VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Hong Huynh

# VAASA STUDENT GUIDE

# MOBILE APPLICATION IN FLUTTER

School of Technology
2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLICATIONLIED SCIENCES
Information Technology

## ABSTRACT

| | |
|---|---|
| Author | Hong Huynh |
| Title | Vaasa student guide mobile application in Flutter |
| Year | 2023 |
| Language | English |
| Pages | 37 + 0 |
| Name of Supervisor | Anna-Kaisa Saari |

The aim of the thesis was to develop a mobile application built in the Flutter framework created by Google. Flutter is known as a cross-platform framework to support developers to create applications that can run on Android, IOS, Web, and Desktops depending on only one codebase.

The tools and methods used in the thesis project include Flutter, Google Maps API, Agile methodology for project management and the implementation of the BLoC (Business Logic Component) pattern for state management within the mobile application.

The application helps new students adapt to the new school as well as to the new city more conveniently and rapidly. Furthermore, the application provides recommendations for some places for the students to play sports, to enjoy meals, and to purchase food at affordable prices. In addition, the students will be able to find more information how they can get part-time jobs, and sell or trade goods.

Keywords Flutter, Firebase, Google Maps

# Contents

# List of Figures

# 1. INTRODUCTION

## 1.1 Objectives

The main objective of the thesis was to practice developing a Flutter application along with the Google ecosystem. The app was developed in the BLoC design pattern in order to manage the state of the application, ameliorate the file structure organizing mindset, and maintainability of the codebase. Another objective was to use Firebase Authentication and Cloud Firestore of Google for administrators to sign in and manipulate data. Also, Google Maps and Openrouteservice API are utilized for the users to navigate from where they are to the locations of the recommended places in the app. The application will be released on Google Play Store and App Store in the near future when the app has been tested on multiple platforms so that new students of VAMK specifically and students in Vaasa, in general, will be able to approach to make their life in Vaasa easier and more convenient.

Not only did the project contribute to the author strengthening programming skills, but it also helped the author to gain a deeper understanding of the fundamentals of BLoC architecture, which is one of the most used patterns when developing with Flutter. Additionally, the author was able to keep track of the list of tasks to finish the product within the given timeframe. Hence, the project brought opportunities to apply project management skills that the author had learnt from school and internships into practice.

## 1.2 Contributions

New students might feel uncomfortable when arriving in a new city, or new environment. This has happened to many students especially international students so this is the reason why this project has been started. Students might not have any acquaintances living in Finland and there may be a lot of questions unsolved or it might take time to get used to a peaceful but freezing country.

With this app, newcomers are able to reach the latest and various information, such as how they can rent an apartment, show directions to the school they are about to study at or a restaurant from their current location, or free courses they can attend. As a result, they will adapt to live in Vaasa as fast as possible and people who already lived in Vaasa might not be asked the same questions repetitively.

## 2. APPLICATION DESIGN

### 2.1 Application Features

Purpose of the application is to make living in Vaasa more convenient and comfortable for new students. The application needs to show information about places or social media in community section that can help the users to know more about Vaasa and meet new people. Application should reveal the address, website, or contact information of the mentioned places or social media. Also, the users can contact article places, such as by sending an email directly to the mentioned place in the article, visiting its website, or calling with only one tap on the screen. Also, map can be zoomed in and out, and help in navigating the users to the place mentioned in the article.

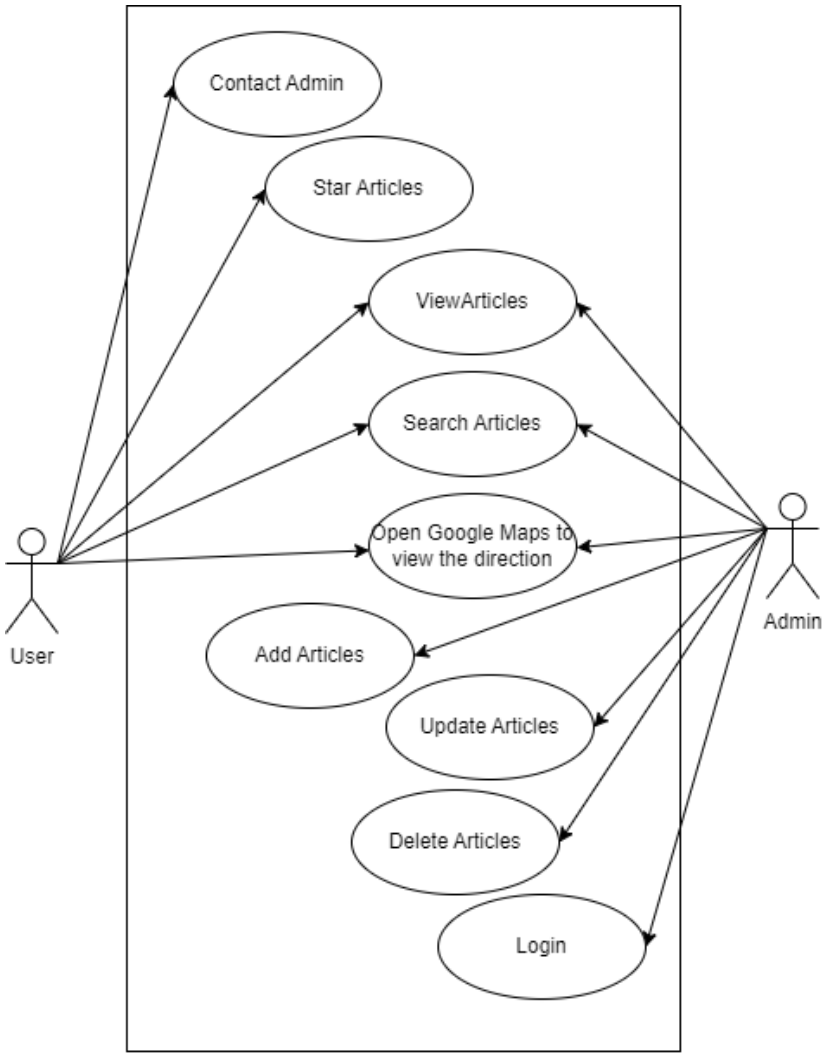



**Figure 1.** Vaasa Student Guide application use cases

Figure 1 shows the application use cases. All shown information in application cannot be manipulated freely because there might bring misleading or privacy-violating information, hence, only admins have the authority to manage the articles within the application to ensure that the information is not violating any laws or ethical dignity. The Admin can manage the articles including creating, reading, updating, and deleting articles. Basic users don't need to log in because all the information within the application is publicly released. Also, the articles can be starred so that the user does not need to reach to a specific category to read that article. In addition, if there is any feedback, or information inaccurate or the users intend to join the admin group, the users can contact the admin via the Contact Screen in application.

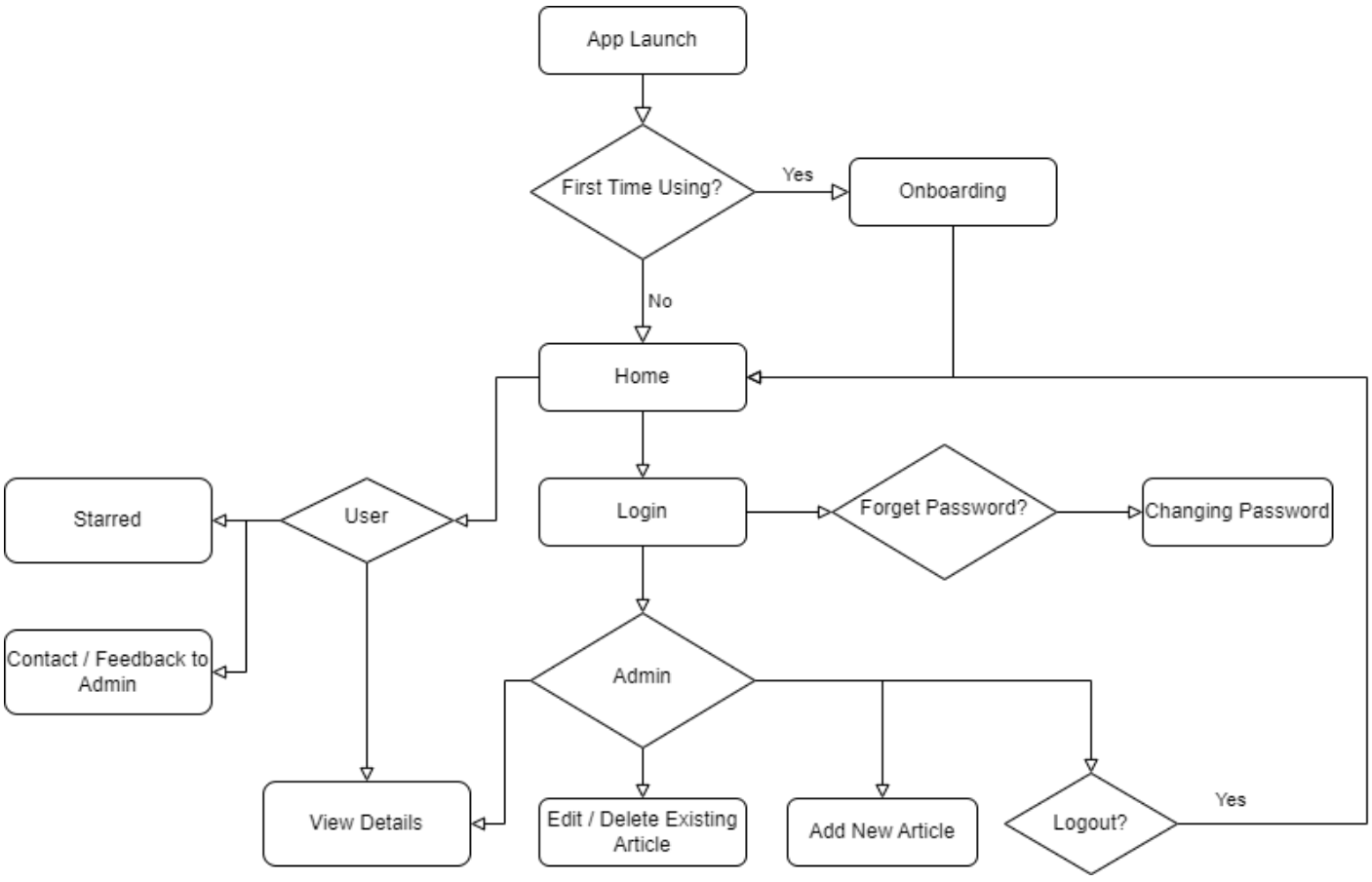



**Figure 2.** Flowchart of the project's application

The flowchart, as shown in figure 2, reveals how the application works based on the corresponding conditions and types of users. The application gives permission to the users and admins with suitable features. For instance, only basic users can star their favorite articles so that they can read them again without choosing a specific category and finding those articles. Also, basic users can contact the admins for feedback. Features for logging into the application, being authorized to manage the articles, or resetting the password can only be used by the admin users.

