MSc in Computer Science and Engineering

Design and Implementation of Mobile Applications

CovTrack Software Design Document

Professor:
Luciano Baresi

Author:
Marco Gelli - 901470

CovTrack Software Design Document

Design and Implementation of Mobile Applications

prof. Luciano Baresi

Marco Gelli - 901470

https://github.com/gellaz/covtrack

A.A. 2019/2020

Contents

0 Document Structure

This document is structured as follows:

Section 1: Introduction. This section introduces the Design Document. It explains the purpose, the scope and the conventions used in the document.

Section 2: General Overview. This section contains a general overview of the project. In this section, the reader could find the core features of the application and the requirements of the system.

Section 3: Architectural Design. This section describes the components used for the system and the relations between them, providing information about their deployment and how they works. It also specifies the architectural styles and the design patterns chosen to design the system.

Section 4: User Interface Design. This section contains the screenshots of the application with some comments to give to the reader a general overview of the user interfaces.

Section 5: External Services & Packages. This section aims to explain the main frameworks, external services and packages, used to make our application functional.

Section 6: UML Diagrams. This section contains a series of UML diagrams that will be used by the reader to better understand how the system works.

Section 7: Testing. This section contains the list of test cases with their relative outcome, performed to ensure the robustness of the system.

At the end of the document the reader can find the reference and an appendix containing all the software and tools used and the revision history of this document.

1 Introduction

1.1 Purpose

The purpose of this document is to give a detailed description of the design of *CovTrack*, a cross-platform mobile application. The main goal is to provide an overall analysis of the product architecture.

The next sections will focus on the implementation of the application, providing insights into the structure and design of each component and in-depth details of all the internal and external services that the system offers and needs in order to reach the given goals.

This document is addressed more specifically to the developers, managers, testers and system administrators that will have to implement, manage and maintain the system.

1.2 Scope

CovTrack is a mobile application which aims to support regional and state authorities in the fight against the spread of the Covid-19 virus through the voluntary monitoring of citizens' movements. To achieve this goal, the application will completely replace the paper version of the mandatory self-certification form introduced in Italy during the first phase of the pandemic.

1.3 Definitions, Abbreviations and Acronyms

1.3.1 Definitions

- **User**: any client of the service, a person the logs in the system and uses it.
- **Authority**: a police officer or any other person in charge of controlling citizens' movements.
- **Trip**: the movement of a citizen which must be justified by a valid reason.
- **Stop**: geographical point where a user stops (is stationary).

1.3.2 Abbreviations

- **Dn**: n-th domain assumption.
- **Gn**: n-th goal.
- **Rn**: n-th functional requirement.

1.3.3 Acronyms

- **API**: Application Programming Interface.
- **BaaS**: Backend-as-a-Service.
- **BLoC**: Business Logic Component.
- **GPS**: Global Positioning System.
- **JSON**: JavaScript Object Notation.
- **OS**: Operating System.
- **REST**: Representational State Transfer.
- **SDK**: Software Development Kit.
- **UI**: User Interface.
- **UUID**: Universally Unique Identifier.
- **UX**: User Experience.

2 General Overview

2.1 Concept

The idea behind *CovTrack* is very simple: before each trip, the user must enter the starting point, the destination and the reason for the trip. In case an authority needs to verify the validity of the trip, he will be able to scan a QR Code identifying the currently active trip directly from user's device, making the verification process much faster and more efficient than the paper system. In addition to that, each time the user stops (stationary) during an active trip, the application will save the coordinates and the visiting time of the place.

When an active trip ends, all the collected data are saved in a remote database: in this way statistics, analysis and data processing may be carried out and in case a user is found positive for Covid-19 virus, it will be possible to trace his previous movements and any meeting with other people who stopped in the same place at the same time.

The idea for *CovTrack* came to me during the lockdown period: I thought that an app like this could be an effective solution in the fight against the spread of the Covid-19 virus, because it helps to speed up the monitoring process and allows a fine-grained control over citizens' movements.

2.2 Stakeholders

The people who would be most interested in this product could be:

- **Citizens**: citizens will constitute the most relevant percentage of the user base.
- **State and Regional Authorities**: the application will help the authorities simplifying the pandemic containment operations.
- **Medical Staff**: doctors and nurses will be able to use the tracking data to reconstruct the patient's contacts.
- **Epidemiologists & Data Analysts**: thanks to the collected data it will be possible for these experts to carry out more accurate analyses on the evolution of the pandemic.
- **Developers**: developers may help extending the application by adding new features and maintaining it.
- **System Administrators**: system administrators could handle the management and maintenance of the server-side architecture.

2.3 Goals

- **[G1]**: *CovTrack* must allow users to manage trips: create a trip to a new destination, create a trip to a previously visited destination, browse a list containing all the reached destinations and consult information about an active trip.
- **[G2]**: *CovTrack* must be able to collect all information on users' movements and stops and store them in a remote database.
- **[G3]**: *CovTrack* must be able to provide the user with up-to-date statistics regarding the pandemic situation in the country where the user is located and in the world.

2.4 Domain Assumptions

- **[D1]**: Users' devices support the GPS technology.
- **[D2]**: Users' devices always have an internet connection available during the interaction with the system.
- **[D3]**: User's devices support the mobile application (including Google Maps services).

2.5 Functional Requirements

In this section I provide a list of all the functional requirements necessary to ensure a correct behavior of the application. I decided to divide them considering the sections in which the application is organized, allowing the reader to easily understand where they could be found.

2.5.1 Authentication Requirements

- **[R1]**: Allow users to authenticate using an email and password combination.
- **[R2]**: Allow users to authenticate using their Google accounts.
- **[R3]**: Allow users to register themselves in the application using an email and password combination.

2.5.2 Home Requirements

- **[R4]**: Allow users to create a new trip by picking the destination on a map or from a list of suggested places.
- **[R5]**: Allow users to choose the motivation for the trips they create.
- **[R6]**: Allow users to check the information about an active trip: source, destination, reason, number of stops, elapsed time.
- **[R7]**: Allow users to show the QR Code identifying an active trip.
- **[R8]**: Allow users to end an active trip and/or start the corresponding return trip (source and destination swapped).
- **[R9]**: Allow users to browse a list of all the destinations reached in the past and to create a new trip with the same destination as one of the past ones.

2.5.3 News Requirements

- **[R10]**: Allow users to check the latest news regarding the pandemic situation in his country and around the world.

2.5.4 Settings Requirements

- **[R11]**: Allow users to be able to change their login password.
- **[R12]**: Allow users to be able to delete their account and all the registered data associated to it.

2.6 Non-Functional Requirements

This is a list of all the non-functional requirements necessary to ensure the application works properly:

- **Reliability**: the system must be available 24/7. Since this requirement is quite strict, small service down-times will be tolerated.
- **Availability**: the services must be always up and running. In case of failure it must be restored as soon as possible.
- **Portability**: to facilitate user adoption, the application must be cross-platform and will be available for both Android and iOS devices.

- **Security**: The data managed by the application must be as reliable as possible: no malicious user must be able to corrupt the data or snort the traffic. For this reason the database must be protected and secure and communication should be encrypted.
- **Scalability**: The architecture of the system must be simply scalable as the number of users grows over time.
- **Extensibility**: the application must be developed trying to keep a simple structure in order to allow further extensions easily.
- **Maintainability**: to make the software as maintainable as possible, the code must follow a modular design. In this way, various system components can be modified and others can be introduced into the system when necessary and with minimal impact on the rest of it. In addition, standard design models must be adopted and coded best practices, so that our code can be easily understood and modified by any future developer. Finally, the code should be fully and adequately documented to facilitate understanding.
- **Internationalization**: to facilitate adoption, the application must offer multi-language support. The language of the application must be chosen based on device's language and if not supported yet, English is chosen as the default language. For now only two languages are supported: Italian and English.
- **Usability**: the application must be developed to make the UI as simple as possible keeping all the functionalities needed to provide the best UX.
- **Nice User Interface**: the application must be as appealing as possible, for this reason I developed it following Google's Material Design guidelines to have a clean and simple design.

3 Architectural Design

3.1 System Architecture

CovTrack can be divided into two main sub-systems: a Front-End and a Back-End.

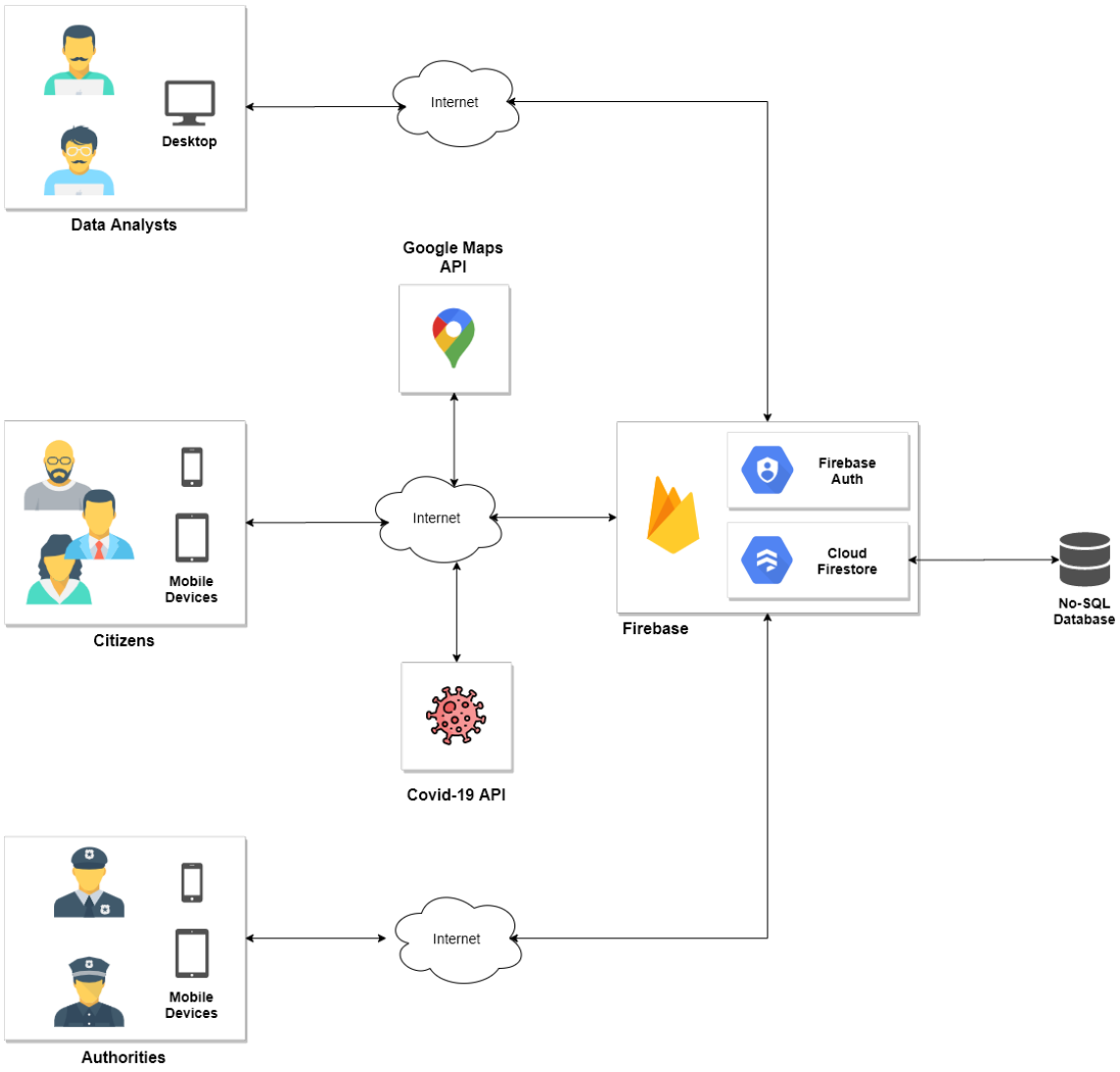


Figure 1: System Architecture Diagram

The Front-End consists in a series [1] of mobile and desktop applications, in particular:

- A cross-platform mobile application used by citizens to manage their trips.
- A cross-platform mobile application used by authorities to check citizens' trips.

