POLITECNICO
MILANO 1863

Computer Science and Engineering

Design and Implementation of Mobile Applications - A.Y. 2021/2022

sApport

Design Document

Luca Colombo - mat. 952240

Simone Ghiazzi - mat. 944782

Contents

Chapter 1

Introduction

1.1 Purpose

This document is the Design Document (DD) for the *sApport* application, developed for both Android and iOS systems. It aims to provide an in-depth description of the application, giving technical details and explaining how it has been implemented and integrated, in terms of architecture and software components.

1.2 Document structure

Chapter 1 is an introduction to the Design Document. It explains the purpose and exposes the scope. All definitions and acronyms used in the document are listed, as well as the reference documents.

Chapter 2 provides a detailed description of the system's architecture design. The components that make up the application are described in their functionalities and interactions and the decisions made with respect to the architecture are explained and justified.

Chapter 3 exposes the user interface design. In order to give a better description of the interaction of users with the application and their flow through the different pages, some UX diagrams have been inserted.

Chapter 4 provides the description of unit, widget and **SYSTEM TESTING????**.

Chapter 5 lists the references and the tools used for creating the application and writing the

documentation.

1.3 Definitions and Acronyms

1.3.1 Definitions

- User: a normal user, a person who signs up for the application because he needs support
- Expert: a psychologist enrolled in the system to provide support to users and to increase his visibility and the number of his patients
- Report: the report of a physical or psychological violence, a threats or harassment anonymously stored in the system's database and forwarded to the authorities
- Push notification: a notification sent to a smartphone
- Marker: a pin placed on the map at the medical office of a registered psychologist
- Diary: a calendar containing notes (diary pages) written by the user. One note can be written each day, can be modified withing that same day and can be marked as favourite at any time

1.3.2 Acronyms

- DD: Design Document
- API: Application Programming Interface
- UI: User Interface
- GPS: Global Positioning System
- DB: Database
- DBMS: Database Management System
- BaaS: Backend-as-a-Service
- MVVM: Model-View-ViewModel

1.4 Scope

sApport is an application that aims to help people who are victims of physical or psychological violence, who have been subjected to threats, who have been harassed or who fear for their health due to addictions. The biggest obstacle to overcome for these people is often the shame or fear of revealing themselves and opening up to others, and this is exactly the core of the application developed.

The registration process is simple: an user inserts his name, surname and birth date. The an email and a password are required and the registration is completed via *confirmation link* sent directly to the specified email. Alternatively, a new account can be created using an existing *Google* or *Facebook* one

Thanks to *sApport*, an *ordinary user* has access to four features:

1. **Chat with an expert:** this gives the possibility to find a psychologist nearby, thanks to the integration of the Google Maps API, and start ***anonymously*** a conversation. Once a conversation has started, it appears in the list of *experts chats* and can be accessed and continued as many times as wanted by the user. An important feature, is the possibility to access the ***expert's profile page***, in which his/her email, phone number and studio address are listed.

2. **Chat with another user:** this feature gives the possibility to interact with any other user by means of an ***anonymous*** chat. Once a conversation has started, it appears in the list of *anonymous chats* and can be accessed and continued as many times as wanted by the user.

3. **Anonymous report:** the user has the possibility to fill in a form for an anonymous report which will then be forwarded to the competent authorities. He can also access the list of all submitted reports in order to check their details.

4. **Write a diary:** this feature gives the possibility to write a new note every day and edit it freely during the same day. The diary is presented in the form of a monthly calendar: each cell corresponding to a note contains a book icon. The color of the icon changes depending on the whether the diary page is marked as *favourite* or not. The user thus

has the possibility to reread the old pages of the diary and to highlight the favorite ones.

PSYCHOLOGISTS' version of the application consists of one main feature:

1. **Chat with patients**

An expert is free to register to the system by inserting his personal information, the address of his studio and his phone number: after inserting a valid email and creating a password, the registration will be confirmed by clicking on the received confirmation link.

The three figures below briefly summarize the main features of the application and underline the distinction between *ordinary users* and *experts*.

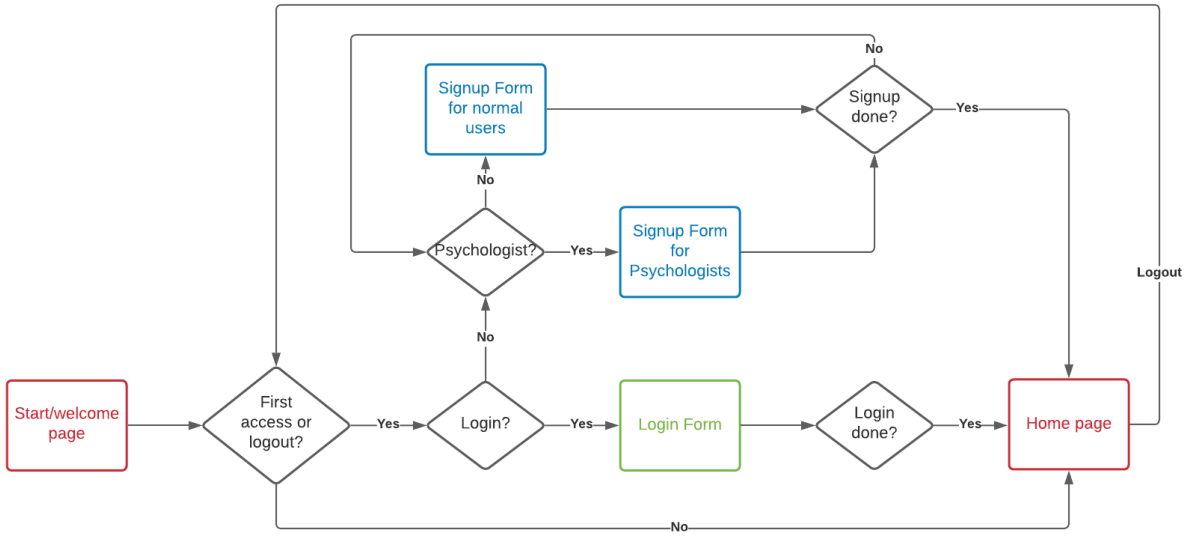


Figure 1.1: Access to the application

Figure 1.1 depicts how both types of user interact with the same application (experts do not need a dedicated system). For this reason, is important to distinguish between them and ask for different information in order to complete their registration and, eventually, login.

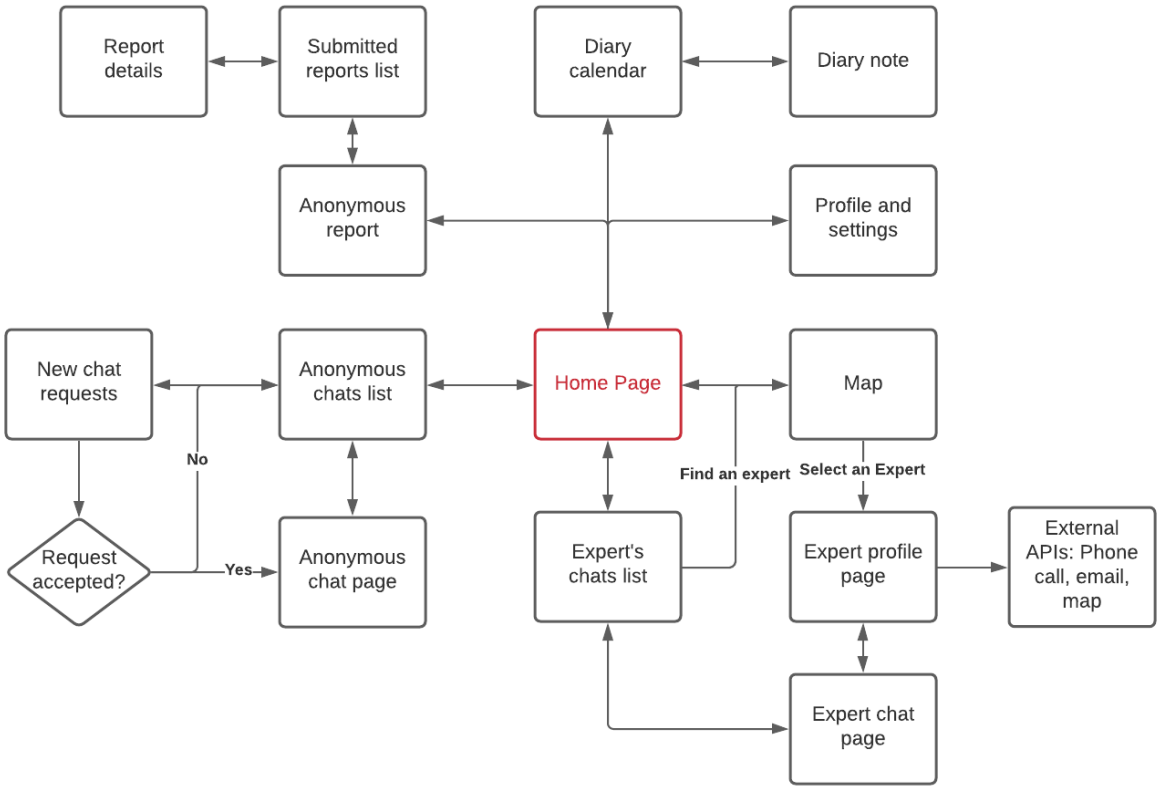


Figure 1.2: Features for ordinary users

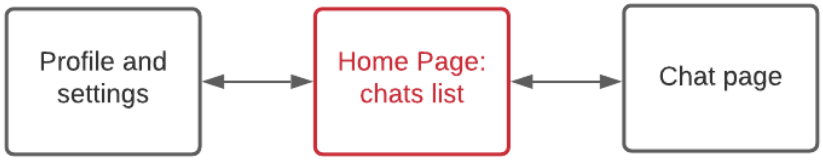


Figure 1.3: Features for experts

Figure 1.2 and 1.3 show the features accessible to *ordinary users* (the former) and *experts* (the latter) as described in the two lists at the beginning of the paragraph.

1.5 Revision history

- First version of the document: 20/04/2021
  - Introduction and description of the project
  - Sketch of the architectural design
  - Sketch of the UI

- Second version of the document: 10/09/2021
  - Almost complete architectural design
  - Almost complete UI design
  - Sketch of test plan
- Final version of the document: 11/01/2022
  - Complete architectural design
  - Complete UI design
  - Complete test plan
  - Final revision of the whole document

1.6 Reference documents

- Teaching material: slides provided by *professor Baresi*
- *Flutter* documentation
- *Firebase* documentation

Chapter 2

Architectural Design

This chapter illustrates the system architecture from both a logical and a physical point of view. Several diagrams are inserted in order to expose it more clearly.

2.1 Overview

sApport has been developed following the so called ***model-view-viewmodel*** paradigm and, for this reason,can be divided into three different tiers:

- **The presentation tier (P)**, the layer at which the users and experts' interaction with the system is managed. It includes the UI (that is the *view*) and its related components.
- **The application logic tier (L)**, the level at which the logic and the functionalities of the application are handled and data is processed. This tier is handled by the *viewmodel* components of the application which are in charge of managing the interaction between the *views* and the *models*.
- **The data persistence tier (D)**, the layer at which data is stored and made persistent, in order to be accessed and never lost. This tier is handled by the *viewmodel* components as well.

This structure is thought in order to guarantee scalability and flexibility to the system.
The layers are separate but interconnected: coupling is kept low (changing one layer doesn't affect the others), but obviously there's data flow between the layers.
For what concerns users' interaction with the system architecture, there's no difference between

ordinary users and experts: they both use the *sApport* application on their mobile devices.

2.2 Class diagrams

In this paragraph the class diagrams representing the structure of the application are shown. The three subsections reflect the framework imposed by the adopted **Model-view-viewmodel** paradigm adopted and described more in depth in *paragraph 2.5*.
Apart from all the attributes and methods listed, the importance of these UMLs is that of showing how the three components of the paradigm strongly interact with each other yet maintaining *low coupling*.

2.2.1 Views

The views are the actual UI. The dart files they are built with contain the widget composing the whole application's interface.
Views access the necessary *ViewModels* via ***MultiProvider***, which is a common access point to all ViewModels' instances and that is initialized in the application's main.
The same role, related in this case to the *Services*, is played by the ***GetIt*** plugin.