## 2.2 Overview of UI

The aim to make a good-looking and useful mobile application is a simple but user-friendly application design and helpful features that meet the users' requirements. Figma, a web application for interface design, was used to design the UI of the application. Moreover, Figma is a free tool and it does not take much memory to launch or use as Adobe XD provided by Adobe does. The application screen designs are shown in Figure 3.

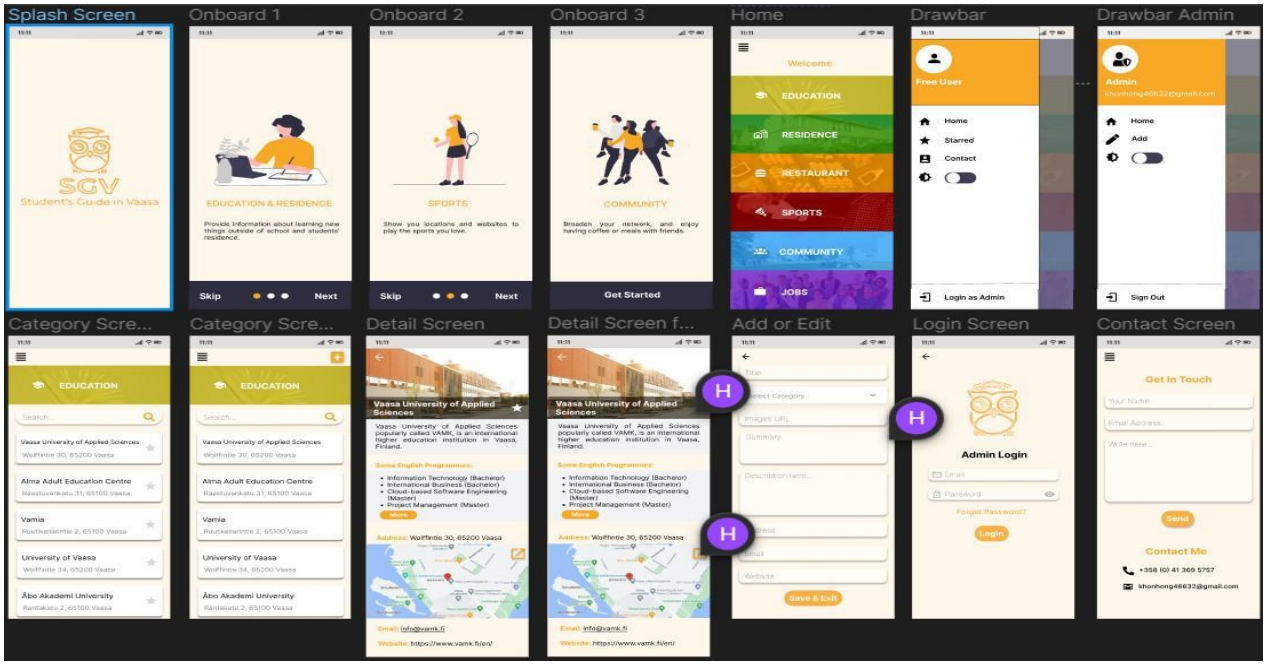



**Figure 3.** Figma Sketch

## 3. RELEVANT TECHNOLOGIES

### 3.1 Flutter Framework

Flutter is known as a cross-platform framework to support developers to create applications that can run on Android, IOS, Web, and Desktops depending on only one codebase. The Flutter framework makes use of Dart programming language also developed by Google and an incredibly huge set of customizable widgets that allow developers to apply their creativity to responsive, user-friendly, and modern user interfaces (Flutter documentation, 2023).

One essential and beneficial feature of Flutter is its "hot reload" functionality, which enables developers to immediately see the alterations of the code in the application as they are programming, without having to start rebuilding the entire application. As a result, the process of building the application will be more rapid and more productive (Flutter documentation, 2023).

Flutter also comes with a wide range of tools and clean architectures, such as the BLoC architecture, which is a design pattern that assists to manage the state of the application and organize the code structure. Furthermore, Flutter and Firebase combination undoubtedly works really well, which offers features to the mobile application such as authentication and cloud storage.

In other words, Flutter is a reasonable option for mobile application development because of its flexibility, productivity, and straightforwardness. Due to the fact that Flutter is created by Google, Flutter has its own documentation website storing concise and coherent information along with Flutter examples (Flutter Documentation, 2023).

Additionally, there are numerous growing communities on different platforms including LinkedIn, Facebook, and more for developers who are keen on learning and using Flutter so that they can exchange their ideas and knowledge (Sharma, S., 2020). Flutter UI consists of widgets. The widgets can be reused and customized without spending too much time and effort as using HTML (Tutorials Point, 2023). Flutter widgets' UI is based on Material UI and Cupertino UI design commonly for Android devices and IOS devices, respectively. However, the developers have to set up manually or with existing packages to choose the suitable design of UI with the corresponding types of devices.

### 3.2 BLoC Design Pattern

In order to maintain the file structure organized, manage states and separate business logic from UI components, an architectural pattern is essential in Flutter applications. BLoC, Business Logic Component, is one of the most powerful and used architectural patterns, which provides a straightforward separation of concerns, stream-based state management and testability and code reusability. For instance, once a button on the mobile screen is clicked, an event is triggered, and Bloc emits the component state along with queried data which can be fetched from the API or a repository which can be local storage or cloud storage. If the requesting data process has errors or depending on there are bugs in system that cause errors, the bloc will emit a state called Error which can for example show an error message. Otherwise the state will be Success and the view will be rebuilt and data be shown on the screen. Furthermore, there is a subset of the BLoC design pattern called Cubit that only needs to trigger functions instead of events for emitting states (Goswami, 2023).

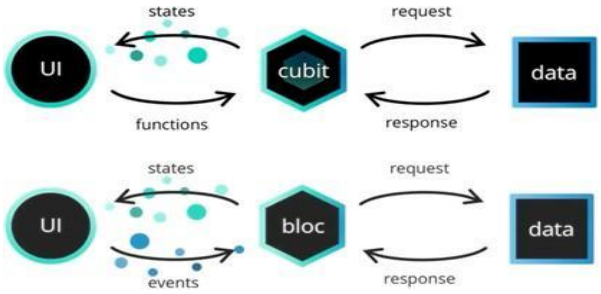



**Figure 4.** Illustration of how BLoC and Cubit work

As Figure 4 shows, BLoC and Cubit have the same responsibility which is emitting a new state to change the UI once a response is received or a request is sent. However, there is one difference, cubit starts to work when a function is called while bloc starts running when an event is added.

In the project, BLoC was used to manage the theme and authentication state of the application throughout the application lifetime. To implement BLoC architecture, its dependencies need to be installed in the pubspec.yaml file. Required dependencies are equatable, bloc and flutter_bloc packages. With these dependencies, developer can create three classes namely AuthBloc, AuthEvent, and AuthState by a few clicks instead of creating every class manually. Whereas, Cubit only needs to trigger functions instead of events for emitting states. Hence, there are ThemeCubit and ThemeState class only.

Figure 5 shows the files needed for making Cubit, and BLoC patterns possible within the project. Cubit is created to be a simpler alternative version of BLoC; thus, it only needs two files while Bloc needs three. First file is theme_cubit.dart file, that contains a class inherited from the Cubit class provided by flutter_bloc. Theme_cubit.dart is responsible for managing triggered functions as input events and changing the state according to those input events. Second file is theme_state.dart, that stores unmodified theme states containing read-only properties. There is one difference shown for BLoC to manage the authentication state which is auth_event.dart file which is storing the various authentication event such as login event, logout event or reset password besides auth_state.dart and auth_bloc.dart file.