[1] In this project I only developed the mobile application used by citizens. In this section I simply want to give an overview of how the system could be composed in its entirety

- A desktop application used by data analysts to analyze trips data.

The mobile application used by citizens must also interact with some third party services: Google Maps API to manage maps and receive information on destinations and the Covid-19 API to receive information on the pandemic.

On the other hand the Back-End is the server part of the system. The main role of this sub-system is to manage the database containing trips data. This sub-system relies on Firebase and in particular on Cloud Firestore, Firebase's No-SQL database.

3.2 Front-End: Mobile Application

The mobile application is a Flutter application composed by several screens that will be described in section 4. On the other hand, in this section I will present the chosen development framework and patterns used for the development of the mobile application.

3.2.1 Development Framework

Developing a production-level application paradoxically means developing two applications: one for iOS devices and one for Android devices. For this reason, updating and maintaining the application over time, means changing the functionality of two totally different codebases, increasing the development costs over the long term and thus making it extremely difficult to address both iOS and Android markets.

Considering that the target of my application is an open set (the entire market), this approach is not feasible. For this reason, I decided to adopt Flutter as my development framework, in order to reach the largest number of users and containing the development costs.

Another reason that led me to choose Flutter is its great portability. As stated on Flutter's homepage:

"*Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.*"

Flutter has managed to emerge almost thanks to the same slogan which determined Java's success: "*write once, run anywhere*" and even if it uses Dart as programming language (which is not yet widespread), the extreme portability will make it easier to develop a web or desktop version of *CovTrack* in the future.

At last, I wanted to learn a new technology to expand my knowledge and since the adoption of this framework is growing very quickly I think this project is a good opportunity to learn it.

3.2.2 State Management & BLoC

Native development frameworks (e.g. Android and iOS) typically use an *imperative* style of UI programming, where you have to manually construct a full-functioned UI element and later, when there is a change, explicitly mutate it using methods and setters.

On the other hand, Flutter is a *declarative* UI framework entirely based on the concept of **state**: it builds the UI to reflect the current state of the app. When there is a change in the state of the app, that triggers a redraw of the UI. There is no imperative changing of the UI itself, you change the state and the UI rebuilds from scratch.

Figure 2: Declarative UI scheme

Flutter has a built-in way of managing the state: the `setState` method. However, as the size of the application grows, there comes a time when you need to share the application state between different widgets (UI elements). For this reason, it becomes increasingly difficult to manage the state, making the app not scalable.

In order to make the development process easier there are several *state management patterns* and among these I have chosen BLoC. Using BLoC allowed me to separate the application into three layers:

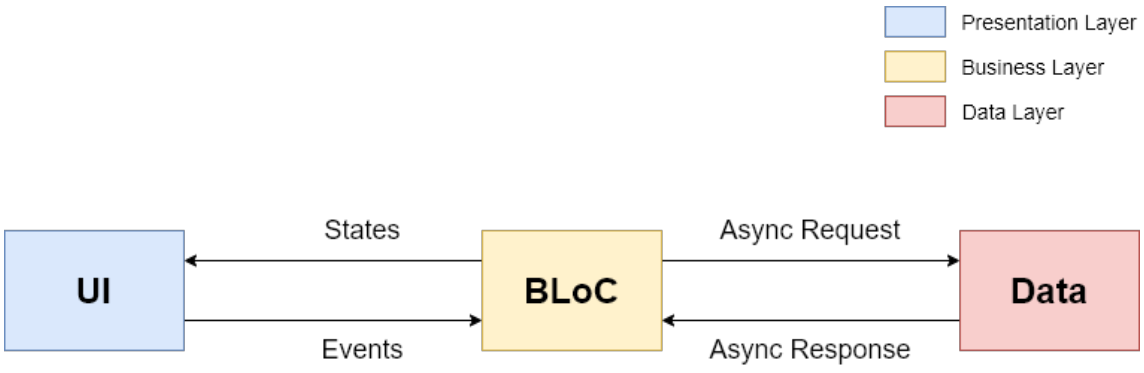


Figure 3: BLoC pattern architecture

- **Data Layer**: lowest level of the application. The responsibility of this layer is to retrieve/manipulate data from one or more asynchronous data sources (e.g. databases, APIs, ...). It can be further divided into two parts:

  - **Data Provider:** responsible for providing raw data.
  - **Repository:** wrapper around one or more data providers with which the BLoC layer communicates.

- **BLoC/Business Layer**: bridge between the UI (Presentation Layer) and the Data Layer. The responsibility of this layer is to map the events coming from the presentation layer with new states: it takes the events generated by user input and then communicates with repository in order to build a new state for the presentation layer to consume. This layer can depend on one or more repositories to retrieve data needed to build up the application state.

- **Presentation Layer**: this layer consists of all the application screens and the widgets that make up each screen. The responsibility of this layer is to figure out how to render itself based on one or more BLoC states. In addition, it should handle user input and application lifecycle events.

3.2.3 Why BLoC?

BLoC makes it easy to separate presentation from business logic. This leads to some obvious advantages:

- Code modularity and re-usability.
- Easily testable.
- The overall layered architecture makes it easy to swap in and out sections. For example switching from a Firebase provider to a similar REST API provider can be easily done.

3.3 Back-End: Firebase

The Back-End will be managed using Firebase, an external service provided by Google. It's a development platform that follows the BaaS paradigm that provides several services (e.g. authentication, real-time database, cloud storage, ...). In particular, two of these services will be used: Firebase Authentication and Cloud Firestore.

3.3.1 Firebase Authentication

This service handles all the authentication procedures for the users: registration, login, password change and account deletion. Before being able to use the service, a preliminary configuration is required, where you have to register the app and specify

which authentication methods will be used. In my case I only selected authentication with email and password, and authentication via a Google account.

3.3.2 Cloud Firestore

This service is of fundamental importance: it provides the application with a cloud-hosted, non-relational database with live synchronization, feature that allows us to rebuild the UI in real time when the data stored in the database is updated. All trips data and destinations data will be stored here.

In particular, following Cloud Firestore's No-SQL data model, there will be a root collection called `users` which will contain a document for each user registered in the system with at least one trip done. Each of these documents is automatically identified by the `uid` (a.k.a. user ID), a unique string automatically assigned by the Firebase Authentication service upon registration.

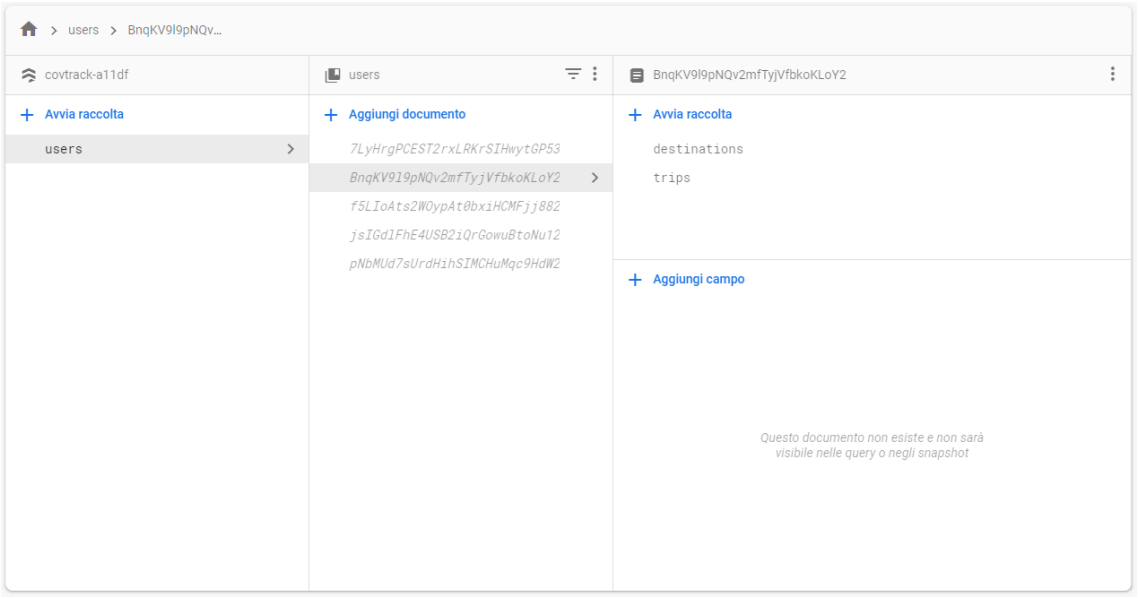


Figure 4: Cloud Firestore `users` collection

Each of the user documents contains two sub-collections: a `trips` sub-collection and a `destinations` sub-collection.

The first one will contain the documents with all the data about the trips completed by the user over time. Each document will be identified by a unique string ID that is automatically assigned by Cloud Firestore at the time of insertion.

