Mobile Application

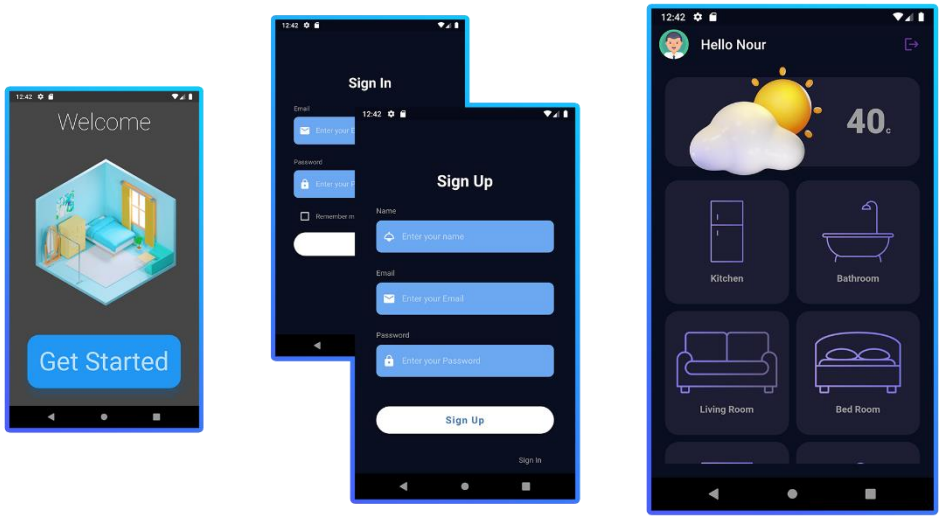

Figure 1. ####.1

The mobile application is created to ease the use of the functionalities that the smart home is providing for the user, as well as using those functionalities remotely, either from the house or outside.

We considered different technologies to implement this mobile application, and mention a few (Unity 3d, Flutter, React Native, Xamarin, Switfic, etc.), and we finally choose flutter.

Flutter is the one that balanced all the important parameters of creating an optimized, library-rich and easy to use framework, it's a very popular framework, and therefore there's a lot of guides online if one got stuck trying to solve a problem, as well as being compatible with most APIs like firebase.

The application screens are the following:

- Welcome Screens:
- Welcome to the app screen.
- Tips and hints.
- Authentication Screens:
- Login Screen.
- Sign up screen.
- Home screen.
- User Settings Screen.
- Rooms.

Chapter ##: Mobile Application:

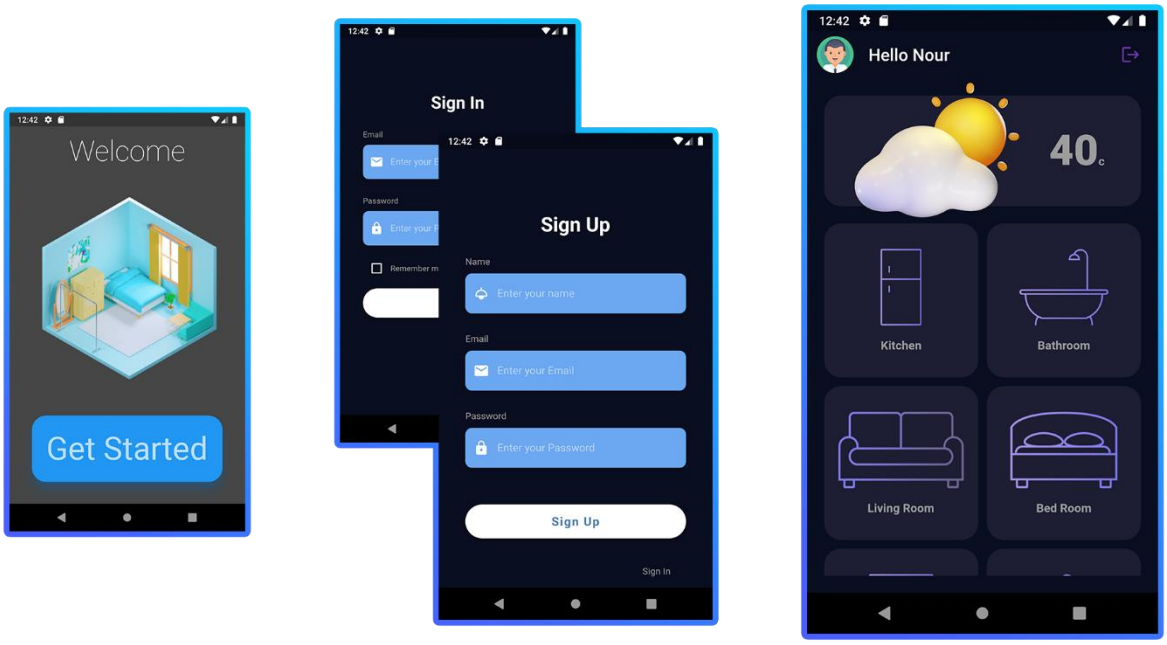

Figure ##.1

##.1 Introduction:

The mobile application is created to ease the use of the functionalities that the smart home is providing for the user, as well as using those functionalities remotely, either from the house or outside.

Part of the idea behind upgrading your house to a smart house is to:

- Monitor your house's sensors (temperature, light levels, etc.).
- Control your house's usual functionalities (lamps, fans, doors, etc.).
- Add smart functionalities (communicating with your house vocally, using face recognition to open doors, giving orders remotely).

##.1.1 Motivation:

The importance of creating a mobile application is to use the smart house remotely, either from outside the house (because it uses public internet not a local network) or from inside the house itself; furthermore, the mobile application could be used in the future (As future improvements) to add more custom smart features, like adding a pattern for switching on/off lights automatically, or increasing/decreasing the temperature automatically to balance the temperature inside the house to a specific temperature.

The ability to control the house remotely is fundamental for the smart house; since the smart house is supposed to increase the welfare of its users, and since it's common for devices like the TV, or the air conditioner to be remotely controlled instead of having to do an additional effort to control it by moving physically to the device to control it, therefore, it was a must to have a mobile application to remotely control the house's features.

Another future features that could be added is a permission system which could allow users based on some parameters to control some functionalities and other users to control all and decide permissions for other users, and you can see how this could be useful for parents, so parents can lock the doors, and stop their kids from accessing the functionality of unlocking the door, or some users won't be allowed to use the windows locks, or change the temperature of the house above or below a certain degree, and so on, all those functionalities will be very difficult to tune if there's no mobile application.

##.1.2 Functionalities:

Even though there's a lot to be done, we prioritized the most imported and the most usable functionalities in a smart house, which are the ability to:

- Access different rooms, each of which have different functionalities within.
- Turn on/off lights.
- Turn on/off Fans.

- Turn on/off Windows.
- Turn on/off Heater.
- Authentication:
    - User Customization:
        - User name.
        - User avatar.
        - Application Color.
    - User Sign in.
    - User Sign up.
- Monitor Temperature.

##.2 Technology:

In order for us to implement this application we need to find a framework that provides optimized applications as well as providing easy to use tools for us as developers so that we can implement features quickly.

One other criterion is the popularity of the framework; the more popular the framework, the easier it will be to debug, since there are lots of people using it, and therefore when a problem comes up, the chances are that someone else has had the same bug/problem before and therefore, you can use this information to solve yours.

