Frunds

Run with friends!

POLITECNICO
MILANO 1863

Design and Implementation of Mobile Applications
Design document for the course project.

Authors
Federico Gibellini
Luca Rondini

Professor
Luciano Baresi

June 2023

Frunds - Design Document

Federico Gibellini, Luca Rondini

June 2023

Contents

1 Introduction

Frunds is an application targeted at runners and aimed at improving their training experience, thanks to the multiple functionalities it offers. Indeed, this application can be adopted for both personal and social purposes. In the first case, a runner could adopt the application to keep track of his performances thanks to the running tracker functionality, or he could try to discover new places in the search section or, finally, challenge himself by setting goals for his future trainings. In the second case, instead, a user could search for friends and see their latest performances or formulate and accept training proposals, in order to hang out with other people to share the training moment. This possibility of getting in touch with friends to share a run is at the basis of the application name: "Frunds" is, indeed, a combination of the words "friends" and "run".

1.1 Definitions and Acronyms

This sections contains some useful information to help the reader of this document. Here are provided some definitions and explanations for terms which the reader might encounter in this document and which might not be part of his knowledge.

1.1.1 Definitions

- **Friend**: a user of the application, different from the current one, whom the current user accepts to invite to his private trainings.
- **Session**, **Training session**: the instance of a run which is performed by a user in the real world.
- **Proposal**, **Training proposal**: the invitation by a user towards other users to meet in a defined place on a defined date and time and to undertake a running session together in the real world.
- **Private proposal**, **Friend proposal**: an invitation open only to the people that the inviting user has added as friends.
- **Public proposal**: a proposal open to all the users of the application.
- **Goal**: a personal athletic objective to achieve during one or more sessions.
- **Speed goal**: a goal based on the metric of the overall speed to keep during a session.
- **Duration goal**: a goal based on the metric of time during a session.
- **Distance goal**: a goal based on the metric of covered length during a session.
- **User**: a physical person who uses the application.

1.1.2 Acronyms

- **SDK**: Software Development Kit. It is a set of tools to allow development. In the following, it will be mostly mentioned alongside "Flutter" as "the Flutter SDK"; in such case, it refers to the whole set of tools and packages for the development and testing of a Flutter application.
- **UI**: User Interface. Here, used as a synonym for the graphical aspect of the application.
- **GPS**: Global Positioning System. In this context, a set of services commonly available on mobile devices that allow identification of the user position.

1.2 Usage scenarios

1.2.1 Tracking a session and setting personal goals

Elisa is preparing for an upcoming running competition by following an individual training schedule. In order to push herself to improve, she decides, before starting a training, to set some goals, such as the duration of the session or the overall pace to keep during the run. She opens the app, taps the button to create a new goal and fills in the form by selecting the type of goal (duration, distance or speed) and choosing an appropriate target value.

Later that day, she leaves her house, reaches her favorite running place and starts tracking her session. At the end of the training, she stops the tracking and, back in the home page, she can see if the goals were reached or not. The goals not yet reached will remain in the home page to be completed in the following sessions.

1.2.2 Proposing a training session to the app community

Francesco is a runner who spends a lot of time training, but for this reason he is not always able to find friends available to go with him. Thus, he thinks that a good idea could be to search for other people in the app community willing to meet him and have a training session together. Francesco knows a wood which is perfect for running, so he decides to publish a training proposal in the application by tapping the "+" button in the app home page and by filling in the form with the place, date and time. He also selects the "Public" option in the dropdown menu to make sure everyone is able to find it.

A few hours later, Chiara, another user of the app who likes to challenge herself at running while meeting other people, opens the "Search" section and then selects the "Map" tab, where she can find the training proposals by searching in her surroundings. Here, she finds Francesco's proposal in a nearby location and, since she is also fine with the proposed date and time, she accepts the invitation. On the scheduled date, they will be able to meet at the proposed location and train together.

1.3 Use cases

The functionalities provided by the application are:

- **Registration/login**: any user can register or login on the application. Authentication is currently provided with only email and password.
- **Session tracking**: users can track the sessions they take part to. In order to do this, they are allowed to start, pause, resume and stop the tracker, which leverages the device GPS services to collect the set of coordinates in which the users pass. In the meanwhile, a timer records the time spent by the user running. In any moment of the session, the tracker provides the users with information about the current duration and run distance. This information will always remain available to all the users after the session.
- **Search**: by means of a search functionality, users can find other runners or places by searching for their name, whereas they can find proposed trainings by moving a map which will display all the available proposals within its boundaries.
- **Set goals**: users are allowed to set their personal athletic objectives, which can currently be based only on the metrics of distance, duration or pace. These goals are achievable by means of sessions and their level of fulfilment is checked after every session.
- **Interact with other users**: users can interact with other users by proposing them trainings and joining theirs or adding such users to their friends.
- **Training proposal**: users can propose training sessions with different levels of visibility and they can also join trainings proposed by other people. In order to propose trainings, users have to fill in a form specifying a location, a date, a time and a privacy level. If this level is set to 'Public', all the users of the application will be allowed to join the proposal; if the level is set to 'Friends', only people added by the user as friends will see the proposal and, thus, will be able to join it. In any case, a user who has already joined a proposal can decide to withdraw and leave it.