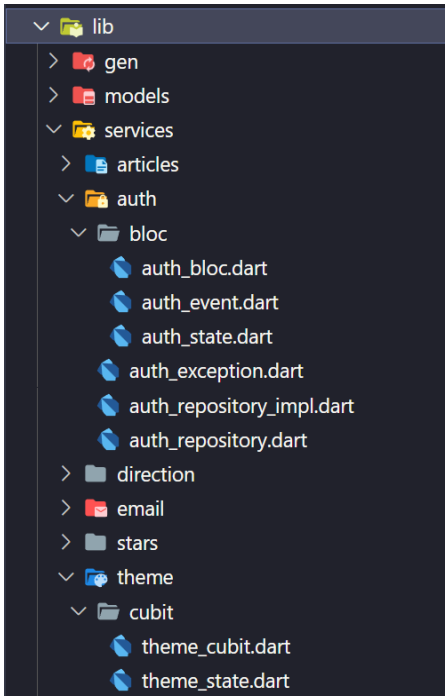



**Figure 5.** File Structure when using Bloc and Cubit

Moreover, BlocProvider needs to be used to provide a single instance of a bloc inside the widget tree. However, in this project, MultiBlocProvider can be used instead because there is more than one bloc which is ThemeCubit and AuthBloc so that instances of AuthBloc and ThemeCubit can be used to update state across multiple widgets. For instance, to get the current theme mode via the current state of the application, an instance of ThemeCubit can be called by including BlocProvider.of<ThemeCubit>(context).state.themeMode method within the BlocBuilder widget.

```dart
@override
Widget build(BuildContext context) {
  return MultiBlocProvider(
    providers: [
      BlocProvider(create: (context) => AuthBloc(AuthRepositoryImpl())),
      BlocProvider(create: (context) => ThemeCubit()..getTheme()),
    ],
    child: BlocBuilder<ThemeCubit, ThemeState>(
      builder: (BuildContext context, ThemeState state) {
        return PlatformApp.router(
            debugShowCheckedModeBanner: false,
            title: 'Student Guide App',
            material: (context, _) => MaterialAppRouterData(
                  theme: lightTheme(context),
                  darkTheme: darkTheme(context),
                  themeMode:
                      BlocProvider.of<ThemeCubit>(context).state.themeMode,
                ), // MaterialAppRouterData
            routerDelegate: _appRouter.delegate(initialRoutes: [
              showHome ? const HomeRoute() : const OnboardingRoute()
            ]),
            routeInformationParser: _appRouter.defaultRouteParser()); // Pl
      },
    ), // BlocBuilder
  ); // MultiBlocProvider
}
```

**Figure 6.** MultiBlocProvider in the main.dart file

After making the ThemeCubit and AuthBloc available in the application, BlocBuilder is required to rebuild the widgets for switching between states as shown in Figure 6. The theme mode of the application will change to light or dark based on the application current state.

```dart
class ThemeCubit extends Cubit<ThemeState> {
  ThemeCubit() : super(const ThemeState(themeMode: ThemeMode.system));

  void setTheme(int index) {
    ThemeState themeState;
    final themePref = ThemePreferences();
    themePref.setTheme(index);
    switch (index) {
      case 0:
        themeState = const ThemeState(themeMode: ThemeMode.system);
        break;
      case 1:
        themeState = const ThemeState(themeMode: ThemeMode.light);
        break;
      case 2:
        themeState = const ThemeState(themeMode: ThemeMode.dark);
        break;
      default:
        themeState = const ThemeState(themeMode: ThemeMode.system);
        break;
    }
    emit(themeState);
  }

  Future<void> getTheme() async {
    final themePref = ThemePreferences();
    final theme = await themePref.getTheme();
    ThemeMode themeMode;
    if (theme == Themes.system) {
      themeMode = ThemeMode.system;
    } else if (theme == Themes.dark) {
      themeMode = ThemeMode.dark;
    } else {
      themeMode = ThemeMode.light;
    }
    emit(ThemeState(themeMode: themeMode));
  }
}
```

**Figure 7.** Functions of theme cubit

Theme cubit has two functions, setTheme() and getTheme(). setTheme() function saves the user theme preference into the local storage of the device. After that, this function will trigger ThemeCubit to emit the corresponding theme state. getTheme() method is used for getting the theme from the local storage and for emitting that theme to change the related theme mode of the application once the user starts using the application. These functions are shown in Figure 7.

```
return BlocBuilder<AuthBloc, AuthState>(
  builder: (context, state) {
    if (state is AuthStateUninitialized) {
      context.read<AuthBloc>().add(AuthEventInitialize());
    }
    final isLoggedIn = state is AuthStateSignedIn;
    return Scaffold(
      appBar: CustomAppBar(),
      drawer: DrawerMenu(isLoggedIn: isLoggedIn),
      body: Column(
        children: [
          CustomText(
            isLoggedIn ? 'Welcome back' : 'Welcome',
            size: 22,
            fontWeight: FontWeight.w500,
          ), // CustomText
          const SizedBox(height: 24),
          Flexible(
            child: ListView.builder(
              itemExtent: MediaQuery.of(context).size.height * 0.14,
              itemCount: categoryList.length,
              itemBuilder: (BuildContext context, int index) {
                return HomeItem(
                  categoryList[index],
                ); // HomeItem
              },
            ), // ListView.builder
          ), // Flexible
        ],
      ), // Column
    ); // Scaffold
```

**Figure 8.** BlocBuilder using AuthBloc

When the users open the application, the current state needs to be checked. The starting state for the application is AuthStateUninitialized as shown in Figure 8. In starting state, Bloc initializes the Firebase repository via AuthRepository to ensure that the user has already logged in as an admin and then emit the following state as shown in Figure 9. If the current state is AuthStateSignedIn, the drawer will show the options that only administrators can only see and choose.

```
class AuthBloc extends Bloc<AuthEvent, AuthState> {
  AuthBloc(AuthRepository repository)
      : super(const AuthStateUninitialized(isLoading: true)) {
    //* initialize
    on<AuthEventInitialize>((event, emit) async {
      await repository.initialize();
      final admin = repository.currentAdmin;
      if (admin == null) {
        emit(const AuthStateSignedOut(exception: null, isLoading: false));
      } else {
        emit(AuthStateSignedIn(admin: admin, isLoading: false));
      }
    });
```

**Figure 9.** AuthBloc class

### 3.3 Google Firebase

Google Firebase, a back-end software brought by Google, is a development tool that enables developers to create applications for iOS, Android, and web platforms with a range of necessary features namely authentication, databases, or tracking analytics (Rosencrance, L., 2019). Despite various features, this project only uses two features from Firebase: Firebase Authentication and Cloud Firestore.

Firebase Authentication is a secure authentication service that offers straightforward authentication methods either for mobile or web applications. The users are able to register and sign in using numerous identity providers such as Facebook, GitHub, and Google, which will push the authentication process more quickly and effortlessly (Firebase Documentation, 2023).

Cloud Firestore is developed to store and synchronize data in real-time across multiple clients and devices, which is suitable for building cutting-edge, scalable, and offline-capable applications. It automatically synchronizes the data in real-time. Cloud Firestore is able to handle huge datasets and the developers can set the read and write permissions via built-in security rules for specific clients. Hence, it gains security to protect the clients' data. Moreover, Cloud Firestore provides advanced querying capabilities including support for complex queries, sorting, filtering, and so on, which helps developers retrieve and manage the data from the database without any difficulties (Pathik, 2021).

At first, a Firebase project must be created from the Firebase console. The easiest way to initiate a Firebase project is by installing and using Firebase CLI which stands for Command Line Interface. By using Firebase CLI, developers do not need to spend time registering for the Android and iOS part of the Flutter application, downloading and setting up files provided by Google for the project (Firebase Google, 2023). Firebase CLI will help the developers configure the Flutter application connected to the existing Firebase project or a new one.

```dart
class AuthRepositoryImpl implements AuthRepository {
  @override
  AdminModel? get currentAdmin {
    final admin = FirebaseAuth.instance.currentUser;
    if (admin != null) {
      return AdminModel.fromFirebase(admin);
    } else {
      return null;
    }
  }

  @override
  Future initialize() async {
    await Firebase.initializeApp(
      options: DefaultFirebaseOptions.currentPlatform,
    );
  }
```

**Figure 10.** Firebase Authentication implementation

Code to implement Firebase and FirebaseAuth methods for initializing Firebase inside the application and be ready to use to get and check the current logged-in admin is shown in Figure 10. Email/Password will be the sign-in provider for the application. Only administrators can sign in to the application to add and edit articles.

Speaking of Cloud Firestore, the application can make use of it to store data in the cloud-based environment instead of local storage. Firestore utilizes a collection-document-data model which means the data will be stored in collections, documents, and fields. Collections can be understood as tables of a relational database in MongoDB. Documents are simply single records within the collections and stored in the JSON-like format, which enables complex data structures. Meanwhile, fields can be compared as columns of a table in MongoDB.

Moreover, Cloud Firestore offers real-time data synchronization, which means fetching the latest updated data to the application. The application uses the Cloud Firestore so that the admins can save and manipulate the written articles in the database. The addNewArticle() function is shown below illustrating that the ArticleModel class needed to be converted into JSON-like format and then inserts into the cloud storage as a new document. The other methods namely updateArticle() and deleteArticle() are used for passing an id as an argument to query the document having the id field same as the passed id so that the Firestore database can update or delete it. Otherwise, it throws an error.

```
addNewArticle(ArticleModel cloudArticle) async {
  try {
    final document = _collection.doc();
    cloudArticle.documentId = document.id;
    await document.set(cloudArticle.toJson());
  } catch (e) {
    throw CannotAddNewArticle();
  }
}

updateArticle(ArticleModel cloudArticle) async {
  try {
    final document = _collection.doc(cloudArticle.documentId);
    await document.update(cloudArticle.toJson());
  } catch (e) {
    throw CannotUpdateArticle();
  }
}

deleteArticle(String id) async {
  try {
    await _collection.doc(id).delete();
  } catch (e) {
    throw CannotDeleteArticle();
  }
}

Stream<Iterable<ArticleModel>> getArticlesByCategory(
    {required String category}) {
  try {
    return _collection
        .where(categoryField, isEqualTo: category)
        .snapshots()
        .map(
          (event) => event.docs.map((doc) => ArticleModel.fromSnapshot(doc)),
        );
  } catch (e) {
    throw CannotGetAllArticles();
  }
}
```

***Figure 11.*** *Cloud Firestore implementation*

### 3.4 Google Maps and OpenRouteServices API

Google Maps is software from Google that can be used to navigate the current location of the users and the destination by any means of transportation. OpenRouteServices API, an open-source API, works as the Direction API of the Google Maps platform for developers that only have the Google Maps API key but do not have any billing account or have problems when creating a billing account on the Google Cloud Platform. With OpenRouteServices API, developers can generate dynamic polylines as direction routes between two coordinates; the current location of the user and the location of the given address within the application (Rohan, A, 2020).

Google Maps is one of the most popular applications coming from Google for navigation. In order to use Google Maps API, developers need to create an account for the Google Cloud Console so that they can enable the Maps SDK for Android and iOS devices, and Directions API.