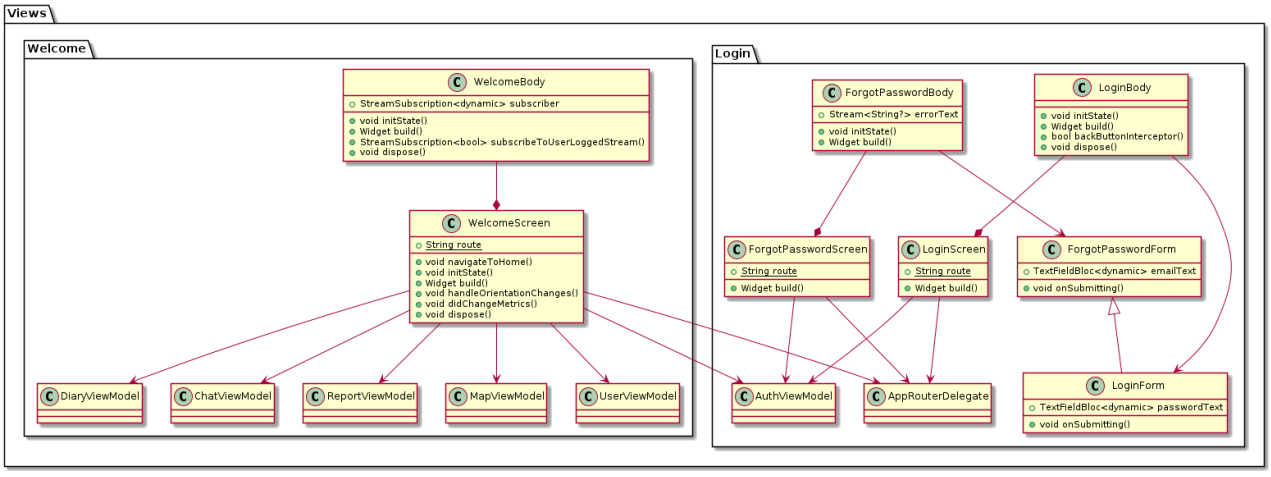


Figure 2.1: Access the application - Class diagram

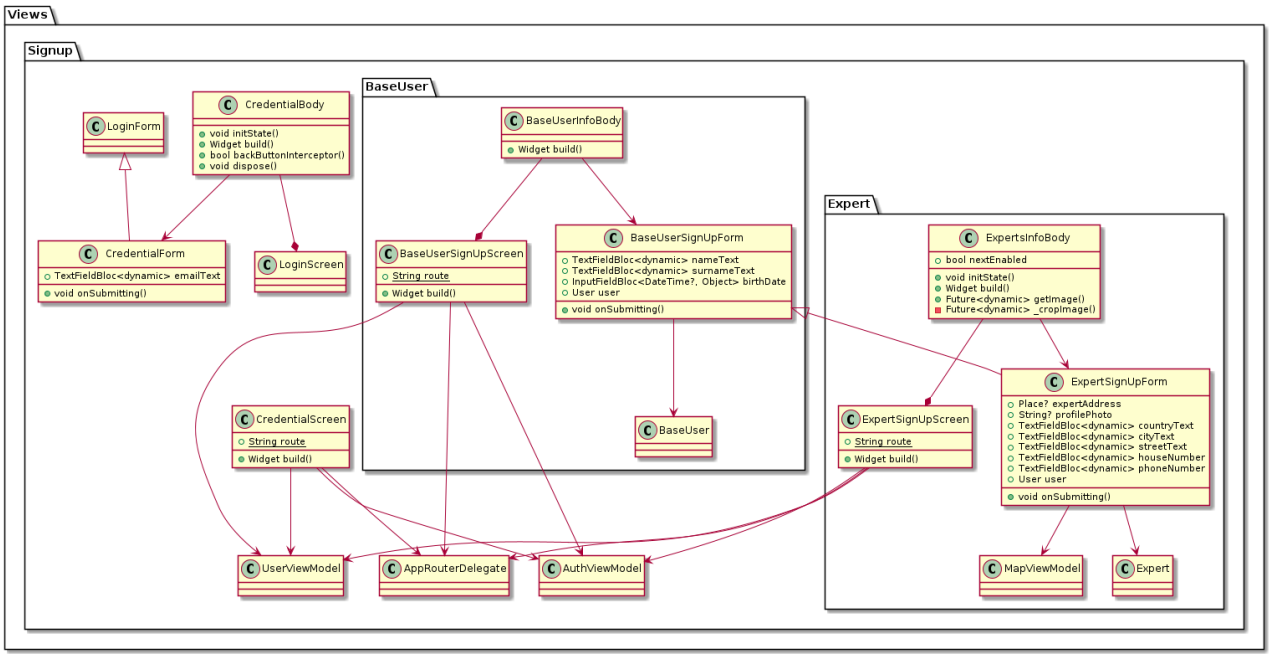


Figure 2.2: Signup - Class diagram

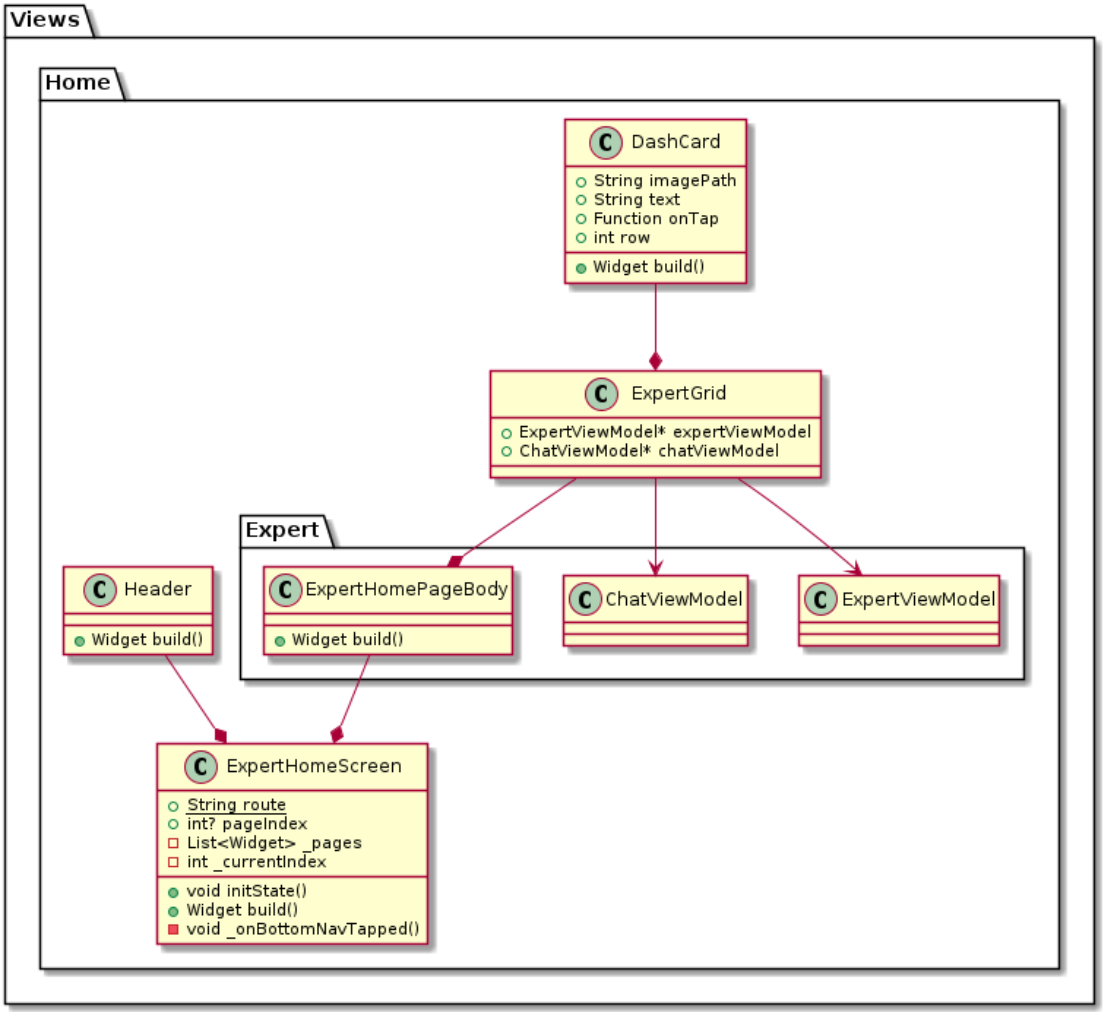


Figure 2.3: Expert home page - Class diagram

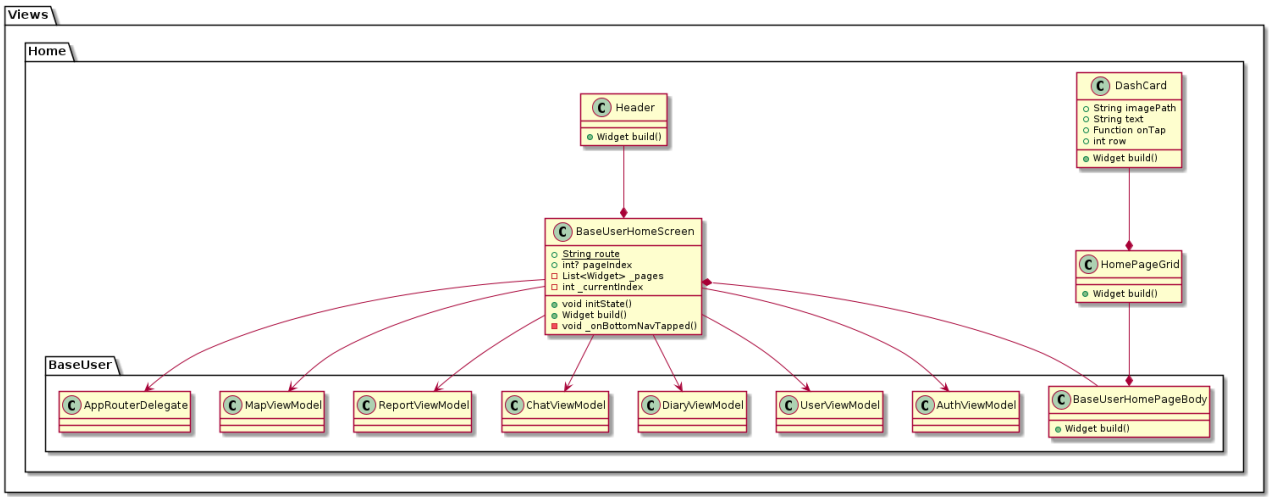


Figure 2.4: Base user home page - Class diagram

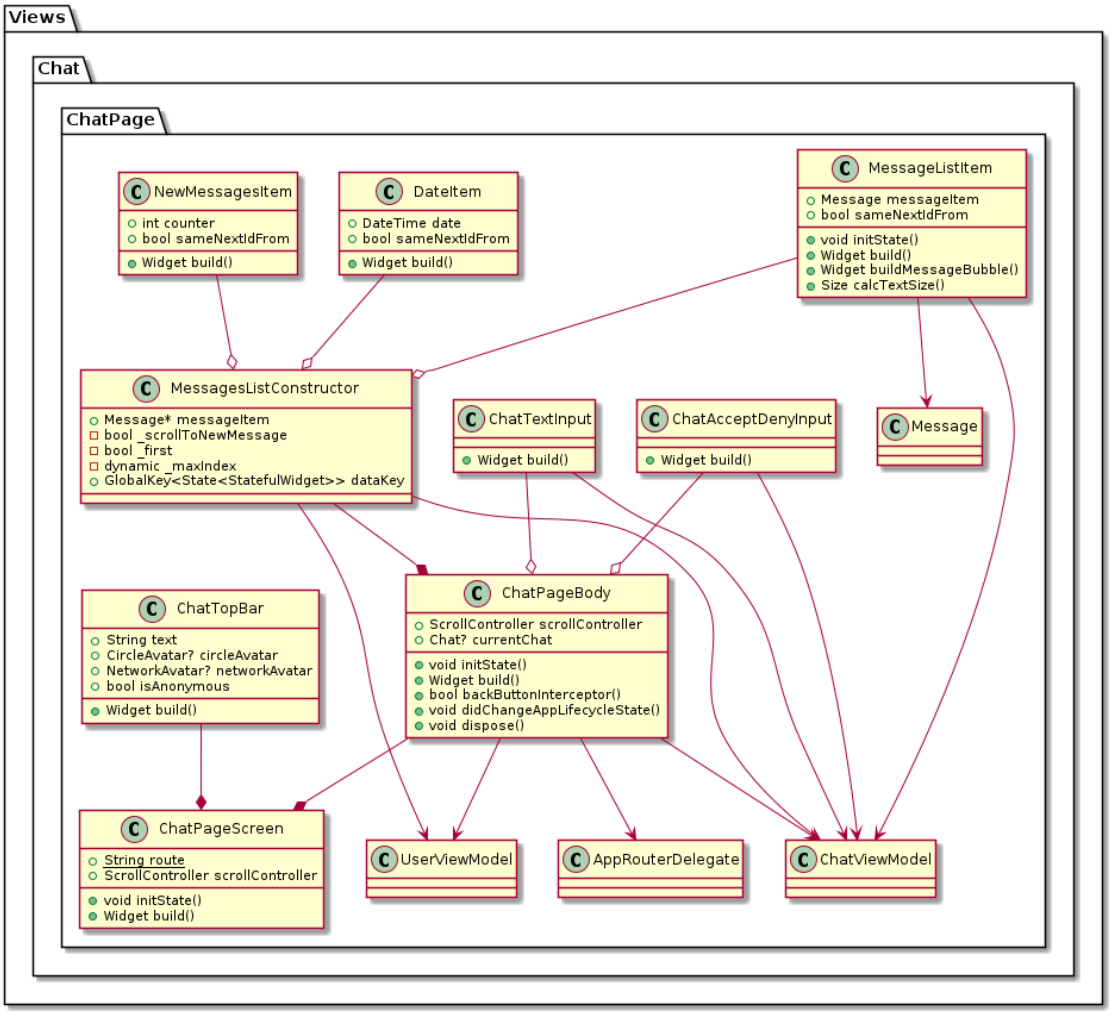


Figure 2.5: Chat page - Class diagram

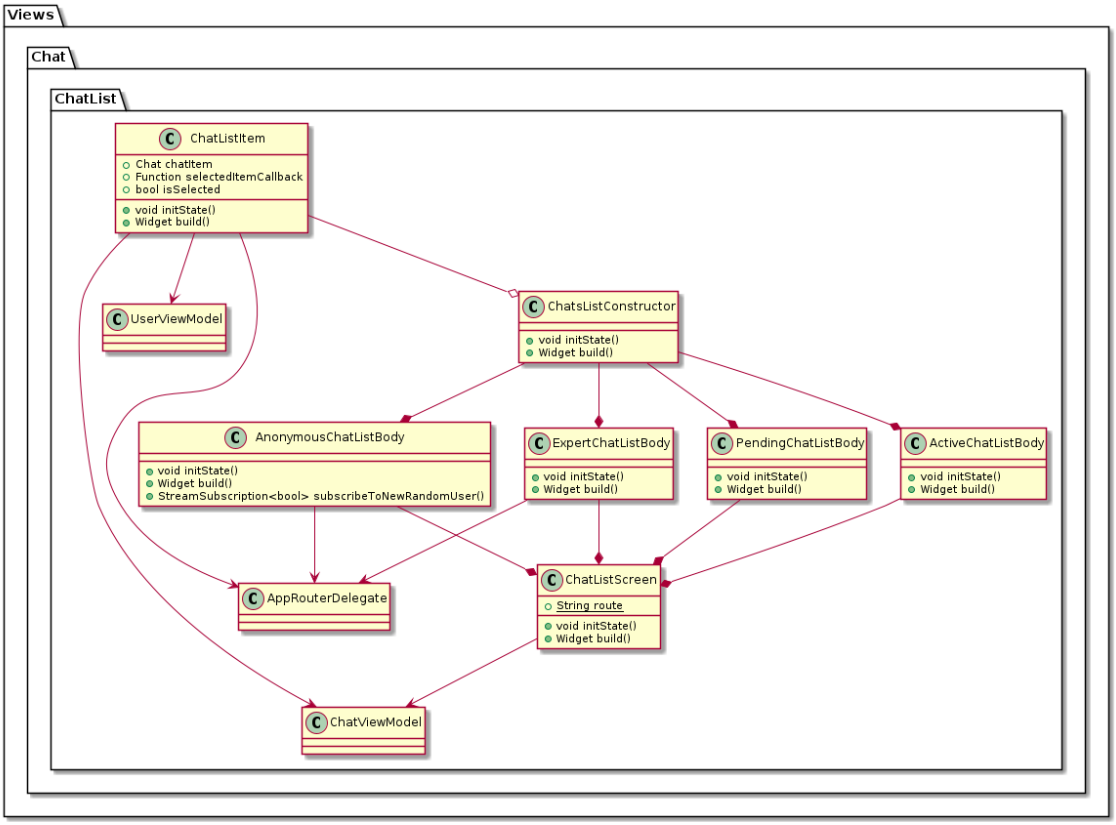


Figure 2.6: Chats lists - Class diagram

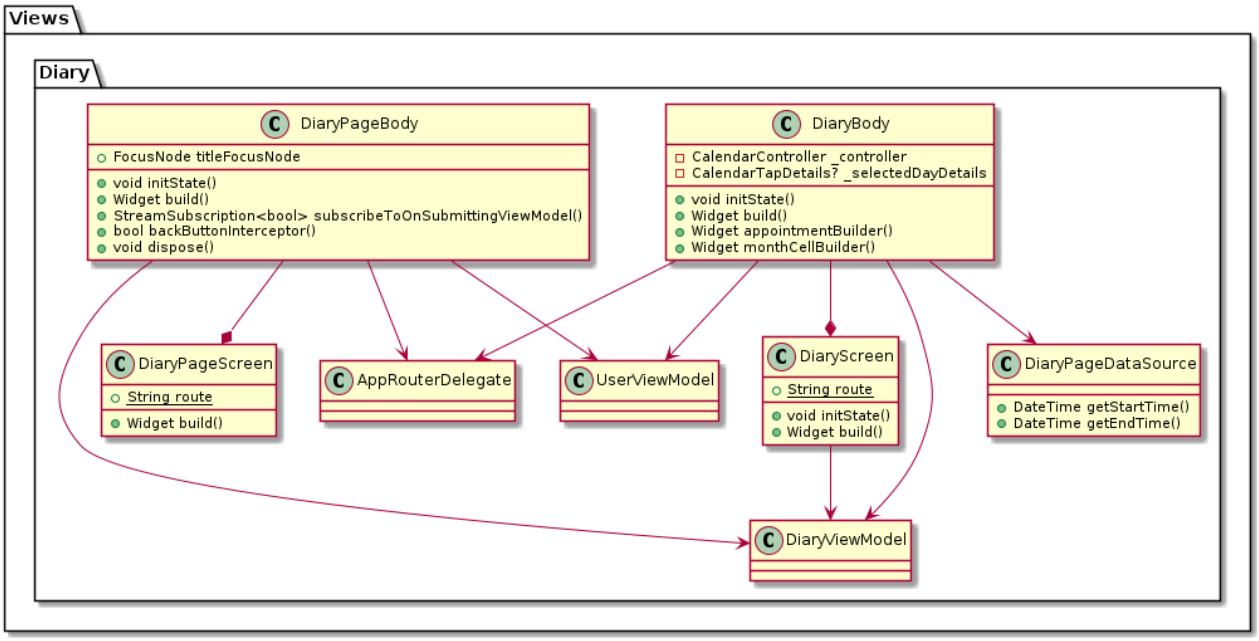


Figure 2.7: Diary - Class diagram

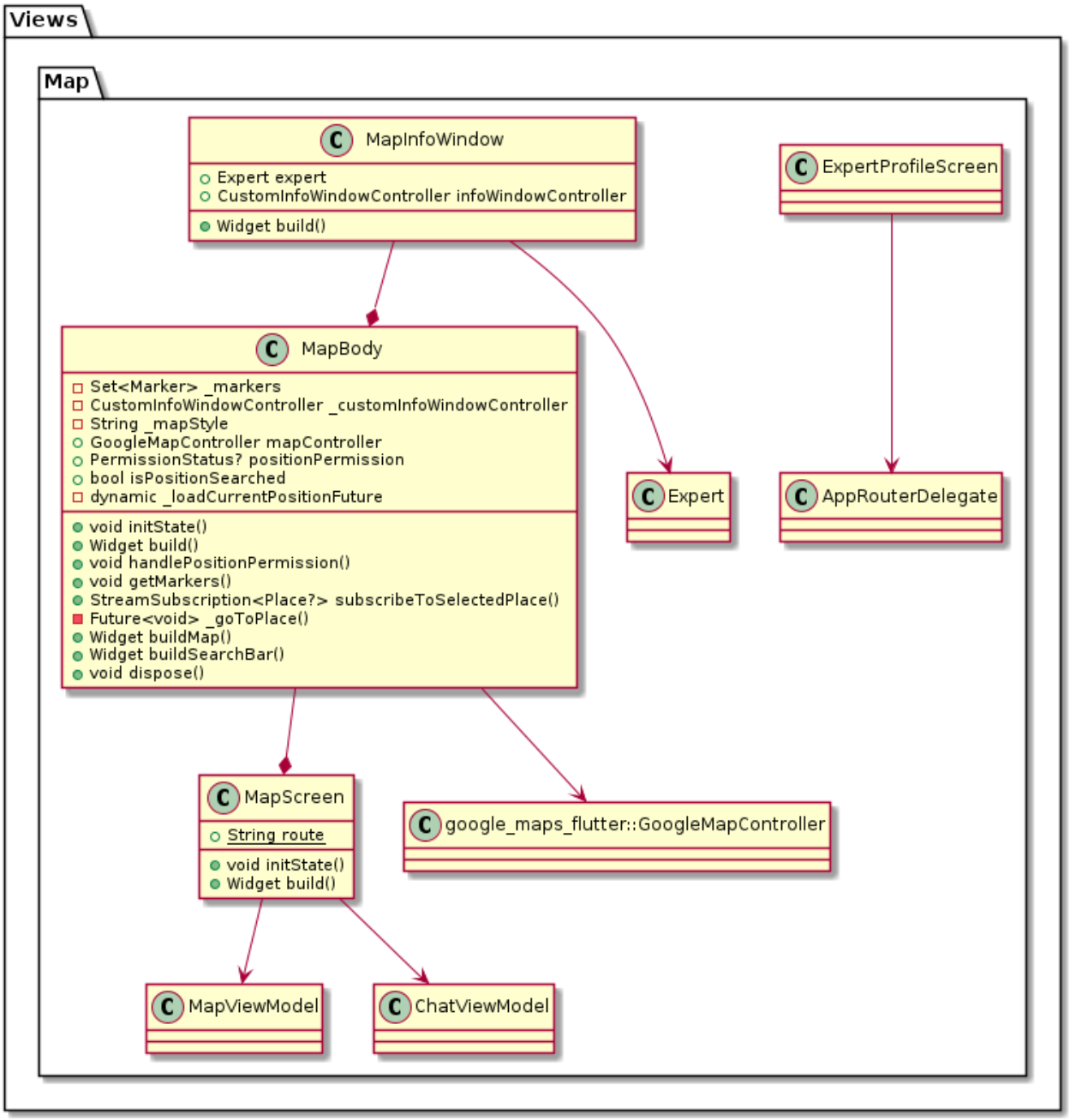


Figure 2.8: Map navigation - Class diagram

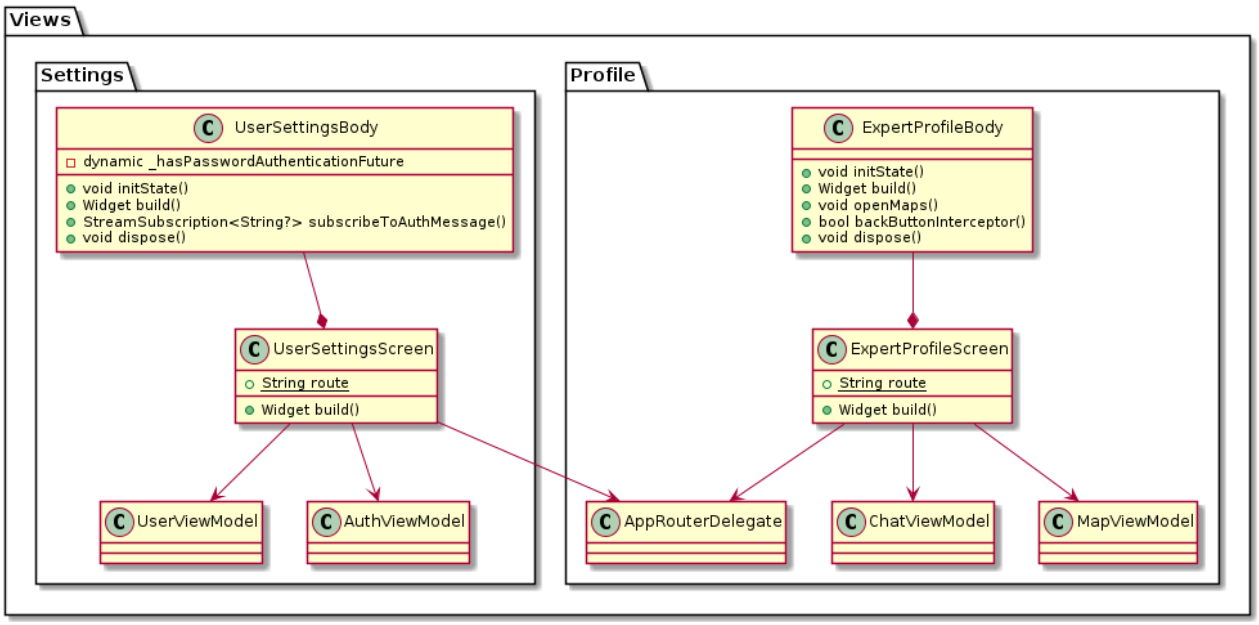


Figure 2.9: Diary - Class diagram

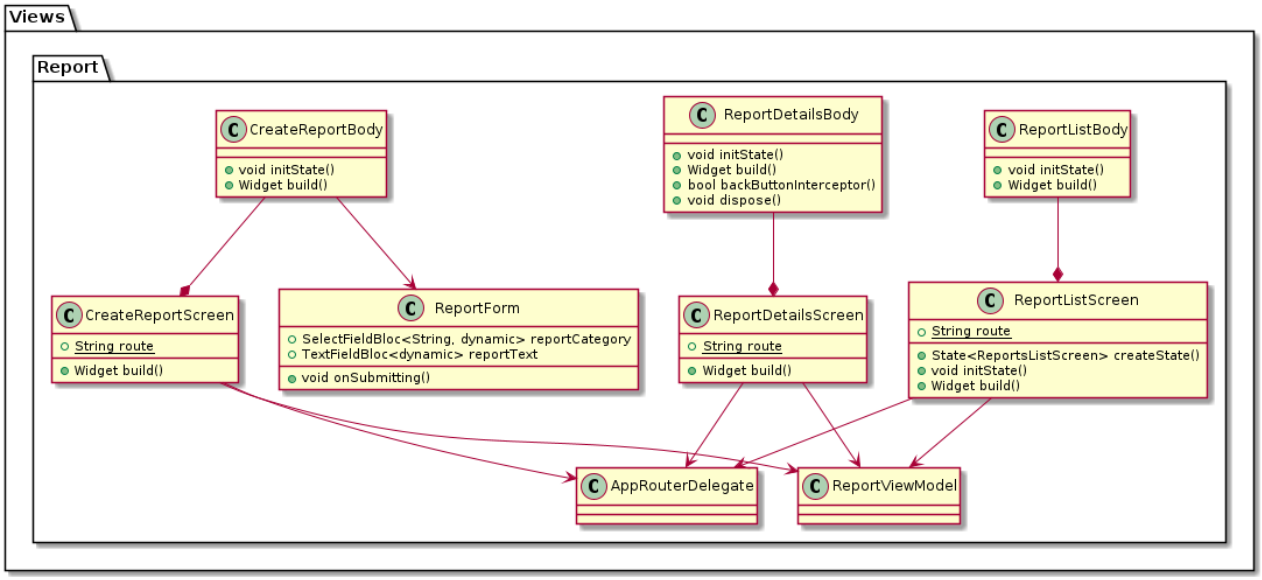


Figure 2.10: Profiles pages - Class diagram