Also the more popular the framework is the more likely you will find tutorials and documentation of the tool which guide you on how to use it.

Another factor is that the more popular the framework the more likely you will find old code that implement common features, for example, there are common transactions in video editing software, you can find those transactions online already made, so you can use them without having to recreate them all from scratch.

And the last benefit is that popular frameworks always give updates that provide tools that are trending, for example, if you're using a popular game engine, once the VR technology is common, they will make updates that equip you with the tools needed to port your game to this new medium.

##.2.1 Technology comparison:

There are many frameworks that we have considered some of them are not even made specifically for creating a mobile application like Unity 3d, which is originally made as a game engine, but in the following pages we will go through each one of those technologies and consider the pros and cons of them.

##.2.1.1 Flutter:

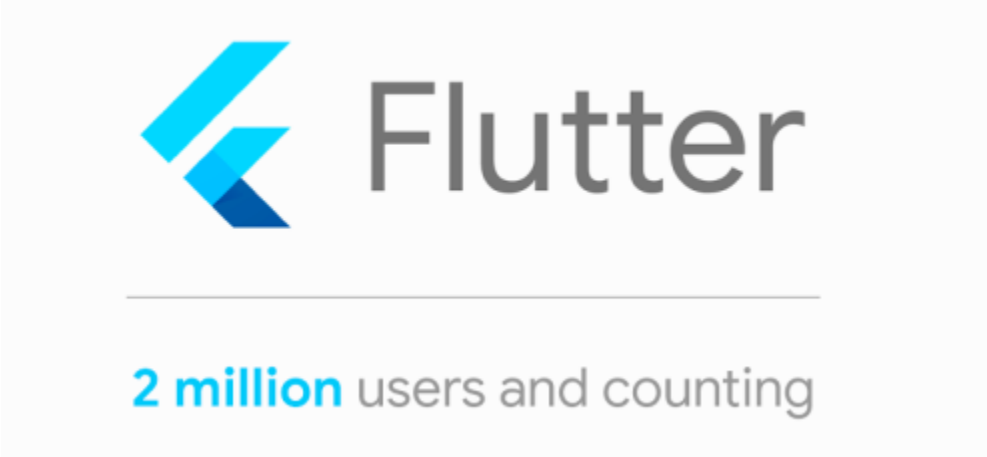

Figure ##.2.1.1.1

Flutter is a multi-platform development kit, developed by Google, and could be deployed/built for Android, IOS, web and Desktop (currently in beta) applications.

Flutter is unique duo to the fact that it gives the same programming tools C/C++ offers, which give access to low level abstractions, however it uses “Dart”, a programming language which offers high level abstractions which make using flutter accessible to beginners at the same time it offers advanced tools to its optimize code with.

Flutter offers accessible tools as well as low level abstractions by which a developer can optimize his/her code.

Flutter offers a set of high quality material design widgets, which give a professional look to the app, without having advanced skills in creating UI, but also makes it easy to develop your own custom UI designs.

Flutter offers Testing APIs and debugging tools.

It supports all modern technologies which is always added duo to the fact that this tool is developed by Google.

Dart is a beginner-friendly programming language that could be learned easily, and supports OOP and its implications, however, it also gives some low-level access to the resources of the machine.

It's supported on different IDEs, which gives each developer the ability to use his/her own environment to work with.

It's extremely popular now, and getting more popular, and this large community makes it easier to find help with common issues, and also makes it quick to improve a tool.

##.2.1.2 React Native:

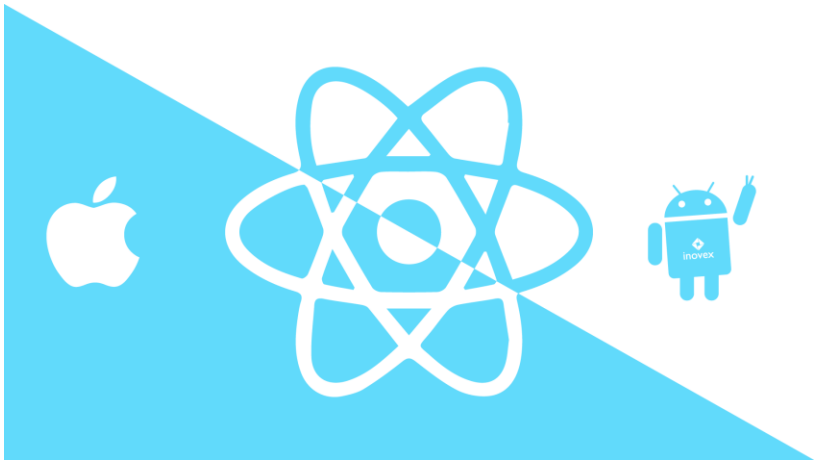

Figure ##.2.1.2.1

React Native is an open-source JavaScript framework, developed by Facebook which is designed to make build for multiple platform from the same source code, platforms like IOS, Android and web.

React is not the same as React Native, React came first is 2011 as an internal solution for Facebook framework, which was necessary to release newer versions with newer features and better, it was a solution for scalability problems, however in 2012 when Facebook acquired

Instagram, and now it was necessary for them to develop a multi-platform framework, they created React Native, which allows them to build for mobile as well as web which was already supported by React.

React Native uses the concept of “bridge”, which allows for asynchronous communication between the JavaScript and Native elements, the bridge concept lies at the very heart of React Native’s flexibility. Native and JavaScript elements are completely different technologies, but they are able to communicate.

This type of architecture offers the benefit of using a lot of OS-native features, but also comes with important challenges; e.g. constant use of bridges inside the app may significantly slow down its performance. If you’re building an app that involves many events, a lot of data, etc. React Native might not be the best option. More on that below.

Code reusability & faster development: The effective development for multiple platforms at once is the biggest and strongest advantage of React Native. Utilizing the same code base for different platforms carries other benefits: faster development and time-to-market of your app, easier and cheaper maintenance (you take care of one not multiple code bases), and a smoother onboarding process for new developers joining the project.

Thanks to the "hot reloading" feature, React Native enables developers to see the changes in the code in a live preview, without the need to refresh anything. This seemingly small tweak can actually tangibly improve the development process as it provides real-time feedback to anything that has been altered inside the code.

Performance: Compared to other cross-platform development solutions, React Native’s “bridge” concept can be seen as revolutionary. Since React Native apps allow usage of natively written code it is not as laggy as web-based cross-platform solutions. The official claim is that React Native gives “native-like” performance, but it is not necessarily true, the best way to put it is that it gives “near-native” experience.

Cost efficiency: Cost efficiency is the heart and the very reason for cross-platform development. Thanks to reusing code on multiple platforms, you usually need a smaller team to deliver the project. As opposed to native development where you need two separate teams to deliver basically two similar operating products instead of one.

Growing developer community: React Native is an open source framework, and as of now its community is thriving and constantly expanding. We can't forget the involvement of Facebook, as they are constantly working on improvements and elements widening the framework. What this means for you is that even if you encounter a problem that has not yet been solved in React Native you might find a bunch of people eager to help you out as they are concerned with making the framework more comprehensive and stable.

A relatively young technology (but improving with time): React Native is still relatively new, and as we mentioned earlier, definitely has some limitations, glitches, and issues needing to be addressed. Some custom modules do not exist in the framework, which means developers might need more time to build and create their own new ones from scratch. Your partner company or your developers should let you know about this during the app estimation process.