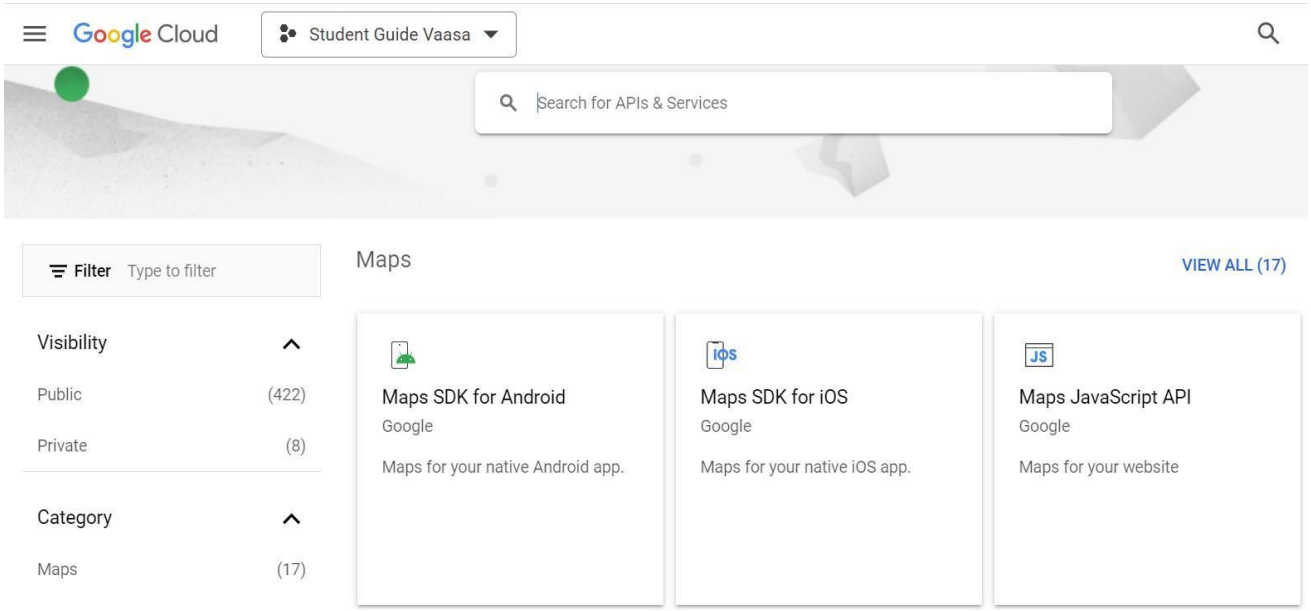



**Figure 12.** Enable APIs and Services

After account creation, credentials must be created to generate an API key and add it to 2 important files: AndroidManifest.xml for Android and AppDelegate.swift for iOS. This is required for the Flutter application can fully integrate Google Maps.

Directions API is needed for the project to draw a dynamic polyline instead of a plain straight polyline from the current location of the user to the destination. If facing any problems when enabling a billing account for Directions API, OpenRouteServices API is one of the best solutions to address.

Creating one or more polylines between two locations is conducted on the GoogleMap widget of the google_maps_flutter package verified by Google's Flutter developers. It requires the longitude and latitude of two coordinates of the current location and the destination, and also the API key for OpenRouteServices API in order to make the mentioned feature possible.

```dart
Future getData({
  required double startLat,
  required double startLng,
  required double endLat,
  required double endLng,
}) async {
  final response = await http.get(
    Uri.parse(
      '${constants.url}${constants.pathParam}'
      '?api_key=${constants.apiKey}'
      '&start=$startLng,$startLat&end=$endLng,$endLat',
    ),
  );

  if (response.statusCode == 200) {
    String responseBody = response.body;
    return jsonDecode(responseBody);
  } else {
    log('API error: ${response.statusCode.toString()}');
  }
}
```

```dart
Future<List<LatLng>> getPolylineCoordinates({
  required double startLat,
  required double startLng,
  required double endLat,
  required double endLng,
}) async {
  try {
    final json = await getData(
      startLat: startLat,
      startLng: startLng,
      endLat: endLat,
      endLng: endLng,
    );

    List coordinates = json['features'][0]['geometry']['coordinates'];

    //NOTE: In the OpenRouteService API, the first value is longitude (lng)
    // and the second value is latitude (lat).
    return coordinates
        .map((coordinate) => LatLng(coordinate[1], coordinate[0]))
        .toList();
  } catch (e) {
    log(e.toString());
    return [];
  }
}
```

**Figure 13.** Open Route Services API implementation

As shown in Figure 10, the latitude and longitude of two locations are passed into the getData() function. getData() function passes the arguments in a GET query to the API. Query returns a String response which is converted into JSON format using jsonDecode() function. The getPolylineCoordinates() function claims the returned value from the previous function and then creates a list of coordinates between the current location and destination coordinates. As a result, google_maps_flutter allows generating a polyline instance by connecting them in order.

## 4. METHODOLOGY

### 4.1 Project Methodology

Regardless of the project scale, a time schedule has to be followed to finish the project before the expected deadline arrives and to ensure that each feature of the final product works at least normally as planned. To reach these goals, Agile and Scrum methodologies were used.

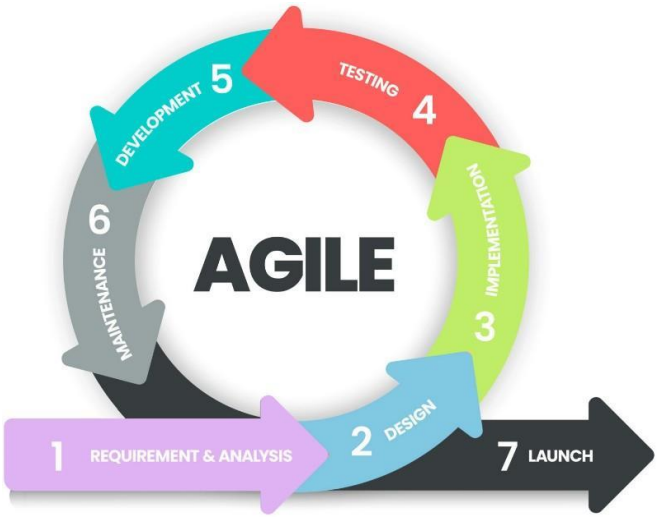



**Figure 14.** Agile lifecycle methodology

As the diagram shows above, small or big projects have to go through all of these Agile methodology phases namely Requirement analysis, Design, Coding, Testing, Deploy, Maintenance, and finally Release. Agile software development practice cannot be neglected because a big project can be divided into small parts so that daily or weekly tasks will always be on track and must be completed. Agile is a project management and software development approach involving working iteratively to meet customers' requirements faster and more efficiently. Instead of waiting for a final gigantic launch at once, agile teams' target is bringing well-qualified value in smaller increments. (Atlassian, 2023)

Scrum is a framework for managing complex projects in an iterative manner and helping teams work collaboratively and better-organized structure. Agile Scrum methodology has been used by many companies to bring high-end teamwork and efficiency to project-based work. Hence, Agile Scrum is based on Agile principles such as satisfying customer needs, and continuous amelioration along with the benefits of Scrum.. (Peek, S, 2023)

Even though Agile and Scrum are two distinct methods they can be worked individually. Nevertheless, their combination will bring a huge benefit either for the project or for the one who is working on the project (Peek, S., 2023). The tasks will be separated into small chunks within a specific timeframe which is known as a sprint. Hence, using Agile Scrum assists the project being built more quickly because the goals have been set to be done in the planned time and frequent planning and goal setting are required to make the scrum team focus on the current sprint's objectives. In other words, Agile Scrum not only helps the scrum team to catch up but also ensures that there is nothing they are missing, and they know what to be done in the backlog and what to be improved in the future, which is adaptable to change and maintenance (Peak, 2023). Therefore, Agile Scrum can be considered to be the most reasonable and best option for the project.

### 4.2 Overview of the Development Process and Tools

First and foremost, Azure DevOps, a product created by Microsoft for reporting, requirements management, and project management, was chosen to manage the project in order to practice the methodology mentioned above. Project was divided in one-week sprints, as shown in Figure 15. Also, each huge task was separated into small tasks.

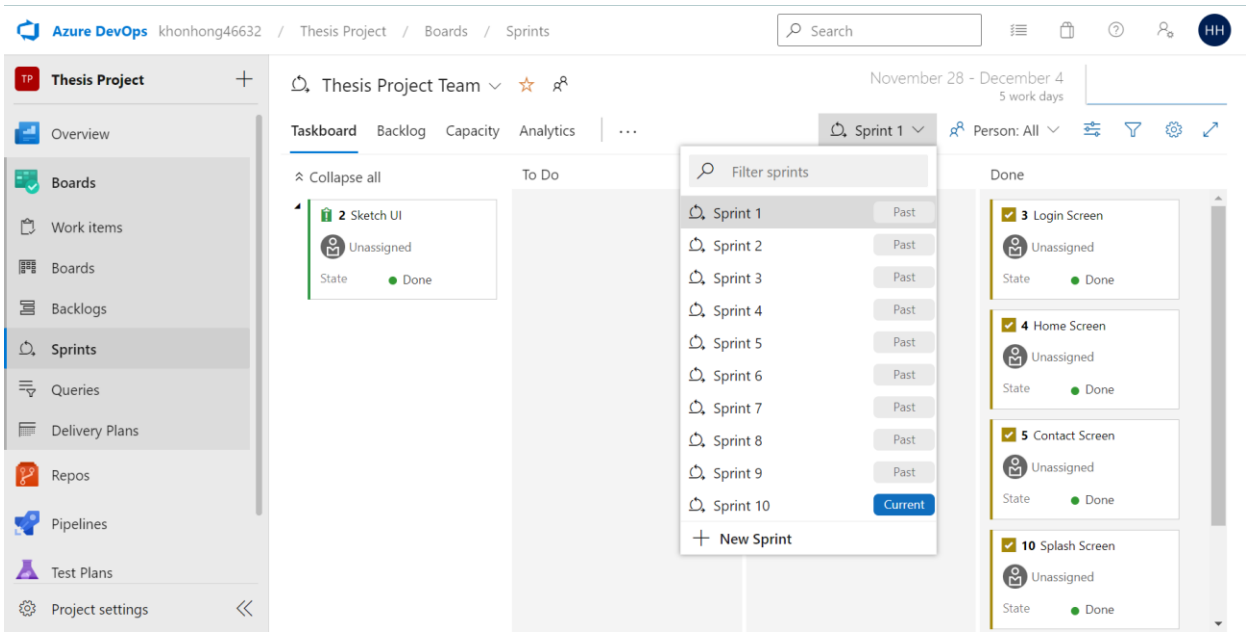



**Figure 15.** Azure DevOps

Afterward, a use-case diagram and flow chart were needed to make progress go further and know how the application would work depending on what state the application is in. Hence, the necessary features for the application were clarified clearly to ensure not to miss any features when building the application. In addition, use-case diagram shows what features are available for the basic users and what are for the administrators.

