Icon
expect(find.byType(IconButton), findsOneWidget);
// Finding home tab
expect(find.text('Home'), findsOneWidget);
// Finding search tab
expect(find.text('Search'), findsOneWidget);
// Finding My Diaries tab
expect(find.text('My Diaries'), findsOneWidget);
// Finding Profile tab
expect(find.text('Profile'), findsOneWidget);
```



Testing for the **“Posts” word**, the **“+” Icon Button** on the top right, the **4 tabs** that make the Bottom Navigation Bar **(Home, Search, My Diaries, Profile)**

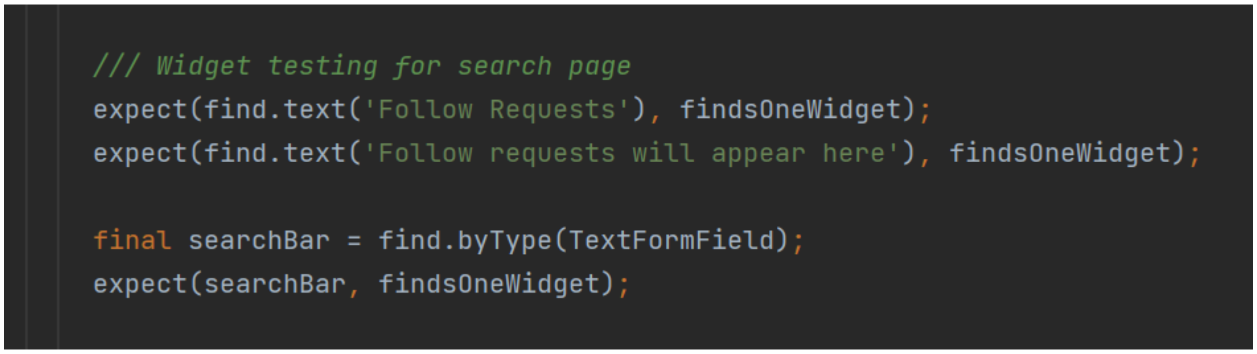
My Diaries screen



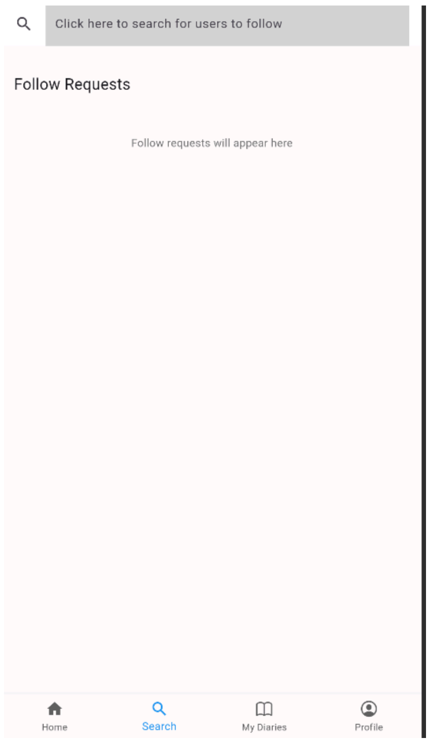
```
/// Widget Testing for My Diaries page
final createDiariesButton = find.byType(FloatingActionButton);
expect(createDiariesButton, findsOneWidget);
expect(find.text('My Diaries'), findsAtLeastNWidgets(2));
expect(find.text("You have no diaries. Create a new one!"), findsOneWidget);
```

Testing for the **Floating Action Button to create diaries,** as well as the standard My Diaries Header Text along with the “You have no diaries. Create a new one!” text for empty pages.

Search screen



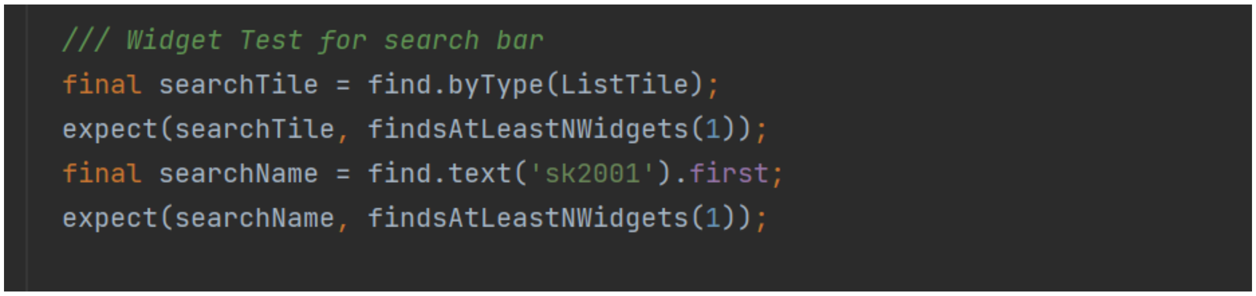
```
/// Widget testing for search page
expect(find.text('Follow Requests'), findsOneWidget);
expect(find.text('Follow requests will appear here'), findsOneWidget);

final searchBar = find.byType(TextFormField);
expect(searchBar, findsOneWidget);
```



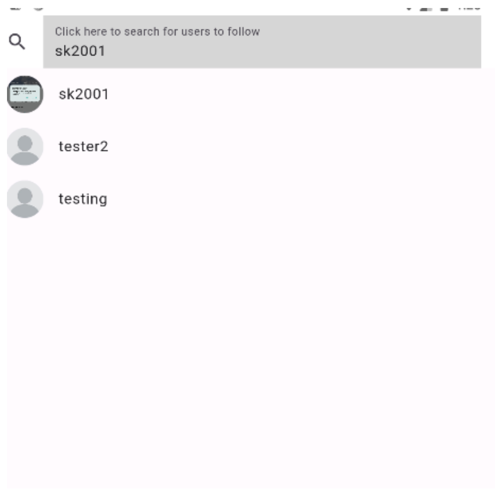
Click here to search for users to follow
Follow Requests
Follow requests will appear here
Home
Search
My Diaries
Profile

We test for the **searchBar** which is a TextFormField, and the **two text displays**. One being ‘Follow Requests’ and the other ‘Follow requests will appear here’ which only shows up if there are no Follow requests.

Search Bar Results



```
/// Widget Test for search bar
final searchTile = find.byType(ListTile);
expect(searchTile, findsAtLeastNWidgets(1));
final searchName = find.text('sk2001').first;
expect(searchName, findsAtLeastNWidgets(1));
```

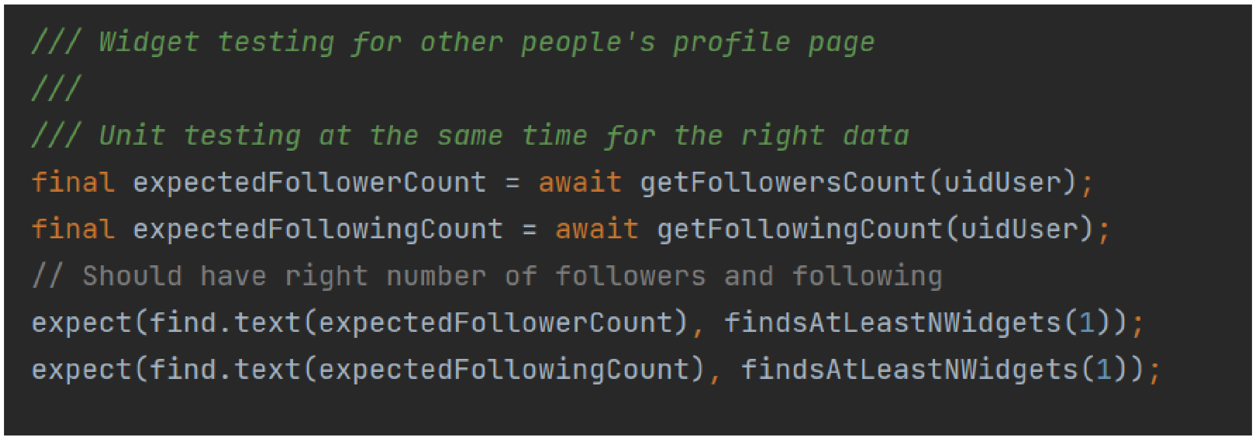


Click here to search for users to follow
sk2001

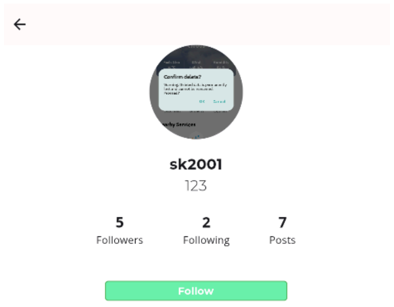
sk2001
tester2
testing

We are testing for the **ListTile** that should show up for a valid user. In this case, we will be using a user we know exist, which is **sk2001**, an account made by Sean.
This are the two information we are checking in our widget test.

Other's Profile Page

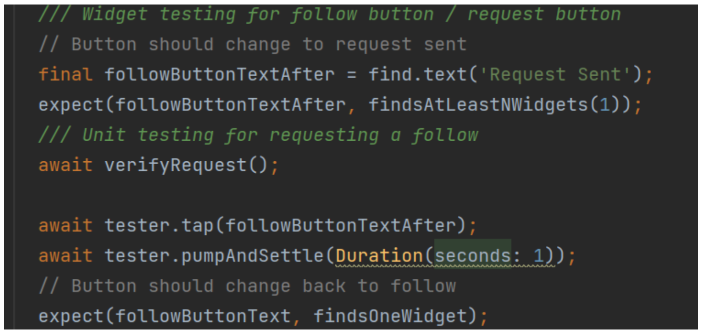