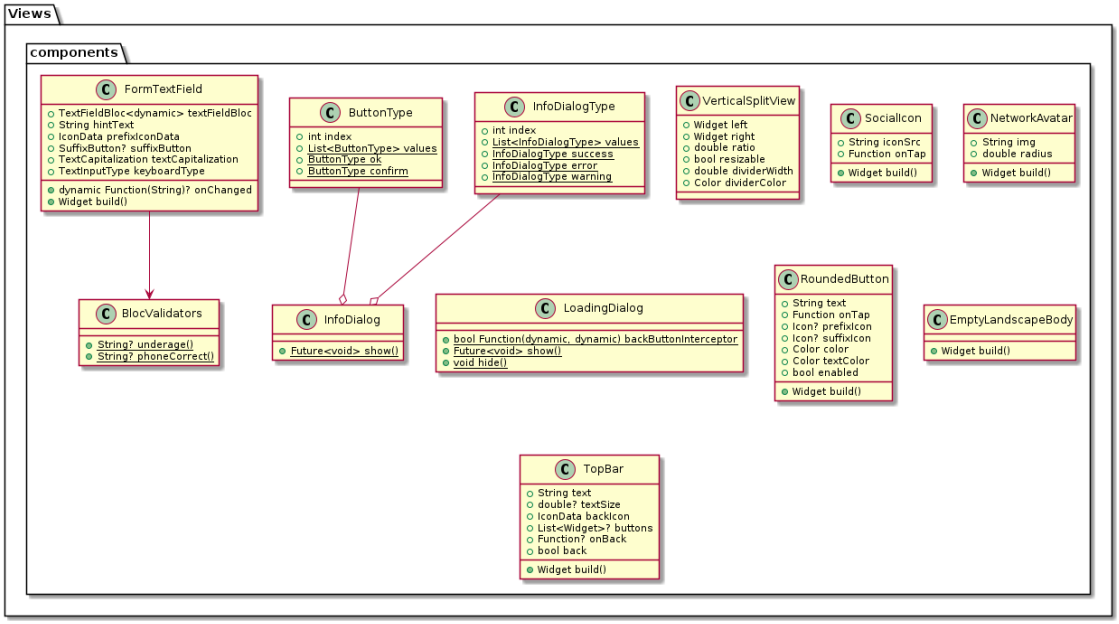


Figure 2.11: Reports - Class diagram

Viewmodels

The viewmodels are the core concept of the adopted paradigm. They work as a bridge between the views, the models, the APIs and the DB.

All ViewModels implement ***ValueNotifiers*** which are in charge of notifying the Views about changes in the data so that the UI can be dynamically updated. In the UMLs, representation of ValueNotifiers has been avoided in order to maintain readability, but their role is fundamental and must be kept in mind.

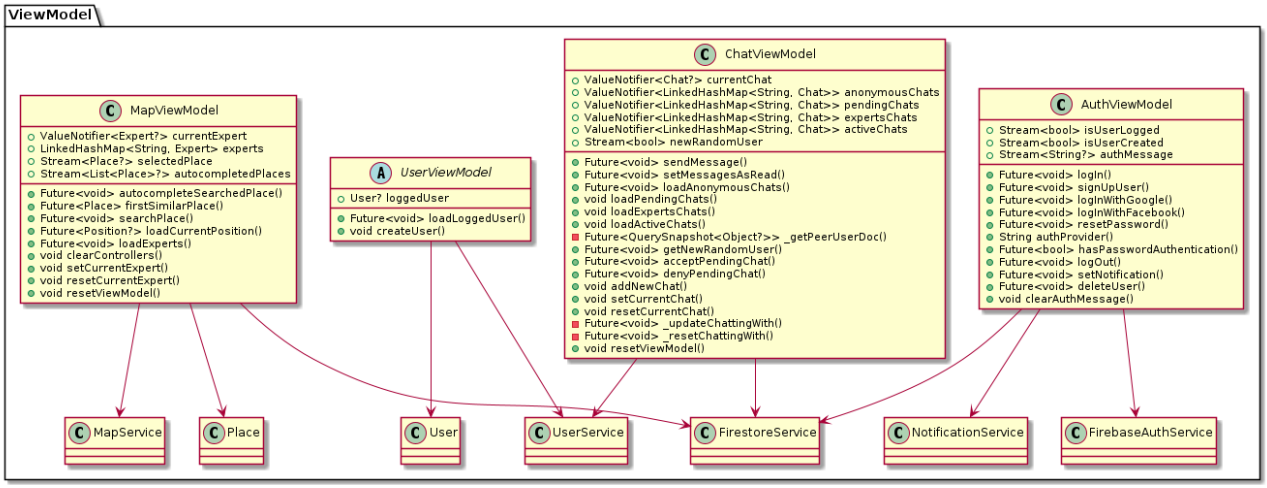


Figure 2.12: Viewmodels - Class diagram 1

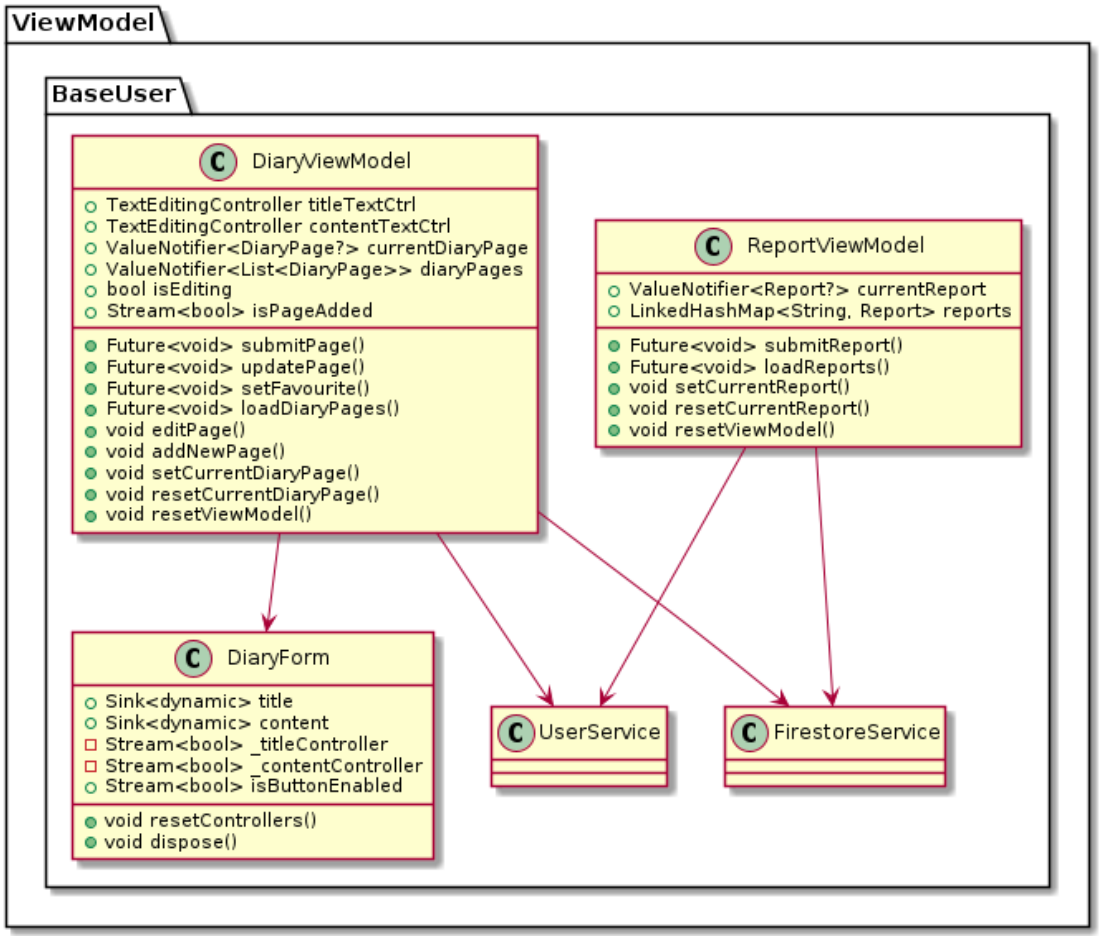


Figure 2.13: Viewmodels - Class diagram 2

Models

The models are responsible of the back-end logic. They represent the objects handled by the viewmodels and exposed to the UI.