The application would not be usable without interfaces for the user to interact, with the design of the application has to be done after use-case diagrams and flow charts are defined.

The main tool of this project was the Flutter framework written in Dart. To start working with Flutter, the Flutter SDK, which stands for Software Development Kit, which provides tools used for programming or as operating system environments, can be installed into the computer by either cloning its repository from GitHub or its documentation website. There are two options for developers to install which are the stable and beta version. The stable version brings a reliable environment while the beta version provides new features and upgrades that are still under development. However, it is recommended by many Flutter developers to choose the stable version to avoid unexpected issues when coding. In order to debug on Android devices or iOS devices, Android Studio or XCode for Android and iOS devices is needed (Janson, T., 2022).

Moreover, Android SDK and Java JDK have to be included as environmental variables of the computer after installing Android Studio to ensure that Flutter applications can be installed and run smoothly on Android devices. Flutter provides a command which can be run on the terminal to check if the environment variable is set so that the Flutter runs correctly without any problems. Command is “flutter doctor”. Not only for environment variables, command can also check for possible new updates and provide more details information about platforms enabled to be run on Flutter by passing a parameter “-v”, which is shown in the picture below.

```
jack_wong@HONG:/mnt/d/Coding/Flutter/students_guide$ flutter doctor -v
[√] Flutter (Channel stable, 3.7.10, on Microsoft Windows [Version 10.0.19044.2728], locale
    en-GB)
    • Flutter version 3.7.10 on channel stable at D:\flutter
    • Upstream repository https://github.com/flutter/flutter.git
    • Framework revision 4b12645012 (9 days ago), 2023-04-03 17:46:48 -0700
    • Engine revision ec975089ac
    • Dart version 2.19.6
    • DevTools version 2.20.1

[√] Windows Version (Installed version of Windows is version 10 or higher)

Checking Android licenses is taking an unexpectedly long time...[√] Android toolchain - develop
 for Android devices (Android SDK version 32.0.0)
    • Android SDK at D:\flutter\Sdk
    • Platform android-33, build-tools 32.0.0
    • ANDROID_HOME = D:\flutter\Sdk
    • ANDROID_SDK_ROOT = D:\flutter\Sdk
    • Java binary at: D:\Android Studio\jre\bin\java
    • Java version OpenJDK Runtime Environment (build 11.0.12+7-b1504.28-7817840)
    • All Android licenses accepted.

[√] Chrome - develop for the web
    • Chrome at C:\Program Files (x86)\Google\Chrome\Application\chrome.exe

[√] Visual Studio - develop for Windows (Visual Studio Community 2019 16.11.9)
    • Visual Studio at D:\Visual Studio IDE
    • Visual Studio Community 2019 version 16.11.32106.194
    • Windows 10 SDK version 10.0.19041.0

[√] Android Studio (version 2021.2)
    • Android Studio at D:\Android Studio
    • Flutter plugin can be installed from:
      https://plugins.jetbrains.com/plugin/9212-flutter
    • Dart plugin can be installed from:
      https://plugins.jetbrains.com/plugin/6351-dart
    • Java version OpenJDK Runtime Environment (build 11.0.12+7-b1504.28-7817840)

[√] IntelliJ IDEA Community Edition (version 2021.3)
    • IntelliJ at D:\IntelliJ IDEA Community Edition 2021.3.1
    • Flutter plugin can be installed from:
      https://plugins.jetbrains.com/plugin/9212-flutter
    • Dart plugin version 213.5744.122

[√] VS Code (version 1.77.1)
    • VS Code at C:\Users\DELL\AppData\Local\Programs\Microsoft VS Code
    • Flutter extension version 3.62.0

[√] Connected device (4 available)
    • SM G970F (mobile) • RF8M63XDP4A • android-arm64  • Android 12 (API 31)
    • Windows (desktop) • windows     • windows-x64    • Microsoft Windows [Version
      10.0.19044.2728]
    • Chrome (web)      • chrome      • web-javascript • Google Chrome 111.0.5563.148
    • Edge (web)        • edge        • web-javascript • Microsoft Edge 111.0.1661.54

[√] HTTP Host Availability
    • All required HTTP hosts are available
```

**Figure 16.** Running "flutter doctor -v

The BloC design pattern, Firebase Authentication, and other Google services that are used in the project, need dependencies. Required dependencies are listed in pubspec.yaml file of the project and those dependencies or external packages can be found and understood by visiting the pub.dev website as known as the Dart programming language package manager. Required packages for project are shown in Figure 17. To get the packages utilized and take effect inside the project, “flutter pub get” command needs to be run. Also, this command can automatically be run when saving the pubspec.yaml file when using Visual Studio Code just by pressing Ctrl + S.

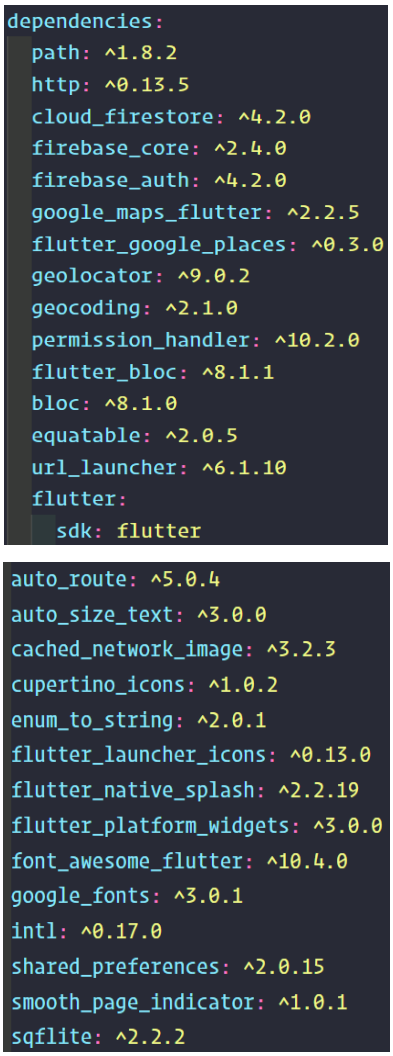



**Figure 17.** Flutter dependencies for the project

Using already-made, high-quality, and trusted libraries created by Google will be more convenient and less frustrating than spending too much time and effort creating complex libraries by self. Hence, these libraries push the process of building the application more rapidly. In addition, tools and services available from the packages make the application's animation or features easily created such as opening a web view or another app when needed, handling the users for permissions of accessing the users' current location and using local storage space memory to store favorite articles for the users.

## 5. IMPLEMENTATION

After creating the Flutter project, a clear and well-organized file structure is essential so that the application can be fixed and improved later on. First and foremost, the assets folder was created to store the necessary images and icons for the application. After that, model classes were created so that the application can manipulate the data by passing a class object.

In this project, there are only three classes, Admin, Article, and Category. The admin model is responsible for getting an admin email which can be passed for an Article instance to record who has added or modified the article in the Cloud Firestore database. Within the Article model class, there are factory constructors for converting the Article object into the suitable format of data for local and cloud databases and vice versa. Whereas, the Category model is used to generate a list of article categories for the home screen. Hence, the admins do not need to select any category every time they tap on the add button when viewing the articles belonging to a specific category.

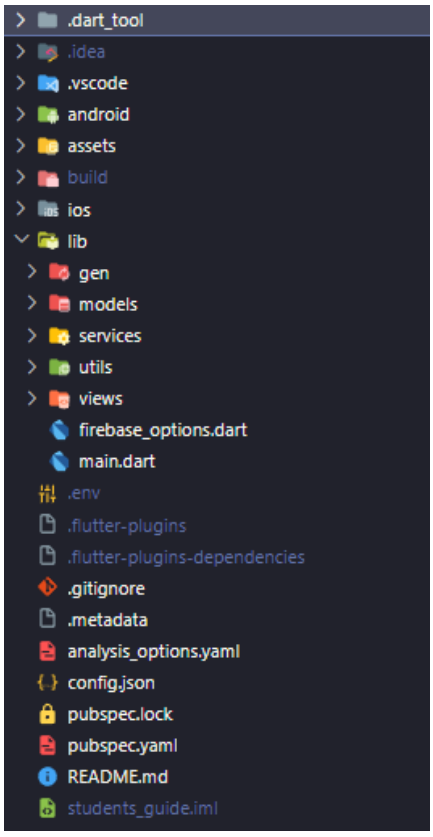



**Figure 18.** Overall view of the project's file structure

As shown in Figure 18, the files are organized in folders. The services folder is the folder containing files to apply the BLoC design pattern, methods for API calls, and local storage provided by Sqflite and SharedPreferences packages. Sqflite is a package that allows SQLite code implementation inside the application which is used to save starred articles for the users when they want to read them once again without choosing a specific category and find them again. Meanwhile, the Shared Preferences package provides a key-value store for saving and retrieving simple data locally on the device as we have mentioned above for saving and loading the theme preferences. The Utils folder stores customized widgets and immutable constants which can be used around the project. Done for the BLoC and data section, moving on to the UI section which is the Views folder, there are portioned widgets for each screen and the completed screens in the widgets and pages folder, respectively.