The need for native mobile developers: The strongest asset of React Native - implementing native code for better performance means that at times React Native developers might find themselves in need of help from native mobile app developers. Same goes for publishing the app in the App-Store and Google Play Store. Typically, native mobile developers are more familiar with the procedure and necessary documentation for a successful launch.

This might not be an issue if you're cooperating with an agency, where there are already native mobile developers that could give the React Native team a hand. But it is definitely something to consider when you're working solely with your own cross-platform team.

React Native does not go well with complex designs & interactions: The React Native's performance pales when confronted with complicated UI design decisions, complex animations, and heavy interactions. Once again, this is because of the bridge concept - all native modules have to communicate with the JavaScript part of the app, and too many of such interactions may slow down the app significantly, making it laggy and simply giving a bad experience.

##.2.1.3 Unity 3d:

Figure ##.2.1.3.1

Unity is a game engine designed to be used as a framework to create games for all platforms, including IOS and Android, however, people have found uses for it outside of this tight domain, for example, it's used to make physics simulations, it's also used to make architectural showcasing, and the idea of using it for this project was to that the user will be in a 3d environment and can navigate the home instead of using icons to represent the features.

Unity creates 3d environments which is mainly used in videogame development, the engine is written in C++, but it allows for the user three different methods to implement functionality and those methods are:

- C# Programming Language.
- Java Programming Language.
- Visual Scripting.

The reason why we considered unity was that it would give a realistic representation of the house, so the user can actually see your house in your phone, and therefore, you can see all the appliances, the lights, the doors, etc.

Unity however isn't supposed to be used as a mobile application develop, it runs lots of background processes, like physics engine, 3d rendering mechanisms, etc., therefore, it's not optimized for this kind of usage, on the other end it's realistic and good-looking.

##.2.1.4 Comparison:

	React Native	Flutter	Unity
Created	**2012**	**May 2017**	**2005**
Created by	**Facebook**	**Google**	**Unity**
Designed for	**Mobile applications**	**Mobile applications**	
App Graphics	**2D**	**2D**	**2D-3D**
Platforms	**Android – IOS – Windows – Mac – Linux.**	**Android – IOS – Windows – Mac – Linux.**	**Windows – Android – IOS – Mac – Linux – PlayStation – Xbox – Wii.**
Popularity	**High**	**Very High**	**Very High**

Machine Resources Required	**Low**	**Low**	**Very High**
Ad-on Quality	**High**	**Very High**	**High**
Online Guides Availability	**High**	**Very High**	**Very High**
Built-In Tools	**Many**	**Many**	**Many**

Table ##.2.1.4.1

##.2.1.5 Conclusion:

We have chosen flutter.

Flutter balances all the elements which we have discussed above, it's a very popular platform therefore, there's lots of open source tools which could be used to advance the framework, as well as compatibility with APIs and closed-source tools, it's very popular online, and therefore, there's a lot of tutorials which one can use as a guide to implement popular and basic features, it's also optimized and its builds work on most phones easily and smoothly.

##.3 Application Overview:

The application is composed from the following screens:

- Welcome screens.
- Authentication Screens.
- Home Screen.
- Rooms.
- User settings.

Welcome Screens:

Welcome screen is a couple of screens which gives the user an overview of the application, what it does, and how he/she can generally us it.

1) Getting Started: Controlling and monitoring your house is the purpose of the application.
2) Control Remotely: You can control any device in your house at your fingertips, just open your phone and use the app.
3) Security comes first: to make sure you're secure you create an account and use it to control your house.

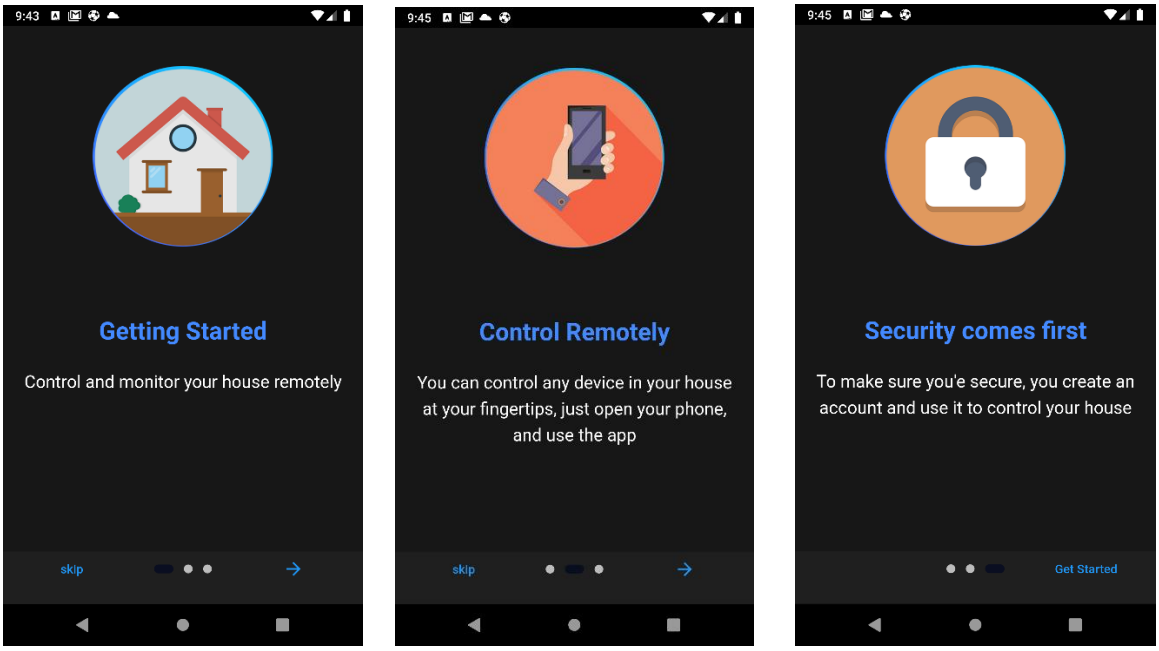

Figure ##.3.1: Intro Screens.

Authentication Screens:

1) Log-in Screen:

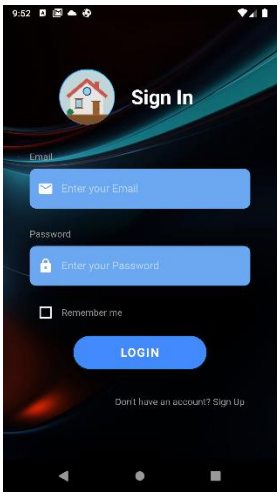

Figure ##.3.2: Log-In Screen.

Log-in screen is the screen that the user uses an account that he/she created in order for them to use it in the app, first the user enters his/her email, and the password, and then either:

- Checks the remember me checkbox and then logs in.
- Or logs-in immediately.

There are a set of error messages based on the type of error the user done:

1) Email is invalid, when the email text doesn't resemble an email
2) No User Found for that email: when the email isn't registered.
3) Password Must be at least 8 characters: when the password is too short.
4) Connection lost, etc.