```
/// Widget testing for other people's profile page
///
/// Unit testing at the same time for the right data
final expectedFollowerCount = await getFollowersCount(uidUser);
final expectedFollowingCount = await getFollowingCount(uidUser);
// Should have right number of followers and following
expect(find.text(expectedFollowerCount), findsAtLeastNWidgets(1));
expect(find.text(expectedFollowingCount), findsAtLeastNWidgets(1));
```



We are doing **widget checking** to ensure that the **numerical display** for the **Followers, Following counts are correct.** This means we have to retrieve data from the server which is thus also part of **Unit Testing.**

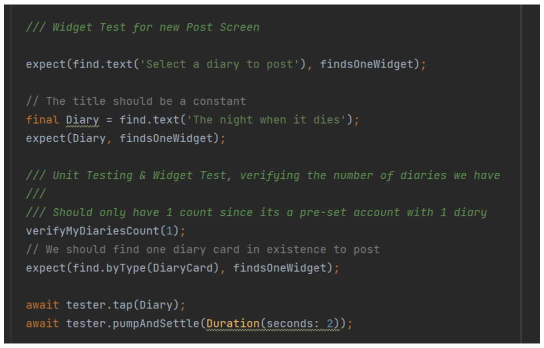
Follow / Request Send Buttons



```
/// Widget testing for follow button / request button
// Button should change to request sent
final followButtonTextAfter = find.text('Request Sent');
expect(followButtonTextAfter, findsAtLeastNWidgets(1));
/// Unit testing for requesting a follow
await verifyRequest();

await tester.tap(followButtonTextAfter);
await tester.pumpAndSettle(Duration(seconds: 1));
// Button should change back to follow
expect(followButtonText, findsOneWidget);
```

We are just checking whether **the UI is being responsive after on Tap.** That is whether it changes to Request Sent and back after deselecting / selecting.

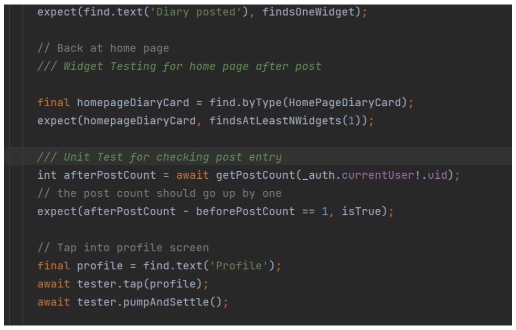
Selecting Diary to post



```
/// Widget Test for new Post Screen

expect(find.text('Select a diary to post'), findsOneWidget);

// The title should be a constant
final Diary = find.text('The night when it dies');
expect(Diary, findsOneWidget);

/// Unit Testing & Widget Test, verifying the number of diaries we have
///
/// Should only have 1 count since its a pre-set account with 1 diary
verifyMyDiariesCount(1);
// We should find one diary card in existence to post
expect(find.byType(DiaryCard), findsOneWidget);

await tester.tap(Diary);
await tester.pumpAndSettle(Duration(seconds: 2));
```

We are checking that there should only by default be 1 diary created, which is specifically for the test purpose, the right heading / title being **‘Select a diary to post**’, a diary with a title named ‘**The night when it dies**’

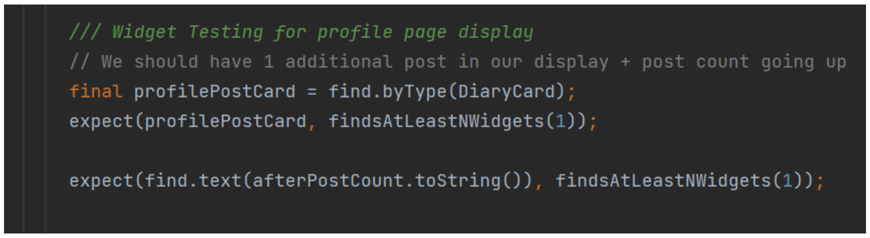
Checking Home page after posting



```
expect(find.text('Diary posted'), findsOneWidget);

// Back at home page
/// Widget Testing for home page after post

final homepageDiaryCard = find.byType(HomePageDiaryCard);
expect(homepageDiaryCard, findsAtLeastNWidgets(1));

/// Unit Test for checking post entry
int afterPostCount = await getPostCount(_auth.currentUser!.uid);
// the post count should go up by one
expect(afterPostCount - beforePostCount == 1, isTrue);

// Tap into profile screen
final profile = find.text('Profile');
await tester.tap(profile);
await tester.pumpAndSettle();
```

We are just checking that there is a diary card being displayed in the homepage after posting, that is we are able to **detect** the presence of such a widget with the type **HomepageDiaryCard**

Own Profile Page



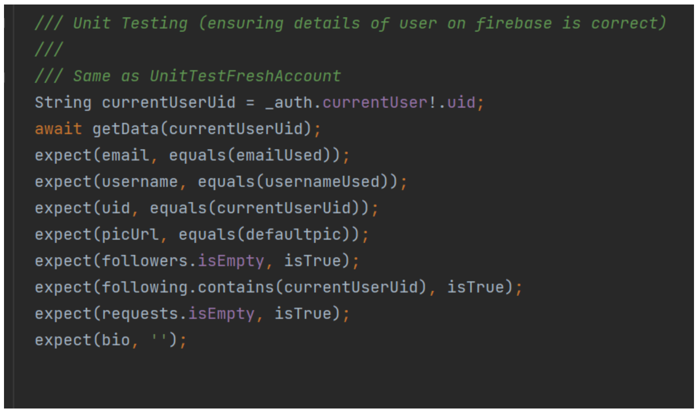
```
/// Widget Testing for profile page display
// We should have 1 additional post in our display + post count going up
final profilePostCard = find.byType(DiaryCard);
expect(profilePostCard, findsAtLeastNWidgets(1));

expect(find.text(afterPostCount.toString()), findsAtLeastNWidgets(1));
```

We also conduct a further check on the profile page where we should see a increment of the post number, this is double verified by doing both unit testing (pulling the after post, length of post number on firebase) and then checking it with the display counter.

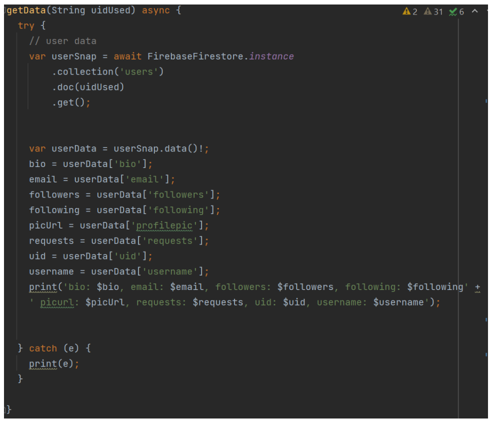
Unit Testing

Unit Testing comes as a form of testing for functionality. In a way, we are testing whether functions return consistent and correct outputs. For e.g, if you have a function like add which adds two numbers together, you will use a unit test to verify that it is supposed to add. Add(1, 1) should therefore return integer 2. Unit Test can also be applied for server-side information retrieval.

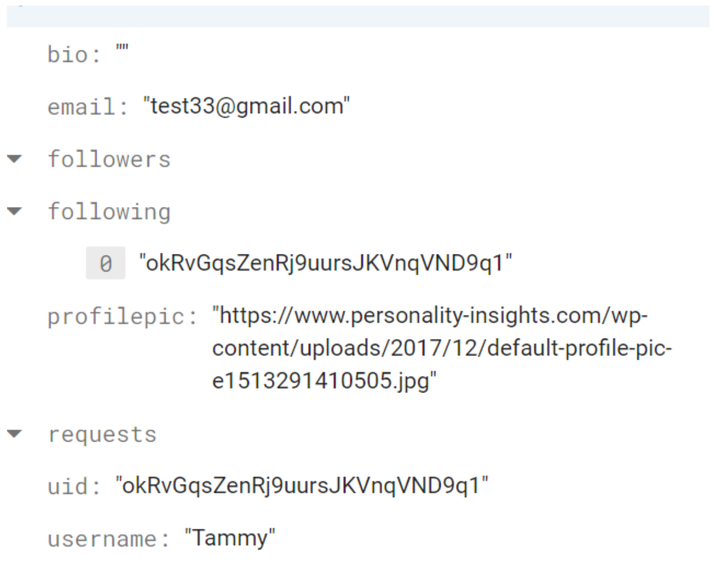
Verifying a new account's firebase information once created