All the use cases mentioned here are also summarized in the Use Case diagram reported in Figure 1.

1.4 Document structure

- **Chapter 1 - Introduction** The introduction provides all the necessary information to induct the reader to the content of the rest of the document. This section provides definitions and explanations for the terms and acronyms that will be used in the document. In addition, the section contains a description of the application functionalities, along with exemplary scenarios in which users might realistically interact with the application.

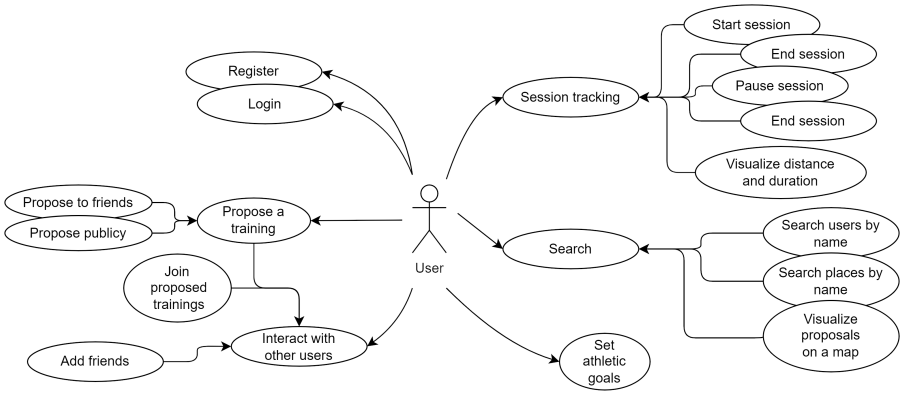


Figure 1: Use case diagram

- **Chapter 2 - Architectural design** This section describes the application architecture and the design choices behind it. First, an overview of the whole system is provided, then the specific features of its component are addressed. The section provides also indications for the deployment of the application and discusses the design patterns and other choices followed during the implementation. The whole discussion is supported by diagrams.

- **Chapter 3 - User interface design** This section addresses the matters of the user interaction with the application. In the first of its two subsections, an interaction flow diagram is presented to describe the allowed flow of actions. In the second subsection, screenshots from both the smartphone and the tablet versions are provided.

- **Chapter 4 - Implementation and testing** The final section reports the procedures followed during the application development process. First, a few notes on the implementation are inserted, then the testing phases and choices are deeply discussed. In the end, the line coverage results are also displayed.

2 Architectural design

2.1 Overall description

Client-server paradigm The architecture designed for the application can be framed in the client-server paradigm. The user is supposed to interact with the main portion of the application by means of his device, thus accessing the client section of the application. The data accessed by the users are, instead, kept on a server.

Hybrid-tier architecture The application cannot be framed in any of the common tiering patterns. Indeed, the architecture shows portions of its behavior which might seem to comply more to a 3-tiered architecture, whereas others seem to adhere to the the 2-tier style. However, this latter case is predominant in this system. Indeed, considering the common taxonomy of application components in 3 layers, Presentation, Application Logic and Data, this application has the first 2 both generally residing on the client side and, in most of the cases, handled by common classes which adequate the User Interface (Presentation layer) based on the output of some computation (Business Logic layer). The Data layer, instead, is totally deployed on the server.

The hybridness of this solution, however, lies in the interaction with the external services, especially those used for the search functionality. In these cases, data extraction and filtering are totally outsourced to those services and the application only handles the UI rendering section. Therefore, in this case, the Application Logic could be devised to be handled in a tier which lies in-between the client and the server, thus framing the application closer to a 3-tier architecture.

External services The application leverages multiple external services for the complete provision of its functionalities. In particular:

- **Google Firebase** is the main suite of services adopted for backend purposes. Within this suite, the adopted services are:
  - **Cloud Firestore** a non-relational database adopted to store all of the users' information;
  - **Authentication** a service to handle the authentication mechanism;
  - **Storage** a file storage on the cloud, adopted to store the users' profile pictures;
  - **Cloud Functions** a service to call specific functions upon certain triggering events, used to introduce automatic actions on the non-relational database.
- **Algolia** is a search engine which the application interacts with in order to allow the search of other application users in the specific search tab.