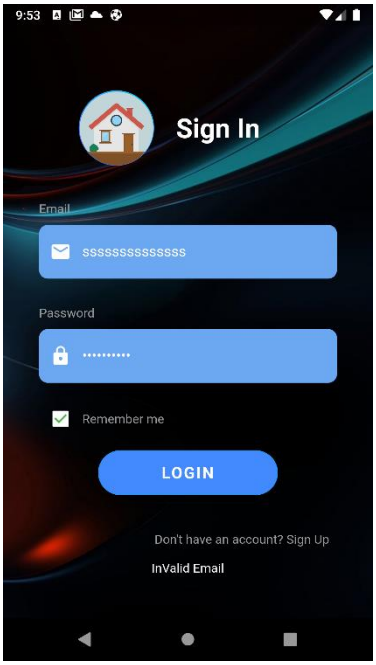

Figure ##.3.3: Invalid Email Error.

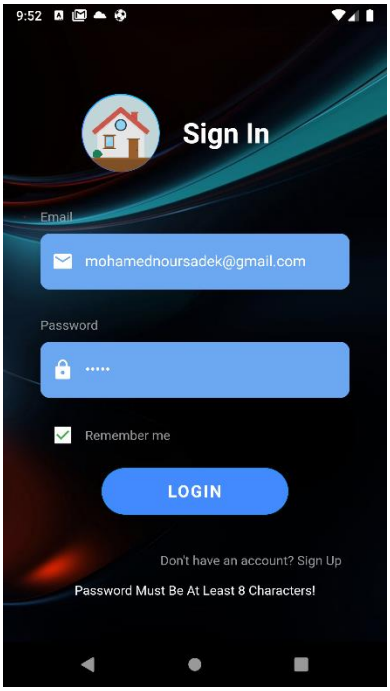

Figure ##.3.4: Password is too short error.

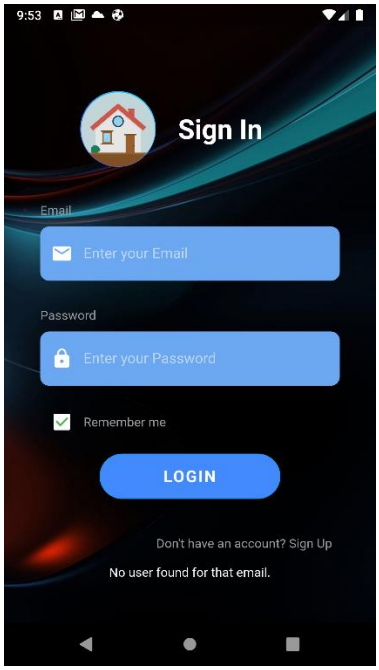

Figure ##.3.5: No user is associated with this email error.

Sign-Up Screen:

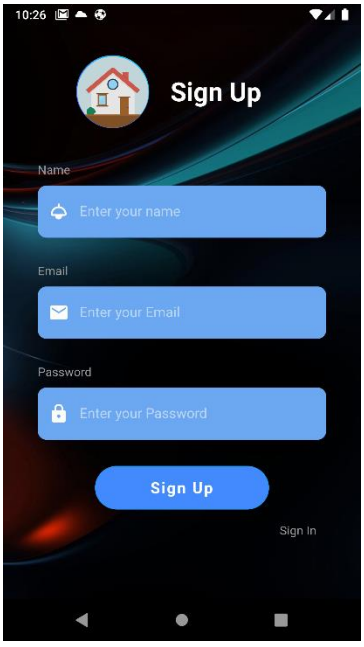

Figure ##.3.6: Sign-Up

It’s Similar to the login-screen, but it’s sign up and it contains three input boxes:

- Name.
- Email.
- Password.

The username must be 8 characters or more.

The Email must be valid with the sign “@” and “.” In the string.

The password must be at least 8 characters long.

With the same errors except a new one when the email is already used:

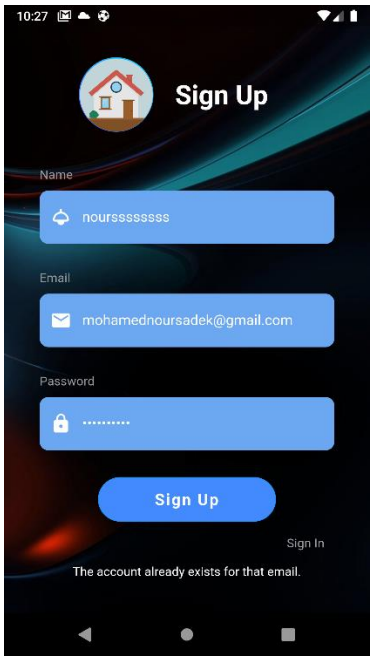

Figure ##.3.7: Sign-Up error.

Home Screen:

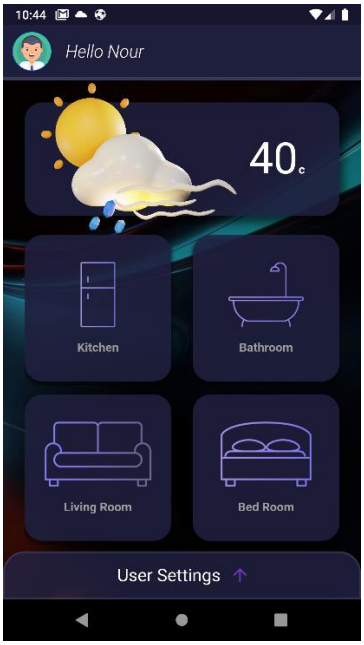

Figure ##.3.8: Home Screen.

It’s the screen where the user sees

- Top Bar:

Contains the user name with the word hello before it, as well as the user avatar, which could be changed from the user settings.

- Temperature inside the house:

Contains the temperature that the sensors in the house read as the average temperature of all the sensors.

- all the rooms.

Each room is represented by an icon which is responsible for opening the room.

- Bottom bar for user settings.

The bottom bar is a slider bar which if slides up, will open the user settings panel.

Rooms:

First When you enter a room, a middle screen shows up until the data is retrieved from the databased.

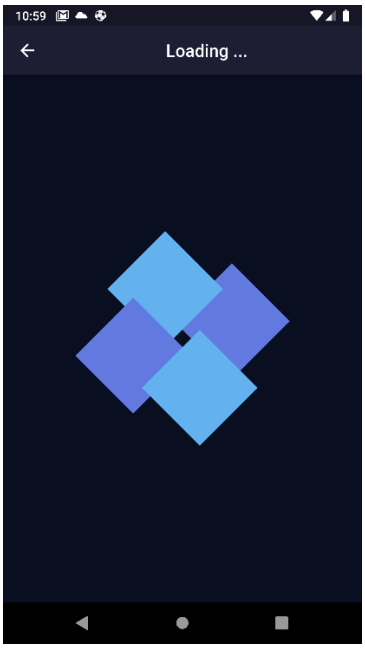

Figure ##.3.9: Waiting.

And then the room opens, based on the room itself, some functionalities show up, here's an example of the living room:

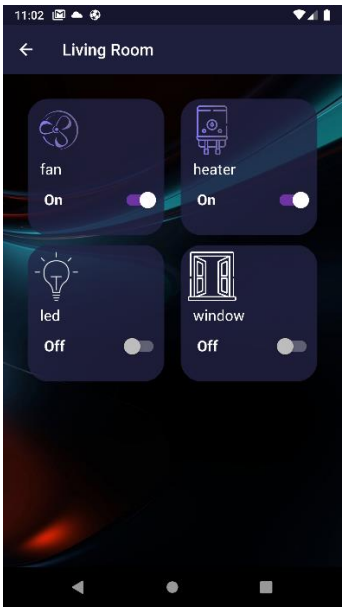

Figure ##.3.10: Living Room.