Without a clear file structure, maintaining and upgrading the application would be difficult. Also, thanks to the BLoC design pattern, there are always three layers which are the UI layer, BLoC, and the data layer. UI or widgets layer is a layer that is visible to the users. The repository or data layer manages the data and works with the back end. The BLoC is the intermediate layer between the UI layer and the data layer. In other words, the UI layer and data layer can only communicate with the BLoC layer separately. The BLoC layer claims events emitted from the UI layer and conducts business logic via the data layer and returns the output event to the UI layer (Islomov, S., 2022). Code for UI, bloc, and repository should not be merged on the same file which would be a bad practice, especially for those trying to maintain and update the code within the project.

In addition, the Bloc state of the application can be tested and observed along with Bloc Observer. In this project, there is a customized bloc observer that prints the current state of the application by assigning the observer instance as the customized bloc observer instance, which is illustrated in Figure 19. With the Bloc Observer, developers can check the current state of the application to avoid unauthenticated users manipulating the articles' data and only to star articles. Bloc Observer also helps developers follow the state change, transition, and debug if there are unexpected errors.

```dart
class CustomBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    log('${bloc.runtimeType} -- $event');
  }

  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    log('${bloc.runtimeType} -- $change');
  }

  @override
  void onCreate(BlocBase bloc) {
    super.onCreate(bloc);
    log('${bloc.runtimeType} -- $bloc');
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    log('${bloc.runtimeType} -- $transition');
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    super.onError(bloc, error, stackTrace);
    log('${bloc.runtimeType} -- $error, $stackTrace');
  }
}
```

**Figure 19.** Customized Bloc Observer

Figure 20 shows the printed logs on the debug console observing the state of the theme cubit. The application sets the system theme as default because of the theme cubit and then the theme mode is saved in the local storage so that the theme mode will be loaded and set automatically for the next usage. It also shows the state of the authentication bloc which makes sure the states transit correctly.

```
[log] ThemeCubit -- Instance of 'ThemeCubit'
[log] ThemeCubit -- Change { currentState: ThemeState(ThemeMode.system), nextState: ThemeState(ThemeMode.system) }
[log] AuthBloc -- Instance of 'AuthBloc'
[log] AuthBloc -- AuthEventInitialize()
[log] AuthBloc -- Transition { currentState: AuthStateUninitialized(true), event: AuthEventInitialize(), nextState: AuthStateSignedOut(false) }
[log] AuthBloc -- Change { currentState: AuthStateUninitialized(true), nextState: AuthStateSignedOut(false) }
```

**Figure 20.** Customized Bloc observer log

Widgets are essential components building up a complete application. They also can be customized. The customized widgets and tools are packaged under custom folder, as shown in Figure 21. Custom folder contains also for example theme settings for dark and light modes and a customized word limit format to ensure that the users can write in the given specific number of words in the contact screen.

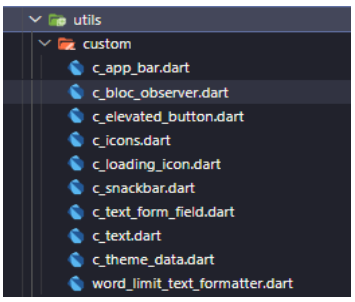
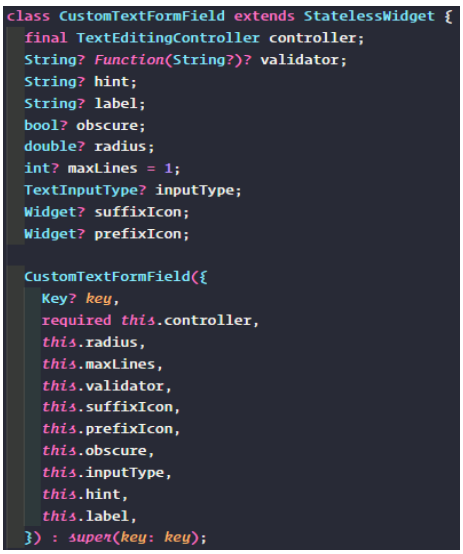



**Figure 21.** Custom folder

Due to the fact that there are widgets used recurrently that have many and the same properties in the project, it brings redundancy of the amount of the boilerplate code if there are customized widgets with pre-set properties. Not only that, this ensures monotony and consistent design across the widgets on the same screen. Additionally, creating reusable customized widgets promotes the reusability and maintainability of the codebase. Developers do not have to enter all the parameters for every widget iteratively which carries the same property settings.

For instance, in the adding or editing article screen or details screen, there are multiple text form fields having the same filled color, label text color, and other same properties when the users tap on the text form field. Customized widget is a crucial solution for the risk of missing needed passed arguments resulting in differences in design or colors between widgets.



```
class CustomTextFormField extends StatelessWidget {
  final TextEditingController controller;
  String? Function(String?)? validator;
  String? hint;
  String? label;
  bool? obscure;
  double? radius;
  int? maxLines = 1;
  TextInputType? inputType;
  Widget? suffixIcon;
  Widget? prefixIcon;

  CustomTextFormField({
    Key? key,
    required this.controller,
    this.radius,
    this.maxLines,
    this.validator,
    this.suffixIcon,
    this.prefixIcon,
    this.obscure,
    this.inputType,
    this.hint,
    this.label,
  }) : super(key: key);
```

**Figure 22.** Optional and required parameters of the customized text form field widget

The constructor of the customized text form field widget has been created as shown in Figure 22. The parameters containing the data types with question marks behind are the optional parameters, for instance, the text field can have a suffix icon or prefix icon. The text form field can also work as a one-line text field in default or a text area by passing a number of lines more than one to the maxLines property. Furthermore, the input keyboard can be varied namely phone, email, or normal input type, which helps the users to input information quicker. The only required parameter is the text editing controller because it is used to pass value when adding new articles or set the value from the existing articles when updating.

```dart
@override
Widget build(BuildContext context) {
  bool isLight = checkIfLightTheme(context);

  double radius = this.radius ?? 10;
  return Container(
    decoration: BoxDecoration(
      borderRadius: BorderRadius.all(Radius.circular(radius)),
      boxShadow: [
        BoxShadow(
          color: Colors.black26,
          blurRadius: radius,
          spreadRadius: 0.1,
          offset: const Offset(2, 1),
        ), // BoxShadow
      ],
    ), // BoxDecoration
    child: TextFormField(
      controller: controller,
      cursorColor: mColor,
      keyboardType: inputType,
      maxLines: maxLines ?? 1,
      obscureText: obscure ?? false,
      enableInteractiveSelection: true,
      validator: validator,
      style: TextStyle(color: isLight ? Colors.black38 : Colors.white54),
      decoration: InputDecoration(
        contentPadding: EdgeInsets.all(radius),
        filled: true,
        fillColor: isLight ? white : gray,
        enabledBorder: OutlineInputBorder(
          borderSide: const BorderSide(color: Colors.black12),
          borderRadius: BorderRadius.all(Radius.circular(radius)),
        ), // OutlineInputBorder
        focusedBorder: OutlineInputBorder(
          borderSide: const BorderSide(color: mColor),
          borderRadius: BorderRadius.all(Radius.circular(radius)),
        ), // OutlineInputBorder
        hintText: hint,
        // hintStyle: const TextStyle(colo r: Colors.black38),
        labelText: label,
        floatingLabelStyle: TextStyle(
            color: mColor,
            fontWeight: FontWeight.bold,
            backgroundColor: isLight ? lBackgroundColor : dBackgroundColor), // TextStyl
        labelStyle:
            TextStyle(color: isLight ? Colors.black38 : Colors.white54),
        prefixIcon: prefixIcon,
        prefixIconColor: white,
        suffixIcon: suffixIcon,
        suffixIconColor: white,
        alignLabelWithHint: true,
      ), // InputDecoration
    ), // TextFormField
  ); // Container
 }
}
```

**Figure 23.** Text form field customization

The customization of the text form field is shown in Figure 23. Without reusable and common customization, it would cause a disastrous codebase as it brings more code duplication for each widget on one screen that is needed to pass the same arguments just for its appearance. This customized widget is used not only in add and editing screen but also in the contact screen and login screen.

The biggest benefit of using customized widgets is that changes occur on one file only and are applied to all the presence of the text form fields of the application in no time. It is possible to use multiple customized widgets having the same appearance even though the number of the passed arguments is slightly different. An example within the add_edit_view.dart file is shown in Figure 24:

```
CustomTextFormField(
  inputType: TextInputType.url,
  controller: image,
  label: 'Image Link',
  validator: (value) {
    if (value!.isNotEmpty) {
      if (!value.validateUrl()) return 'Invalid Url';
    }
    return null;
  },
),
CustomTextFormField(
  controller: intro,
  label: 'Introduction',
  maxLines: 3,
  validator: (value) {
    if (value!.isEmpty) {
      return 'Please enter an introduction';
    }
    return null;
  },
),
CustomTextFormField(
  controller: info,
  label: 'Description',
  maxLines: 5,
  validator: (value) {
    if (value!.isEmpty) {
      return 'Please enter a description';
    }
    return null;
  },
),
CustomTextFormField(
  inputType: TextInputType.streetAddress,
  controller: address,
  label: 'Address',
),
```

**Figure 24.** Customized widgets implementation

As shown in Figure 24, even though CustomTextFormField is reused multiple times, it can pass distinct arguments, which proves that it not only brings the consistency of the design but also the flexibility of the design so that developers can set and alter the properties as they want. For example, several text form fields only have one line while some have more than one line. Also, there are different validators for the application to check if the administrators enter valid information or not.