- **LocationIQ** is search engine for real-world places. It allows, with a single Http GET request, to query a remote database of locations based on OpenStreetMap.

Additional libraries The application also expands its functionalities thanks to 2 external, public and frequently-maintained libraries, available on *pub.dev*:

- **Geolocator** is a library providing all the necessary operations to interact with the user's device GPS services and used, primarily, for the session tracker functionality;
- **File Picker** is a library allowing the user to access his device file system and select files from it, a necessary operation to allow users to upload pictures for their profiles.

2.2 Component view

Client components The client components of the application, as shown in Figure 2, can be divided in 4 main categories:

- **Model objects** are those class instances that represent an actual data content for the application. For instance, the Session object contains all the information describing a session, such as the starting date and time or the list of the user's position during the run.
- **Components** are classes for the basic graphical components used in the application. These classes have been introduced in order to foster code re-usage; indeed, in most of the cases, they correspond to graphical objects whose content needs to be dynamically adapted on the basis of some reference Model object.
- **Pages** are the classes that represent the actual pages a user interacts with. They usually put together multiple Component objects to produce a final result to display. In addition, the SessionPage also contains all the Application Logic for tracking a session.
- **Interfaces** are classes in charge of handling the interaction with external components in order to improve the decoupling between the Pages and the external components and foster testability of the whole system.

Server configuration This section aims simply at providing some information about the structures configured on the server services to support the application functionalities.

- The **Cloud Firestore** environment is composed of 4 collections, mapping the related Model objects: *goals*, *proposals*, *sessions*, *users*. The information mapping Places objects is included in each proposal document since the place search functionality interacts with a different service (LocationIQ).

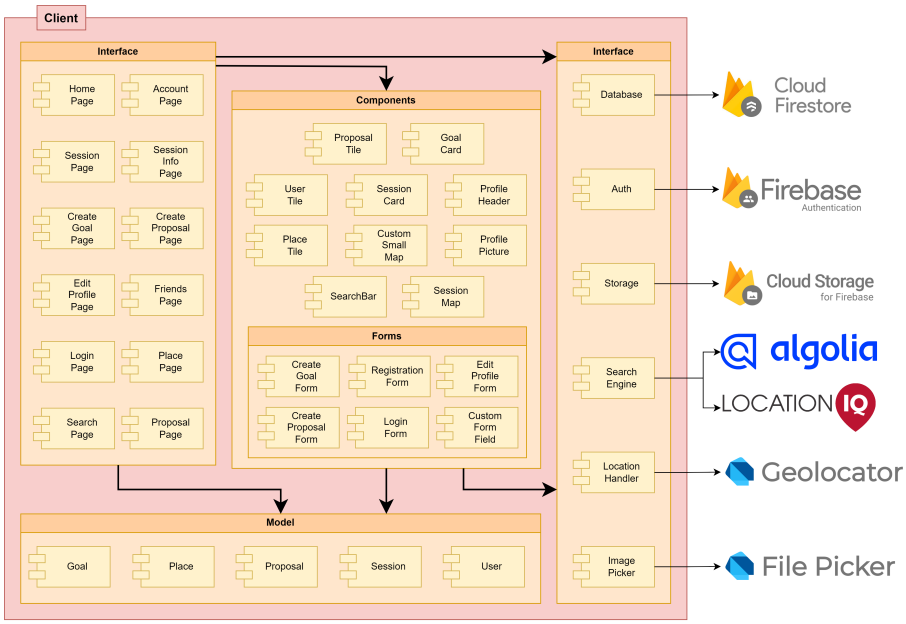


Figure 2: Component diagram

- The **Algolia** environment is composed of a single index for *users* which is constantly updated, thanks to the Google Cloud Functions, with the content of the Firestore *users* collection. This index is configured to allow search of users based on their *name* and *surname* fields.

2.3 Deployment view

Figure 3 describes the overall intended deployment of the application system. On the client side, one can install and use the application on either his smartphone or tablet. On the server side, instead, it is possible to identify three distinct servers, each of them running a specific external service. On one server, which can be assumed to be owned by Google, the whole suite of Firebase services is run. For simplicity, it was assumed that the application data were stored on this server as well, even though it might happen that they are actually stored on two different devices. The second server is property of Algolia and, similarly to the case of the first server, it is assumed both to run the searching application and to store the data for the user index. Finally, the third server is the one hosting the Application Logic for the LocationIQ service and its data to query.