User Settings:

By sliding the bottom bar a screen shows up which gives you the ability to:

1) Change the user Name.
2) Change the user avatar.
3) Change the color of the app.
4) Log-out.

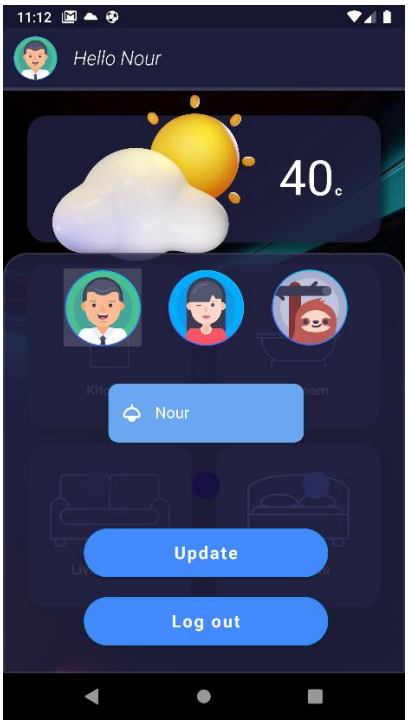

Figure ##.3.11: User Settings.

##.4 Front-End:

The Concept of Widgets

Flutter uses a style of writing code for showing the front end of an application that is called widgets.

A widget is a concept that takes its inspirations from React, and I quote:

"The central idea is that you build your UI out of widgets. Widgets describe what their view should look like given their current configuration and state. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next."

So widgets exist in widget trees, for example, you build an app by building a main widget which will contain all other widgets, this widget is called regularly "Container", and in it you nest other widgets, like panels, and buttons, etc.

So, for example:

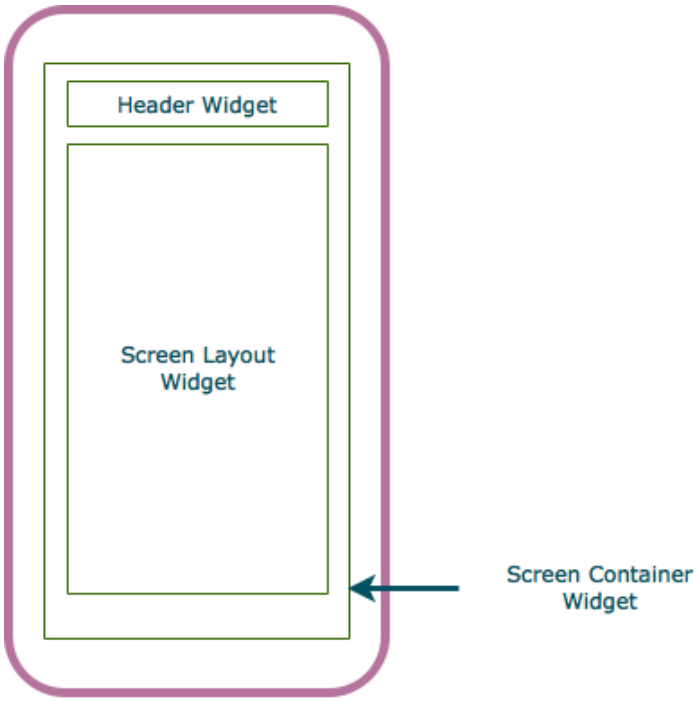

Figure ##.3.12: Widgets layout.

This Screen is made of:

- Screen Widget which contains:
    - Header Widget which contains:
        - Body.
        - Text.
        - Etc.
    - Screen Layout Widget which contains:
        - Body.
        - Text.
        - Etc.

And to give an example from our app:

```
class Loading extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Center(child: Text("Loading ...")),
        backgroundColor: Color(0xFF1D1E33),
      ),  // AppBar
      body: Center(
        child: Lottie.asset('assets/images/98432-loading.json'),
      )  // Center
    );  // Scaffold
  }
}
```

Figure ##.3.13: Loading Code.

Here’s the simplest screen in our app, which is a loading screen, it’s contained all in a widget called

- Scaffold which contains:
    - App Bar which contains:
        - Title text.
        - Background color.
    - Center which contains
        - Child image (animated images).

And that’s how the output looks

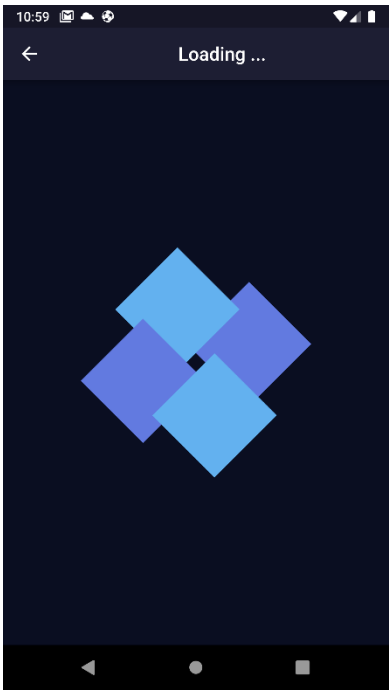

Figure ##.3.14: Loading Screen.

Customized looks:

Throughout the app we have added features which could be customized, for example:

- The user name.
- The color.
- The avatar.

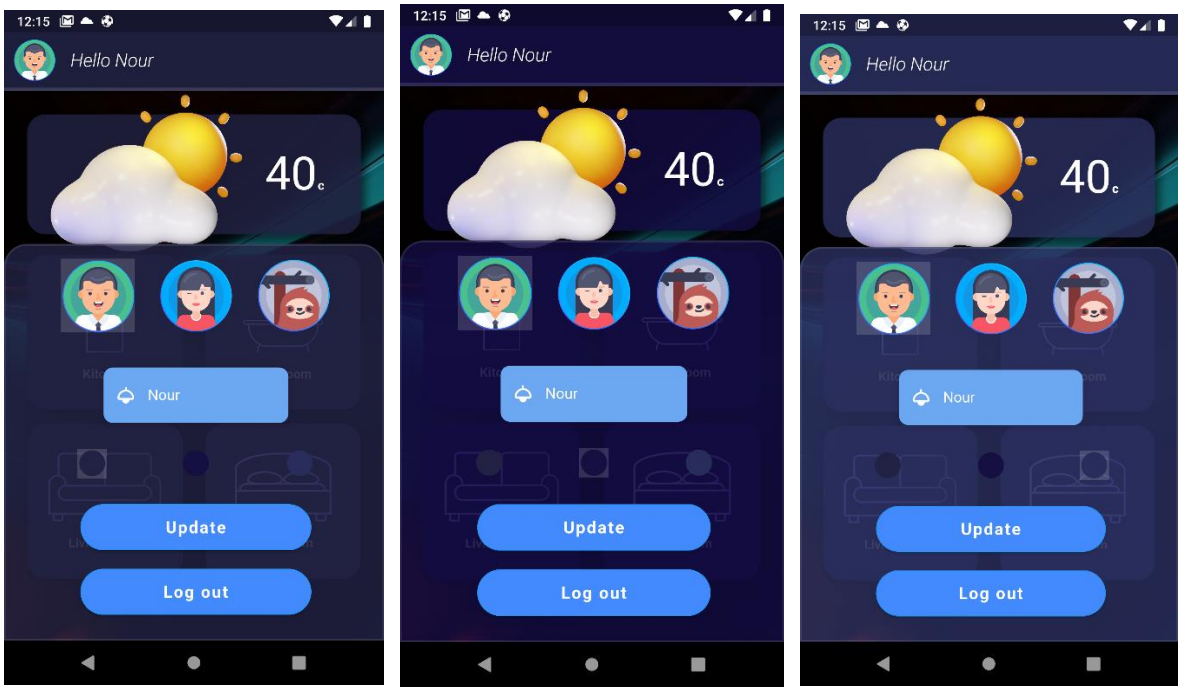

Figure ##.3.15: Color Customization.