```dart
/// Unit Testing (ensuring details of user on firebase is correct)
///
/// Same as UnitTestFreshAccount
String currentUserUid = _auth.currentUser!.uid;
await getData(currentUserUid);
expect(email, equals(emailUsed));
expect(username, equals(usernameUsed));
expect(uid, equals(currentUserUid));
expect(picUrl, equals(defaultpic));
expect(followers.isEmpty, isTrue);
expect(following.contains(currentUserUid), isTrue);
expect(requests.isEmpty, isTrue);
expect(bio, '');
```



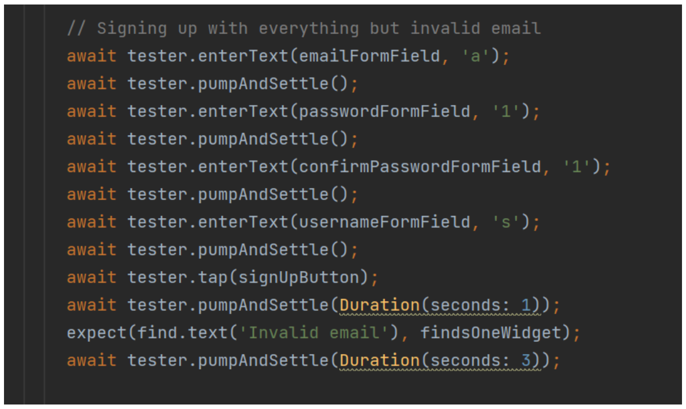
```dart
getData(String uidUsed) async {
  try {
    // user data
    var userSnap = await FirebaseFirestore.instance
        .collection('users')
        .doc(uidUsed)
        .get();


    var userData = userSnap.data()!;
    bio = userData['bio'];
    email = userData['email'];
    followers = userData['followers'];
    following = userData['following'];
    picUrl = userData['profilepic'];
    requests = userData['requests'];
    uid = userData['uid'];
    username = userData['username'];
    print('bio: $bio, email: $email, followers: $followers, following: $following' +
    ' picurl: $picUrl, requests: $requests, uid: $uid, username: $username');


  } catch (e) {
    print(e);
  }

}
```



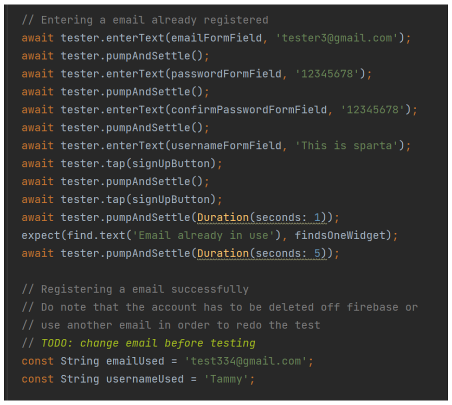
```
bio: ""
email: "test33@gmail.com"
▾ followers
▾ following
    0  "okRvGqsZenRj9uursJKVnqVND9q1"
profilepic: "https://www.personality-insights.com/wp-content/uploads/2017/12/default-profile-pic-e1513291410505.jpg"
▾ requests
uid: "okRvGqsZenRj9uursJKVnqVND9q1"
username: "Tammy"
```

Testing that the user's data is stored in Firebase and can be retrieved with the right details / information. In this case, we had the details we inputted to create the account (mainly **email, username)**. We then use an async function **getData()** to pull data from our firebase firestore to verify the data creation for the user. We check **all user fields** to be **exactly the same** as **any fresh account that is created.** In this case, **your followers list, request list to be empty, pic url set to the default, bio is an empty string, uid being ur user auth uid. For following, we have** default enabled self-following for testing and convenience purpose, this prevents the need to be swapping accounts to verify that you have posted. Thus it will contain a list containing ur own uid.

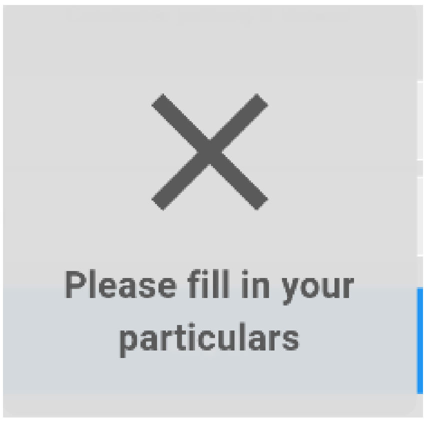
Wrong Inputs / Status Alert displays

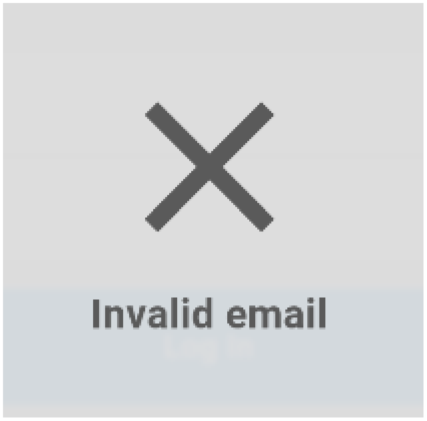


```
// Signing up with everything but invalid email
await tester.enterText(emailFormField, 'a');
await tester.pumpAndSettle();
await tester.enterText(passwordFormField, '1');
await tester.pumpAndSettle();
await tester.enterText(confirmPasswordFormField, '1');
await tester.pumpAndSettle();
await tester.enterText(usernameFormField, 's');
await tester.pumpAndSettle();
await tester.tap(signUpButton);
await tester.pumpAndSettle(Duration(seconds: 1));
expect(find.text('Invalid email'), findsOneWidget);
await tester.pumpAndSettle(Duration(seconds: 3));
```



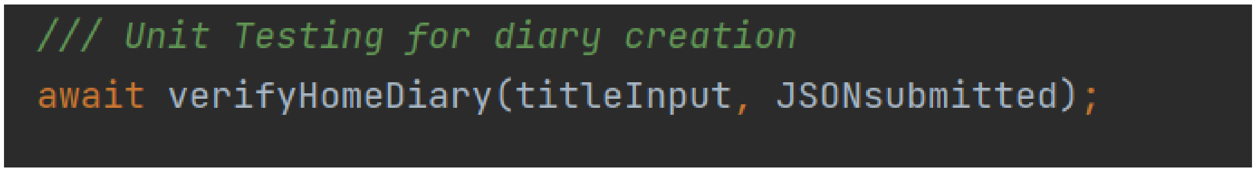
```
// Entering a email already registered
await tester.enterText(emailFormField, 'tester3@gmail.com');
await tester.pumpAndSettle();
await tester.enterText(passwordFormField, '12345678');
await tester.pumpAndSettle();
await tester.enterText(confirmPasswordFormField, '12345678');
await tester.pumpAndSettle();
await tester.enterText(usernameFormField, 'This is sparta');
await tester.tap(signUpButton);
await tester.pumpAndSettle();
await tester.tap(signUpButton);
await tester.pumpAndSettle(Duration(seconds: 1));
expect(find.text('Email already in use'), findsOneWidget);
await tester.pumpAndSettle(Duration(seconds: 5));

// Registering a email successfully
// Do note that the account has to be deleted off firebase or
// use another email in order to redo the test
// TODO: change email before testing
const String emailUsed = 'test334@gmail.com';
const String usernameUsed = 'Tammy';
```



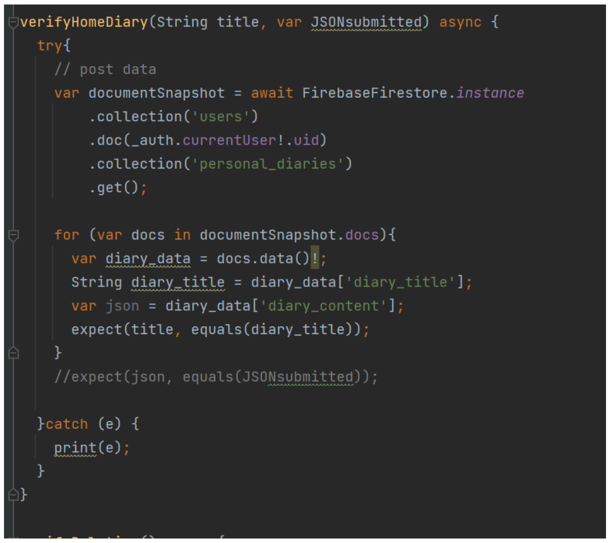


The above images are just some of the cases we worked on for the wrong inputs / different combinations of invalid inputs into the text fields. The cases for invalid inputs are too long to show in images, hence we recommend you to look at the source code. In essence, we test all different kinds of inputs for the register and login screen such as **leaving fields empty,** putting **wrong inputs** in the fields knowing they won't work (cases could include firebase requiring **invalid email, wrong password, fields are empty, email already used**). We then check for the right notifications to pop up

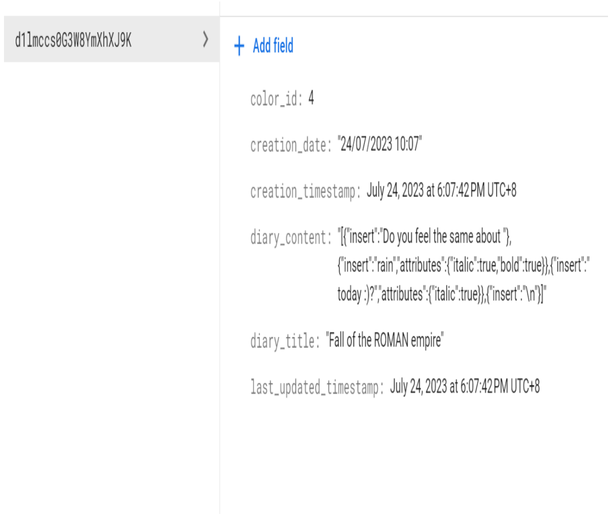
Verifying back-end creation of our diary