An interesting element in the diagram is that the Algolia and Google servers must be assumed able to communicate with one another. Indeed, the Algolia index must be continuously updated by the Firebase Cloud Functions. However,

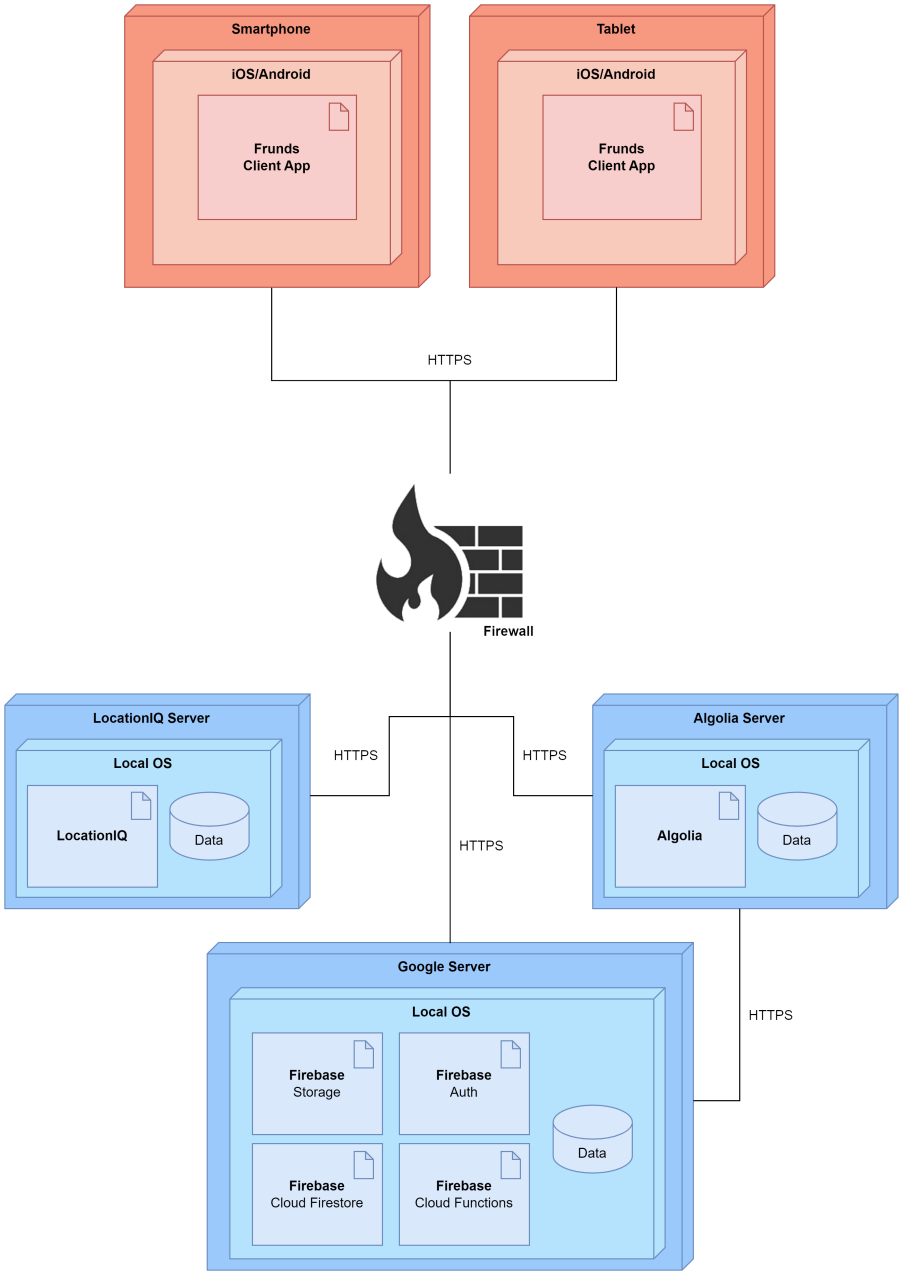


Figure 3: Deployment diagram

this communication channel is completely out of control for the developers of this application. Instead, for security reasons, it is advisable to set up a client-side firewall to protect the application user from malicious attacks from the internet and to adopt a secure protocol, such as HTTPS, for message exchange on the network.

2.4 Architectural patterns

- **Model-View-Controller** On the client side, classes are organized to comply to this pattern. The Model classes are designed to coincide with the Model section of this pattern, whereas the View and the Controller tend to merge in the Pages. This choice was motivated by the aim to clearly decouple the client side from the server side: in this way, indeed, all the contents coming from the server side are converted by the Interfaces into Model objects, which are used throughout the client side. If, for any reason, the structure of objects on the server side changed, it would only be necessary to adjust the conversion mechanism on the Interfaces, without need to adequate also the classes for the User Interface or the business logic, which implement the View and Controller sections of the pattern.