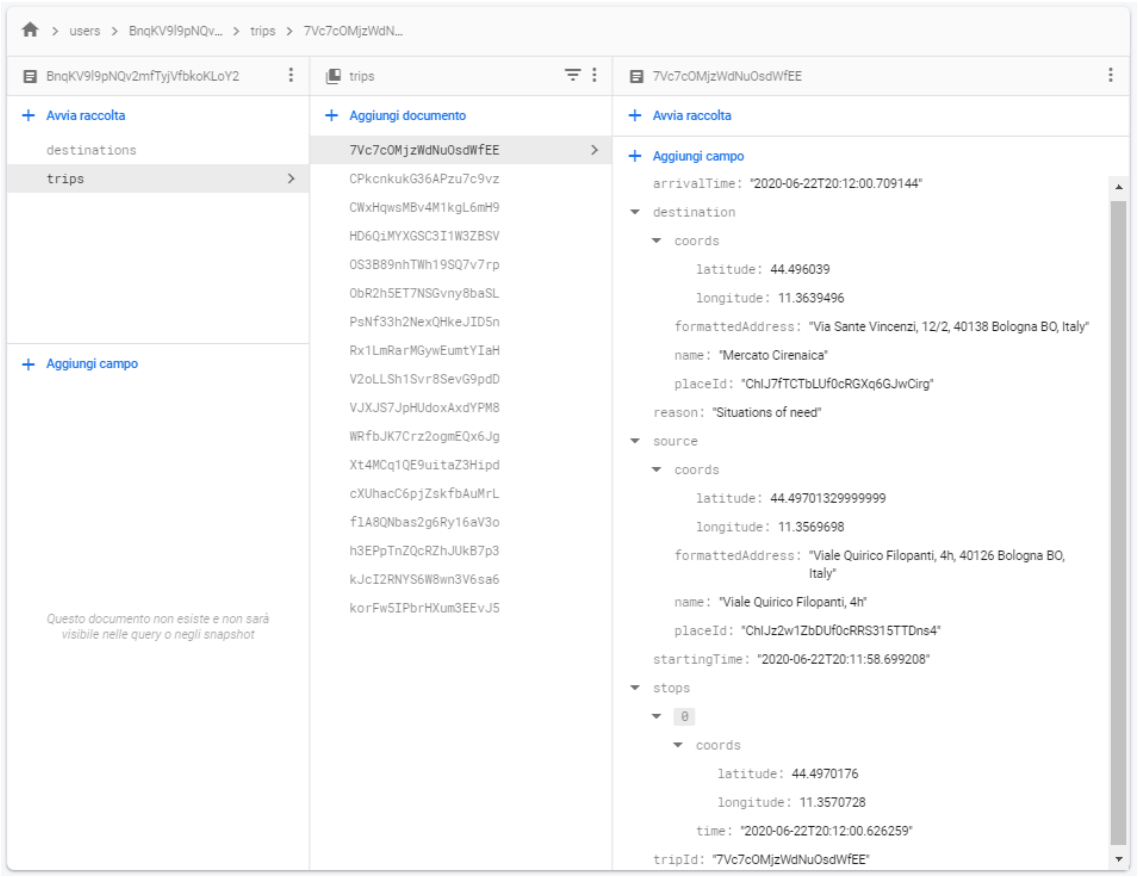


Figure 5: Cloud Firestore `trips` collection

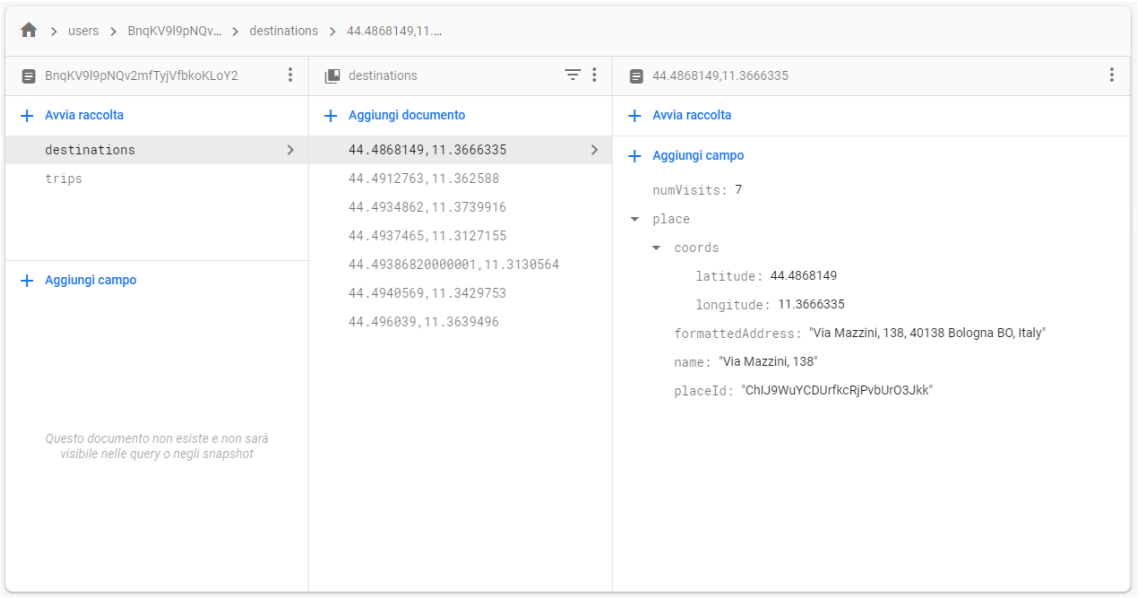


Figure 6: Cloud Firestore `destinations` collection

On the other hand, the `destinations` sub-collection can be considered as a view built from the `trips` sub-collection. In fact, only the destinations reached by the user will be stored in this collection. Each destination document is identified by a string obtained by concatenating the latitude and longitude of the destination and thus obtaining the following structure: `"<destination_latitude>,<destination_longitude>"`. I decided to use this as a unique identifier because each geographical point is uniquely identified by its coordinates.

Both types of document contained in the two sub-collections are structured as JSON. The two example listings below will help the reader to better understand how the data is structured.


```
{
    "startingTime": "2020-06-22T20:11:12.090004",
    "source": {
        "coords": {
            "latitude": 44.49701329999999,
            "longitude": 11.3569698
        },
        "formattedAddress": "Viale Umbria, 4h, 40126 Bologna BO, Italy",
        "name": "Viale Quirico Filopanti, 4h",
        "placeId": "ChIJz2w1ZbDUf0cRRS315TTDns4"
    },
    "arrivalTime": "2020-06-22T20:11:14.333962",
    "destination": {
        "coords": {
            "latitude": 44.4912763,
            "longitude": 11.362588
        },
        "formattedAddress": "Via Verdi, 13, 40138 Bologna BO, Italy",
        "name": "Ospedale Sant'Orsola",
        "placeId": "ChIJC_LSuE4rfkcRbIj8KPoTn68"
    },
    "reason": "Health reasons",
    "stops": [
        {
            "coords": {
                "latitude": 44.497016,
                "longitude": 11.3570698
            },
            "time": "2020-06-22T20:11:13.997942",
            "tripId": "CPkcnkukG36APzu7c9vz"
        }
    ]
}
```


Listing 1: Trip document's structure

```
{
    "numVisits": 7,
    "place": {
        "coords": {
            "latitude": 44.4868149,
            "longitude": 11.3666335
        },
        "formattedAddress": "Via Mazzini, 138, 40138 Bologna BO, Italy",
        "name": "Via Mazzini, 138",
        "placeId": "ChIJ9WuYCDUrfkcRjPvbUrO3Jkk"
    }
}
```

Listing 2: Destination document's structure

3.4 Third Party Interaction

In order to be functional, *CovTrack* must interact with numerous external services. These services will be described in detail in the Section 5 of the document.

3.5 Project Folder Structure

In this section I will explain what the main folders and files that make up the project.

3.5.1 Root Project Structure

In the following figure it is possible to see the overall organization of the project.

- `lib`: this folder contains the application source code.
- `test`: this folder it contains all the tests for the main parts of the application.
- `fonts`: this folder contains the font chosen for the application and all its weights (regular, bold, extra bold, ...) in `.ttf` format.
- `ios` & `android`: these folders contain all the code necessary to run the application on the respective mobile OS.
- `pubspec.yaml`: configuration file through which general settings and dependencies on external packages on which the application depends are managed.

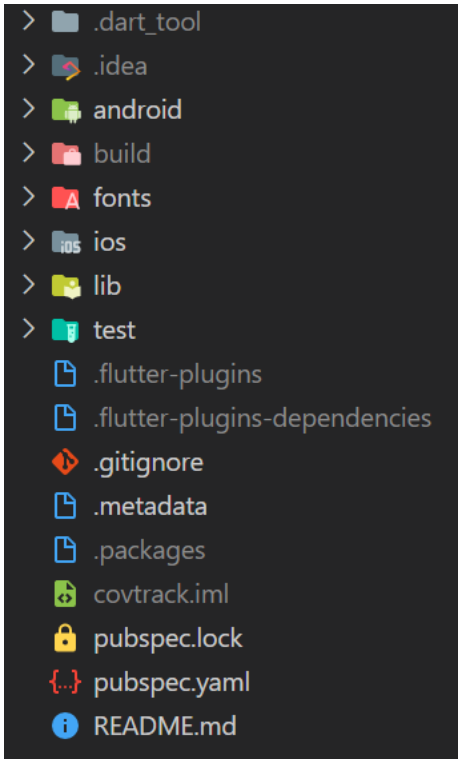


Figure 7: Project folder structure

3.5.2 `lib` Folder

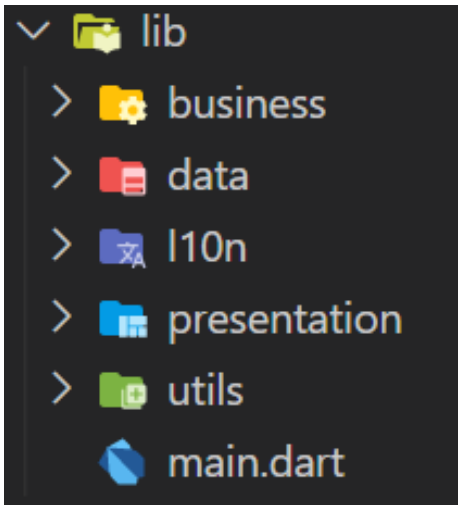


Figure 8: `lib` folder

- `business`: folder containing all the blocs and repositories used to implement the BLoC pattern.

- `data`: folder containing all the the models to manage the application data (simple Dart classes with utility methods).
- `presentation`: folder containing all the UI components (*Widgets*) of the application and the themes used to style the application.
- `l10n`: folders containing the `.arb` files and all the code necessary for the internationalization of the application.
- `utils`: folder containing utility classes: input validators, navigation routers, localization utilities.

In the following figures the first three folders are shown in detail.