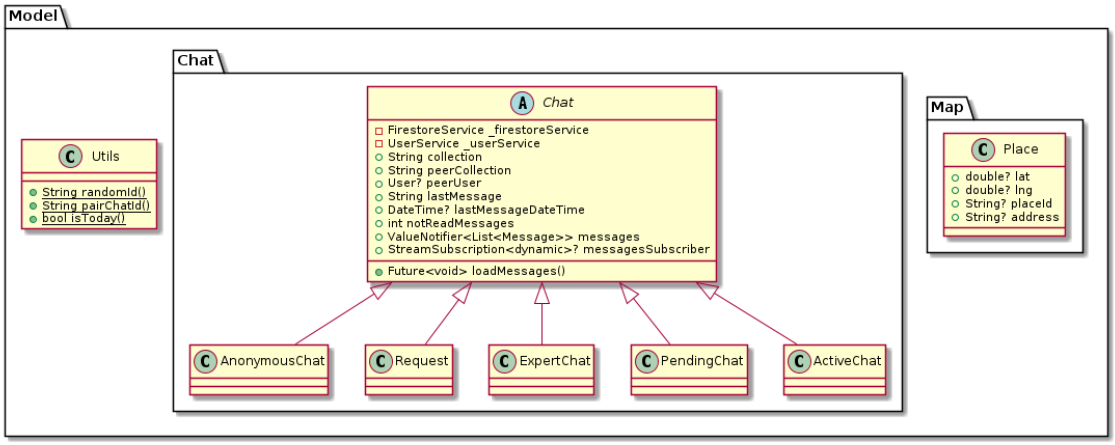


Figure 2.14: Models - Class diagram

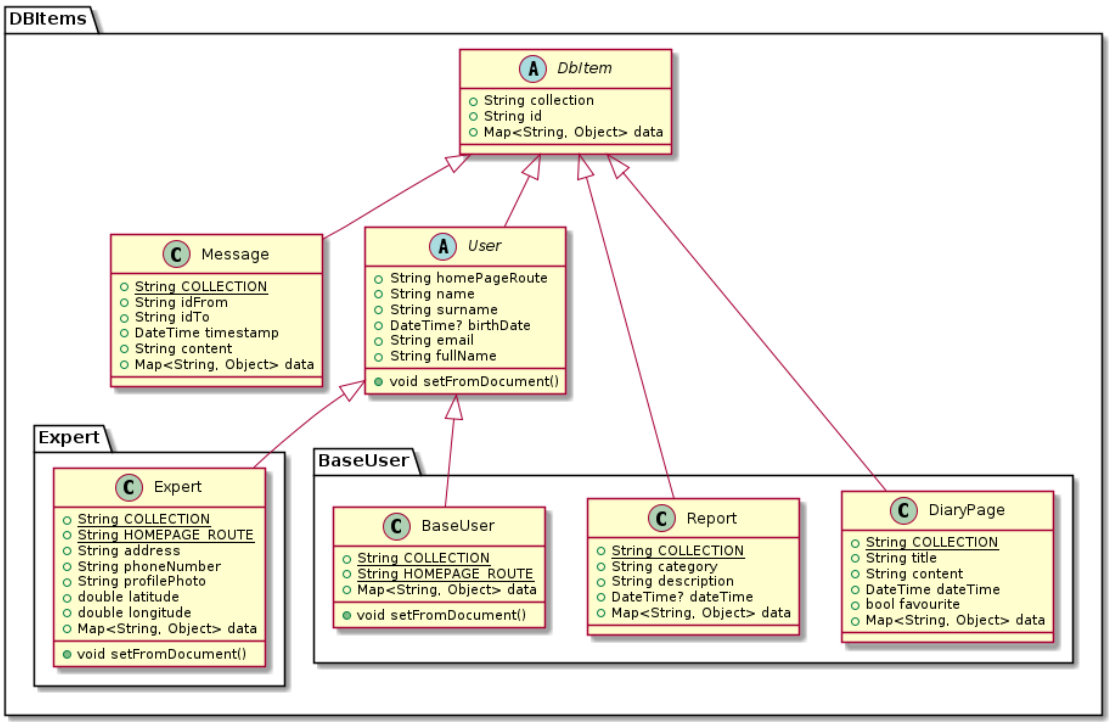


Figure 2.15: Database items - Class diagram

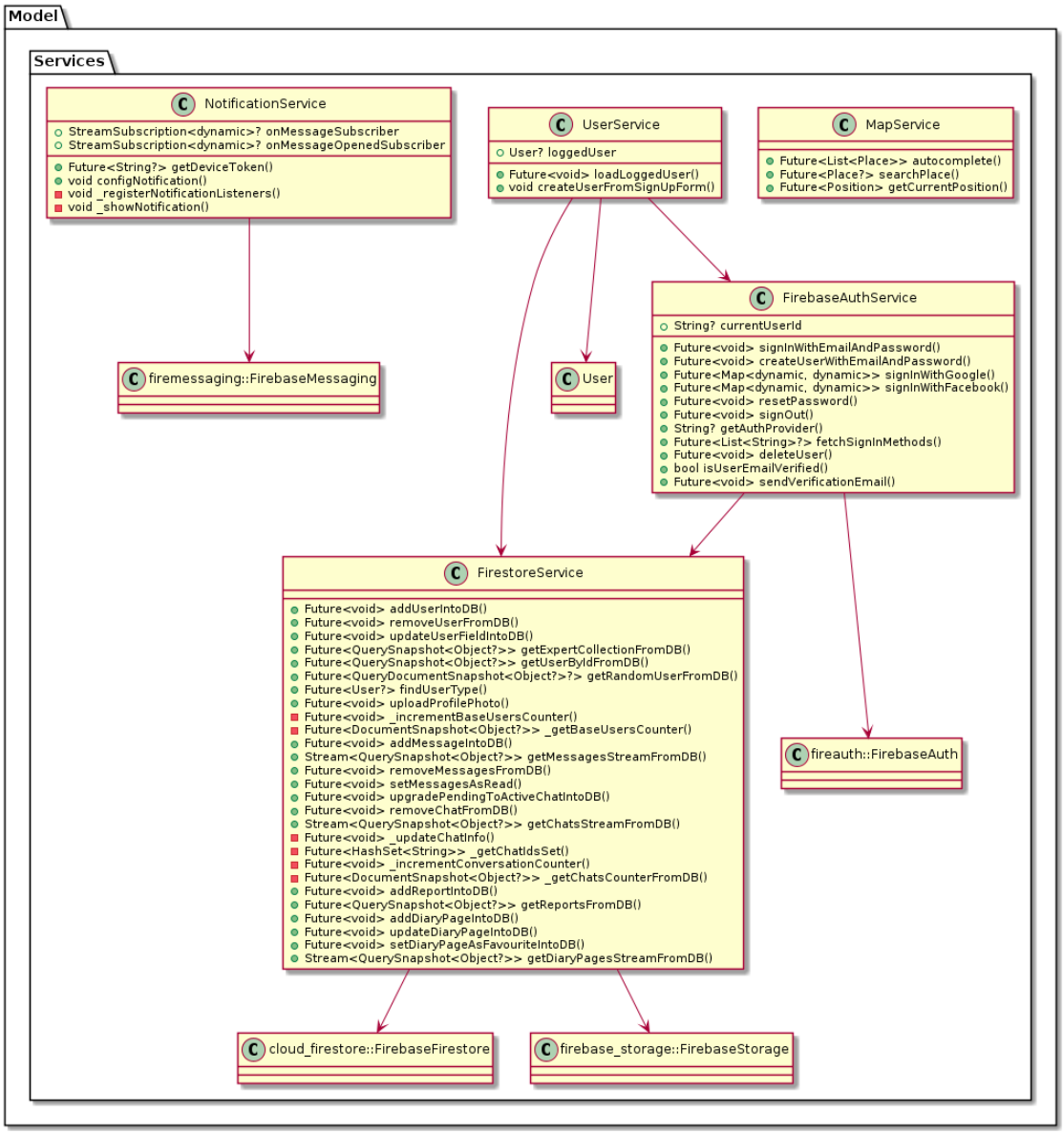


Figure 2.16: Services - Class diagram

2.3 Component view

The following diagram represents all the main ViewModel components of the system as well as the client's internal services and shows the interfaces through which they interact with each other as well as with the external services and the APIs exploited. All components are described and the provided interfaces are listed.

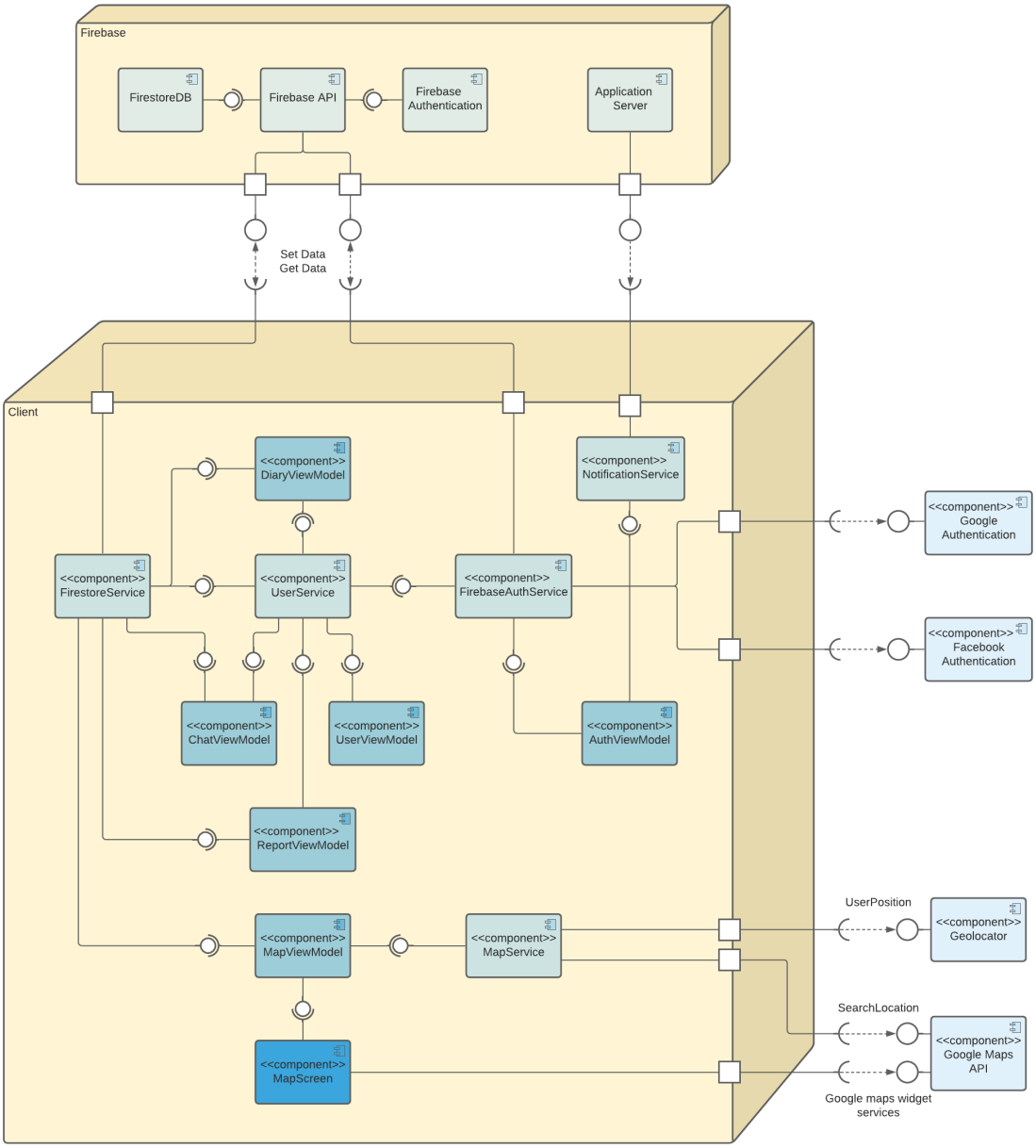


Figure 2.17: Components diagram

- **MapScreen:** The page in which a user can navigate through the world map, search a specific place and tap on any expert's marker. It is the *only screen shown* because it's the only one that directly exploits an external API, that is the *Google maps API*.

  - **MapViewModel:** The necessary ViewModel for the interaction with the map's services. It interacts with the *FirestoreService* in order to retrieve information about registered experts (needed in the MapScreen to render the corresponding markers) and with the *MapService*.
    - **MapService:** handles the autocomplete of the search bar and the actual research of the places by interacting with the external *Google* services. It also handles the identification of the user's current position thanks to the *Geolocator* plugin.

- **ReportViewModel:** The ViewModel that handles the creation of reports and the retrieval of old ones. Both are done via the interaction with *FirestoreService* which requires the identification of the currently logged user, whose ID is retrvied by exploiting the *UserService*
- **ChatViewModel:** as suggested by the name, this ViewModel handles all the chats functionalities. It is the most complex view model as it has to manage both types of conversations (anonymous and with an expert) which include:
  - Identify the currently open chat (via *FirestoreService*)
  - Identify which users are chatting (via *FirestoreService* and *UserService*)
  - Handle chats notifications
  - Handle the messaging process, manage the recovery of old messages and the sending of new ones (via *FirestoreService* and *UserService*)
  - Searching for a new anonymous user to chat with in case of anonymous chats (via *FirestoreService* and *UserService*)
  - Accepting or refusing a new conversation when a request is notified (via *FirestoreService* and *UserService*)
- **UserViewModel:** it plays the role of the bridge between the signup form and the *UserService* when handling a new user registration. It also plays the key role of redirecting to that same service the request of checking whether a user has already logged in with the device on which the application is running

- **DiaryViewModel:** this ViewModel acts as the manager of the diary section of the applications. It is changer of forwarding towards the *FirestoreService* the request of all the written pages as well as the creation of a new page. It also defines which is the currently open page, handles the requests of pages' editing and the change in their favourite status.
- **AuthViewModel:** as the name suggests, this ViewModel manages all the login, logout, password resetting, account deletion and sign up related functionalities. Regarding the latter, those can either derive from the compilation of the dedicated form or through the exploitation of *Google or Facebook authentications*. Finally, this ViewModel is in charge of registering the device to the *NotificationService*.
- **Firebase:**
  - **Firebase API:** allows the application to interface with Firebase storage and authentication services.
  - **Firebase Application Server:** exposes the Firebase functionalities, like its notification service.