- **Bridge/Adapter** The Interface classes are designed to act as intermediaries between the core of the client side and the server side. This decision was motivated by the deriving enhanced flexibility towards future developments: in this case, indeed, if the external components had to be changed, it would be necessary only to re-implement the methods of the Interface classes, with no need to alter the exposed interface of the classes.

- **Singleton** Most of the Interface classes are also designed to comply to the singleton pattern. Therefore, a single instance of each of those classes should be created on startup and then used for the rest of the application usage instance lifecycle. This choice was considered to be compliant to the meaning of the classes themselves: given they should be interfaces to handle the connection with the exterior, it was considered acceptable to assume a single instance of them existed and that all interactions with a specific service passed through such instance.

2.5 Other design decisions

- **Google Cloud Functions** The developed system leverages the Google Cloud Functions, a Google service that allows triggering calls to specific functions upon certain events configured in the development phase. In this project, these functions were used in two occasions:

  – to update the Algolia index for user search by means of an official extension for the integration of Firebase with Algolia. In this context,

the cloud functions are triggered by any insertion, deletion or update on objects in the Firebase *users* collection occur.

– to delegate to the server the check and update of a user's goals upon completion of a session. In this context, functions are triggered when a session object is inserted in the *sessions* collection on the Firestore database.

3 User Interface design

3.1 Flow diagram

Figure 4 is a diagram representing the interaction flow for the smartphone version of the application. For the tablet version, the flow is slightly different because of the presence of dialogs replacing the creation form pages which are, instead, necessary in the smartphone version. The diagram aims at complying the most to the rules of the IFML standard while providing the highest readability and completeness.

3.2 Screenshots

The next pages display exemplary screenshots for all the pages of the application in its mobile phone version. Then, the reader will find also some screenshots from the tablet version, but only for those pages for which the difference with the smartphone version is significant.

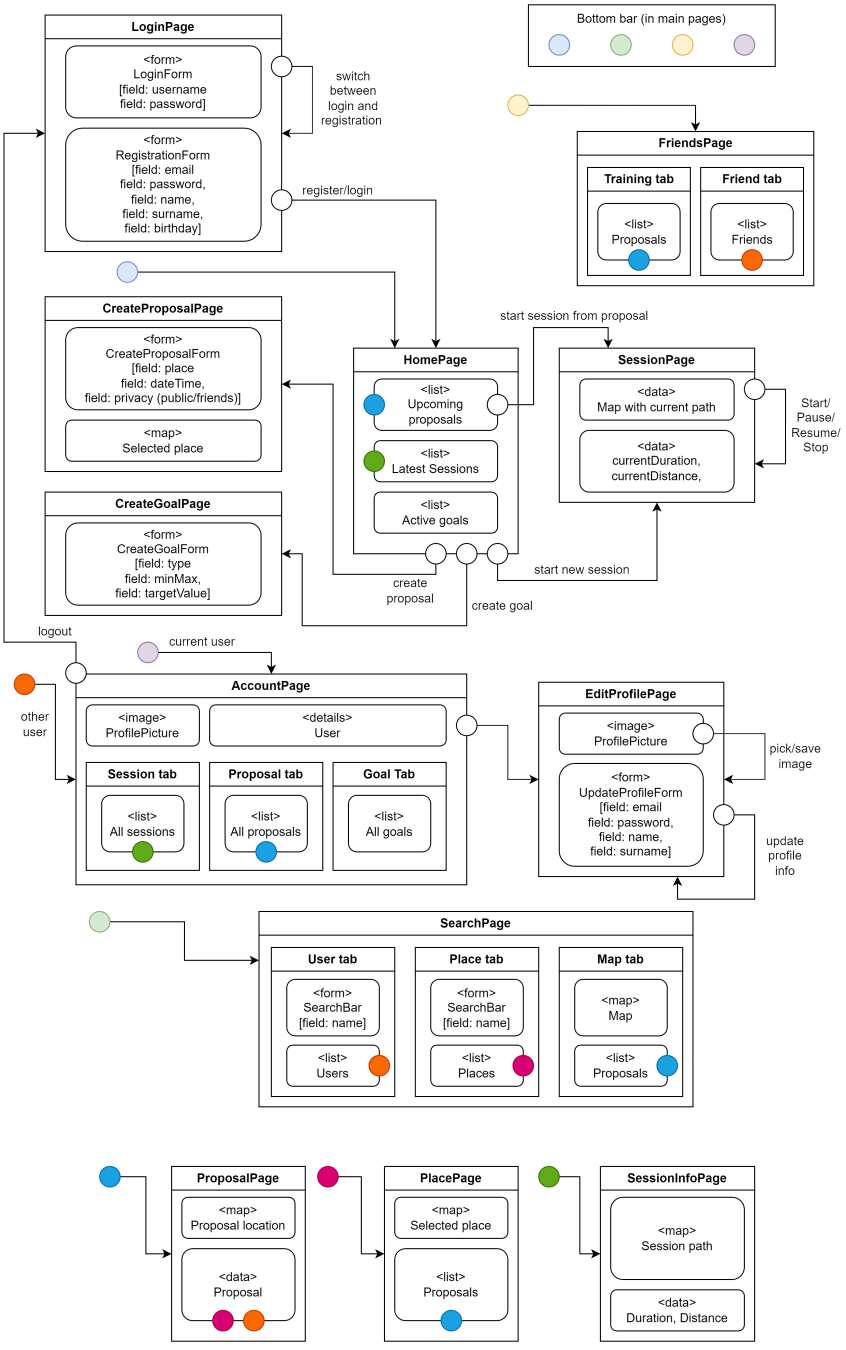
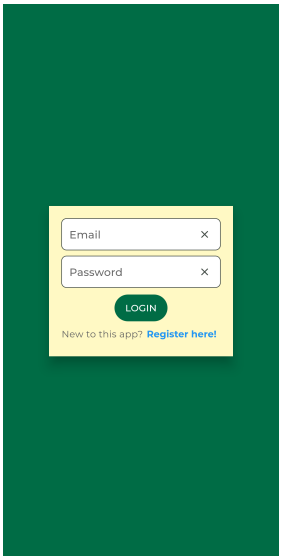
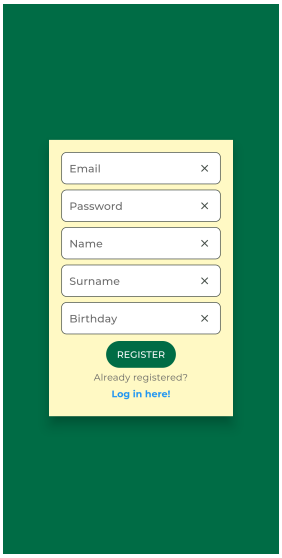