```
/// Unit Testing for diary creation
await verifyHomeDiary(titleInput, JSONsubmitted);
```



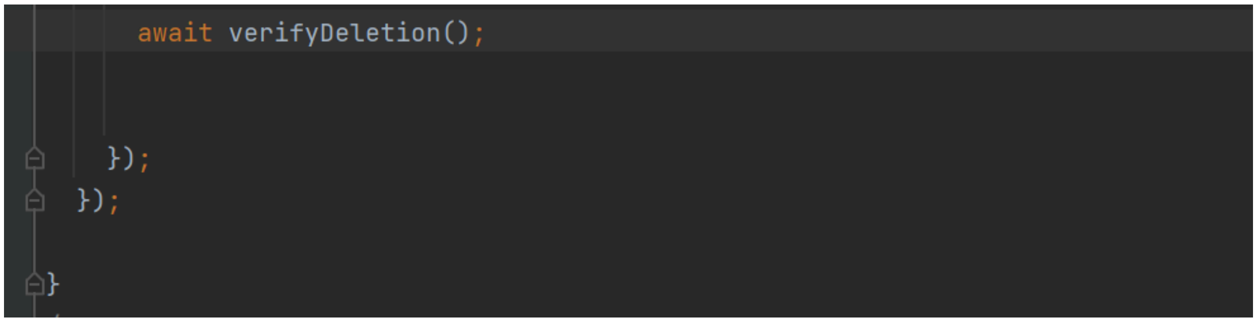
```dart
verifyHomeDiary(String title, var JSONsubmitted) async {
  try{
    // post data
    var documentSnapshot = await FirebaseFirestore.instance
        .collection('users')
        .doc(_auth.currentUser!.uid)
        .collection('personal_diaries')
        .get();

    for (var docs in documentSnapshot.docs){
      var diary_data = docs.data()!;
      String diary_title = diary_data['diary_title'];
      var json = diary_data['diary_content'];
      expect(title, equals(diary_title));
    }
    //expect(json, equals(JSONsubmitted));

  }catch (e) {
    print(e);
  }
}
```



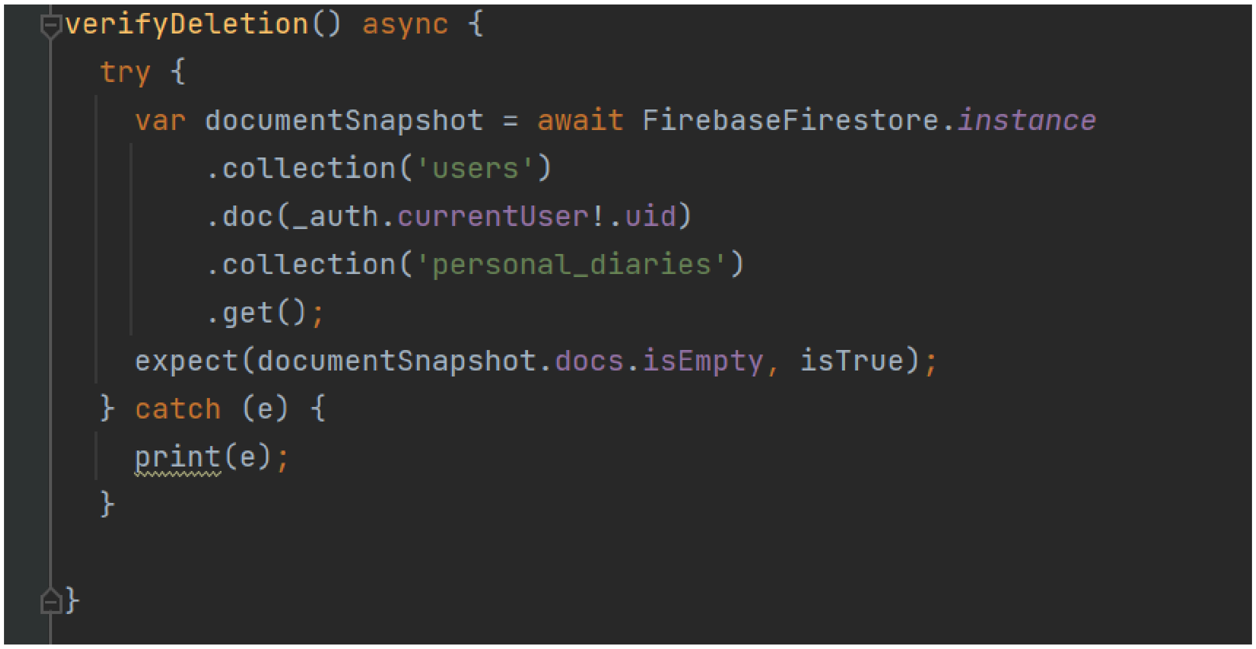
```
color_id: 4
creation_date: "24/07/2023 10:07"
creation_timestamp: July 24, 2023 at 6:07:42 PM UTC+8
diary_content: "[{"insert":"Do you feel the same about "},
               {"insert":"rain","attributes":{"italic":true,"bold":true}},{"insert":"
               today :)?","attributes":{"italic":true}},{"insert":"\n"}]"
diary_title: "Fall of the ROMAN empire"
last_updated_timestamp: July 24, 2023 at 6:07:42 PM UTC+8
```

In this unit test, we check for the creation of the diary on the back-end database. We are not checking the **JSON file in this case**, due to the equals function not encapsulating the entire JSON. Even though both json files print the same output. Hence we only check for the diary_title entry, which in itself verifies a creation of a diary back-end.

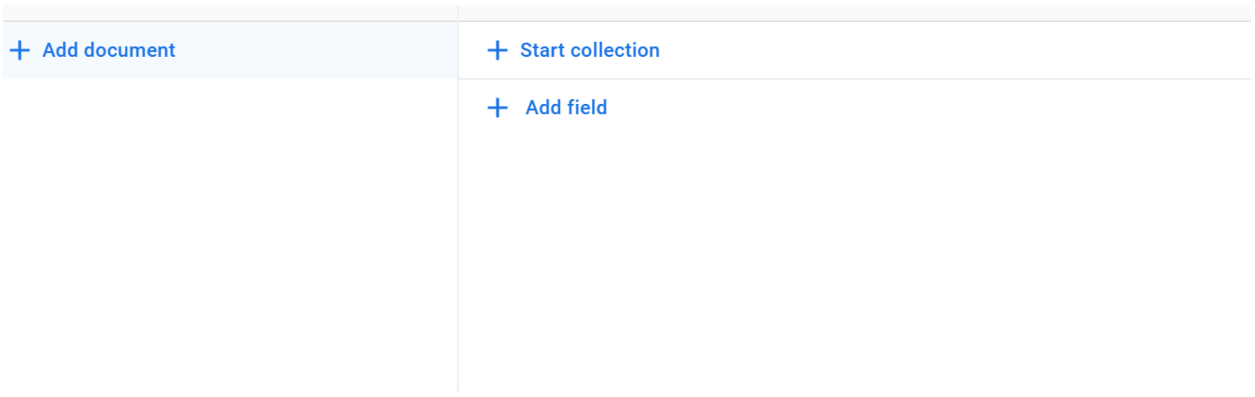
Verifying a diary's deletion



```dart
        await verifyDeletion();


      });
    });