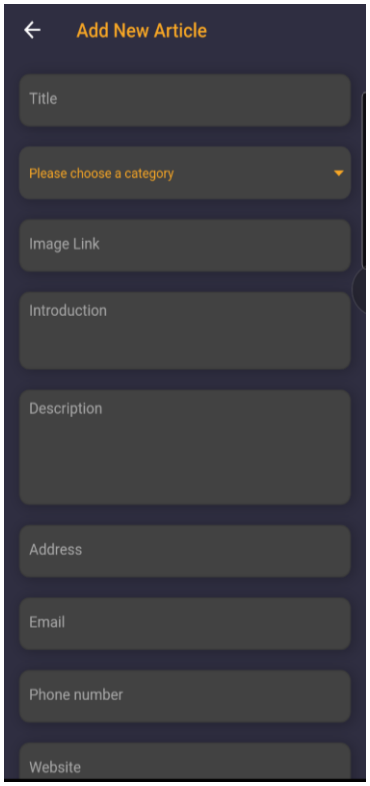



**Figure 25.** Add new article screen

The result showing the consistency and flexibility of the customized widget is shown in Figure 25. Additionally, it might be noticed that adding and editing article screen using the same widgets for their corresponding features instead of creating individual screens for each feature. The way to know the application is in creating a new article or updating an existing article case is that checking if the AddEditView widget which contains the mentioned customized widgets is passed an article as an argument or not. If there is no passed article into this view, all customized text box form fields of this screen are empty, and a new article will be created on the Google Cloud Firestore database once the admin enters all the necessary information and click the “Save & Close “button. Otherwise, text fields will be shown the information of the passed article and update the existing stored on the Firestore database based on the altered information.

Bloc Consumer is the combination of Bloc Listener and Bloc Builder from Bloc packages (Wogu, J., 2022). Bloc Consumer adds a listener to watch what the current AuthState is emitted in the application at the moment and return the corresponding result, for example, showing error dialog to handle invalid login or navigate from one to another screen when logging in. The builder section of the Bloc Consumer is used for building the widget UI also based on the emitted AuthState. For instance, if the user has completed entering the email and the password to login as an admin which causes the current auth state carrying a property called isLoading to change to another kind of auth state, the builder section will rely on the isLoading property to show a loading icon or the button to login the application. BloC Consumer is shown in Figure 26.

```
BlocConsumer<AuthBloc, AuthState>(
  listener: (context, state) async {
    if (state is AuthStateSignedOut) {
      if (state.exception is AdminNotFoundAuthException ||
          state.exception is WrongPasswordAuthException) {
        await showErrorDialog(
            context, 'Invalid Email or Password');
      } else if (state.exception is GenericAuthException) {
        await showErrorDialog(context, 'Unknown Error');
      }
    }
    if (state is AuthStateSignedIn) {
      Future.delayed(const Duration(milliseconds: 1),
          () => context.pushRoute(const HomeRoute())); // Future.delayed
    }
  },
  builder: (context, state) {
    return state.isLoading
        ? const CustomLoadingIcon()
        : CustomElevatedButton(
            onPressed: () async {
              if (_form.currentState!.validate()) {
                context.read<AuthBloc>().add(AuthEventLogin(
                    email.text.trim(), password.text)); // AuthEventLogin
              }
            },
            text: 'Login'); // CustomElevatedButton
  },
) // BlocConsumer
```

**Figure 26.** BLoC Consumer

## 6. TESTING

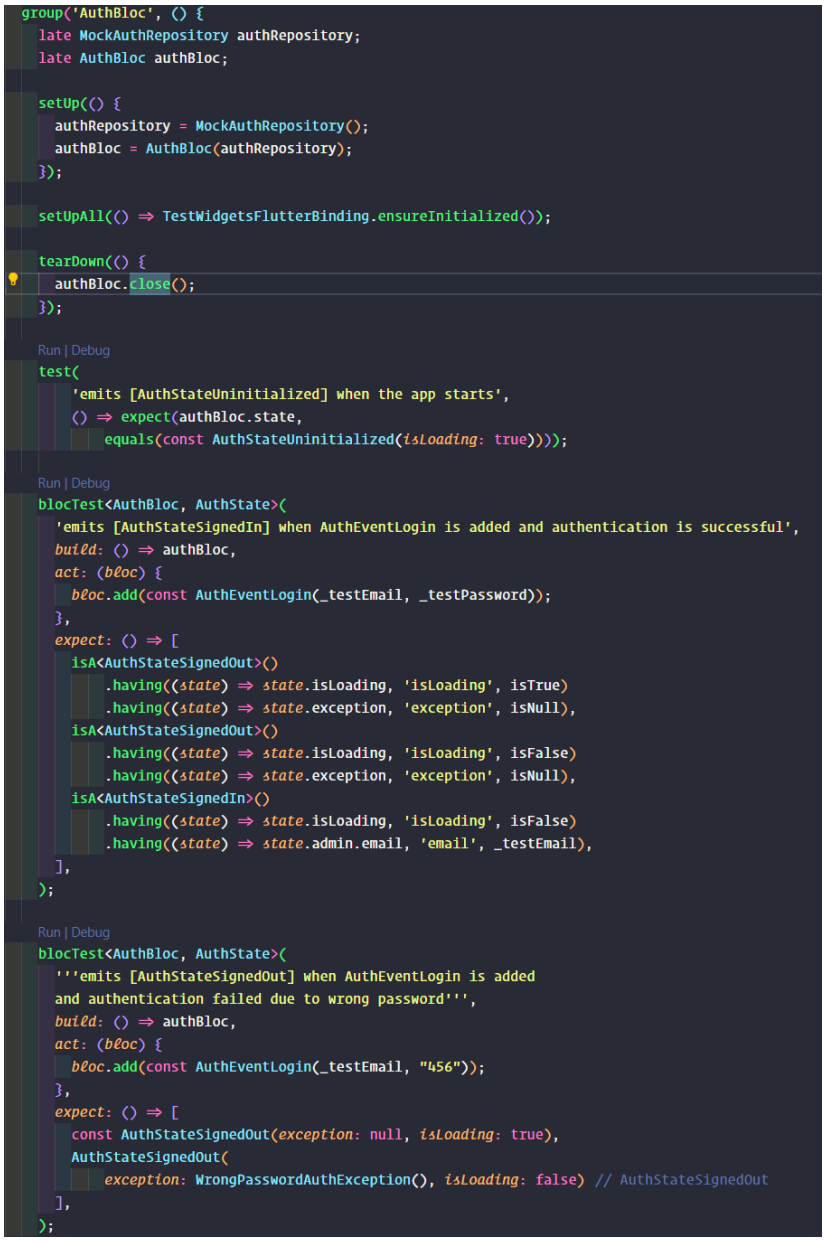
In order to test the Cubit for the application theming and BLoC for the authentication, BLoC offers testability as separating clearly business logic and UI components. Unit test for the BLoC can be written by utilizing mocking events and checking the emitted states.

State and theme management in an application are extremely necessary to bring comfortable user experience to the consumers. Hence, within the project, there are several unit tests when implementing BLoC pattern for the application with the help of Flutter_test and Bloc_test libraries.

```dart
void main() {
  Run | Debug
  group('ThemeCubit', () {
    late ThemeCubit themeCubit;

    setUpAll(() => TestWidgetsFlutterBinding.ensureInitialized());

    setUp(() {
      themeCubit = ThemeCubit();
    });

    tearDown(() {
      themeCubit.close();
    });

    Run | Debug
    test(
      'emits [ThemeState(system)] as initial theme state',
      () => expect(
          ThemeCubit().state, const ThemeState(themeMode: ThemeMode.system)),
    );

    Run | Debug
    blocTest<ThemeCubit, ThemeState>(
      'emits the corresponding ThemeState when setTheme is called',
      build: () => themeCubit,
      act: (cubit) => cubit.setTheme(ThemeMode.dark.index),
      expect: () => [
        const ThemeState(themeMode: ThemeMode.dark),
      ],
    );

    Run | Debug
    blocTest<ThemeCubit, ThemeState>(
      'emits the saved ThemeState when getTheme is called',
      build: () => themeCubit,
      act: (cubit) {
        cubit.setTheme(1);
        themeCubit.close();
        ThemeCubit().getTheme();
      },
      expect: () => [
        const ThemeState(themeMode: ThemeMode.light),
      ],
    );
  });
```