Parameters dependence:

Icons and frontend depend on what the parameters are, for example the temperature of the house is reflected not only in the number but in the icon:

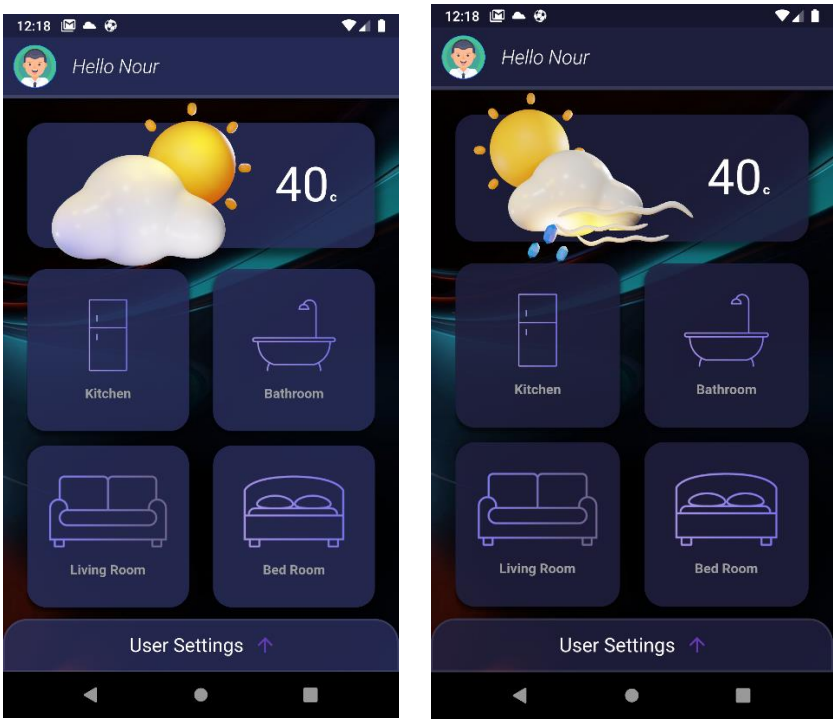

Figure ##.3.16: Parameter dependence.

Space management using sliders:

Using Sliding bars, we can fit much more than we can otherwise.

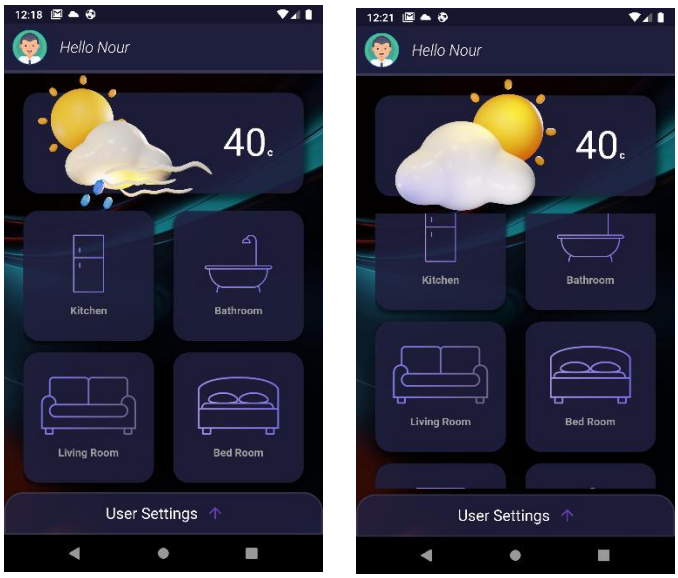

Figure ##.3.17: Space management.

##.5 Back End:

##5.1 Introduction:

IoT is all about connecting devices, or “things” as you may call them, to the internet and then analyzing data collected from these devices to extract an added value.
Most of my IoT projects require some way of communication between the different endpoints. These endpoints can be anything from devices and services to applications, and eventually data needs to be stored somewhere for further processing and analyzing.
So, let’s say you want to build an IoT system where a device will measure temperature and humidity values from sensors and send them to a database service to store them. Then you want to have a web application that will fetch these values and display them in a dashboard
Firebase offers many cloud services that ranges from authentication, storage, and cloud functions to hosting your web application.

We used three services from firebase (**Firebase Realtime Database**, **Firebase Authentication** and **Cloud Firestore Database**)

##5.1.1 Services we used

Firebase Realtime Database.

A real-time database is a database system which uses real-time processing to handle workloads whose state is constantly changing. This differs from traditional databases containing persistent data, mostly unaffected by time
In the case of Firebase Realtime Database, clients will be connected to the database and will maintain an open bidirectional connection via websockets. Then if any client pushes data to the database, it will be triggered and (in this case) inform all connected clients that it has been changed by sending them the newly saved data.
So, we used it as the main database for home sensors and controllers.

Firebase Authentication

Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook, and Twitter, and more.
We use it using email and password, but we got a problem that we cannot add variables with the user data as we want as it limited to some data types like display name and photo URL, so we used Firestore database as third service to save it.

Cloud Firestore

Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. Like Firebase Realtime Database, it keeps your data in sync across client apps through Realtime listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity.
So, we used it to save additional user data like the home code belong to the user
We took the home code as foreign key and saved it with the user id to be easily access anywhere.

##5.2 Features and implementations:

##5.2.1 Authentication

As we must make the app be used only with users that own a Home, we provided a secure key for every home. As we said earlier the firebase authentication doesn't provide save additional data with user data, we had to use Cloud Firestore.

Initially we use a provider to provide user data and functionality of authentication (login, sign up and logout)

Concept Of Provider:

Provider is a wrapper the provide data and functions to all the application without having to pass it to each component we want to use this add.