2.4 Deployment view

The application has been deployed in three nodes: the mobile application (both the iOS one and the Android one) that runs on the device on which it is installed, the *BaaS* that handles both authentication and the communication between the application (and the device cache on which data is temporally saved) and the *Firestore NoSQL database* used for saving data regarding users, messages and reports:

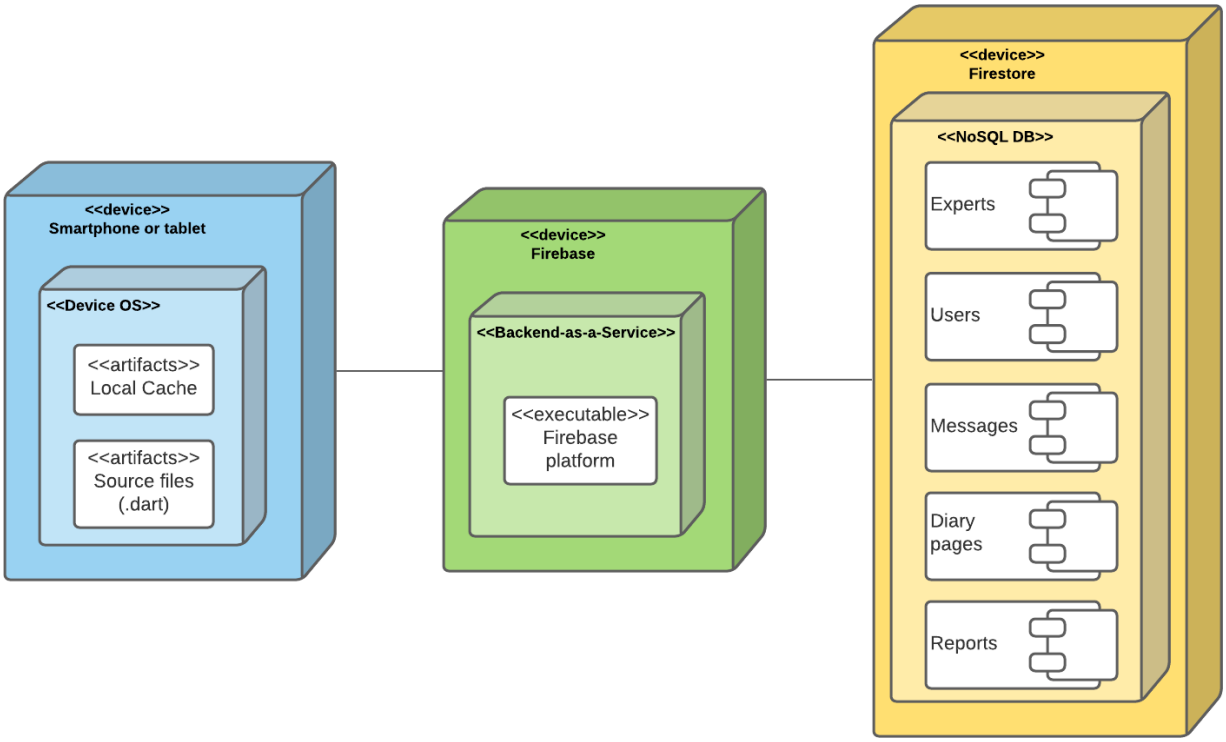


Figure 2.18: Deployment diagram

2.4.1 Technologies

Client application

sApport has been developed using **Flutter** framework in order to exploit its flexibility and ease of implementation of plugins and pre-built widgets. Moreover, a Flutter-developed application is by default available for both Android and iOS devices.

Backend-as-a-Service

Firebase is a BaaS app development platform that provides hosted backend services such as a realtime database, cloud storage, authentication, crash reporting, machine learning, remote

configuration, and hosting for your static files. It supports Flutter.

Database

The DB used is *Google Cloud Firestore*, an easily manageable pre-configured NoSQL database. It helps in storing and syncing data for both client and server-side development; it also supports automatic caching of data for using it even offline.

2.5 Design patterns and principles

- **Model-View-Viewmodel pattern:** is the software architectural pattern chosen. It is based on a strong separation between the UI and the back-end. The core concept of the pattern is the bridging functionality of the *viewmodel* components which are responsible of exposing the models, interact with servers and APIs methods as well as forwarding rendering and update inputs to the views.
- **Low coupling:** thanks to the chosen pattern, is possible to keep the coupling between the different parts and components of the application low by providing stable interfaces, by means of the *viewmodels* to rule their interaction. We especially make sure that no cyclic dependencies among the components. This principle concurs in making the application maintainable and easily updatable.
- **Dependency inversion:** related to the previous pattern, this pattern is used to keep the coupling between the application and the external APIs low by making sure that an API that provides a specific service could be easily substituted with another one. This principle may be relaxed if necessary as in the case of the *Google Maps API*, which is really pervasive in the client and a de facto standard, or in the case of the integration of the *Firebase* DB.

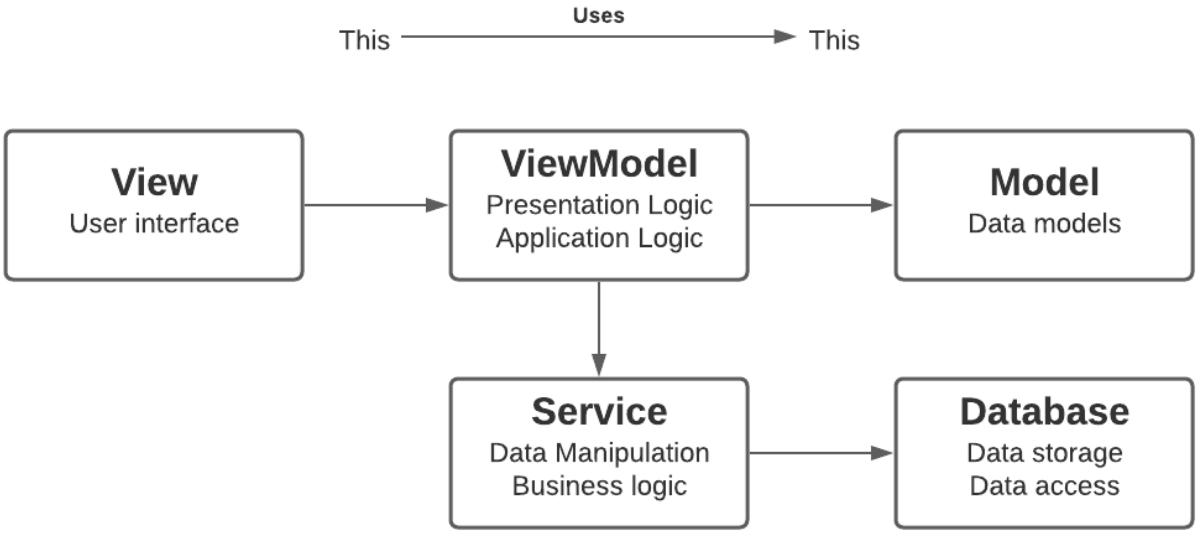


Figure 2.19: Model-View-Viewmodel pattern schema

Before proceeding to the *Runtime view* section, is important to visualize the schema of the MVVM pattern as depicted in the figure above: in fact, in the following sequence diagrams, the (frequent) communication between each ViewModel and the the linked Services has not been represented in order to maintain an easily readable aspect and highlight the devices' local dynamics.

The reader must keep in mind that all the data is stored in the database and its retrieval and modification is based on the usage of the services as already shown and described in *section 2.3*.

2.6 Runtime view

The following sequence diagrams will describe the interaction between the main components of the system during the execution of the most common and most interesting features.

Notice that the trivial processes of sign up and login have not been represented. The reader can refer to *image 1.1* in order to understand the flow of the application in those cases: the only consideration that needs to be added is the fact that information related to newly created users is added to a *Firestore database* (that is the non-relational DB used to store all the useful data of the application) and the same information is used in order to control the login process.

2.6.1 Creating a new report

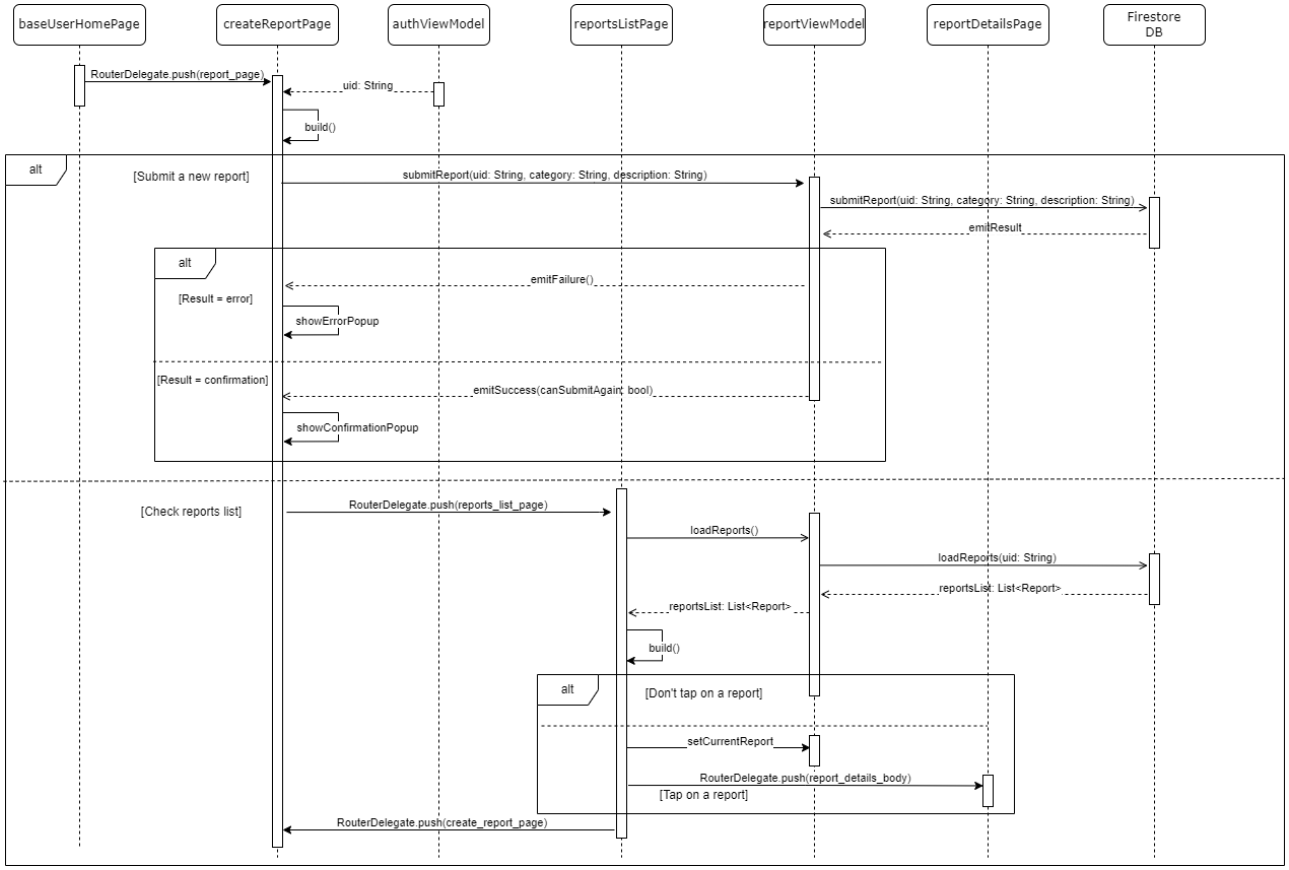


Figure 2.20: Create new report - sequence diagram

The sequence diagram above represents the process through which a user creates a new report or can navigate through all his previously submitted ones. Once the *createReportPage* has been rendered, the user can decide whether to fill it with the necessary information(category of the crime and its description) or to display the list of his reports. In the first case, after submitting the report, the *reportViewModel* is in charge of forwarding its content towards the *Firestore database*, adding the user is (*uid*). The result of this operations is communicated to the user via popup that appears in the *createReportPage*. In the second case, by tapping the dedicated button, the user is redirected towards the *reportsListPage*. Here all the reports are listed in decreasing order of creation and the user can chose to tap on one of them in order to read its details by navigating towards the *reportDetailsPage* or go back the createReportPage.

2.6.2 Searching experts with *Google Maps*

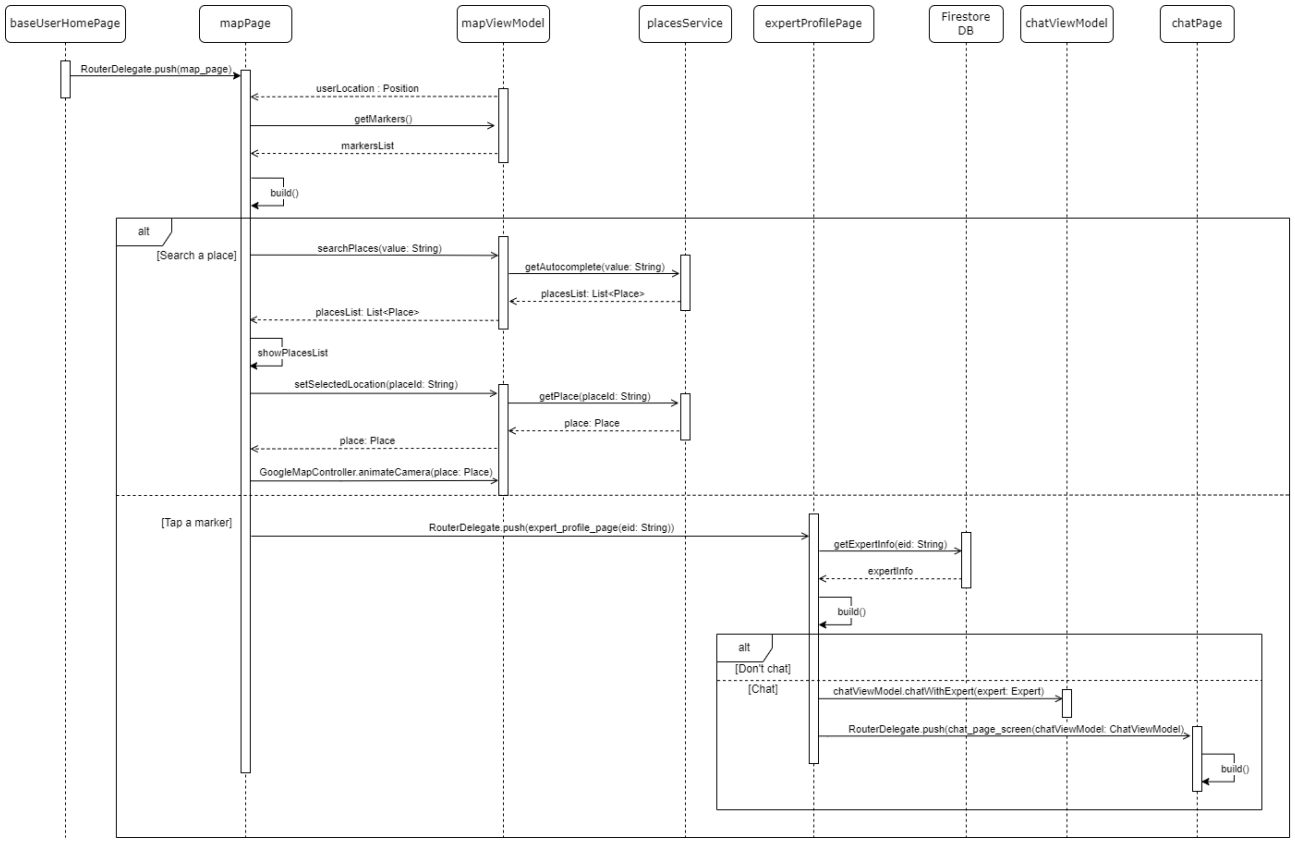


Figure 2.21: Search experts - sequence diagram

The sequence diagram shown in figure above represents the implementations of the *Google Maps API* into the systems and its usage: navigating through the map is the only way to find an expert nearby or in any possible location. Entering the *mapPage*, in fact, the map is centered around the user's position (who needs to give to the application the authorization to access his geolocation, otherwise a standard location is used) and all the markers representing an expert's medical studio are placed in the corresponding addresses. At this point, apart from scrolling though the map, a user can perform two actions:

1. *Search a place:* the idea is not to limit the user to view only the neighborhoods, but to have the possibility to search for any place to find registered psychologists.

2. *Tap on a marker:* this will trigger the opening of a popup containing a brief summary of the corresponding expert's information (such as name, surname, email and phone number). A tap on this newly created popup will lead to the *Expert Profile Page*, with the complete infos about the psychologist where the user has the possibility to start chatting with the expert or even open the smartphone's mail provider (tapping on the email), or

smartphone's call dialer (tapping on the expert's phone number).