}
```

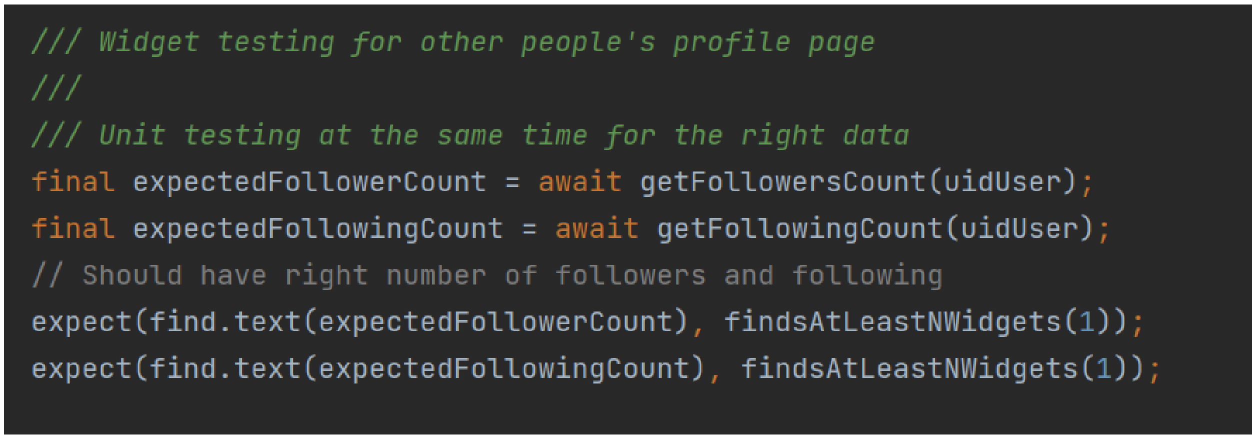


```
verifyDeletion() async {
  try {
    var documentSnapshot = await FirebaseFirestore.instance
        .collection('users')
        .doc(_auth.currentUser!.uid)
        .collection('personal_diaries')
        .get();
    expect(documentSnapshot.docs.isEmpty, isTrue);
  } catch (e) {
    print(e);
  }

}
```

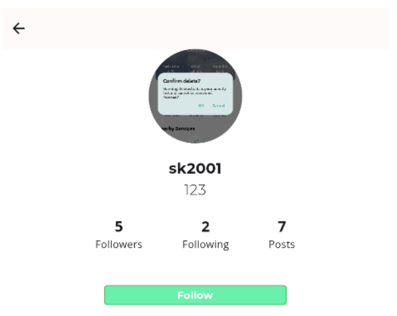


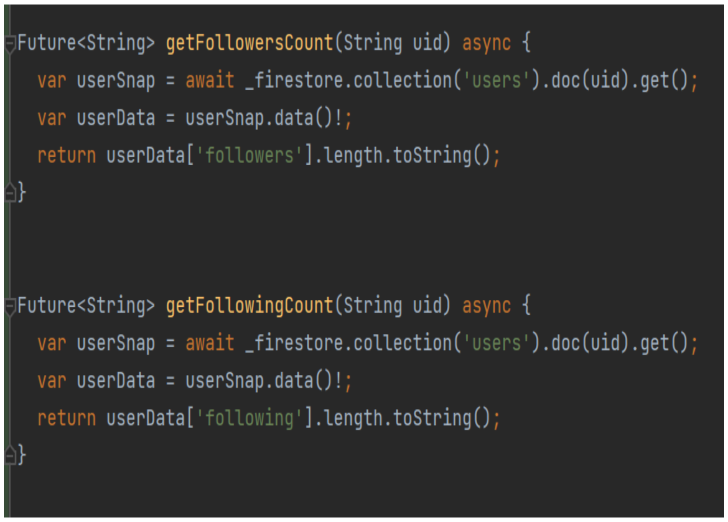
In this case, we are checking whether we are retrieving **a empty list of data** for all of the personal diaries. The verifyDeletion() function confirms this through expecting an empty list when retrieving the data.

Other’s Profile Page



```
/// Widget testing for other people's profile page
///
/// Unit testing at the same time for the right data
final expectedFollowerCount = await getFollowersCount(uidUser);
final expectedFollowingCount = await getFollowingCount(uidUser);
// Should have right number of followers and following
expect(find.text(expectedFollowerCount), findsAtLeastNWidgets(1));
expect(find.text(expectedFollowingCount), findsAtLeastNWidgets(1));
```

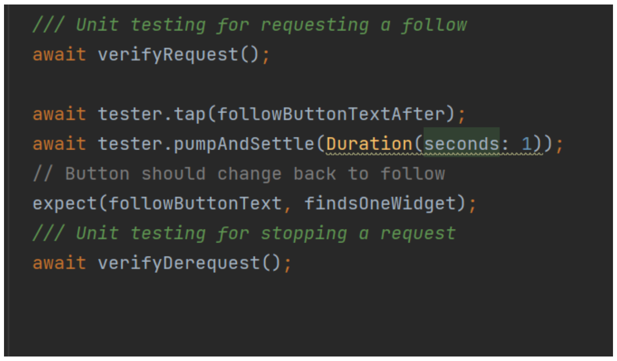




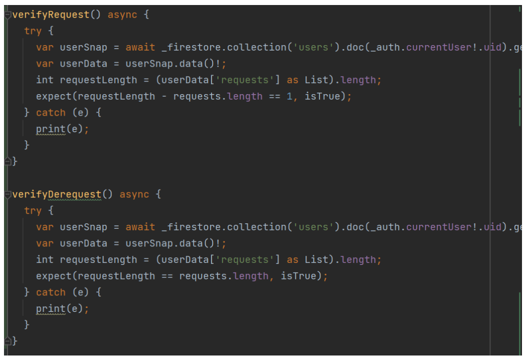
```
Future<String> getFollowersCount(String uid) async {
  var userSnap = await _firestore.collection('users').doc(uid).get();
  var userData = userSnap.data()!;
  return userData['followers'].length.toString();
}


Future<String> getFollowingCount(String uid) async {
  var userSnap = await _firestore.collection('users').doc(uid).get();
  var userData = userSnap.data()!;
  return userData['following'].length.toString();
}
```

We are **retrieving data** from the live server to **double check with what is being displayed.**

Follow / Request Send Buttons



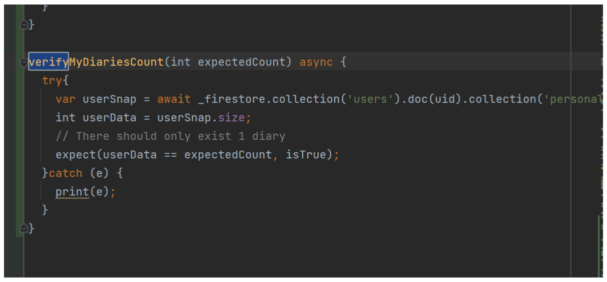
```
/// Unit testing for requesting a follow
await verifyRequest();

await tester.tap(followButtonTextAfter);
await tester.pumpAndSettle(Duration(seconds: 1));
// Button should change back to follow
expect(followButtonText, findsOneWidget);
/// Unit testing for stopping a request
await verifyDerequest();
```



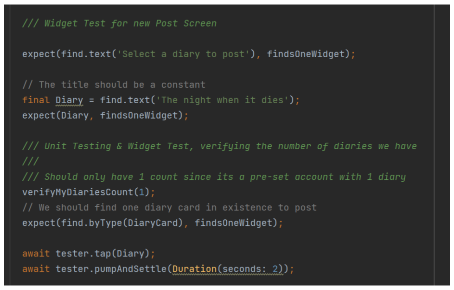
```
verifyRequest() async {
  try {
    var userSnap = await _firestore.collection('users').doc(_auth.currentUser!.uid).ge
    var userData = userSnap.data()!;
    int requestLength = (userData['requests'] as List).length;
    expect(requestLength - requests.length == 1, isTrue);
  } catch (e) {
    print(e);
  }
}

verifyDerequest() async {
  try {
    var userSnap = await _firestore.collection('users').doc(_auth.currentUser!.uid).ge
    var userData = userSnap.data()!;
    int requestLength = (userData['requests'] as List).length;
    expect(requestLength == requests.length, isTrue);
  } catch (e) {
    print(e);
  }
}
```

We are doing unit testing on the back-end database in order to check whether the request has been made or removed accordingly as it should be. In this case, we check for whether the new length of the request has increased by 1, and when we rescind our request, whether the length is the same.

Selecting Diary to post



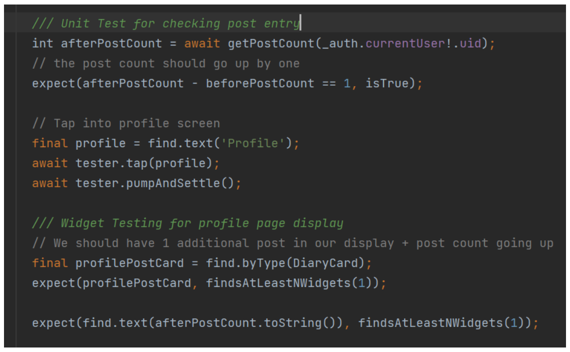
```
  }
}

verifyMyDiariesCount(int expectedCount) async {
  try{
    var userSnap = await _firestore.collection('users').doc(uid).collection('personal
    int userData = userSnap.size;
    // There should only exist 1 diary
    expect(userData == expectedCount, isTrue);
  }catch (e) {
    print(e);
  }
}
```



```
/// Widget Test for new Post Screen

expect(find.text('Select a diary to post'), findsOneWidget);

// The title should be a constant
final Diary = find.text('The night when it dies');
expect(Diary, findsOneWidget);