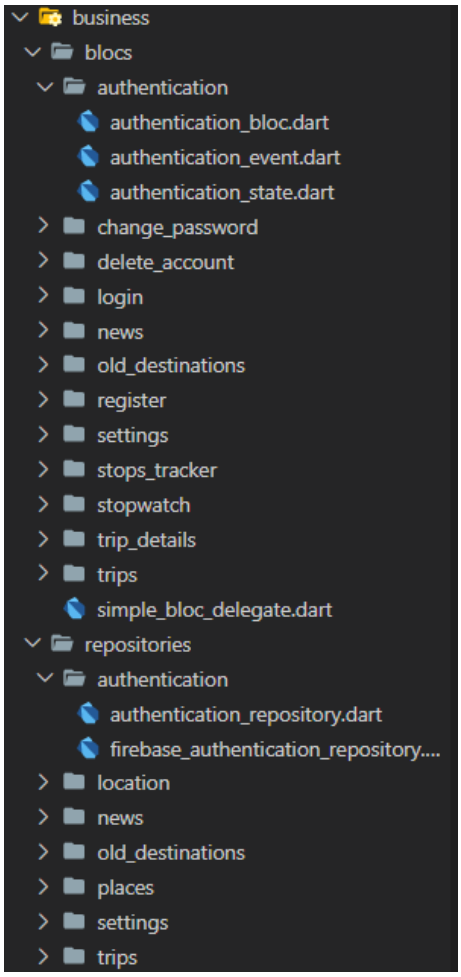


Figure 9: `business` folder

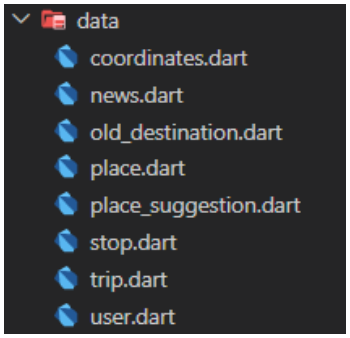
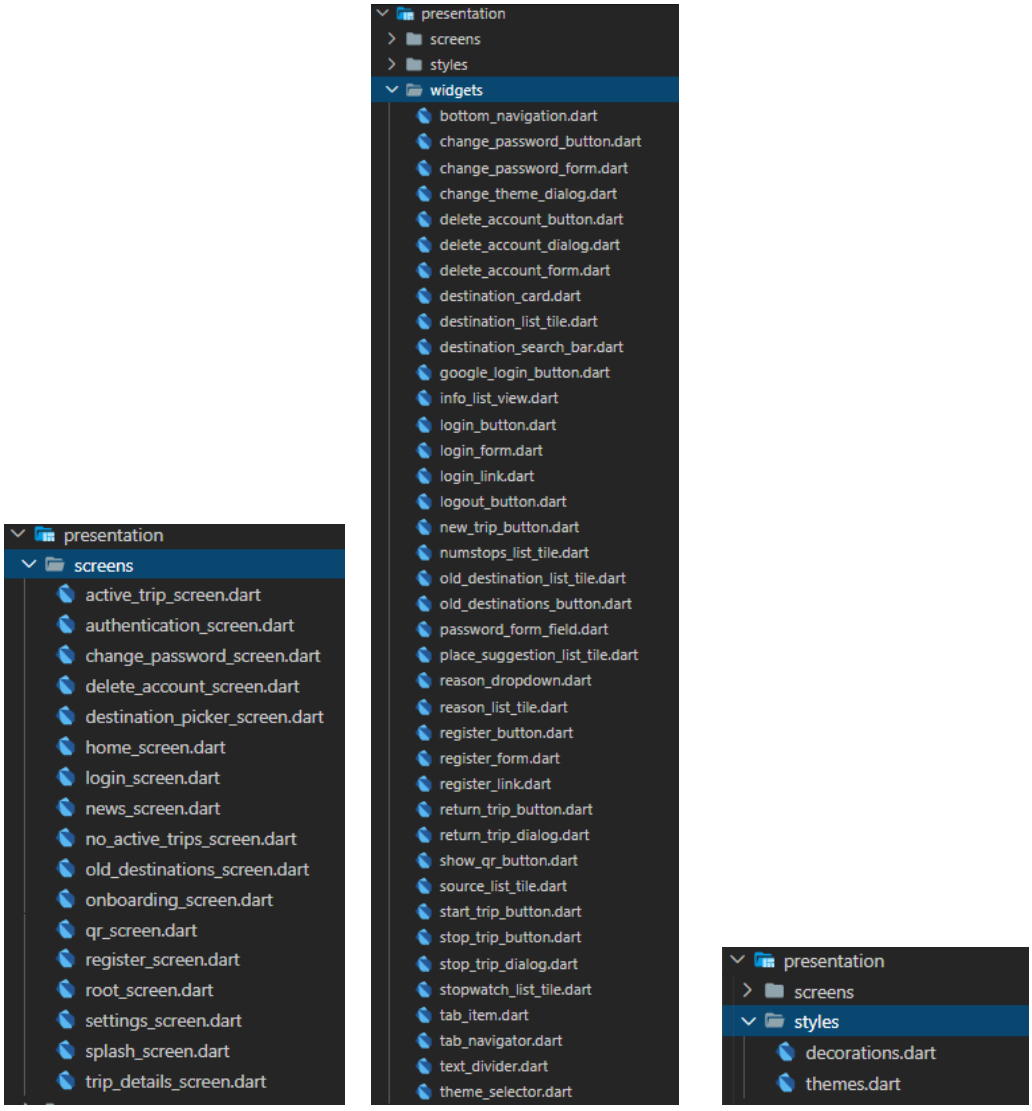


Figure 10: `data` folder



(a) Screens folder (b) Widgets folder (c) Styles folder

Figure 11: `presentation` folder

4 User Interface Design

In this section I provide a certain number of screenshots of the application. I focused my attention on designing the application for mobile phones, even if some portions of code are already prepared to be displayed on larger screens.

4.1 Splash Screen

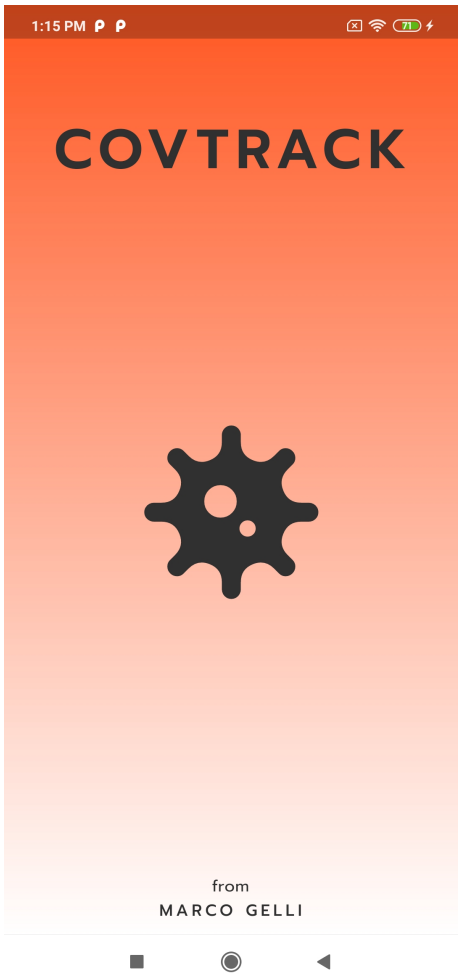


Figure 12: Splash Screen

The splash screen welcomes the user when the application is starting, meanwhile the settings and user preferences are loaded.

4.2 Onboarding Screens

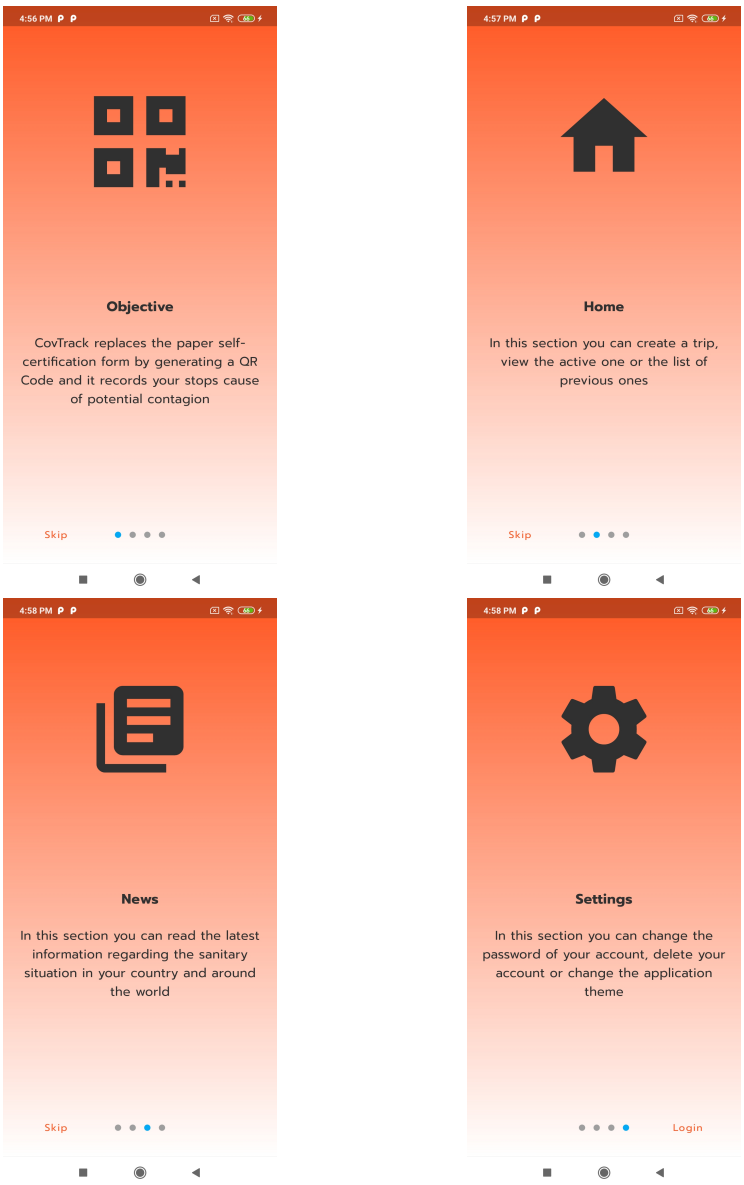
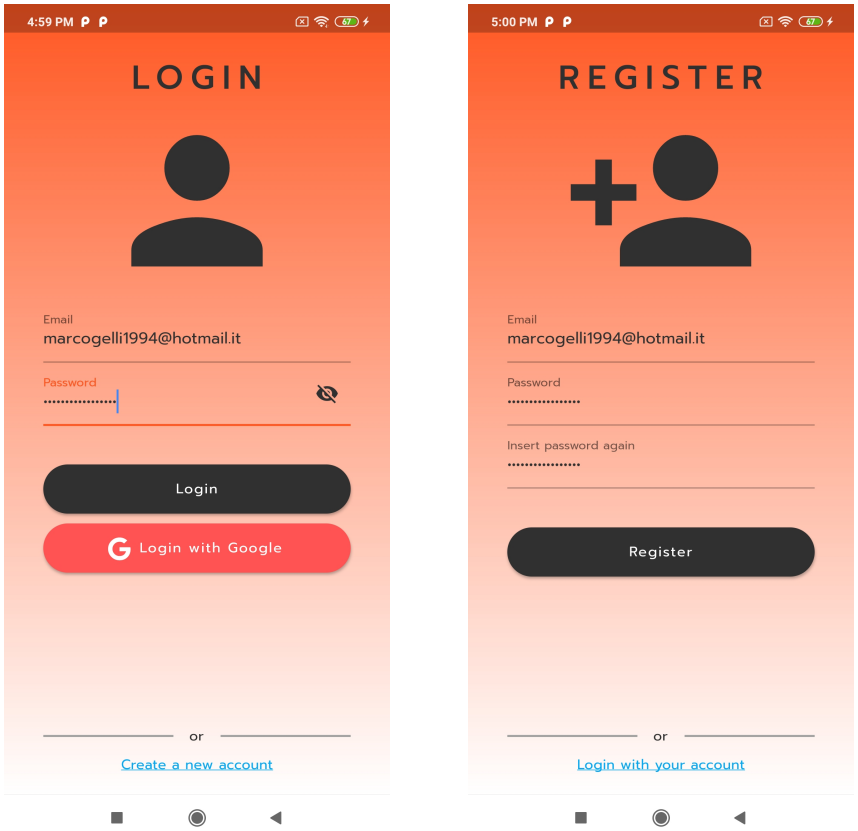


Figure 13: Onboarding Screens

These are the introductory screens that will be displayed when the user runs the app for the first time. Here it's explained the purpose of the application and the features offered by each section. The user can skip this presentation by tapping on the ***Skip*** button which will take him to the last of these where, by tapping on the ***Login*** button, he can navigate to the Login Screen [14a].

4.3 Authentication Screens



(a) Login Screen (b) Register Screen

Figure 14: Authentication Screens

The Login Screen provides two input fields for email and password and two buttons. If the user wants to access with a previously created account he must fill the input fields and tap on the ***Login*** button (which will be active only when the two input field are filled and successfully validated).

On the other hand, if the user wants to use his Google account to access the application he just needs to press on the ***Login with Google*** button and he will be prompted to choose which of his accounts he would like to use inside the application. This is the fastest way to create an account in *CovTrack*, since no fields are required to be filled.

If the user wants to create a new account he just needs to click on the ***Login*** link and the Register Screen [14b] will be displayed. After filling the three input fields, the ***Register*** button will be active and by tapping it, the user will create a new account and will be taken to the Home Screen [15].

4.4 Home Screens

In this section the screens of the **Home** section will be shown. The user can always navigate to another section using the navigation bar at the bottom of the screen.

4.4.1 No Active Trips Screen

Figure 15: No Active Trips Screen

This screen will be displayed if the user doesn't have any active trip going on. At the bottom there is a button bar: if the user taps on the ✚ button, he will be taken to the Destination Picker Screen [17] to choose the destination of the trip to be created. On the other hand, if the user presses the 🕘 button, the Old Trips Screen [20] will be displayed.

Finally, if the user presses the top-right icon, he will be logged out and taken back to the Login Screen [14a].