Diagrams

Sign In Flowchart

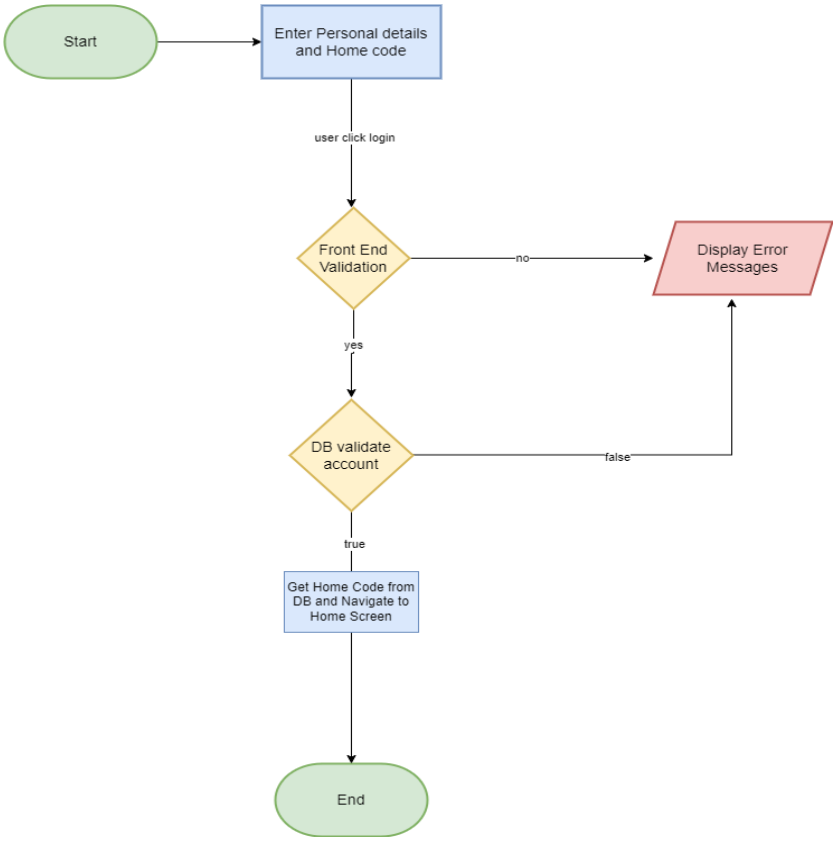

Figure ##.5.1: Sign in Flowchart

Registration Flowchart

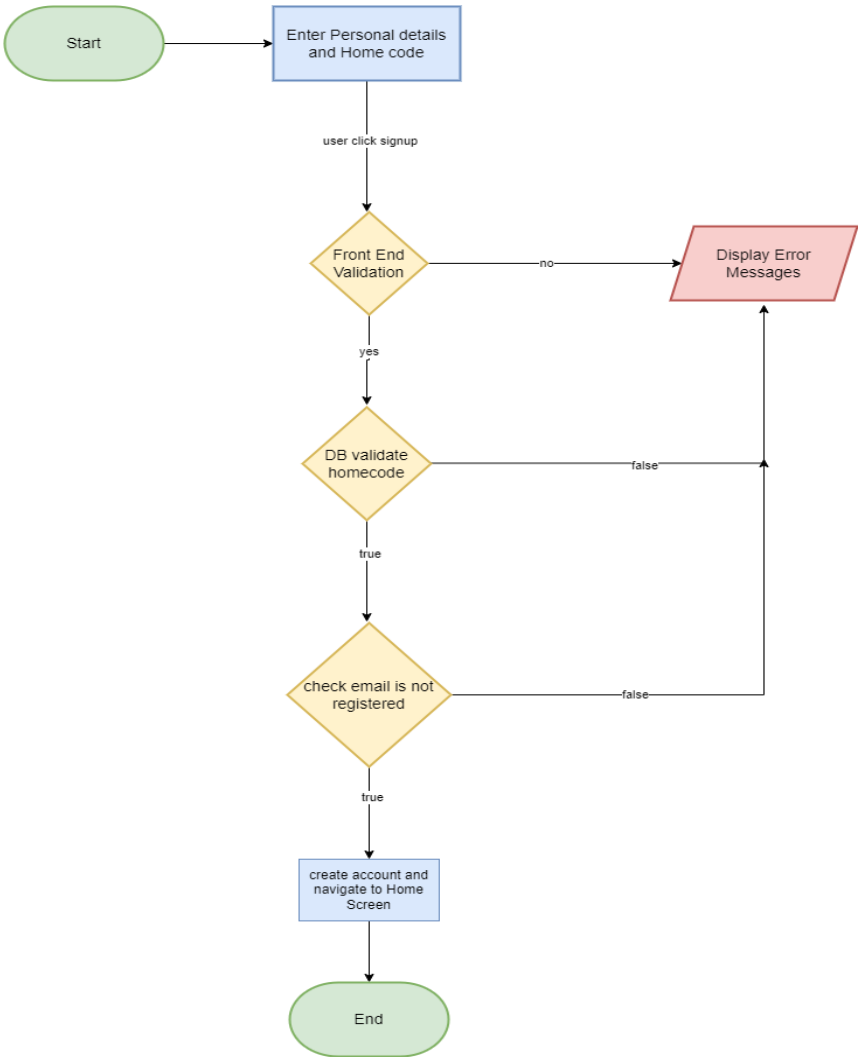

Figure ##.5.2: Registration Flowchart

Implementation

```
class AuthenticationService {
      Stream<User?> get authStateChanges => _firebaseAuth.userChanges();

      Future<void> signOut() async {
            await _firebaseAuth.signOut();
      }

      Future<String> signUp(

      {required String email,
      required String password,
      required fullName,
      required code,
      required photoURL}) async {
            try {
                  DatabaseEvent event = await ref.once();

                  Var homesList = event.snapshot.value as Map;
                  var homes = homesList.keys.toList();
                  var found = false;

                  for (var home in homes) {
                        if (home == code) found = true;
                  }
                  if (found == false) return "Invalid Code";

                  UserCredential userCredential = await _firebaseAuth
                  .createUserWithEmailAndPassword(email: email, password:
                  password);
                  User? user = userCredential.user;

                  await user!.updateDisplayName(fullName);
                  await user.updatePhotoURL("icons/vector.png");

                  await users.doc(user.uid).set({
                  "email": email,
                  "code": code,
                  "uid": user.uid,
                  });

                  return "success";
            } on FirebaseAuthException catch (e) {
                  return e.code;
            } catch (e) {return "error";}
      }
```

Figure ##.5.3 Authentication Code

As shown this an example of **Stream** variable and two functions (Our provided has more functions like **sign in** and **updating user data**)

Why are we using stream?

To keep listen to the user data if it has been changed without requesting with a button or something.

Sign Up Logic

First, we ensure the home code the user has entered is a valid home code then we trying to create a new account with only the email and password.

If it successfully done, then we are adding the display name and photo URL as we cannot do it in one step, the third step we are adding the user id with home code in the cloud firestore to be easily

access after that. Firebase authentication automatically handles all user register failure like short password or duplicated emails.

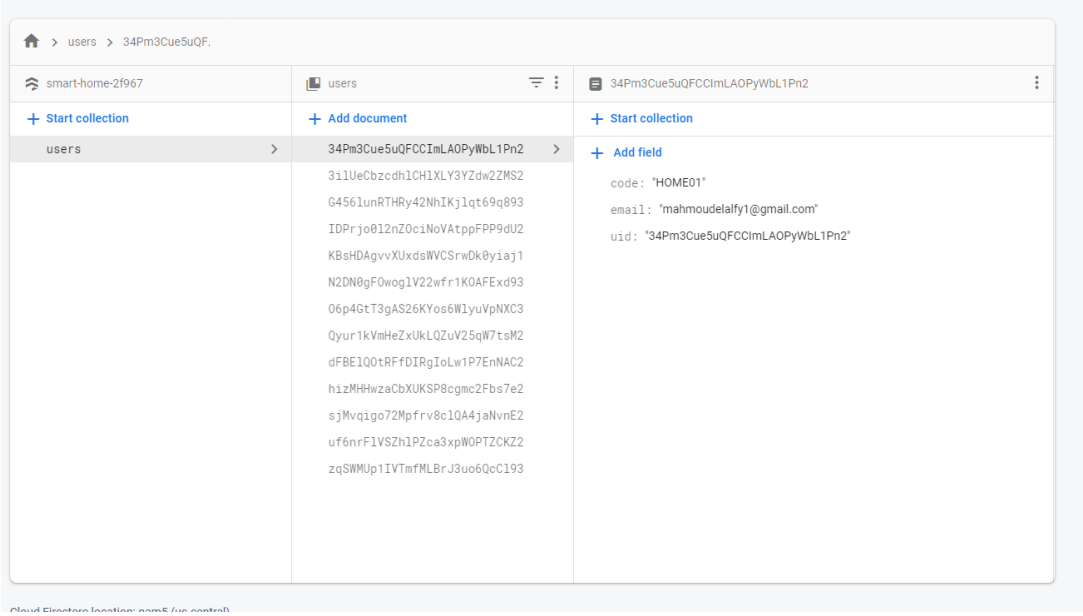

Figure ##.5.4 Cloud Store

Login, Update user and Logout Logic

As we don't need more logic than the normal authentication, we use only Firebase authentication services on it