Figure 4: Interaction flow diagram

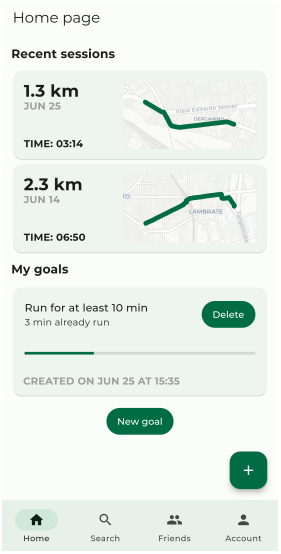


(a) Login

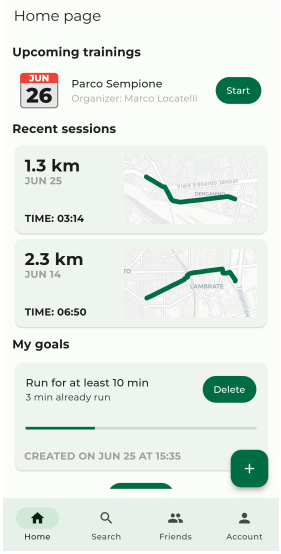


(b) Registration

Figure 5: Login page

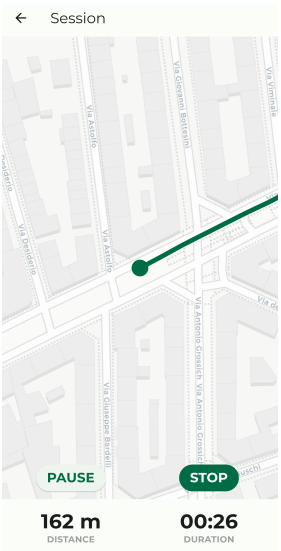


(a) Without upcoming proposals

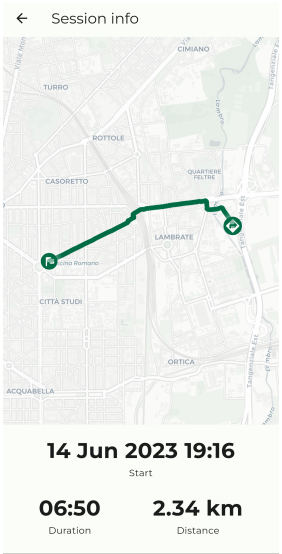


(b) With upcoming proposals

Figure 6: Login page

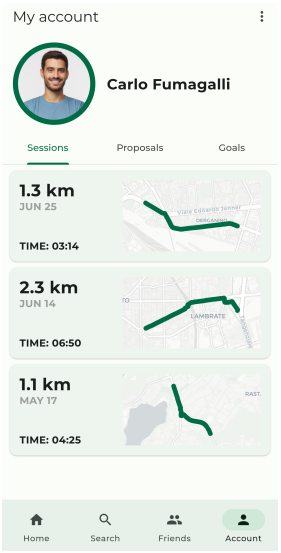


(a) Session page when tracking

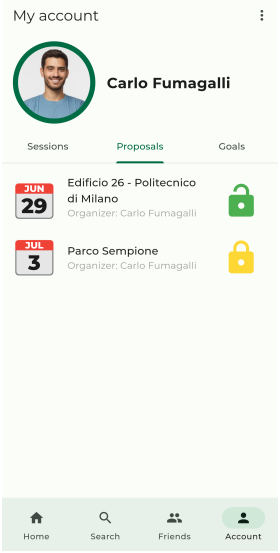