2.6.3 Anonymous chats with other users

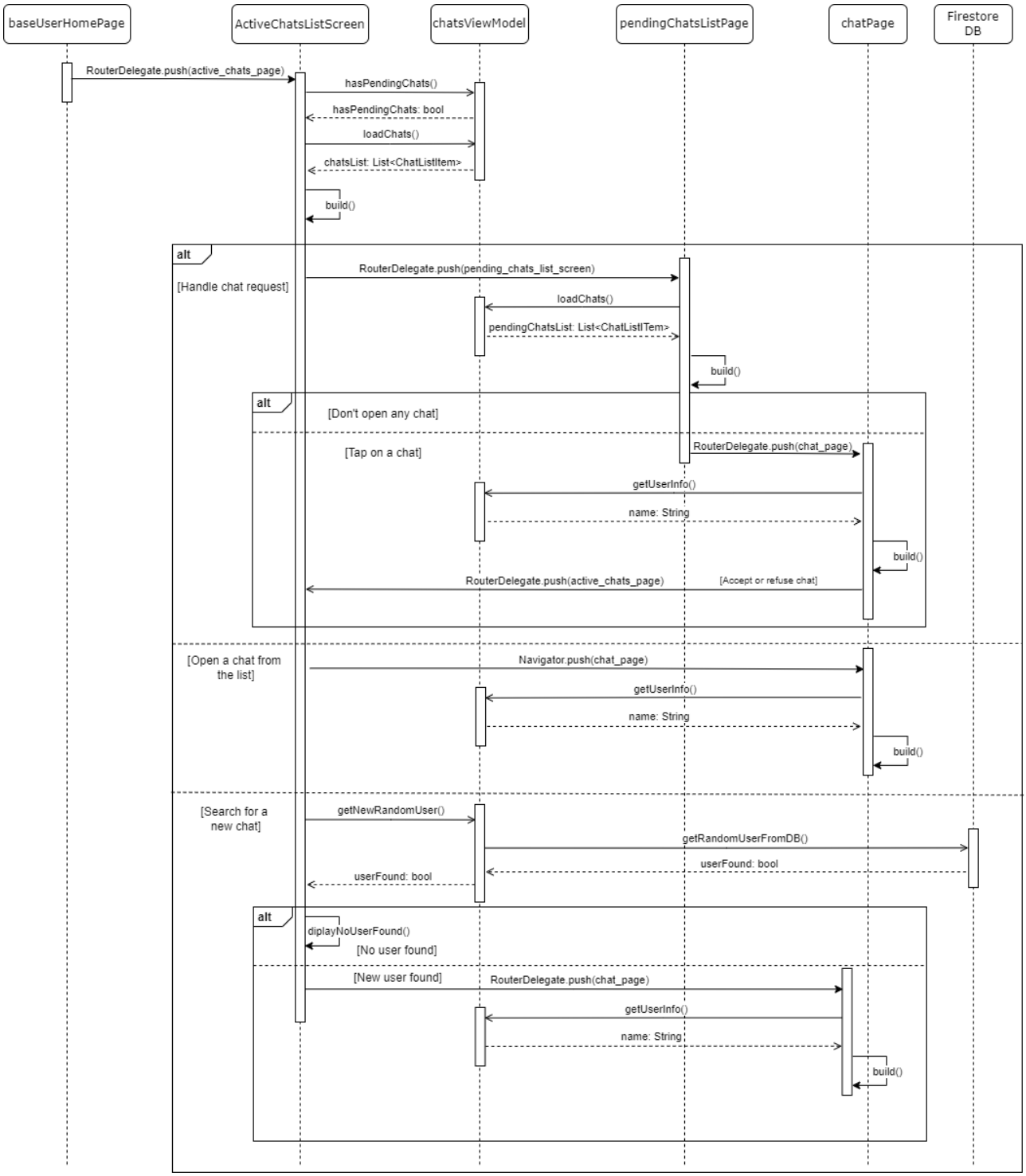


Figure 2.22: Anonymous chats - sequence diagram

The *anonymous chats sequence diagram* shows the three possible features that a user can exploit in order to interact with other users anonymously (the only information shown during a

conversation is the first name):

1. *Search for a new chat:* in order to start a new conversation, the system randomly selects one of the registered users (excepts those we are already talking with or those that have refused to chat with us) who receives a notification after receiving a message. If the pool of possible matches is empty, a message is displayed.

2. *Handle received requests:* as said in point 1, after receiving messages from a new possible match, is possible to navigate to the requests page and open the chat in order to accept to chat or to delete it (and avoid future matchings).

3. *Chat:* all the accepted chats are listed in the active chats page. From there, as in any messaging application, is possible to tap on one of them to open a conversation, read old messages, read and send new ones.

2.6.4 Chats with experts

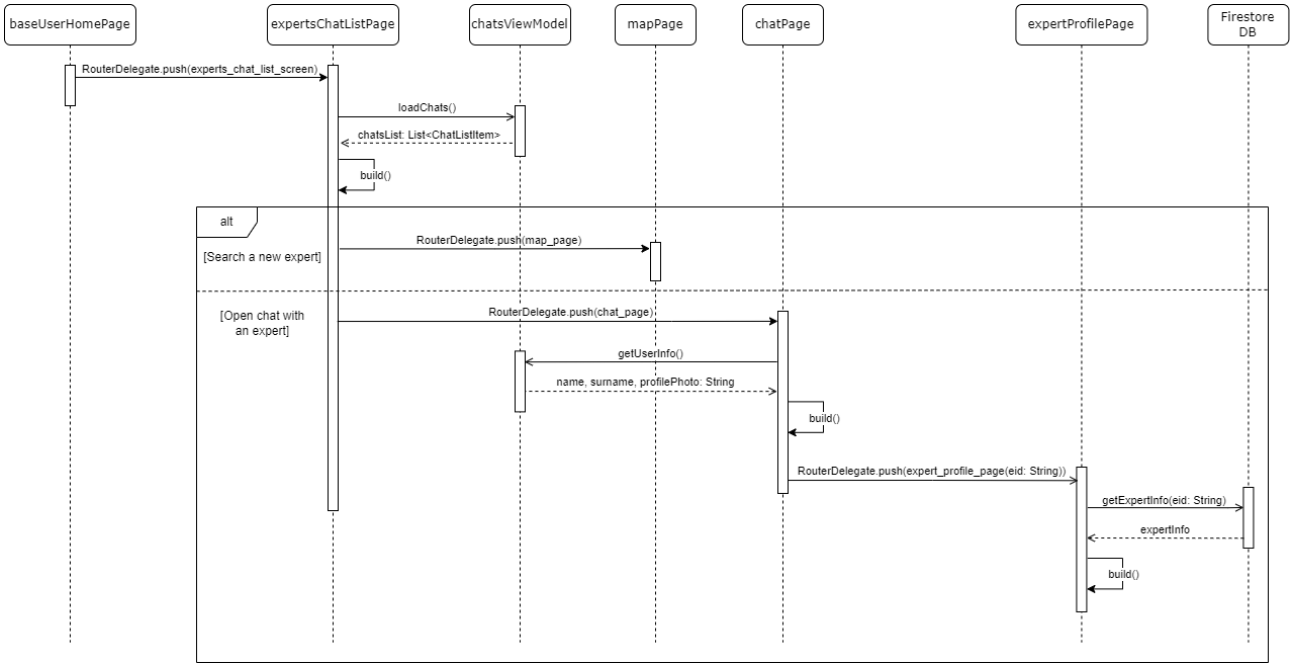


Figure 2.23: Chats with experts - sequence diagram

The figure above shows the sequence of steps that a users needs to take in order to chat with experts. Similarly to the previous case, there's a page (*Experts chat List page*) in which all the active chats are listed. But, differently from the anonymous chats, in this case both the photo as well as the full name of the experts are shown. From there, two paths can be followed:

1. *Look for a new expert:* this path simply leads to the map. From there, all the possibilities have already been described in paragraph 2.5.2 .

2. *Open a chat:* as with the anonymous chats, it is possible to open at any time one of the chats with the experts. There, a user can also decide to **open the expert's profile page** already described in paragraph 2.5.2 .

2.6.5 Experts chatting with patients

The logic and sequence of steps that an expert, from its **Experts Home Page**, has to follow in order to chat with his patients is almost identical to that showed in the previous diagram: the only differences stand in the parameters handled by the viewmodels and forwarded towards the DB that determine how things are rendered and listed in the UI. For this reason, we have avoided repetitions and the reader can refer to the previous paragraph.

2.6.6 Use the diary

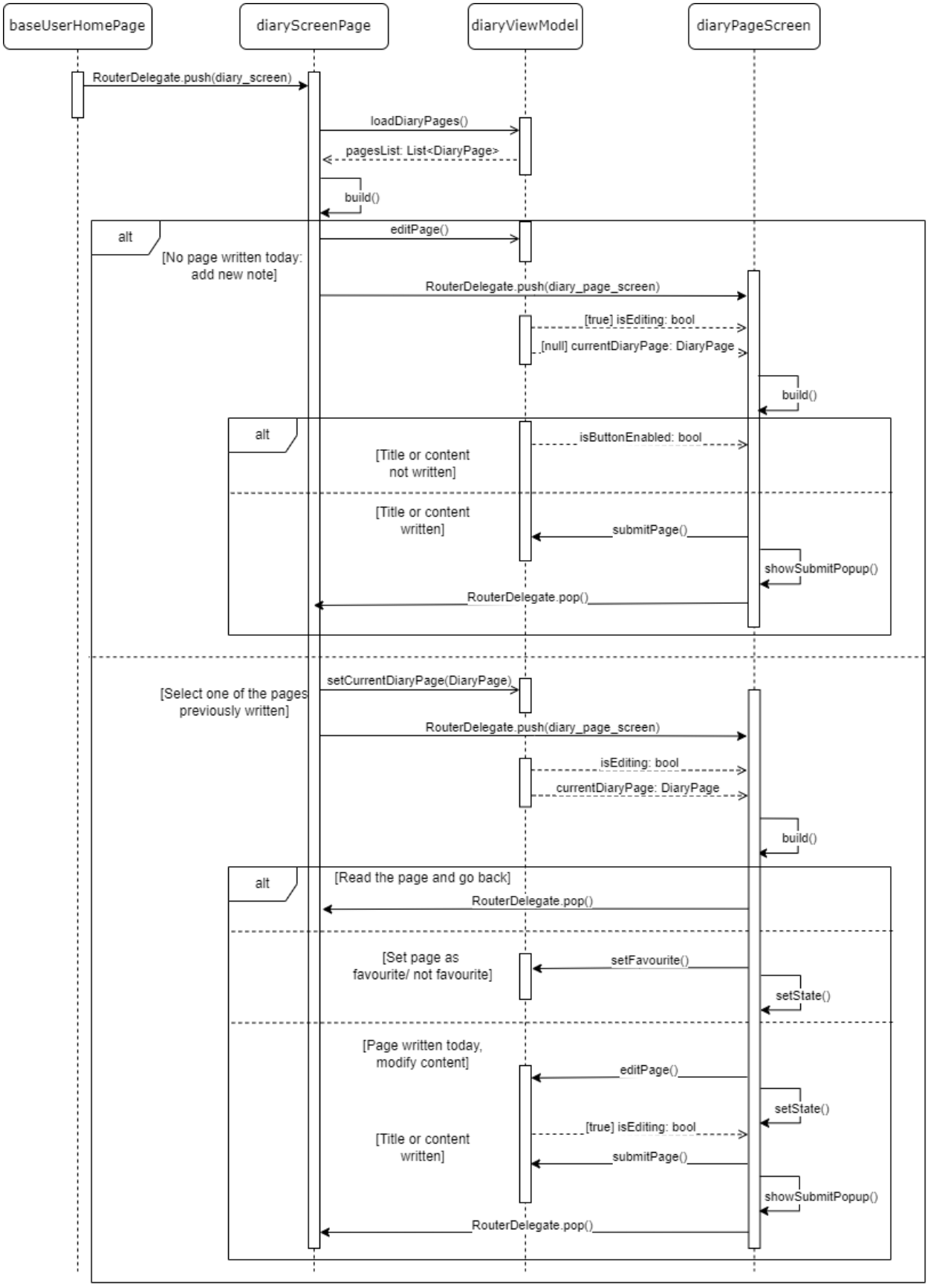


Figure 2.24: Using the diary - sequence diagram

Accessing the *Diary page*, the user can scroll through a calendar whose cells contain an icon if a diary page has been written in the corresponding day.
Two main paths can be followed from this page:

1. No note has been written today, so the user has the possibility to add a new one. Tapping on the dedicated button, he is brought to the *Diary page screen* where he can write the title and the content of the not and then submit. After that, the application goes back to the main diary page.

2. The user scrolls through the calendar and eventually chooses one of the notes. He is then brought to the *Diary page screen* in which the title and the content of the note are shown. Here two possible scenarios open up:

   - The user can decide to mark the note as favourite (or not favourite in case it was already marked as so) by tapping on the dedicated heart-shaped icon

   - If the opened page has been written *that same day* it can be modified by tapping on the appropriate icon. After that, the modifications are submitted and the application goes back to the main diary page

2.7 Plugins

Plugins are a core concept in Flutter applications development as their usage exponentially increases the customability of the UI as well as the ease of integrating new features. Plugins are also fundamental for **testing**, as described more in detail in the dedicate section, because of the need to mock all network functionalities (mainly DB access and image networking).
In this section, the main plugins that sApport depends on are listed with a brief description:

- **rxdart:** extends the capabilities of Dart Streams and StreamControllers.
- **email_validator:** A simple (but correct) Dart class for validating email addresses without using RegEx. Can also be used to validate emails within Flutter apps (see Flutter email validation).
- **google_sign_in**
- **flutter_facebook_auth**
- **Firebase plugins:** firebase_auth, cloud_firestore, firebase_core, firebase_storage, firebase_messaging, flutter_local_notifications, firebase_app_check
- **intl:** Provides internationalization and localization facilities, including message translation, plurals and genders, date/number formatting and parsing, and bidirectional text.
- **google_maps_flutter**
- **geolocator:** used to retrieve the last know device position or the current device position.
- **google_place** The Places API is a service that returns information about places using HTTP requests. Places are defined within this API as establishments, geographic locations, or prominent points of interest.
- **flutter_app_badger:** This plugin for Flutter adds the ability to change the badge of the app in the launcher. It supports iOS and some Android devices (the official API does not support the feature, even on Oreo).
- **image_picker:** A Flutter plugin for iOS and Android for picking images from the image library, and taking new pictures with the camera.
- **open_mail_app:** This library provides the ability to query the device for installed email apps and open those apps.

- **url_launcher**
- **cached_network_image:** A flutter library to show images from the internet and keep them in the cache directory.
- **syncfusion_flutter_calendar:** The Flutter Calendar widget has built-in configurable views such as day, week, workweek, month, schedule, timeline day, timeline week, timeline workweek and timeline month that provide basic functionalities for scheduling and representing appointments/events efficiently.
- **custom_info_window:** A widget based custom info window for google_maps_flutter.
- **permission_handler** This plugin provides a cross-platform (iOS, Android) API to request permissions and check their status. You can also open the device's app settings so users can grant a permission. On Android, you can show a rationale for requesting a permission.
- **shimmer:** A package provides an easy way to add shimmer effect in Flutter project.
- **image_cropper**
- **sizer:** A flutter plugin for Easily make Flutter apps responsive. Automatically adapt UI to different screen sizes. Responsiveness made simple.
- **flutter_form_bloc:** Easy Form State Management using BLoC pattern. Separate the Form State and Business Logic from the User Interface.
- **mockito:** Mocking framework used for testing
- **firebase_auth_mocks:** Mocks for Firebase Auth.
- **firebase_storage_mocks:** Mocks for Firebase Storage.
- **network_image_mock:** A utility for providing mocked response to Image.network in Flutter widget tests.
- **integration_test**

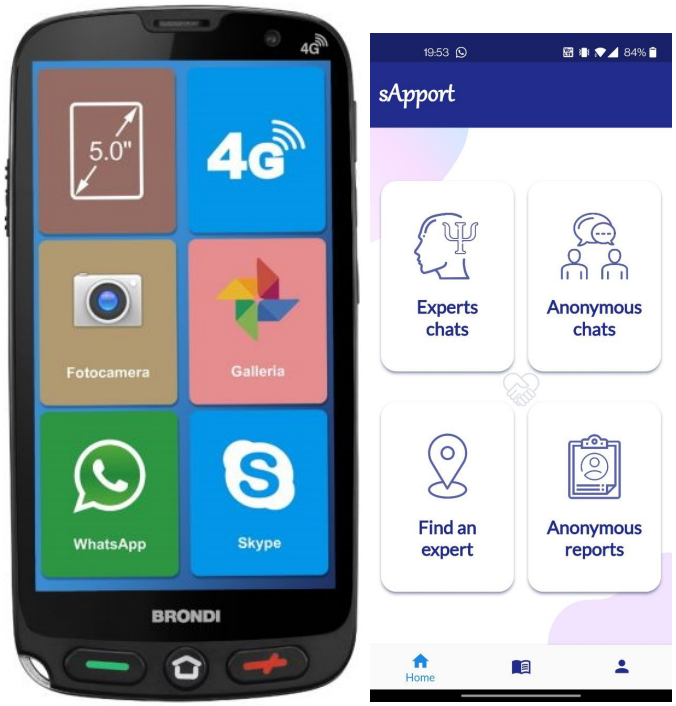
Chapter 3

User Interface Design

3.0.1 Design choices

The main inspiration in the designing the application's look and feel have been the so called *easy smartphones*, like those sold by the Italian phone company Brondi: their products, mainly sold to older people, people with cognitive disturbances and visually impaired people, are characterized by:

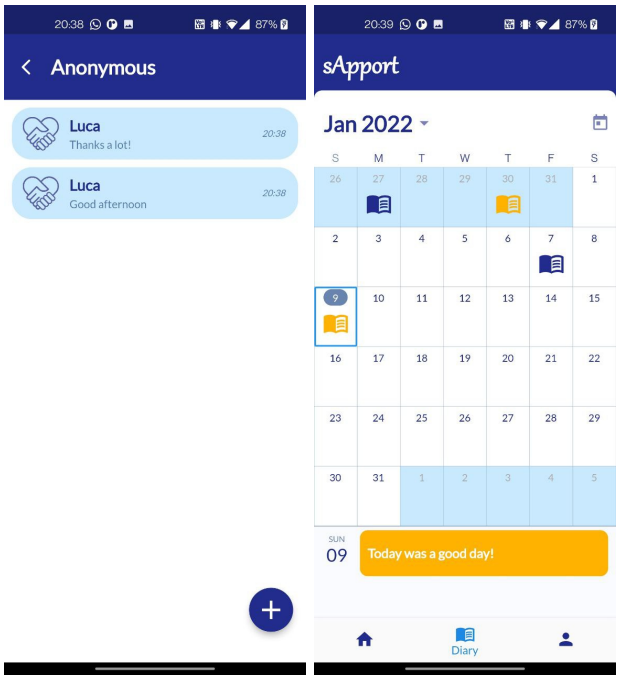
- Big and easily recognizable icons
- Large text font
- Straightforward pages, with a few buttons and text fields and dedicated to a single application's feature



(a) The Amico smartophone from Brondi

(b) *sApport's* home page

Figure 3.1: A comparison between the application's home page and Brondi's Amico smartphone home page



(a) The chats list page

(b) The diary page

Figure 3.2: Other examples of the clean and minimalistic UI developed for the application

3.0.2 Personalized look and feel depending on the device type

One of the strengths of *sApport* is the optimization of the UI **depending on the device** on which the application is run.