##5.2.2 Current Weather

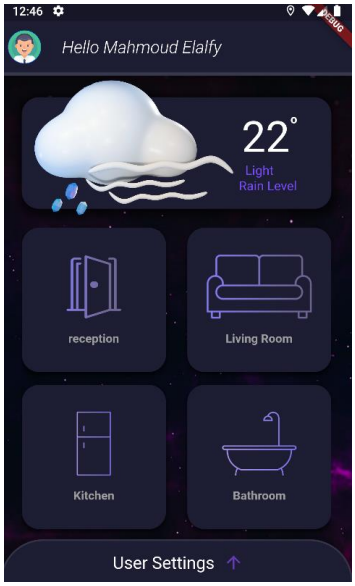

Figure ##.5.5 Weather widget

Our First Feature is providing current temperature from external API depend on the location of the user mobile but when getting the rain sensor, the is implemented from the IOT team.

Implementation

We used open weather API as a service to provide the app with current weather details we are getting the data throw API request then we are decoding the result from JSON format to Flutter Object Format Using Object

```dart
Future<Weather> fetchWeather() async {
  final response = await http.get(Uri.parse(
      'https://api.openweathermap.org/data/2.5/weather?q=zagazig&appid='));
  if (response.statusCode == 200) {
    return Weather.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load Weather');
  }
}
```

Figure ##.5.6 Weather API Code

```dart
class Weather {
  final double temp;

  const Weather({required this.temp});

  factory Weather.fromJson(Map<String, dynamic> json) {
    return Weather(
      temp: json['main']['temp'] - 273.15,
    );
  }
}
```

Figure ##.5.7 Weather Class Code

How We Get Home Code and Rain status from Realtime Database when the application loading?

To get **rain status** from the database, we need to get **home code** first from cloud firestore to be able to get the correct home status.

As we mentioned before we listen to the user data and change through the provider

Figure ##.5.8 Listening to user status

First Step we get the **user id** from the user data

Second Step we get **Home Code** from the cloud firestore using the **user id**

Third Step we get the **Rain Status** from the Real time Database using the **home code**

For Every Step from these steps (also getting weather data) we are using Flutter Future Builder

Concept of Future Builder

Widget rebuilding is scheduled by the completion of the future, using State.setState, but is otherwise decoupled from the timing of the future. The builder callback is called at the discretion of the Flutter pipeline and will thus receive a timing-dependent sub-sequence of the snapshots that represent the interaction with the future.

##5.2.3 Rooms Data

As we already got the Home Code at the Home screen Every room now can got by passing at or importing it from Home Screen

To get rooms data from the database we get an instance from the database where we can access all room data, we got it by using **onValue** method as shown blue which provide a stream that we will subscribe to keep track if the data has been changed in the database

After that we are looping on all on-off devices and building card for everyone so if the user installed new on-off devices it automatically added to the application without any configuration.

Get Data Flowchart

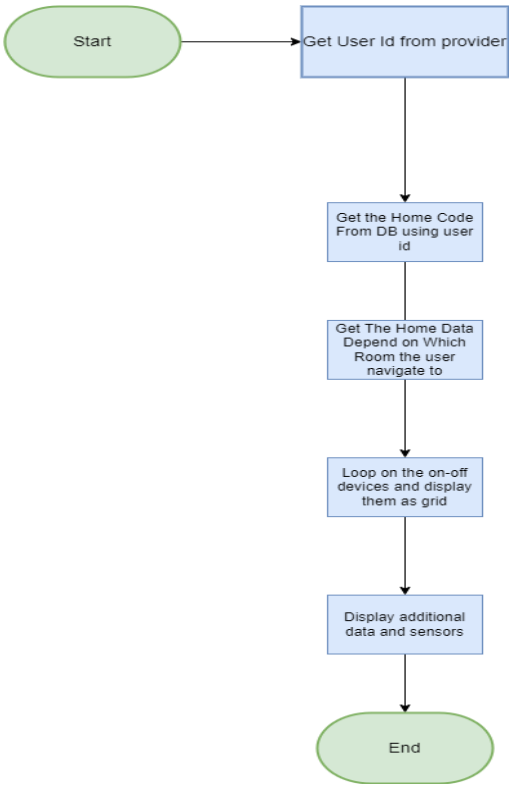

Figure ##.5.9 Get Data Flowchart

Implementation

```
get_Data_from_Firebase() {
    dbref = FirebaseDatabase.instance.ref(Home_Code + "/bathroom/on-off");
    Stream<DatabaseEvent> stream = dbref.onValue;

// Subscribe to the stream!
    stream.listen((DatabaseEvent event) {
      if (!mounted) return;
      setState(() {
        dataBase = event.snapshot.value as Map;
        devices = dataBase.entries
            .map((entry) => {entry.key: (entry.value == 0 ? false : true)})
            .toList();
        is_Loading = false;
        streamController.add(is_Loading);
      });
    });
  }
```

Figure ##.5.10 Get Rooms Data Code

Also, as we got instance from the firebase, we can edit it by function we created to toggle the on-off device.

5.3 Shared Preferences

As the firebase authentication is already persistence, we didn't have to save the user data on the device to get in the app without having to login every time.

But we made a shared preference service to provide saving additional data on the device like the selected color

Implementation

```
class CacheHelper {
  static late SharedPreferences sharedPreferences;

  static init() async {
    sharedPreferences = await SharedPreferences.getInstance();
  }

  static dynamic getData({
    required String key,
  }) {
    return sharedPreferences.get(key);
  }
}
```

Figure ##.5.11 Shared Preference Code

##.6 Future improvements:

Conclusion:

The mobile application is necessary for the user to control and monitor the house either on-site, or remotely, however there are much more that could be done, and to mention a few:

1) Patterns creator: users should be able to customize patterns of controlling devices in the house, for example, they can tell the app to turn the lights automatically at certain times of the day, or based on the light outside of the house, you can also control the heater to work only if the temperature is lower than a certain amount, etc.
2) User permissions: some users might have permission to do some things and others don't.

List of figures:

Chapter ##:

List of Tables:

Chapter ##:

References:

[1] Flutter Docs., (2017). *Introduction to widgets*, available at https://docs.flutter.dev/development/ui/widgets-intro

[2] Firebase Docs., (2022). *Add Firebase to your Flutter app*, available at https://firebase.google.com/docs/flutter/setup