(b) Session information page after run

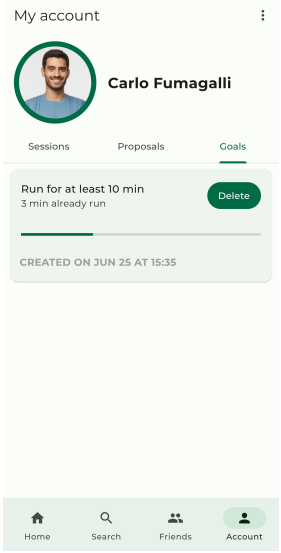
Figure 7: Session tracker



(a) Session tab

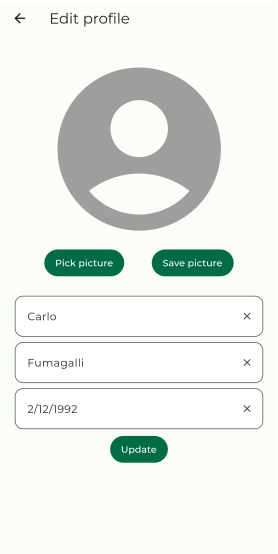


(b) Proposal tab

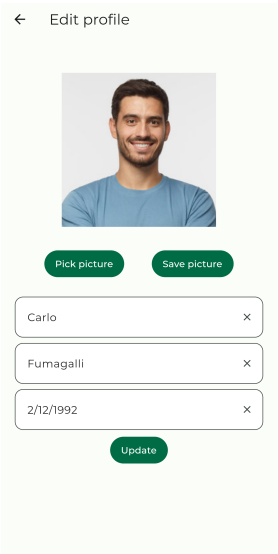


(c) Goal tab

Figure 8: Personal account page

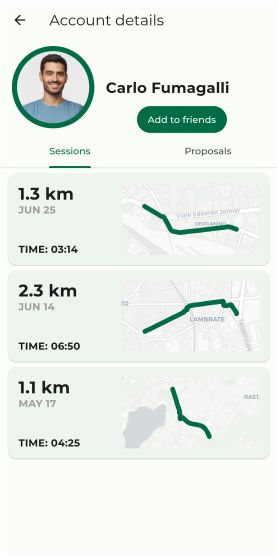


(a) Without picture

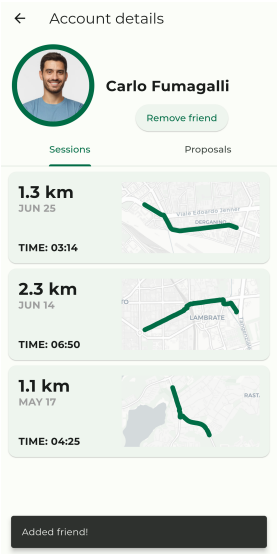


(b) After picture selection

Figure 9: Edit profile page