4.4.2 Active Trip Screen

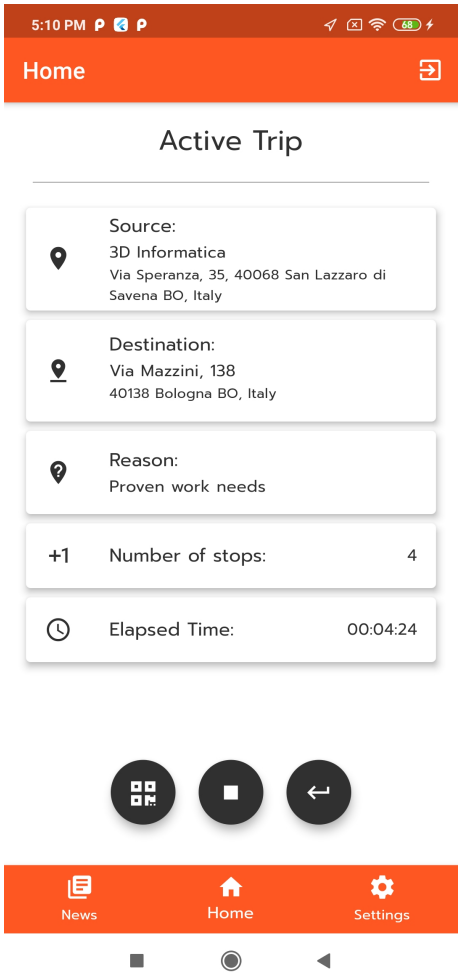


Figure 16: Active Trip Screen

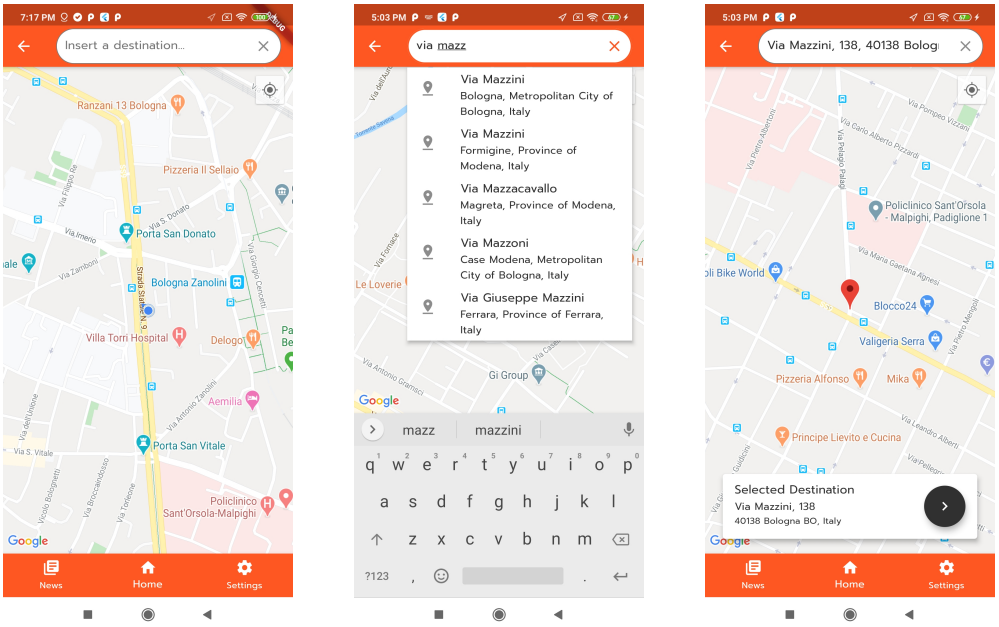
This screen will be shown to the user when there is a currently active trip. All information regarding the active trip is shown: source, destination, reason, number of stops and elapsed time.

At the bottom of the screen there is a button bar:

- If the user presses the ▦ button the QR Code Screen [19a] will be displayed.
- If the user presses the ■ button the Stop Current Trip Dialog [19b] appears.
- If the user presses the ⮐ button the Start Return Trip Dialog [19c] appears.

Finally, if the user presses the top-right icon, he will be logged out and taken back to the Login Screen [14a].

4.4.3 Destination Picker Screen



(a) Initial (b) Typing destination (c) Destination selected

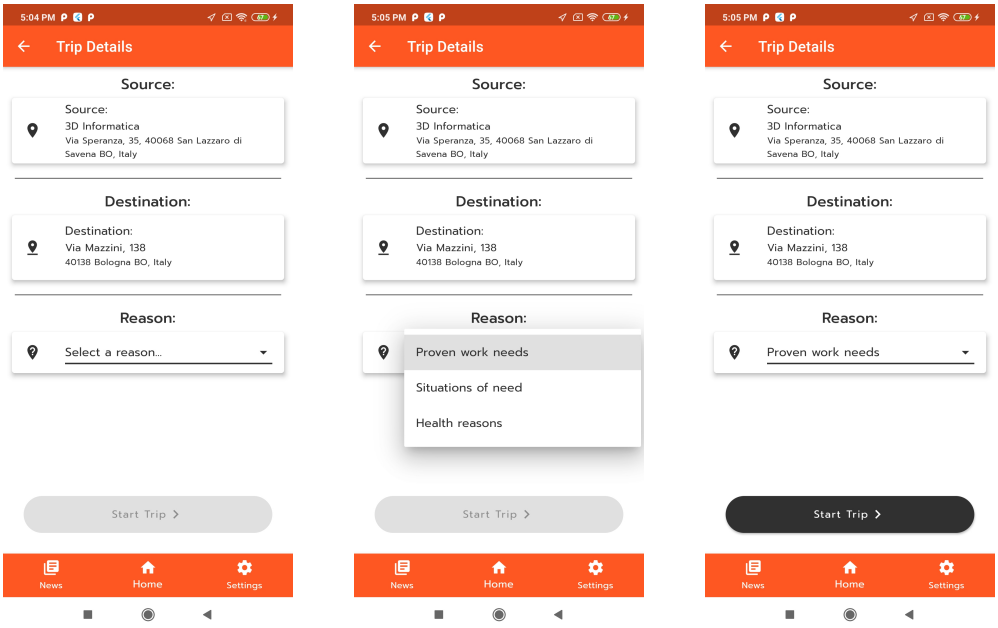
Figure 17: Destination Picker Screen

In this screen the user can select the destination of a new trip. There are two ways to do it: using the search bar or selecting it by pressing on the map.

If the user enters his destination in the search bar, a list of suggestions from the Google Place Autocomplete API will be displayed: to select a destination he just needs to tap on one of them. On the other hand, the user can also directly use the map to select a destination by just tapping on one point on the map.

When a destination is selected, the map is centered on it and a pin is placed on that point. In addition, a card will appear on the bottom of the screen with the place address and the › button which allows the user to navigate to the Trip Details Screen [18].

4.4.4 Trip Details Screen



(a) Initial (b) Selecting reason (c) Reason selected

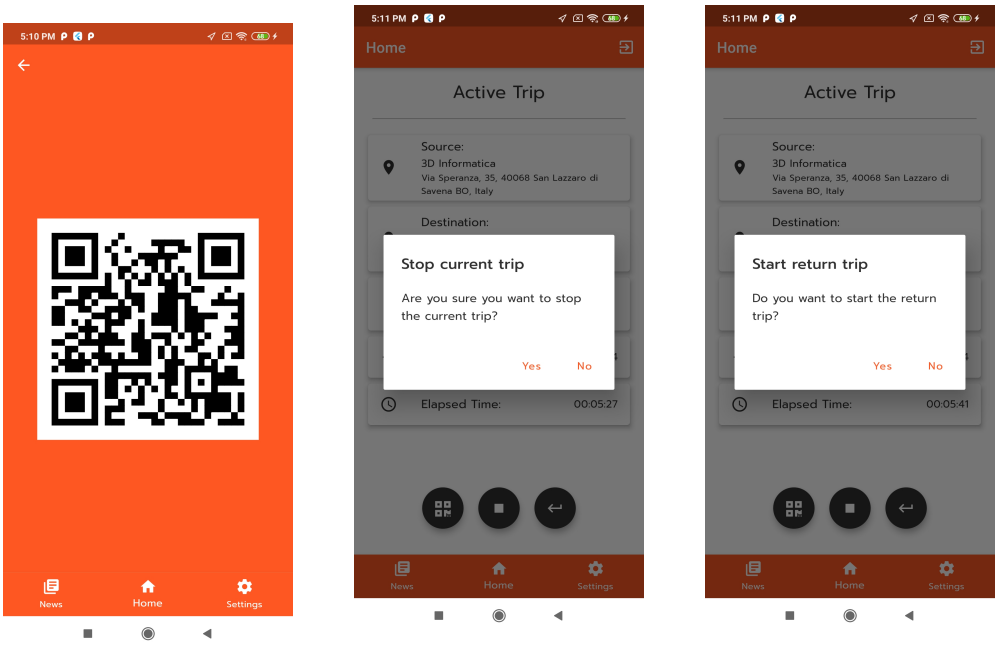
Figure 18: Trip Details Screens

This screen shows the source and the selected destination of the trip. It also provides the user with a dropdown which allows him to select the reason for the trip. The ***Start Trip*** button will be disabled until the user selects a reason for the trip.

If the user presses on the ***Start Trip*** button, the trip will be created and added to the list of user trips. After that, the user will be shown the Active Trip Screen [16] containing all the details of the trip just created.

4.4.5 Active Trip Actions Screens

These are the screens that will be shown to the user if he presses one of the buttons on the Active Trip Screen [16].



(a) QR Code Screen

(b) Stop Current Trip Dialog

(c) Start Return Trip Dialog

Figure 19: Active Trip Actions Screens

The QR Code Screen simply shows the QR Code identifying the trip, which is quite large so that it will be easy for the authorities to scan it.

The Stop Current Trip Dialog asks the user if he is sure he wants to end the active trip. If he presses on ***Yes*** the active trip will be stopped and stored in in the remote database and the No Active Trips Screen [15] will be displayed, On the other hand, if he presses on ***No*** the dialog will be closed.

The Start Return Trip Dialog asks the user if he is sure he wants to start the return trip. If he presses on ***Yes*** the active trip will be stopped and stored in the remote database, and a new trip with swapped source and destination will be created. On the other hand, if he presses on ***No*** the dialog will be closed.

4.4.6 Old Destinations Screen

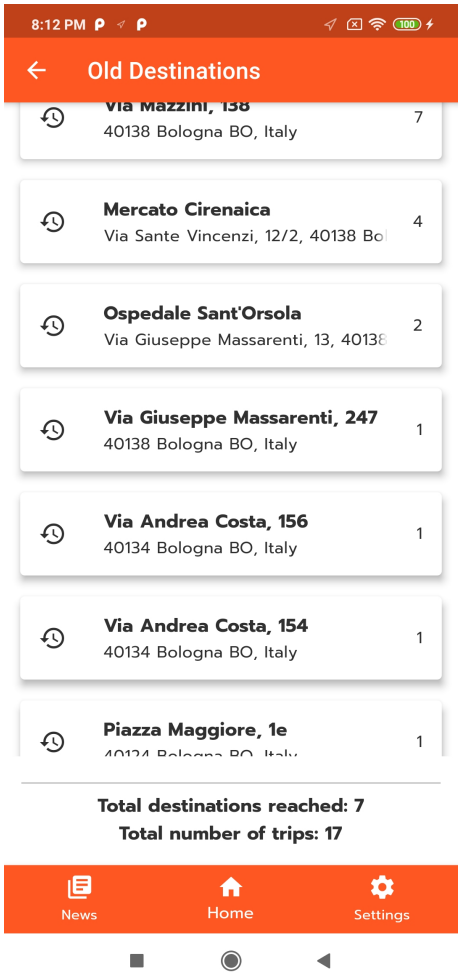


Figure 20: Old Destinations Screen

This screen shows a list of all the destinations reached by the user during his trips. The list is sorted in descending order according to the number of times a destination has been visited. If the user presses one of the destinations, the creation of a new trip will start and the user will be taken to the Trip Details Screen [18] screen, where the source and destination fields will be filled respectively with user's current location and the same destination as the one shown in the tapped element.