/// Unit Testing & Widget Test, verifying the number of diaries we have
///
/// Should only have 1 count since its a pre-set account with 1 diary
verifyMyDiariesCount(1);
// We should find one diary card in existence to post
expect(find.byType(DiaryCard), findsOneWidget);

await tester.tap(Diary);
await tester.pumpAndSettle(Duration(seconds: 2));
```

verifyMyDiariesCount just checks if the current number of diaries we have created in **My Diaries** matches what is shown in the screen to select the diaries we can post.

Verifying the increment in number of post



```
/// Unit Test for checking post entry
int afterPostCount = await getPostCount(_auth.currentUser!.uid);
// the post count should go up by one
expect(afterPostCount - beforePostCount == 1, isTrue);

// Tap into profile screen
final profile = find.text('Profile');
await tester.tap(profile);
await tester.pumpAndSettle();

/// Widget Testing for profile page display
// We should have 1 additional post in our display + post count going up
final profilePostCard = find.byType(DiaryCard);
expect(profilePostCard, findsAtLeastNWidgets(1));

expect(find.text(afterPostCount.toString()), findsAtLeastNWidgets(1));
```

We check for an increment in the post counter between operations through taking the old post count (what we pulled before we posted) and the after post count (what we pulled after our posting operation), and check that the difference is 1..

Testing (Integration Testing)

Integration Test mimics user behavior by clicking through the application, scrolling which recreates user interaction for a specific use case. This is crucial as it should test whether our application functions as **desired**,

Generally, as **Integration Test** require a substantial time to run, it should not be commonly used and written for every test case. Instead, we have choose to only test our **core features**. We have therefore written **these Integration cases.**

Video of Integration Test running:

https://drive.google.com/drive/folders/1FlahtAYKjNeeE4P7h3ZC7M6zFBfugomU?usp=sharing

Github issue for testing (Under Test_Milestone3_SEan)
https://github.com/gingerbreaf/Feelings_Overflow/blob/Test_Milestone3_SEan/integration_test/core_features_feelings_overflow.dart

We did testing on a separate branch as per our software engineering practices for git. With integration testing, we also add a lot of unnecessary code into the source code such as Keys in order to help us navigate and find widgets. Separate branching also ensures that we know the integration test is done on a certain version of the dev branch. This does not allow much re-usability but we prevent the tests from creating side effects in our source code.

These are all done automatically by code, no manual inputs were made.

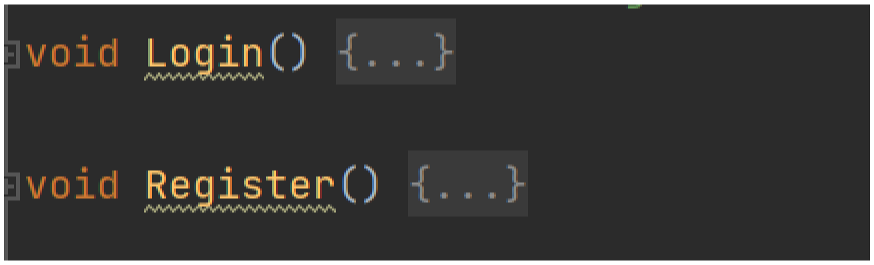
Integration Test

1. **Register**
2. **Login**
3. **Personal Diary, Save and Delete features in my_diaries page(test function: myDiariesFeature())**
4. **Posting of Diary feature in home page (test function: postingFeature(), snippetFeature())**
5. **Following and unfollowing of users in profile page (test function: followFeature())**

We believe that these features represent the core features we have built in our app. The mode of testing we have used is through **Flutter's inbuilt Integration Test** modules, allowing us to extensively test the presence of widgets and their values, input values automatically and test for different cases.

View the full testing code in the github link above or you can view the video!

Register Integration Test(function: Register()) & Login Integration Test(function: Login())

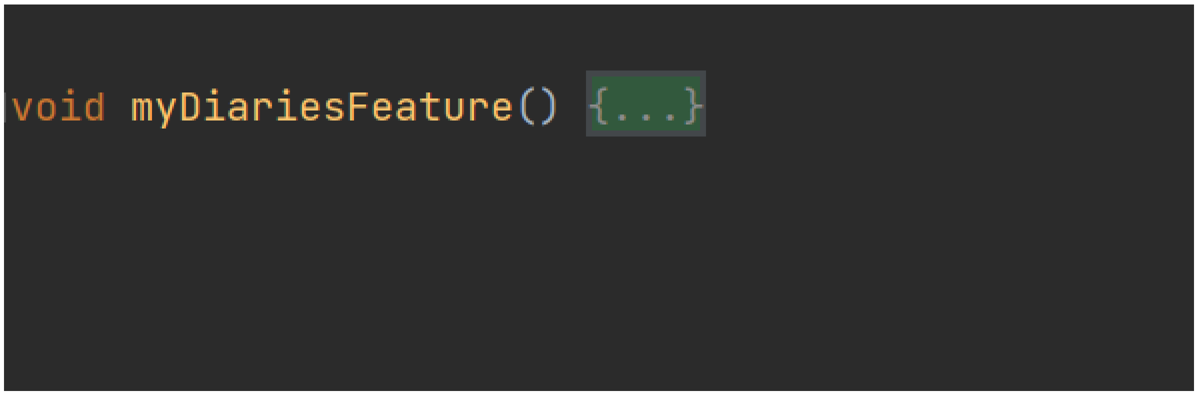


```
void Login() {...}

void Register() {...}
```

We run through the whole process of widget and unit testing as seen in **Widget Testing & Unit Testing**, automatically inputting fields in order test **response.** Through the video and the source code, you see that we used **integration testing** to ensure the correct functionality and display simultaneously. **For example, we conducted the widget test on the whole Register page & Login page, before inputting into the fields and then conducting unit testing** as well as **checking** the right navigation and responses as a whole for the app. Using the automated testing, we have checked extensively for wrong inputs by the users, probing for any deviation from expectations.

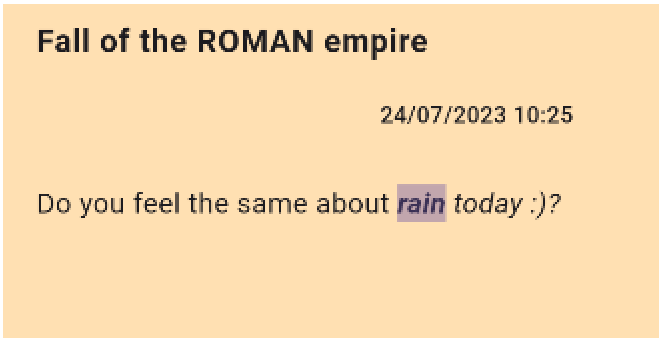
Do note: Register() or the registration Integration Test requires the emailUsed field to be changed after every use. That is because we do not have a function to delete the account both on firebase auth and firestore together.

My Diaries Integration Test