This mainly translates into two types of checks:

- **Device screen size:** all the application's widgets have been built defining their size as proportional to the device's width and/or height. This translates into an homogeneous appearance regardless of the screen.
- **Device type:** a further customization of the experience depends on the device on which *sApport* is run: in fact, it can be installed not only on smartphones (as seen in the previous section) but on **tablets** as well. On those handheld systems, **screen rotation is allowed** and gives the users the possibility to interact with the main functionalities in a *split screen* fashion while the device is kept in *landscape orientation* (that is, horizontally).

The following two screenshot exemplify how the UI experience changes while using the application in *landscape* mode by showing again an anonymous chat page and a calendar page.

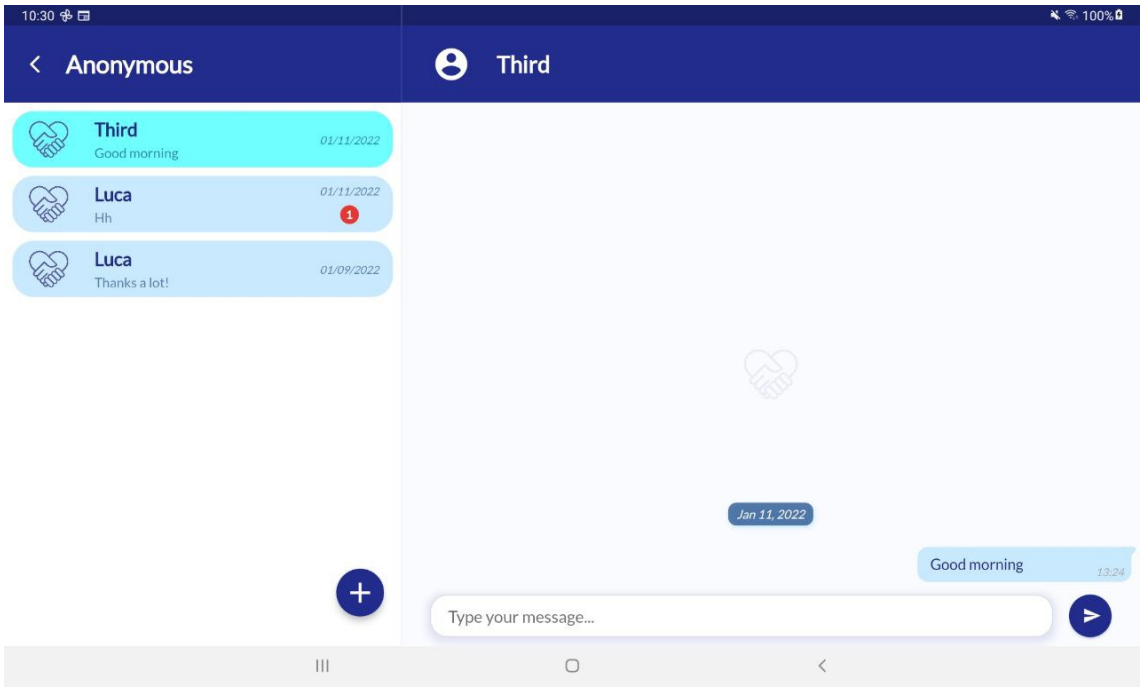


Figure 3.3: The anonymous chats page on a tablet in landscape mode

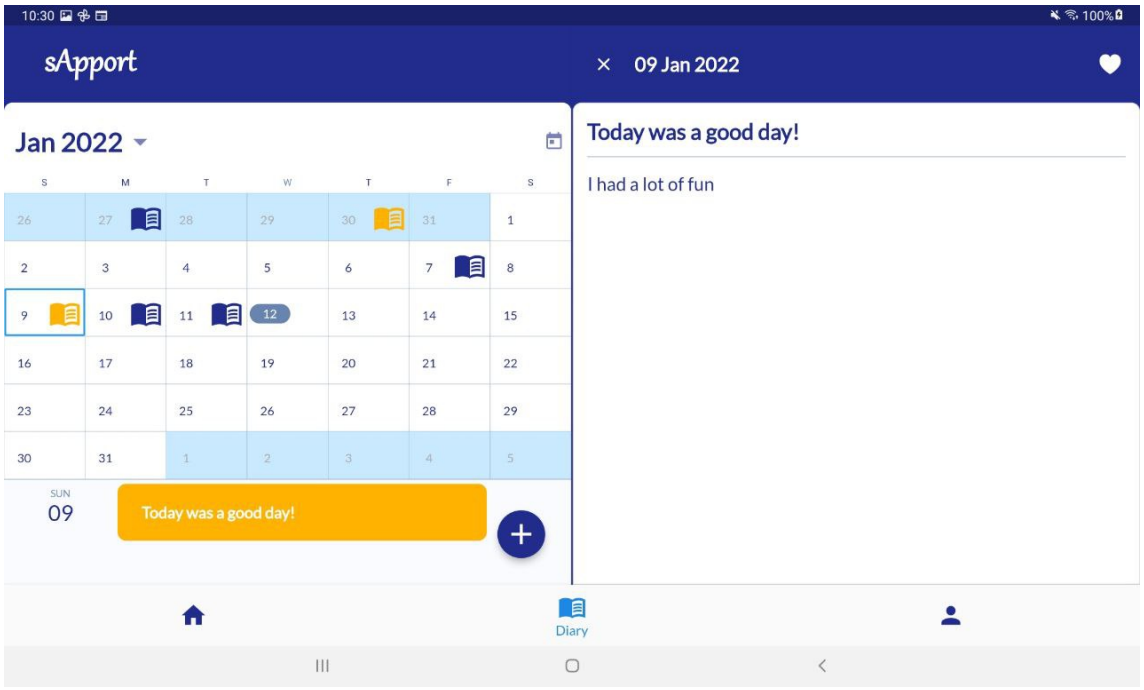


Figure 3.4: The diary page on a tablet in landscape mode

3.0.3 UX Diagrams

The following UX diagrams show the flow which the two types of users (*ordinary users*r and *experts*) will follow through their navigation inside the *sApport* application.

A special diagram is dedicated to the operations of sign up that the users must perform the first time they use *sApport* on a new device, in order to underline the difference in information asked depending on their category.

In order to increase readability, the majority of arcs going from a block to the main page (or in general to the previous page) have not been represented. However, it is implied that users have the possibility to perform these operations.

Moreover, as in any modern application, the system doesn't require any login operation apart from the first access on a device or after a logout. For this reason, the first page in the two diagrams that describe the application's features is *Home page*.

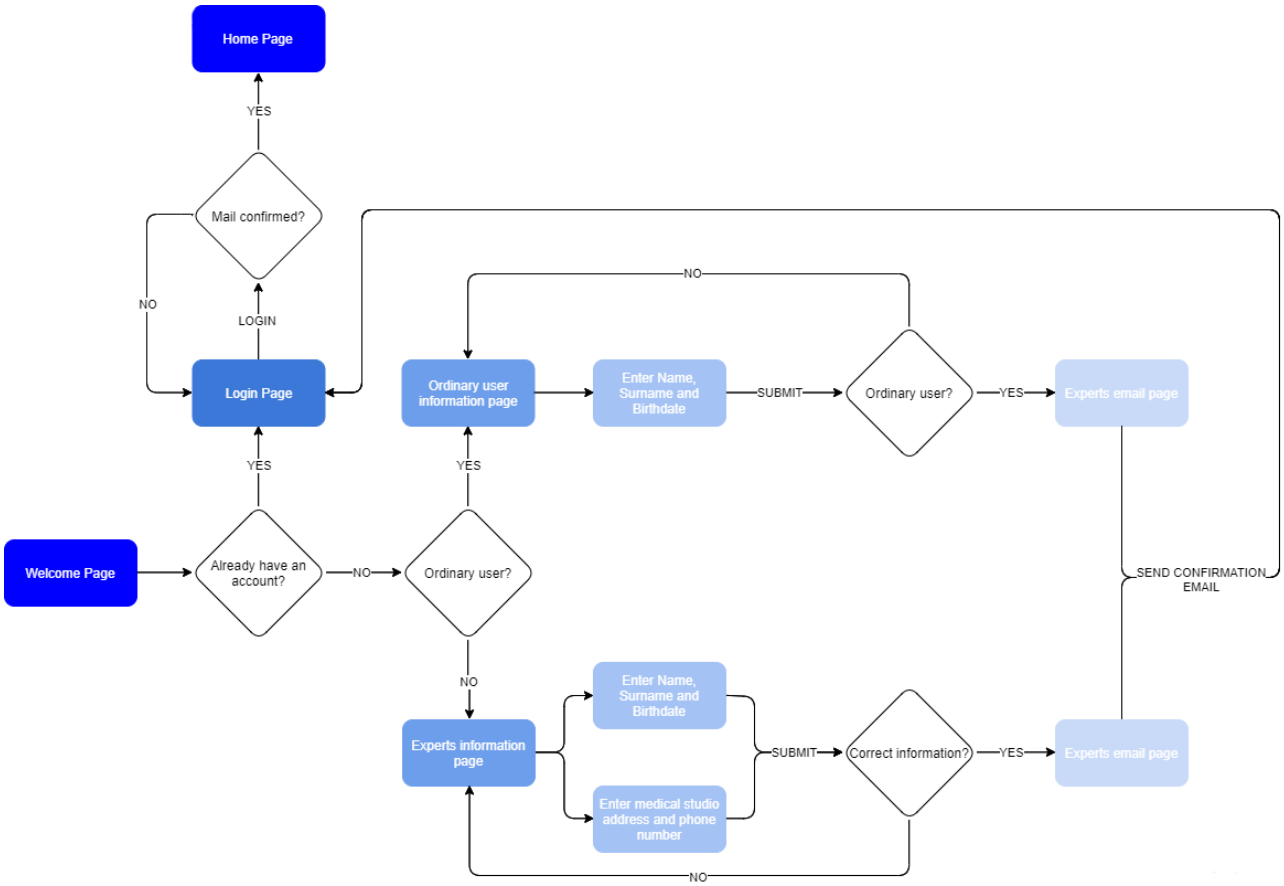


Figure 3.5: UX diagram - First access on the device

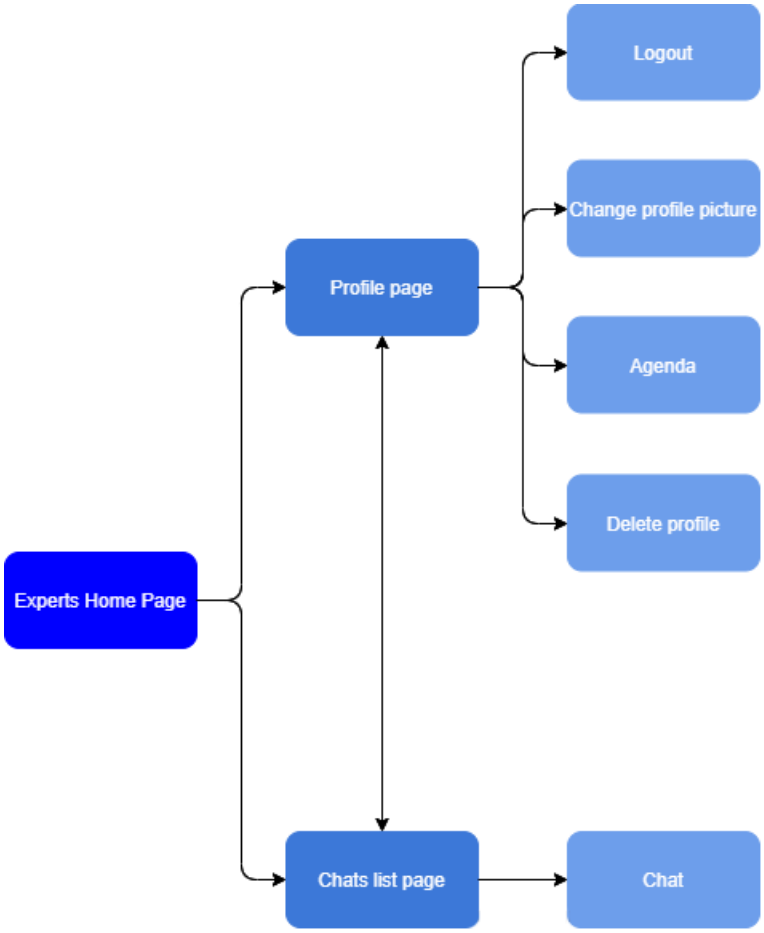


Figure 3.6: UX diagram - Experts flow

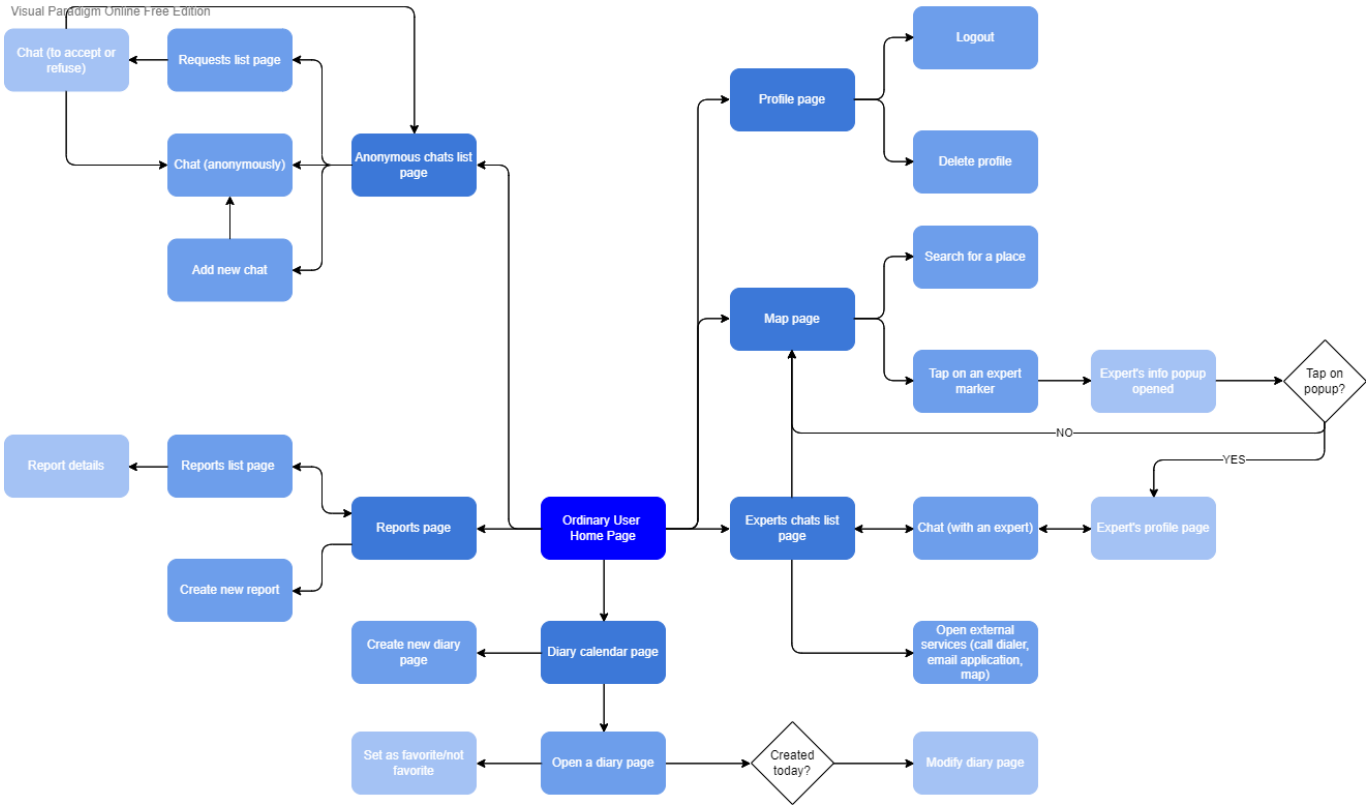


Figure 3.7: UX diagram - Ordinary users flow

Chapter 4

Unit, widget, integration and hands-on test plan

The test plan has had a key role in the development process of the *sApport* application because of the increasing number of widgets, files, functionalities and services implemented.