Finally, at the bottom of the screen is shown how many destinations the user has visited in total and how many trips he has made in total.

4.5 News Screen

This is the only screen of the **News** section.

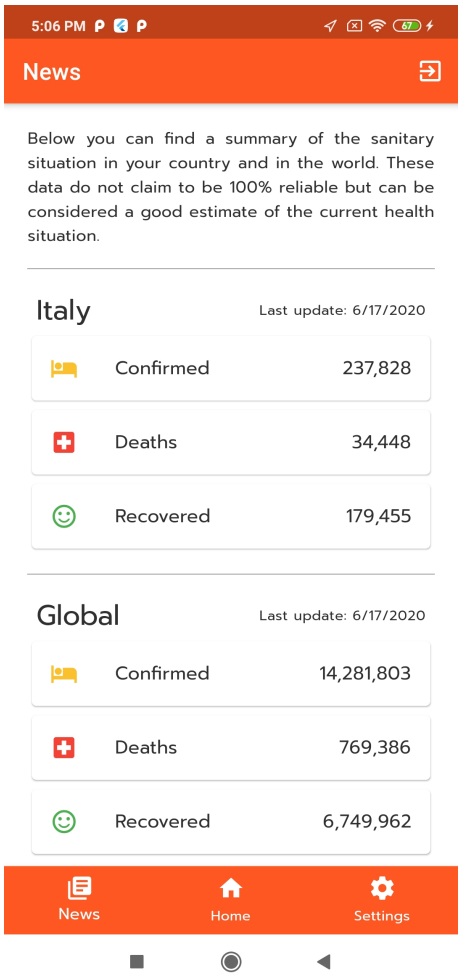


Figure 21: News Screen

This screen shows the user a summary of the health situation in his country and around the world. In particular, for each of the two sections are shown the total number of infected people, the total number of deaths and the total number of recovered. The last update date is also shown.

By dragging the screen down, fresh data are fetched from the API and when loaded displayed to the screen. Finally, at the bottom of the screen there is a clickable link that opens the device browser to the API home page from which the news data is collected.

Also in this screen, if the user presses the top-right icon, he will be logged out and taken back to the Login Screen.

4.6 Settings Screens

In this section the screens of the **Settings** section will be shown. The user can always navigate to another section using the navigation bar at the bottom of the screen.

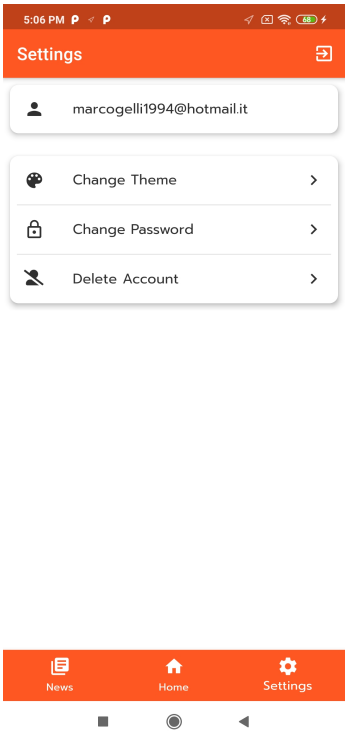
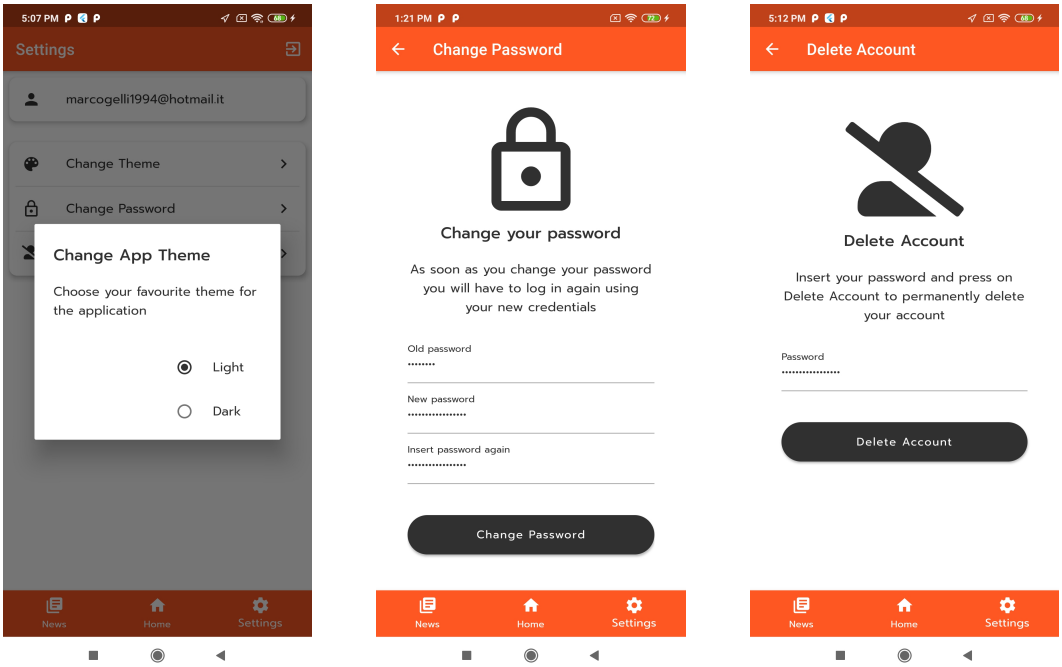


Figure 22: Settings Screen

In this first screen the user is presented with a text showing the email with which the user has registered and a menu with all the available settings of the app. By pressing one of the menu items the user will be taken to the corresponding screen.

Also in this screen, if the user presses the top-right icon, he will be logged out and taken back to the Login Screen [14a].



(a) Change Theme Dialog (b) Change Password Screen (c) Delete Account Screen

Figure 23: Settings Screens

When the user presses on ***Change Theme*** item an alert dialog with two radio buttons appears: by tapping on one of the two he can select the theme of the application by choosing between ***Light*** and ***Dark***. The user can close the dialog by tapping outside of it.

On the other hand, if the user selects ***Change Password*** item, he is taken to the Change Password Screen. Here the user is presented with a form with three fields: one in which the user must enter the current password (as a security measure) and two for the new new password. Once all the fields of the form are filled and successfully validated the ***Change Password*** button will be active and if the user presses it he will be logged out and taken back to the Login Screen [14a].

Finally, if the user taps on the ***Delete Account*** item, he is taken to the Delete Account Screen. Here the user is shown a form with only one field in which to enter the account password. Once the user enters the password, the button will be active and pressing on it will display an alert dialog in which he is asked if he is sure he wants to delete the account. If the user confirms, the account is deleted and all the data of the trips saved on the remote database will be deleted as well. Finally the user will be returned to the Login Screen [14a].

5 External Services & Packages

In this section are listed all the external services used in the mobile application. Some of them are necessary for the proper behavior of the system and other only to enrich the overall experience of the user. All of them are completely transparent to the user and fully integrated within the application. When a Dart package is used to integrate a particular service I provide the corresponding `pub.dev` link.

5.1 Firebase

Here are listed all the Firebase back-end services used by the application.
→ **Dart package**: `firebase_core`

5.1.1 Authentication

This service is a very important service since it handles all the authentication procedures for the users. In the client-side part of the application, it is used to allow user to register or login using their preferred method among Google or classic email and password combination.
→ **Dart package**: `firebase_auth`

5.1.2 Cloud Firestore

This service is also of fundamental importance: it provides the application with a cloud-hosted, non-relational database with live synchronization. Following Cloud Firestore's NoSQL data model, I store trips data in documents, which are grouped into collections divided by user id. The live synchronization feature allows us to rebuild the UI in real time when the trips data is updated.
→ **Dart package**: `cloud_firestore`

5.1.3 Remote Configuration

Service used to store and retrieve the Google Maps API key.
→ **Dart package**: `firebase_remote_config`

5.2 Google Authentication

Service used to obtain the Google user profile and email, that is then used by Firebase Authentication to handle new user creation, deletion or authentication.
→ **Dart package**: `google_sign_in`

5.3 Google Maps

Service which provides the user with an interactive map to select and view the destination of his next trip.
→ **Dart package**: `google_maps_flutter`

5.4 Google Maps Web Services

It's an API that groups a very important set of services for the application, since they return information about places using HTTP requests. To make the integration of these services easier I used the package shown below, which is simply a wrapper used to mask the complexity of the JSON responses and spared me from having to implement specific decoder classes so that the code is cleaner.
→ **Dart package**: `google_maps_webservice`

5.4.1 Place Autocomplete

Web service that returns place suggestions given an input query string. The service can be used to provide the auto-complete functionality to the search bar of the Destination Picker Screen.

5.4.2 Place Details

Web service which starting from the id of a place, returns more comprehensive information about it such as its complete address, the coordinates and many others. In the application it is used in the Destination Picker Screen to provide the user with as detailed information as possible about his destination.

5.4.3 Reverse Geocoding

Web service used to convert geographic coordinates into human-readable addresses. This service is used when the user has to select a destination by tapping on the map:

the place information will be retrieved starting the coordinates of the selected point (latitude and longitude).

5.5 Device Location

The package reported below provides easy access to platform specific location services. It's used to find the current position of the user and to receive continuous position updates. In the application this service is widely used during the trip creation process (to center the map and to retrieve place suggestions given user's current location) and to track the stops of the user during an active trip. Finally, thanks to it, the complexity of managing permissions will be masked.
→ **Dart package**: `geolocator`

5.6 Local Storage

To store simple key-value pairs data such as the chosen app theme or other settings I use the plugin reported below. Data is persisted to disk asynchronously.
→ **Dart package**: `shared_preferences`

5.7 QR Code Generation

The package below is used to generate the QR Code identifying the trip, which allows me to insert the QR Code into the code like any other Widget.
→ **Dart package**: `qr_flutter`

5.8 Internationalization & Localization

Multi-language support and localized date/number formatting to my application.
→ **Dart package**: `intl`

5.9 Covid-19 API

API from which the data presented in the News section is collected. There's no authentication required and the responses are in JSON format. The user is presented with two types of news: the news about the health situation in his country and the news about the health situation in the world. The following figures show two examples, one for each type of news.

HTTP request to the following endpoint:
https://covidapi.info/api/v1/country/ITA/latest

produces the following JSON response:

```
{
    "count": 1,
    "result": {
        "2020-07-08": {
            "confirmed": 242149,
            "deaths": 34914,
            "recovered": 193640
        }
    }
}
```

Listing 3: Local News JSON response structure

HTTP request to the following endpoint:
https://covidapi.info/api/v1/global

produces the following JSON response:

```
{
    "count": 1,
    "date": "2020-07-08",
    "result": {
        "confirmed": 12041480,
        "deaths": 549468,
        "recovered": 6586726
    }
}
```

Listing 4: Global News JSON response structure

5.10 Other Packages

- `http`: platform-independent high-level functions and classes that make it easy to consume HTTP resources.
- `flutter_bloc`: provides multiple components used to reduce the boilerplate code needed for the correct implementation of the BLoC pattern.
- `equatable`: allows me to compare Dart objects in a simple way without having to override the `==` and the `hashCode` operator. Particularly useful to compare BLoC event and state objects.
- `rxdart`: implementation of the popular reactiveX API for asynchronous programming, leveraging the native Dart Streams API. Used to manipulate BLoC events (e.g. add debounce time).
- `font_awesome_flutter`: icon pack used for app embellishments.
- `introduction_screen`: set of widgets used for the onboarding process.
- `flutter_typeahead`: auto-complete widget used for the search bar of the Destination Picker Screen.
- `url_launcher`: platform-independent package that allows to open the device browser on a given URL. Used to open the browser on the Covid-19 API home page in the News section.
- `uuid`: used for generating RFC4122 UUIDs.