```
void myDiariesFeature() {...}
```

For the My Diaries feature, we test the functionality of being able to **create, edit and delete diaries**. Firstly, the test runs through logging in with the right credential, then navigation to the **My Diaries** tab through the bottom navigation bar. The tester then does a widget test for the My Diaries page in order to ensure we are on the right page. This is seen in the **widget testing for My Diaries screen**.

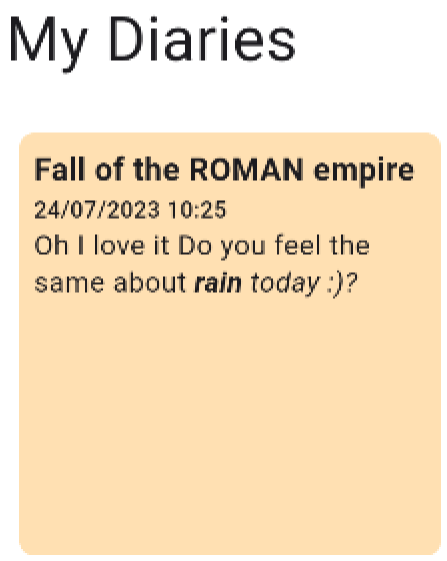


Fall of the ROMAN empire

24/07/2023 10:25

Do you feel the same about ***rain*** *today :)?*

We then create a diary through locating the add diary button, navigating into the **diary creation screen**.
In this case, unlike our Milestone 2 diary, we use a rich text widget instead, this thus allows us to input words and format them in a particular way. In this case, we mimic the selection of different words to be *italic* and **bold,** before proceeding to create the diary. We then used **unit testing** via Firebase data retrieval as seen in Unit Testing **Verifying backend creation of Diary,** This ensures we double confirm the creation on our back-end.



My Diaries

Fall of the ROMAN empire

24/07/2023 10:25

Oh I love it Do you feel the same about ***rain*** *today :)?*

We then also look for the creation of a Diary Card at the My Diaries Tab now, verifying the creation of a diary. We then look to **edit** the diary, clicking on the diary card and adding words to the Diary before clicking the back button, showing that the diary can be edited and saved.

We then proceed to click on the diary again, go into the **Diary Editing** screen, locate and tap on the delete button. This creates a pop up which allows us to confirm the deletion of the diary.

After that, we finish up by checking that the back-end no longer possess any data of the diary that was deleted. We see this in **Unit Testing Verify a diary's deletion**.

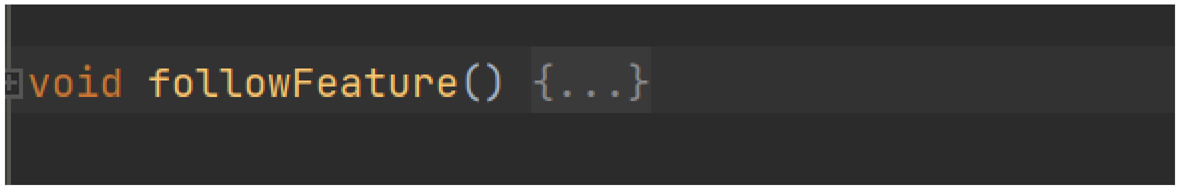
Side note: The testing was extremely challenging since we did not use flutter in-house Text Fields, or Text Widgets. This means the integration test methods like enterText did not work for the rich-text editor / controller we are using. There was a need to amend the libraries original file to allow Keys to be attached to editor and controller. There was also a need to code out the creation of the words, in a delta format and format the selection of the Quill Controller.

At this point, we have checked for both the front-end and back-end functionality

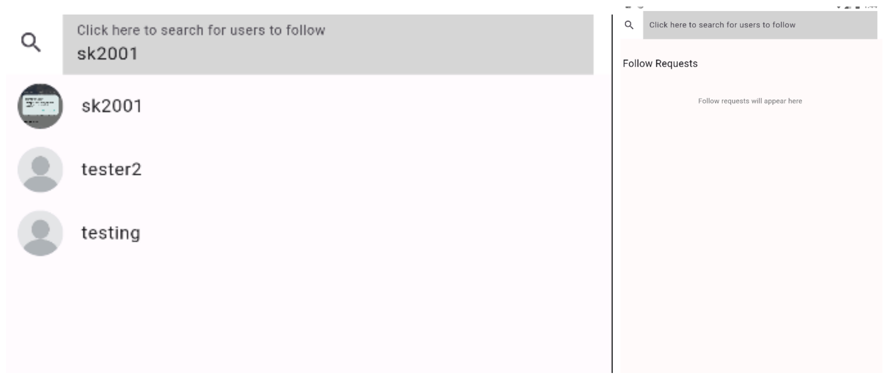
1. **Login Screen (Authentication)**
2. **My Diaries (UI & Back-End)**
3. **Custom Diary Card Widget (UI)**
4. **Diary Creation Screen (UI & Back-End functions)**
5. **Diary Editing Screen (UI & Back-End functions)**

Thus, if the test succeeds, it guarantees the functionality of creation, editing function and deletion of **diaries** on the **my_diaries** tab as well as **back-ends** update relating to diary creation and deletion

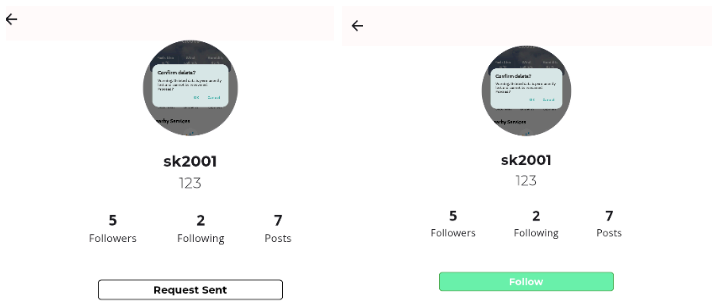
Following Feature / Requesting



```
void followFeature() {...}
```


Click here to search for users to follow
sk2001
sk2001
tester2
testing
Click here to search for users to follow
Follow Requests
Follow requests will appear here

In the followFeature() integration test, we test the functionality of the search tab, effectively searching, finding a user, sending a request and unsending that particular request. We also do widget testing and unit testing along to determine the accurate widgets and numbers to display (**Other people's profile page, Follow / Request send buttons** for both **Unit Test and Widget Test, additionally Search Bar widget test)**. The test goes as follows, we login, head into the search tab. We proceed with a widget test for the search tab given we have no followers. We then proceed to search and scan the search bar for the right searches. Then we click into the profile, to do another unit test and widget test to determine whether the UI is in sync with the back-end server.



We then request to follow and re-check the request validity on our back-end before rescinding the request. All of these were done automatically, and ensured a consistent result, that is everytime you were to run the test, the results were consistent and the test case should pass

At this point, we have checked for both the front-end and back-end functionality

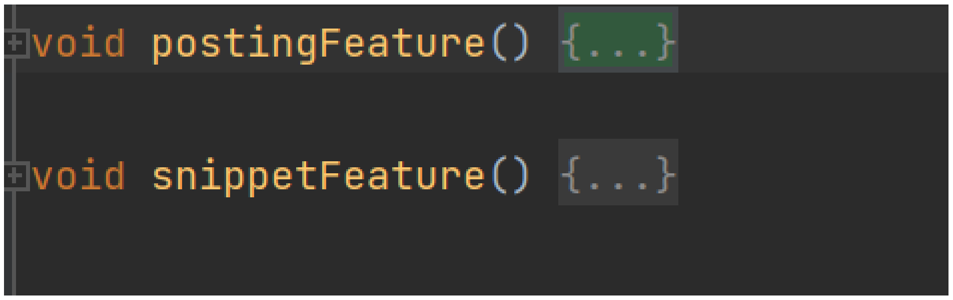
1. **Login Screen (Authentication)**
2. **Search Tab (UI)**

3. **Search bar (UI & Back-end)**
4. **Other_user_profile_screen (UI & Back-end)**.

Posting Diary / Snippet feature

Side note: while we have created integration test videos for both features, predominantly in terms of widget testing and unit testing, the test are identical hence in widget testing, you will only see it for posting normal whole diaries, however it applies to snippets feature as well.

Side note 2: The code was written such that I followed myself and thus was able to see my own post, this was done due to limitations of several factors. The limitations include the fact that the Integration Test is only able to simulate the actions of one device, this means i cannot have two concurrent emulators running.This prevents the need for toggling between accounts as well to verify posting This means that to ensure simplicity and ease of testing.