Before proceeding into a more detailed explanation of the test campaign, is important to underline which have been the main concerns into writing the test files.
As introduced and described in previous sections (see in particular *Chapter 2*) , *sApport* does not implement any local computation and data manipulation but instead relies on DB data storage and retrieval which implies networking. Moreover, other network functionalities such as the usage of network images (like expert's profile images, whose address is again stored in the Firebase DB) are implemented.
This has forced the usage of different **mocking frameworks and plugins** (see section 2.7) for the definition of *unit tests* and *widget tests* as they "verify the behavior of a method or class" (the former) and "verify the behavior of Flutter widgets without running the app itself" (the latter).
In particular:

- **Mockito:** This framework has been used for mocking higher level classes such as View-Models (and the related Models) for testing the Views and Services for testing the View-Models.
- **FakeFirebaseFirestore:** used for mocking the actual Firebase DB.

- **MockFirebaseAuth:** used for mocking all the authentication functions.

4.1 Unit and widget tests

4.1.1 Unit tests

Unit tests are handy for verifying the behavior of a single function, method, or class. Regarding the *sApport* application, which relies on the MVVM pattern, unit test have been used for testing at first all the *Models and the Router Delegate* behaviors and functionalities as shown in the following listings.

Figure 4.1: Unit tests - Models and Services

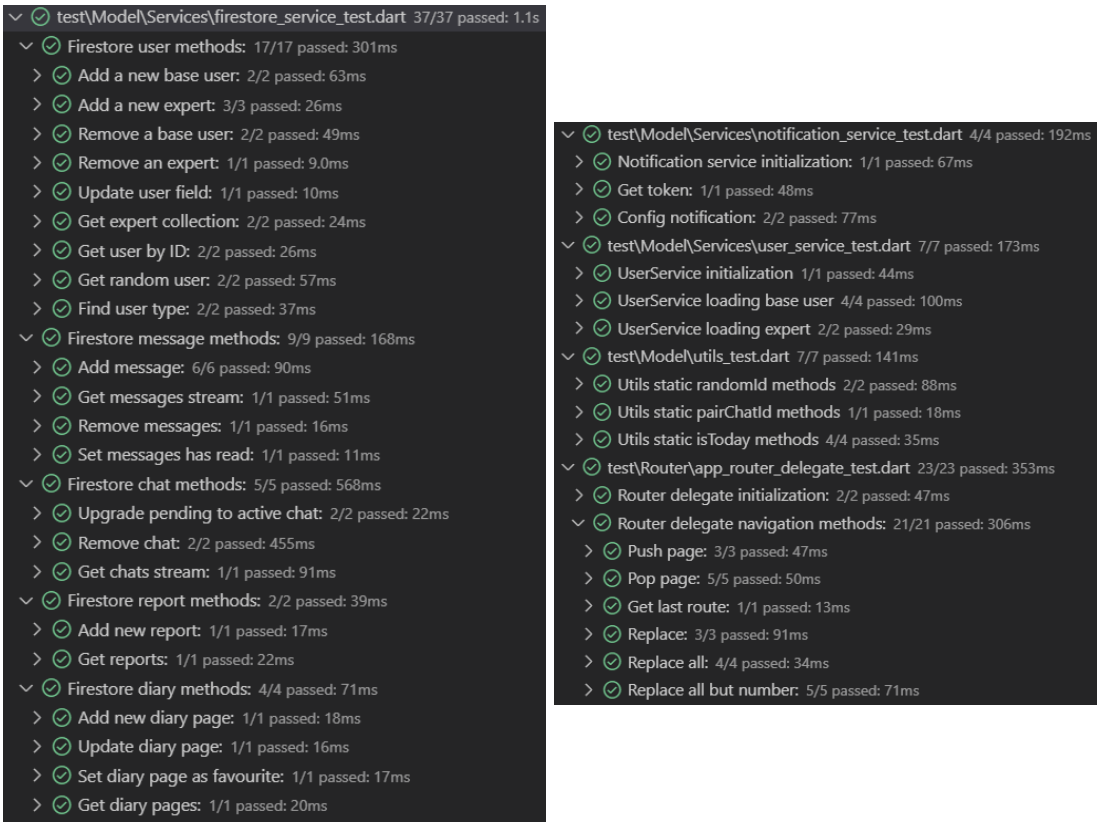


Figure 4.2: Unit tests - Models, Services and Router delegate

The successful creation and implementation of Models and Services, has then been followed by that of the *ViewModels* which have been unit tested as well.

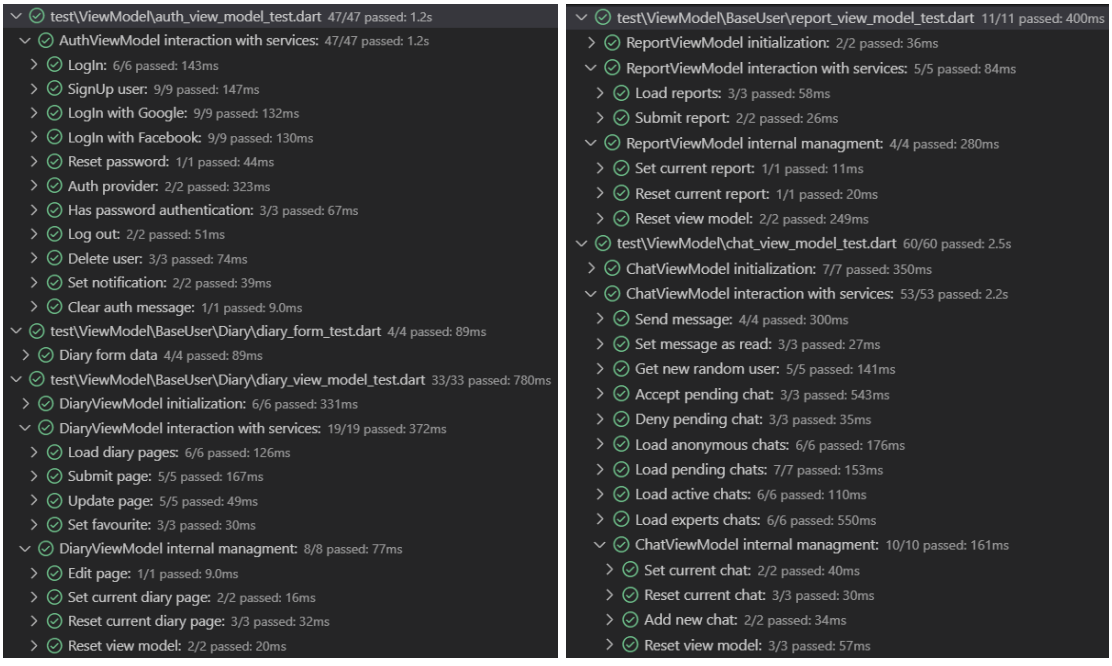


Figure 4.3: Unit tests - ViewModels

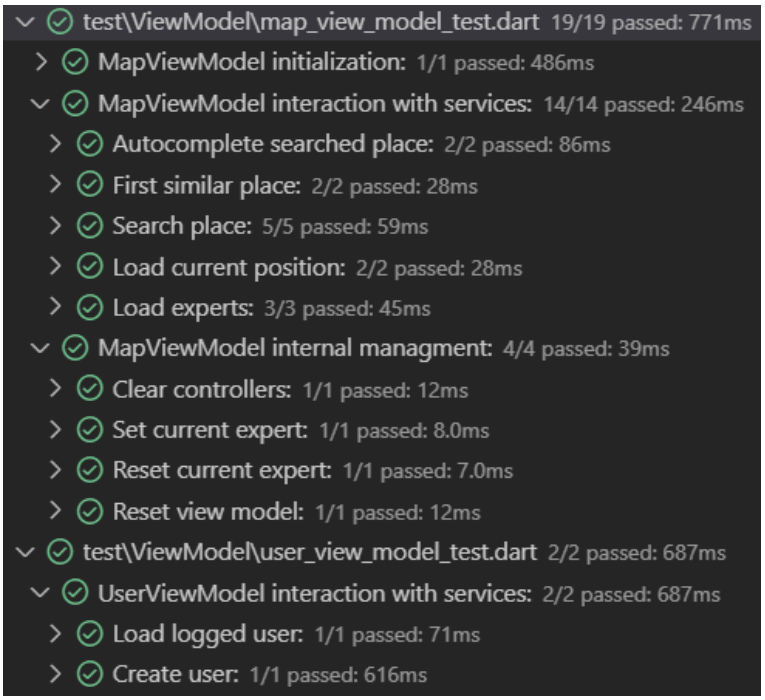


Figure 4.4: Unit tests - ViewModels

All the files listed above, tested and correct, are then exposed and exploited by the application's UI whose files have required a different kind of testing as described in the next section.

4.1.2 Widget tests

Widget tests play a key role in testing widget classes, their correct rendering and behaviour. These tests have been used for checking the correct rendering of the views, that is no widgets overflow, correct behavior of the widgets (for example buttons taps, TextField compilation etc.) and correct display of data coming from the (mocked) DB through the mediation of ViewModels and exploitation of Models, Services and Router delegate.

As shown in the following images, during the test campaign all the main functionalities have been widget tested and this ha brought to the creation of a smooth and clean UI, free from errors regardless of the device.

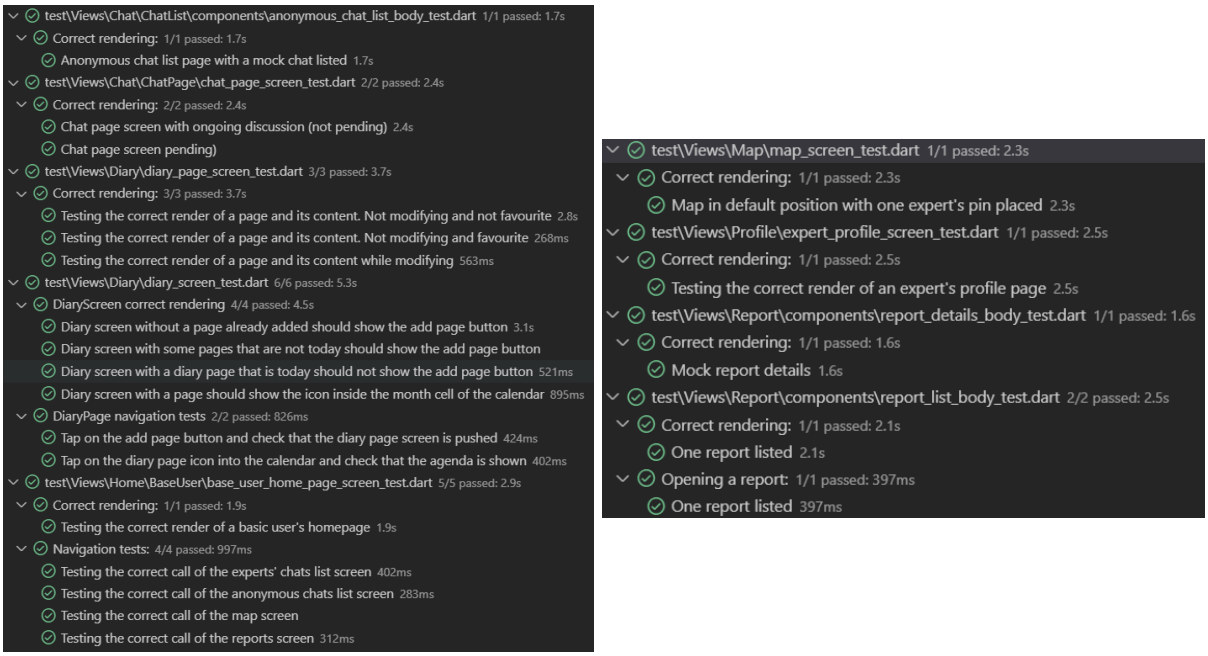


Figure 4.5: Widget tests

4.2 Integration tests

Unit tests and widget tests are handy for testing individual classes, functions, or widgets. However, they generally don't test how individual pieces work together as a whole, or capture the performance of an application running on a real device. These tasks are performed with integration tests. These tests are performed following the integration of a component into the system environment, their aim is to test interfaces and components' interactions and find possible integration faults like inconsistent interpretation of parameters or values, side effects on parameters or resources, omitted functionalities and, in general, communication and data flow through components

Integration tests are written using the integration_test package, provided by the SDK: a complete integration test campaign is part of the list of *future work*. The main functionalities of the application have already been tested with unit and widget tests, the next step is to perform integration tests in order to move to the phase of distribution for beta testing.

4.3 Hands-on tests

Apart from the automated and scripted tests described in the previous sections, *sApport* has undergone a period of real hands-on testing: 5 people, 2 with computer science-related knowledge and 3 without particular computer skills, have tried the application following some predefined use cases and giving their feedback. Moreover, out of the 5 testers, 2 have used the application on a tablet.
This process, has had a major role in simplifying and improving the user experience and UI.

4.3.1 Tested Use Cases

The straightforward cases asked to the testers were:

- Create an account of you choice (anonymous user or expert) and then log into the application
- After a login navigate through the home page and then (each alternative tested, random order left free to the user at each test run);
  - Open the map and try to get in touch with an expert
  - Open the experts chats list screen. Interact with the chats or try to create a new one
  - Open the anonymous chats list. Interact with the chats or try to create a new one
  - Create a report and then have a look at its details
  - Create a diary page, then do whatever you want with it (set as favourite, modify)
  - Check your profile and the logout
  - **(Only two testers who have tried the application simultaneously)** Tester A (anonymous user) try to get in contact with Tester B (anonymous user). Tester B decides to accept or refuse the chat
- Use freely the application and give your feedback during the activity and after it

Chapter 5

References and used tools

5.1 References

- Slides of the *Software Engineering 2* course by Prof. Di Nitto and Prof. Rossi
- Slides of the *Desing and implementation of mobile applications* course by Prof. Baresi
- *Flutter official documentation*
- *Firebase official documentation*
- *An Introduction to MVVM in Flutter*, by Mohammad Azam
- *Model-View-ViewModel (MVVM) Explainedr*, by Jeremy Likness
- *Brondi website*

5.2 Tools

- *Overleaf* for the LaTeXdocument;
- *Github* for version control;
- *PlantUML* for class diagrams
- *Lucidchart* for deployment diagrams
- *VisualParadigm* for the architecture diagrams
- *Draw.io* for component diagrams and sequence diagrams