6 UML Diagrams

In this section I present some useful diagrams that helps the reader to better understand the interaction between the user and the application.

6.1 Use Case Diagrams

These diagrams show the flow of operation triggered by a specific actor when it tries to perform some task. The actor can be the user (a human actor) or a system. The use cases we propose show the main operations that are possible to perform in the system.

6.1.1 Unauthenticated user interaction

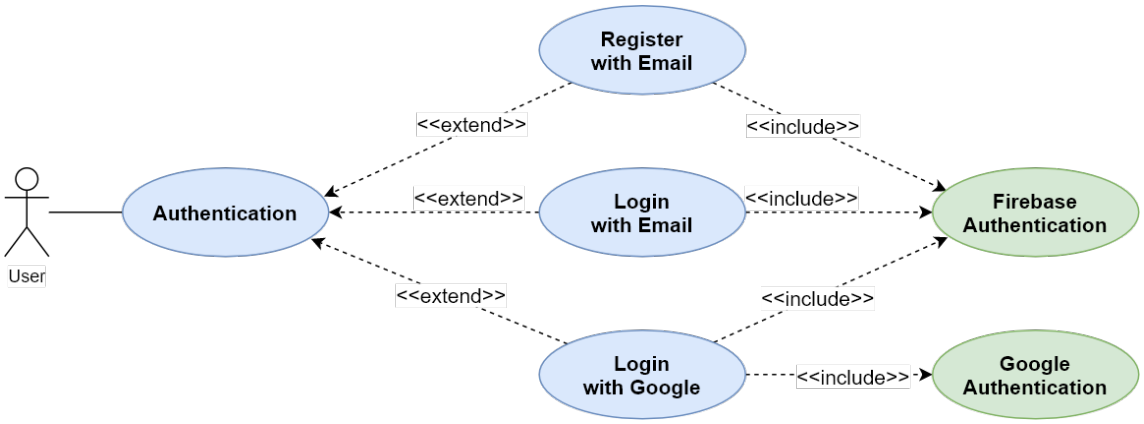


Figure 24: Use case: unauthenticated user

6.1.2 Home section user interaction

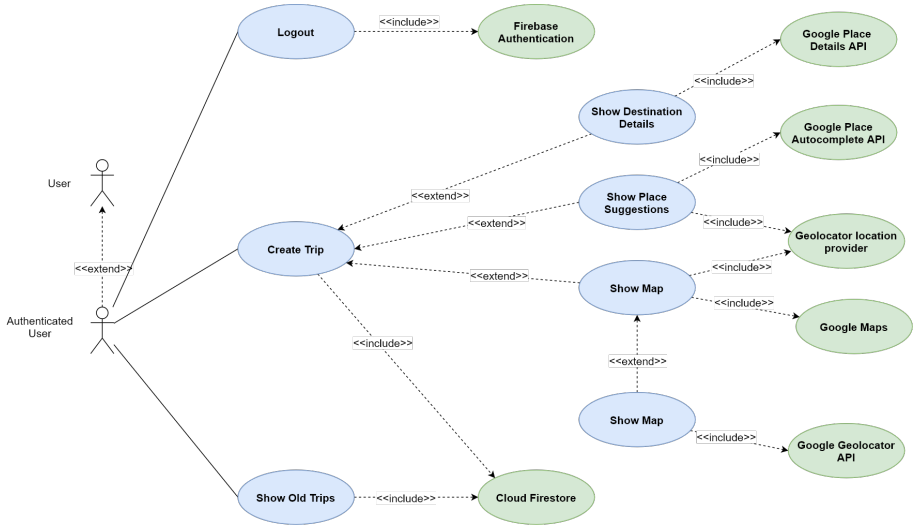


Figure 25: Use case: home section with no active trip

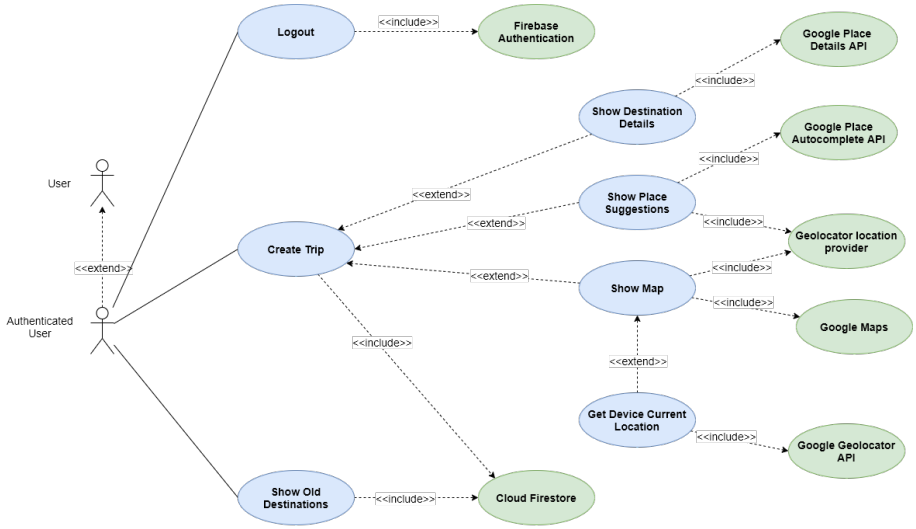


Figure 26: Use case: home section with active trip

6.1.3 News section user interaction

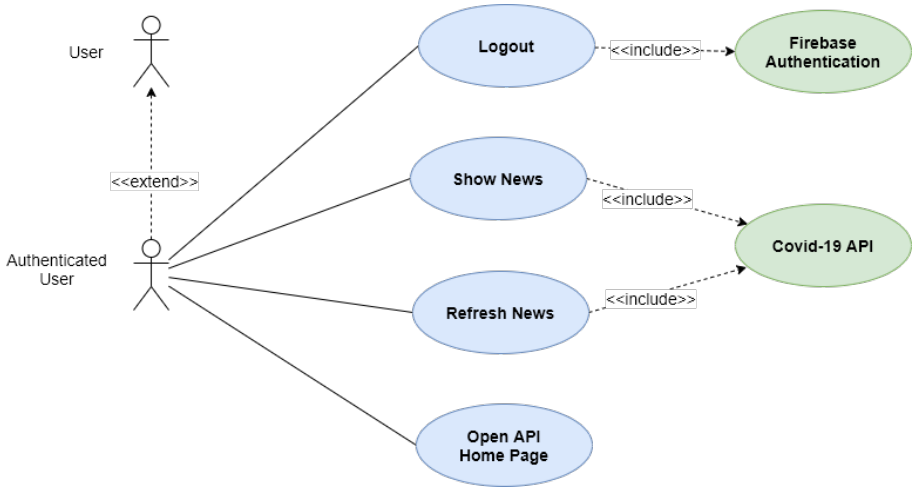


Figure 27: Use case: news section

6.1.4 Settings section user interaction

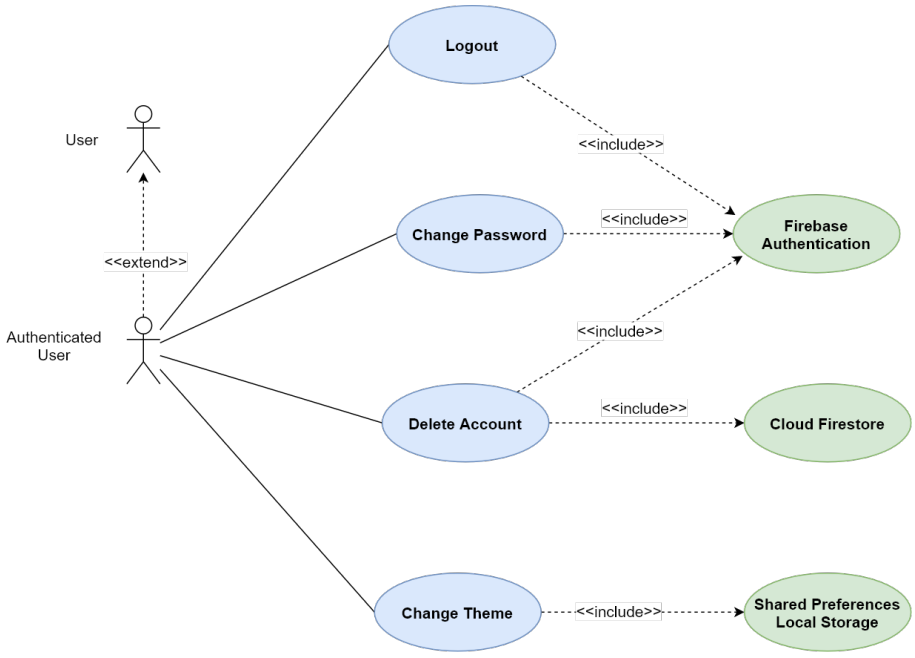


Figure 28: Use case: settings section

7 Testing

This section describes the results of the main tests done on *CovTrack* application. I focused my attention on testing the business layer consisting of the BLoCs and the data layer.

To write the tests I used the following libraries:

- `firebase_test`: Flutter's built-in testing library.
- `bloc_test`: library to reduce the boilerplate when testing BLoCs.

7.1 BLoCs Testing

The following section shows all the tests performed on the main BLoCs responsible for managing the business logic behind the application screens.