```
void postingFeature() {...}

void snippetFeature() {...}
```

Lastly, for postingFeature & snippetFeature, we test the **functionality** of posting a pre-existing diary, via two routes, the posting route or the snippet route. The difference being that posting just involves posting the **entire diary**, while snippet allows us to **style and format a certain word selection**, creating a **postcard** format instead. We login into the app, similar to other integration tests, using a pre-created diary, we test the different screens for **their widgets** (Selecting diary screen, Home page after posting, profile page) as well as **unit testing** (the number of diaries we can select from, verifying the increment in post count). With this, we extensively **verified** a successful post from both back-end and front-end

At this point, we have checked for both the front-end and back-end functionality

1. **Login Screen (Authentication)**

2. **Home_tab (UI & Back-End)**
3. **Custom Home page Diary Card Widget (UI)**
4. **Custom Post Card Widget (UI)**
5. **Diary selection screen(UI & Back-End)**
6. **Snip_screen (UI & Back-End)**
7. **Diary_posting_screen (UI & Back-End)**
8. **Profile page (Back-end)**

User Acceptance Test

In order to show evidence of the user acceptance test, we decided to use google forms to document them. The forms and their results are posted below (forms are not shared as collaborators to prevent edits, instead the responses are shown in a google sheet)

Things to note: We didn't set a name field to enter into the google form initially, so the responses are manually entered into the google sheet, as seen here. We edited it so some forms do contain name inputs

Structure of user acceptance test: The **Google Form** breaks into a **structured way** for a user to approach the app for the first time. The questions flow in a manner such that the core functions and features interact in a way they are supposed to (for e.g in order to post, you first need to create a diary). The questions are generally yes or no answers (others to indicate suggestions or errors) to whether they are able to perform a functionality of a core feature. Then we generally ask them to rate the UI / UX aspect at the end of it. This was done for **most of our core features.** Generally, we also try to ask about intuitiveness as well.

User Acceptance Google Form

Google sheet result

Result Analysis & Addressing concerns: Functionality wise, most of the users are able to follow the word guide in the google form and put that they were able to use different functions of our core features. The majority of answers was yes, however there were some answers that included others which really helped us identify certain bugs and errors in our code. Due to constant changes to the code, we often miss out certain components that might not function as expected. In this case, a response from George and Jun wen, identified that upon clicking the search bar, there was a pop up that they felt was not very intentional or deserving. This was originally done for catching errors however it also happened when there was no active follow request from other users, thus throwing the same error. This was indeed not very intuitive and we added a **null check** to address this. This error was thus addressed in the later builds. Additionally, George further identifies that the follow button UI was not working very well, and upon changing screen, the request sent button was changed back to the follow button. This turned out to be a back-end method error as one of our firebase methods was throwing a **null error,** however, this was due to the account being a **legacy** account that was created previously and did not have such a user field on firebase. This has also allowed us to simplify the logic in the backend method (used to only have requestFollow encapsulating multiple cases -> split into requestFollow and stopFollow). This prevents the methods from having side-effects that cause other back-end changes other than what was intended. Jun wen did point out that he was unable to change the font after the post however, this is not an intended function of our app hence it will not be addressed.

In terms of UI / UX, the feedback generally was a **4 out 5** rating where there was a belief that there could be more intuitiveness and generally more guidance in leading the user, it was believed that aesthetically it might not be the best form to represent certain aspects of the diary. This included **Profile page,** and **overall UI,** with **half of the responses** indicating a 4 out of 5 satisfaction. We however decided not to intervene, since the **UI** actually went through multiple iterations of changes, and we were intending to keep to a minimalistic feel inspired by instagram.

After **User Testing,** we have thus addressed the concern as mentioned before, and got the participant, which in this case was George, to use our updated build and proceed to do a google form which highlighted if everything has been addressed.
For the search bar intuitiveness that he mentions, we fix it in a further later build, highlighting the search bar grey and including the hint words 'Click me to search here'

User Acceptance response to concerns form

Google sheet result(Addressed concerns)

Generally, our user acceptance testing was sent to both computing and non-computing friends and we understand that our numbers are limited but it has indeed help us to improve and correct our functions

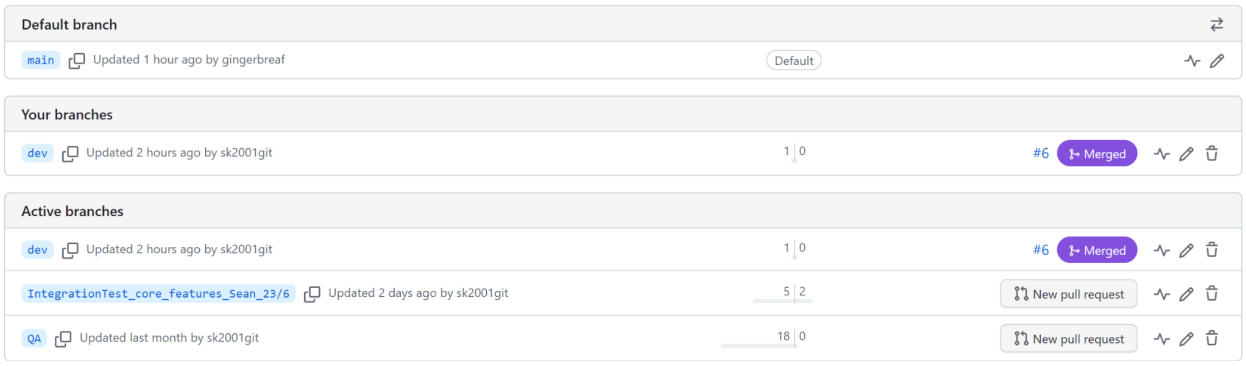
Documentation

We documented the methods that are called during build, with brief summaries of what they do and how they work. Variables in each file are also documented with their purpose. The build functions are not documented since they are simply used by flutter to build the screen.

Management

Git and github features such as add, commit, push and branching were used during development. We mostly worked on the dev branch and branched off dev to implement features on a WIP branch before merging into dev. Testing was also done in a separate testing branch. After sufficient testing, we then decided to merge dev into the main branch. However, we did not realize from the start that we are not supposed to git track files generated by the app, and used git add . to track all files. Hence we were unable to merge dev branch into main branch due to numerous merge conflicts from the cache files and decided to overwrite main branch instead.

Git pull request was also extensively used, in order to merge code from WIP branches into the dev branch. The main branch is only to be merged after extensive testing.



Default branch

main	Updated 1 hour ago by gingerbreaf		Default

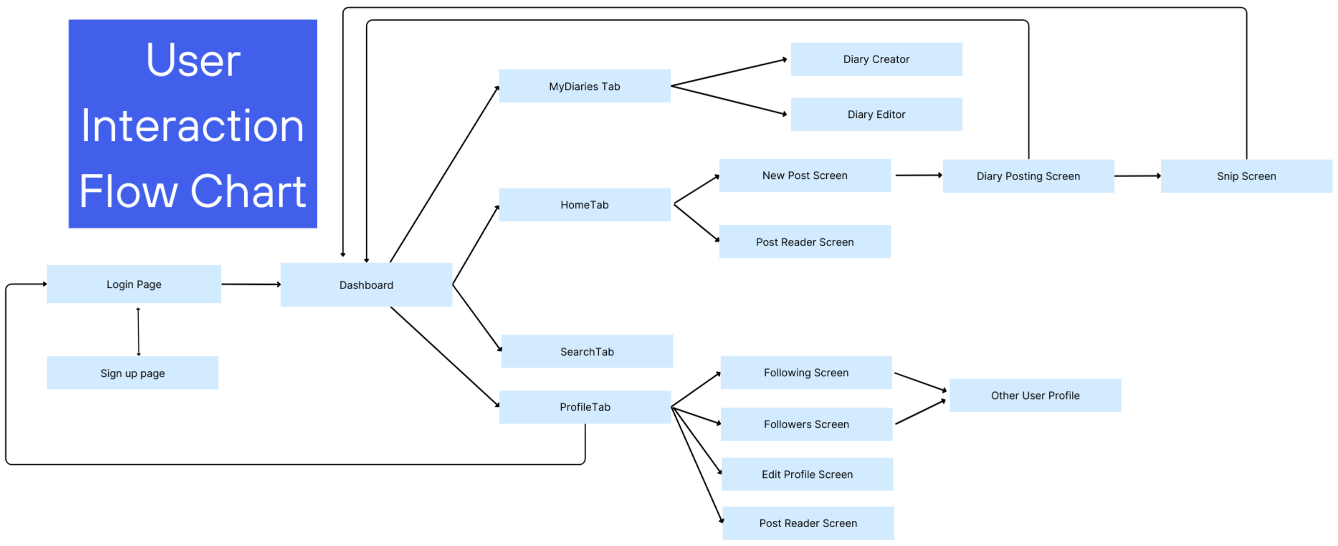
Your branches

dev	Updated 2 hours ago by sk2001git	1 | 0	#6	Merged

Active branches

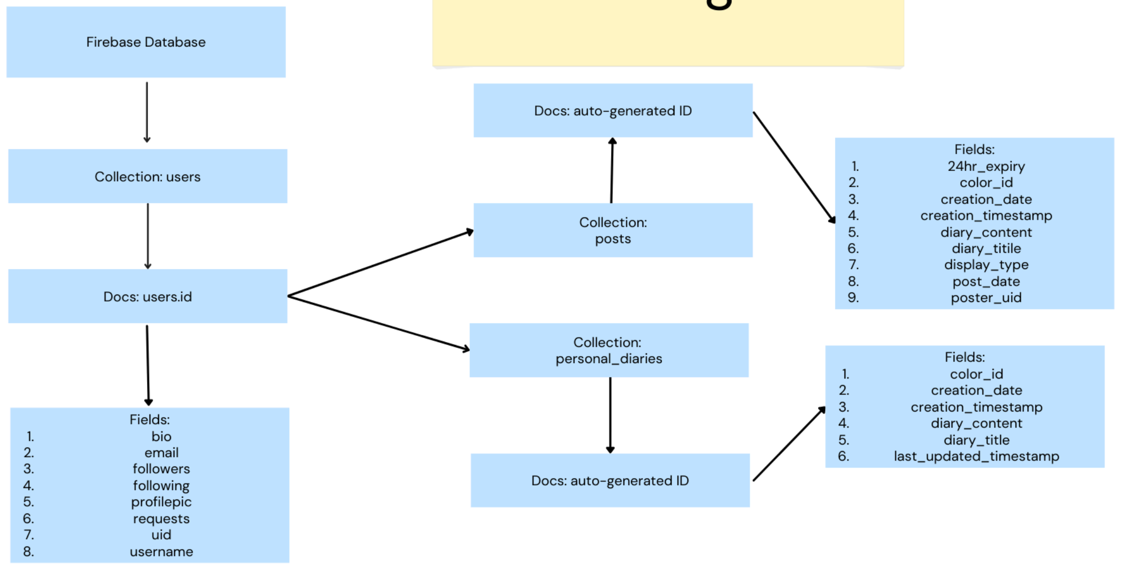
dev	Updated 2 hours ago by sk2001git	1 | 0	#6	Merged
IntegrationTest_core_features_Sean_23/6	Updated 2 days ago by sk2001git	5 | 2		New pull request
QA	Updated last month by sk2001git	18 | 0		New pull request

Diagrams



User Interaction Flow Chart
Login Page
Sign up page
Dashboard
MyDiaries Tab
Diary Creator
Diary Editor
HomeTab
New Post Screen
Diary Posting Screen
Snip Screen
Post Reader Screen
SearchTab
ProfileTab
Following Screen
Followers Screen
Other User Profile
Edit Profile Screen
Post Reader Screen

Data Management



Test our app here: [Feelings Overflow MS3.apk]
(apk file, android phone required)

Github repository link: https://github.com/gingerbreaf/Feelings_Overflow

Video link: MS3 demo.mp4

Poster link: StackUnderflow MS3 Poster.png