**Figure 27.** Tests for Theme Cubit

In Figure 27, there are test cases for Cubit's theme management. Each test case has its own description, code procedure and result expectation. For example, the last test case within the Figure 27, whose actions including calling the setTheme() method at first to set and save the selected theme. In this case, the selected theme is light theme as it is located at the first index of the enumerated list called enum. After that, the current instance of ThemeCubit will be closed and a new instance of ThemeCubit is created to get the saved theme. The expect property asks for a list of states which will be emitted during the methods are called in the act property.



```dart
group('AuthBloc', () {
  late MockAuthRepository authRepository;
  late AuthBloc authBloc;

  setUp(() {
    authRepository = MockAuthRepository();
    authBloc = AuthBloc(authRepository);
  });

  setUpAll(() => TestWidgetsFlutterBinding.ensureInitialized());

  tearDown(() {
    authBloc.close();
  });

  Run | Debug
  test(
      'emits [AuthStateUninitialized] when the app starts',
      () => expect(authBloc.state,
          equals(const AuthStateUninitialized(isLoading: true))));

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    'emits [AuthStateSignedIn] when AuthEventLogin is added and authentication is successful',
    build: () => authBloc,
    act: (bloc) {
      bloc.add(const AuthEventLogin(_testEmail, _testPassword));
    },
    expect: () => [
      isA<AuthStateSignedOut>()
          .having((state) => state.isLoading, 'isLoading', isTrue)
          .having((state) => state.exception, 'exception', isNull),
      isA<AuthStateSignedOut>()
          .having((state) => state.isLoading, 'isLoading', isFalse)
          .having((state) => state.exception, 'exception', isNull),
      isA<AuthStateSignedIn>()
          .having((state) => state.isLoading, 'isLoading', isFalse)
          .having((state) => state.admin.email, 'email', _testEmail),
    ],
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    '''emits [AuthStateSignedOut] when AuthEventLogin is added
    and authentication failed due to wrong password''',
    build: () => authBloc,
    act: (bloc) {
      bloc.add(const AuthEventLogin(_testEmail, "456"));
    },
    expect: () => [
      const AuthStateSignedOut(exception: null, isLoading: true),
      AuthStateSignedOut(
          exception: WrongPasswordAuthException(), isLoading: false) // AuthStateSignedOut
    ],
  );
```

```
  blocTest<AuthBloc, AuthState>(
    'emits [AuthStateSignedOut] when AuthEventLogout is added',
    build: () => authBloc,
    act: (bloc) => bloc.add(AuthEventLogout()),
    expect: () =>
        [const AuthStateSignedOut(exception: null, isLoading: false)],
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    'emits [AuthStateResetPassword] when AuthEventResetPassword is added',
    build: () => authBloc,
    act: (bloc) => bloc.add(const AuthEventResetPassword(_testEmail)),
    expect: () => [
      const AuthStateResetPassword(
          exception: null, isLoading: true, emailSent: false),
      const AuthStateResetPassword(
          exception: null, emailSent: true, isLoading: false)
    ],
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    '''emits [AuthStateResetPassword] having AdminNotFoundAuthException
     when AuthEventResetPassword is added but the mail is not found''',
    build: () => authBloc,
    act: (bloc) => bloc.add(const AuthEventResetPassword("test1@gmail.com")),
    expect: () => [
      isA<AuthStateResetPassword>()
          .having((state) => state.isLoading, 'isLoading', isTrue)
          .having((state) => state.emailSent, 'isEmailSent', isFalse)
          .having((state) => state.exception, 'exception', isNull),
      isA<AuthStateResetPassword>()
          .having((state) => state.isLoading, 'isLoading', isFalse)
          .having((state) => state.emailSent, 'isEmailSent', isFalse)
          .having((state) => state.exception, 'exception',
              isA<AdminNotFoundAuthException>()),
    ],
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    '''emits [AuthStateResetPassword] having InvalidEmailAuthException
    when AuthEventResetPassword is added but invalid Email''',
    build: () => authBloc,
    act: (bloc) => bloc.add(const AuthEventResetPassword("test.com")),
    expect: () => [
      isA<AuthStateResetPassword>()
          .having((state) => state.isLoading, 'isLoading', isTrue)
          .having((state) => state.emailSent, 'isEmailSent', isFalse)
          .having((state) => state.exception, 'exception', isNull),
      isA<AuthStateResetPassword>()
          .having((state) => state.isLoading, 'isLoading', isFalse)
          .having((state) => state.emailSent, 'isEmailSent', isFalse)
          .having((state) => state.exception, 'exception',
              isA<InvalidEmailAuthException>()),
    ],
  );
});
```

**Figure 28.** Test cases for AuthBloc

For BLoC, an event is needed for the act property and a list of states is expected in order once the needed event is added as shown in Figure 28. The “isA” matcher is extremely helpful for asserting the instances of the AuthState. Moreover, “having” matcher can be used along with “isA” to ensure the property value of the instance same as expected.

For the last test case shown in Figure 28 as an example, the act property adds the reset password event to the bloc with an email in invalid format. The expect property determines the expected sequence of states emitted by the bloc. Even though there is only the same type of state which is AuthStateResetPassword, they carry distinct property values. The first one carries true value of isLoading property and no exception but the second one contains an exception which is InvalidEmailException and isLoading returns to false.

After writing several test cases to ensure that the application works normally relating to authentication and theming, the tests can be run and revealed by using “Text Extension UI” extension on Visual Studio Code or simply clicking on the run test button next to the test cases or group of test cases. The test results of ThemeCubit and AuthBloc that ensures whether they are performing correctly or not are shown in Figure 29. The passed tests will be ticked in green next to the test case description. Otherwise, there will be red crosses if the actual results are not as same as the expected results.

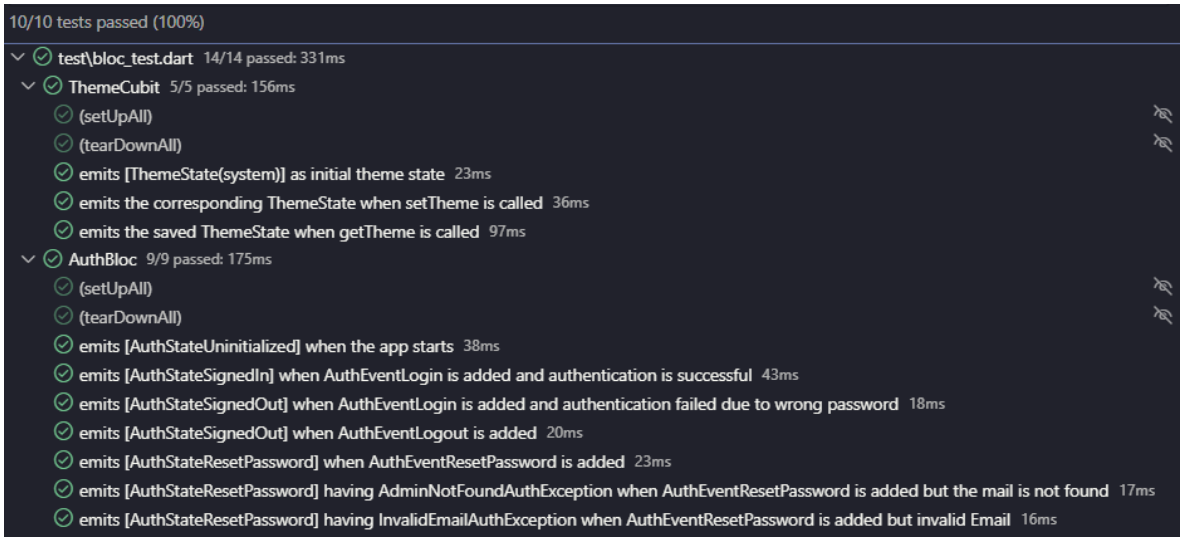



**Figure 29.** Test results

# 7. CONCLUSION

As conclusion, Flutter is undoubtedly a useful tool to build cross-platform mobile applications instead of programming different codebases for Android and iOS devices. Having the ability to create an application that can run on any device platform, the importance of Flutter's existence might be noted and allured later on. Flutter might not be an ideal tool at the moment to build big-scale projects which might bring lagging and low performance especially when the project is getting bigger but hopefully, Google upgrades Flutter more in the near future. Even though the complexity of the BLoC design pattern, the file structure of the Flutter project becomes more straightforward and neater.

With the help of the Agile Scrum methodology, the project was completed before the given deadline. Hence, it can be considered a finished project and treated as a usable application but there are still some improvements that can be taken place on this application to bring it to the best performance and usage. The developed application is a prototype, whose functionalities are tested using mobile devices.

Even though author did not have much experience with UI/UX design in advance, during the project sketching the UI design with Figma was learned. Application is simple to use and user-friendly for the users.

While building the project, there were several problems and challenges encountered and needed to resolve. The biggest problem was related to Google Maps. The billing account on Google Cloud could not be created, which most probably was because of banking issues. Open Route Services was used as an alternative to the Direction API of Google.

From this project, the author has a clearer understanding of the design pattern so that the application can be improved in the near future. Even though this application is simple and easy for some professional developers, it is helpful for authors' future careers by allowing apply the knowledge of Flutter. Regardless of the challenges and the shortage of finance, the project was finally completed into a functional application in the given timestamp.

**REFERENCES**


Atlassian (2023) What is Agile? Accessed: 17.04.2023 https://www.atlassian.com/agile

Cherednichenko, Sveta. (2023) How to implement the Bloc architecture in flutter: Benefits and best practices, Mobindustry. Accessed: 17.04.2023. https://www.mobindustry.net/blog/how-to-implement-the-bloc-architecture-in-flutter-benefits-and-best-practices/.

Flutter - introduction (2023) Tutorials Point. Accessed: 10.04.2023. https://www.tutorialspoint.com/flutter/flutter_introduction.htm#

Flutter documentation (2023) Flutter. Accessed:10.04.2023. https://docs.flutter.dev/.

Flutter Documentation (2023) Hot reload, Flutter. Accessed: 17.04.2023. https://docs.flutter.dev/development/tools/hot-reload.

Goswami, R. (2023) Bloc v/s cubit in Flutter application. Medium. FlutterDevs. Accessed: 17.04.2023. https://medium.flutterdevs.com/bloc-v-s-cubit-in-flutter-application-6caa2825b26a.

Islomov, S. (2022) Getting started with the Bloc pattern. Accessed: 17.04.2023 https://www.kodeco.com/31973428-getting-started-with-the-bloc-pattern

Janson, T. (2022) How to install and configure flutter SDK on Windows 10, Liquid Web. Accessed: 17.04.2023 https://www.liquidweb.com/kb/how-to-install-and-configure-flutter-sdk-windows-10/.

Pathik. (2021). 7 reasons to choose google cloud firestore as your database solution, BlueWhaleApps. Accessed: 17.04.2023 https://bluewhaleapps.com/blog/7-reasons-to-choose-google-cloud-firestore-as-your-database-solution

Peek, S. (2023) What is agile scrum methodology? Business News Daily. Accessed: 17.04.2023. https://www.businessnewsdaily.com/4987-what-is-agile-scrum-methodology.html

Rohan, A. (2020) Drawing route direction in flutter using openrouteservice API and google maps in Flutter. Accessed: 17.04.2023 https://medium.com/@rohanarafat86/drawing-route-direction-in-flutter-using-openrouteservice-api-and-google-maps-in-flutter-4431a2989dd5.

Rosencrance, L. (2019) What is Google Firebase? Definition from TechTarget, Mobile Computing. TechTarget. Accessed: 17.04.2023. https://www.techtarget.com/searchmobilecomputing/definition/Google-Firebase.

Sharma, S. (2020) The growth of flutter development- 3years after the birth of alpha. Accessed: 17.04.2023 https://medium.flutterdevs.com/the-growth-of-flutter-development-3years-after-the-birth-of-alpha-78baee809dff.

Wogu, J. (2022) Flutter bloc made easy. Accessed: 17.04.2023 https://medium.com/@Ikay_codes/flutter-bloc-made-easy-d244dd369e9a.