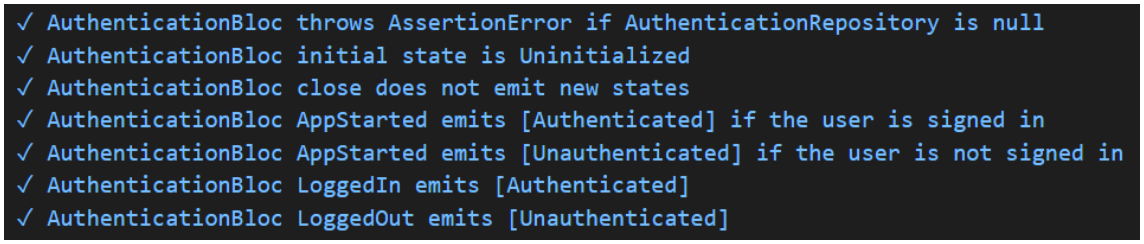


Figure 29: Authentication BLoC test cases

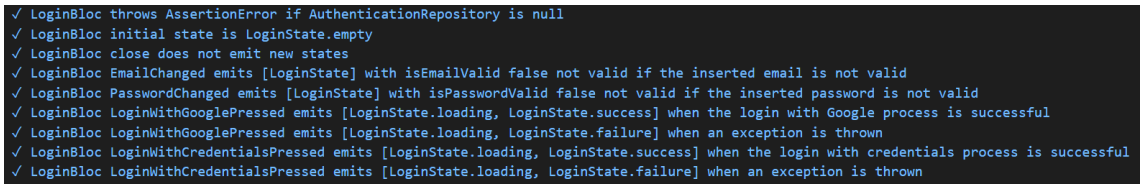


Figure 30: Login BLoC test cases

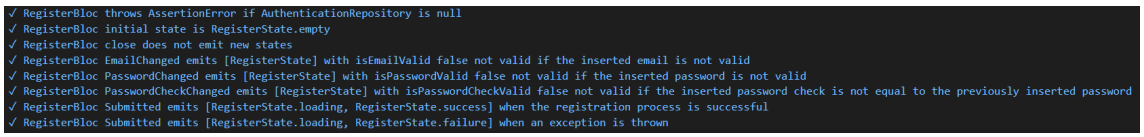


Figure 31: Register BLoC test cases

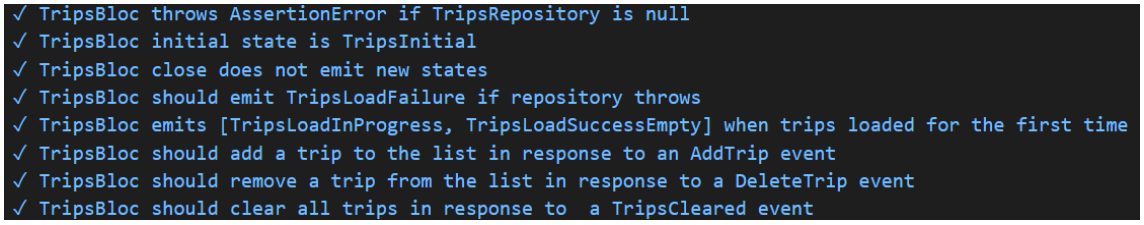


Figure 32: Trips BLoC test cases

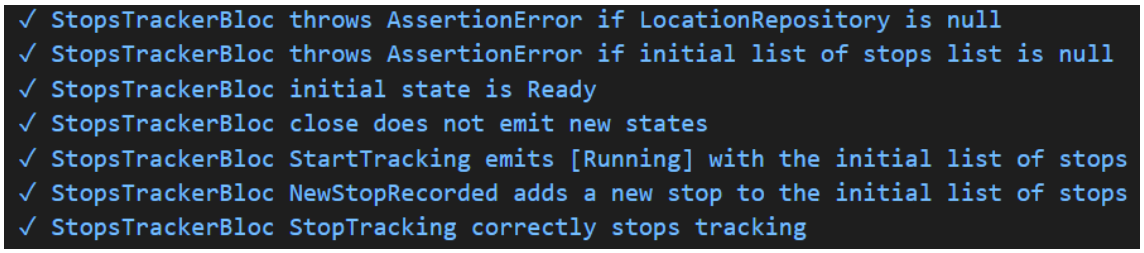


Figure 33: Stops Tracker BLoC test cases

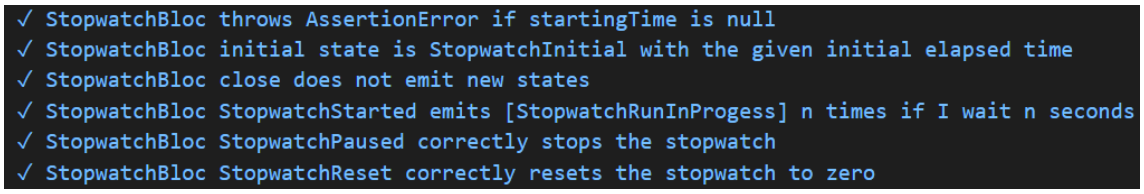


Figure 34: Stopwatch BLoC test cases

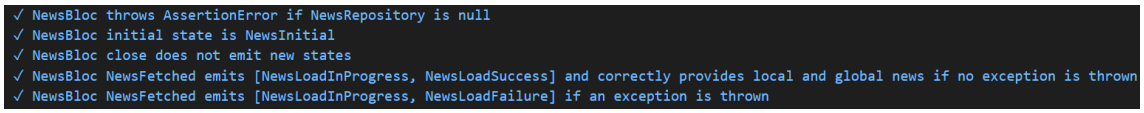


Figure 35: News BLoC test cases

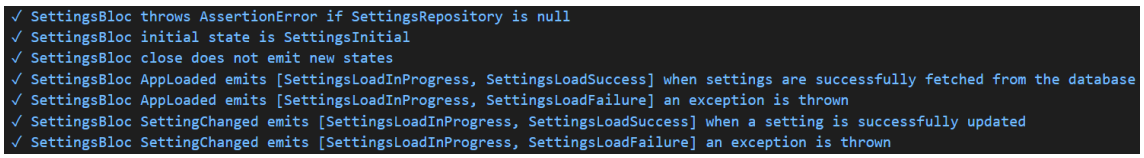


Figure 36: Settings BLoC test cases

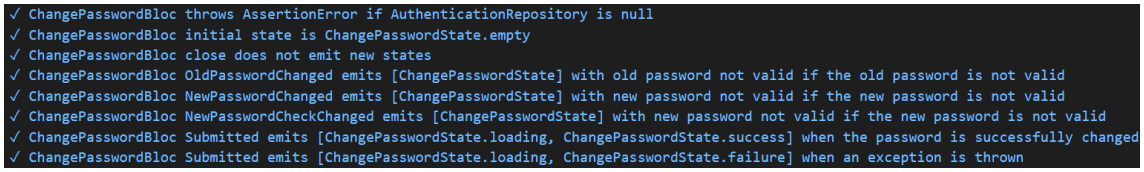


Figure 37: Change Password BLoC test cases

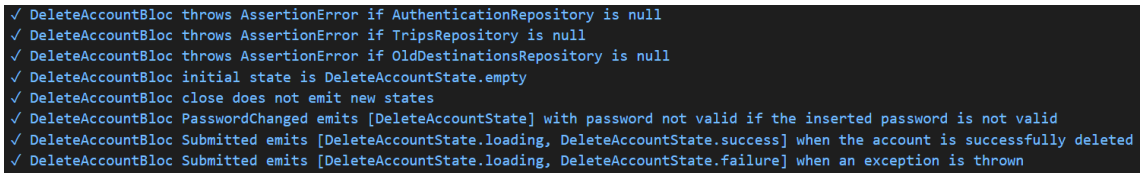


Figure 38: Delete Account BLoC test cases

7.2 Data Testing

The following section shows all the tests performed on the models that make up the application data layer.

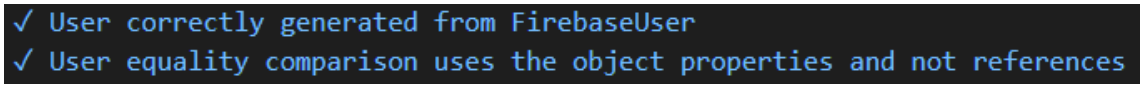


Figure 39: User model test cases

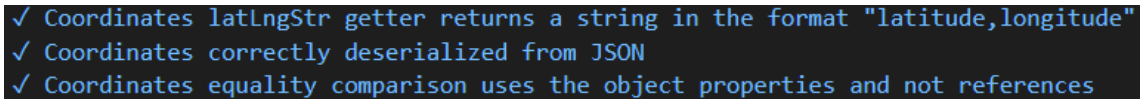


Figure 40: Coordinates model test cases

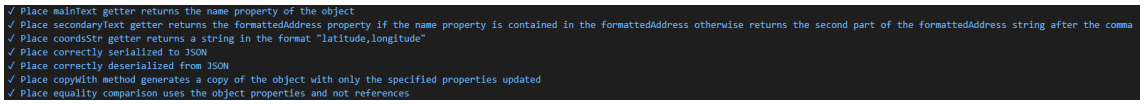


Figure 41: Place model test cases

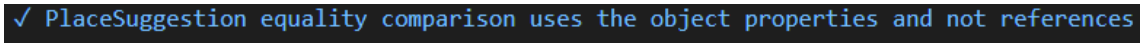


Figure 42: Place suggestion model test cases

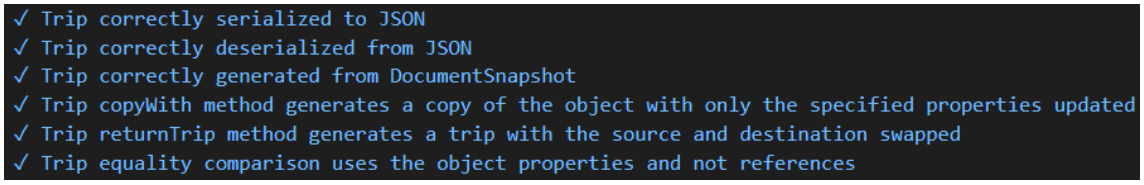


Figure 43: Trip model test cases

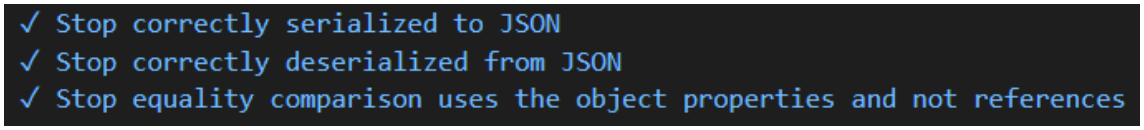


Figure 44: Stop model test cases

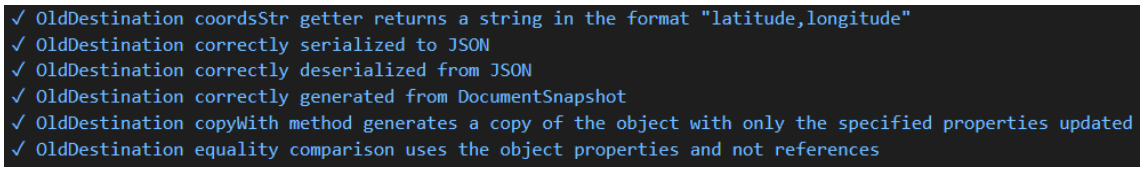


Figure 45: Old destination model test cases

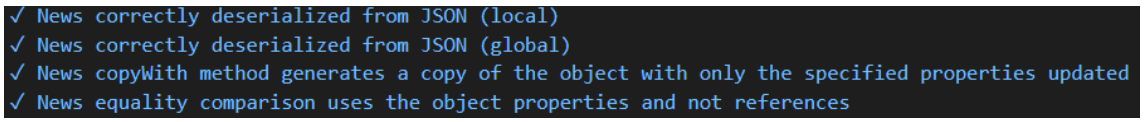


Figure 46: News model test cases

References

[1] Course Material
https://baresi.faculty.polimi.it/dima.htm

[2] Flutter documentation
https://flutter.dev/

[3] pub.dev - Dart package catalog
https://pub.dev/

[4] Felix Angelov's BLoC library documentation
https://bloclibrary.dev/#/

[5] Firebase Authentication
https://firebase.google.com/docs/auth

[6] Firebase Cloud Firestore
https://firebase.google.com/docs/firestore

[7] Google Place Autocomplete API
https://developers.google.com/places/web-service/autocomplete

[8] Google Place Details API
https://developers.google.com/places/web-service/details

[9] Google Geocoding API
https://developers.google.com/maps/documentation/geocoding/

[10] Covid-19 API
https://covidapi.info/

A Appendix A

A.1 Software & Tools

- *Visual Studio Code*: used as IDE for coding.
- *Git*: used as version-control system (VCS).
- *GitHub*: used as hosting service for version-control using Git.
- *Android Studio*: used for the Android emulator.
- *Figma*: used to draw mock-ups.
- *draw.io*: used to draw diagrams.
- *Overleaf*: used for writing this LaTeX document.
- *Final Cut Pro*: used for the elevator pitch.

A.2 Revision History

Revision	Date	Changelog
1.0	9/6/2020	First document issue