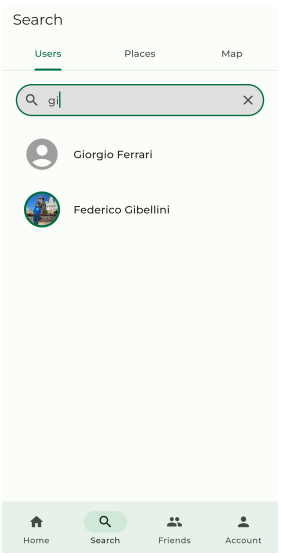


(a) Not friend yet



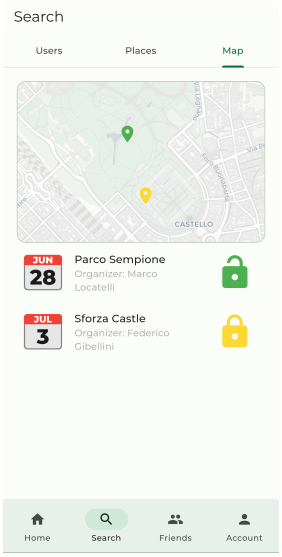
(b) Just added as friend

Figure 10: Account page, seen from other user



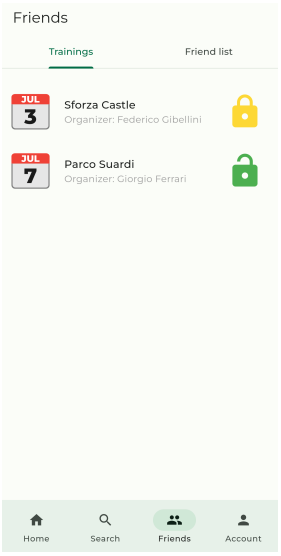
(a) User tab

(b) Place tab

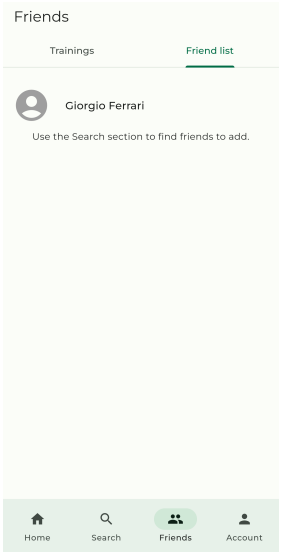


(c) Map tab

Figure 11: Search page

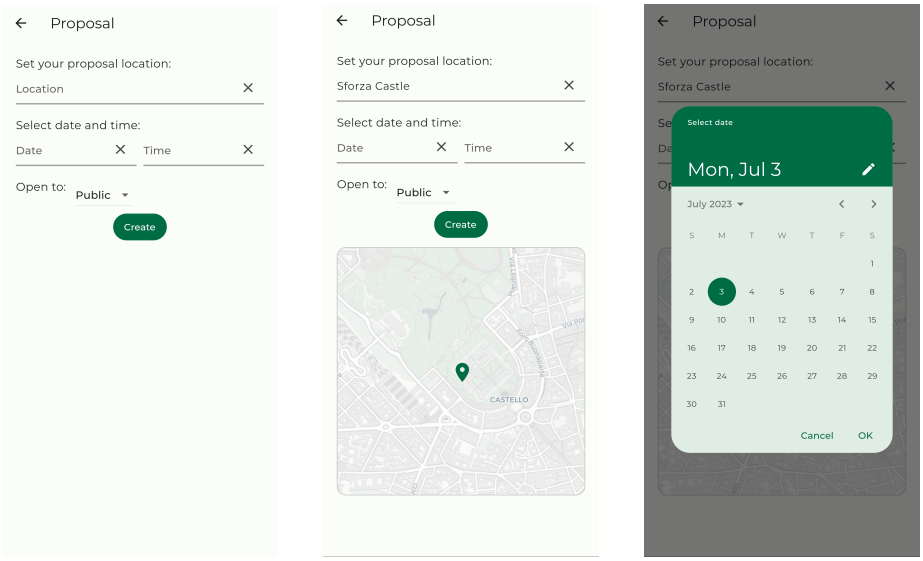


(a) Training tab



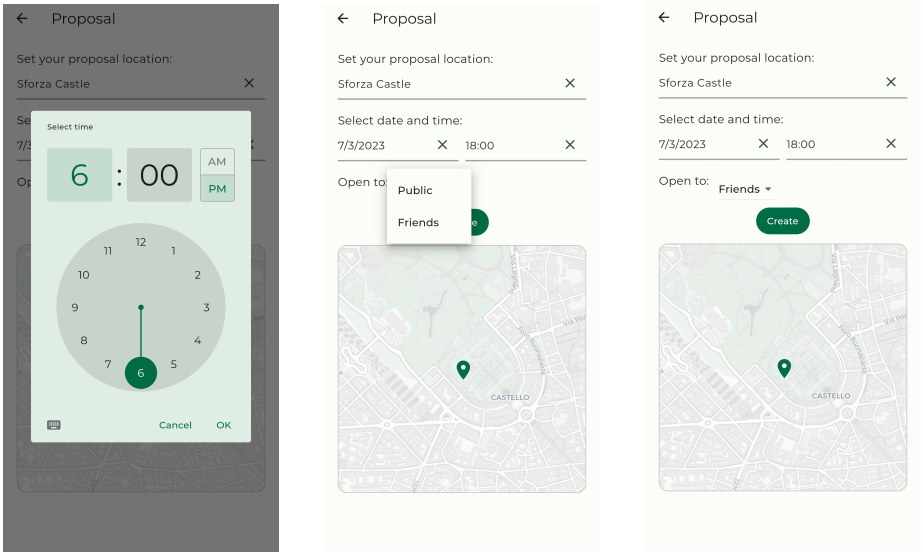
(b) Friend tab

Figure 12: Friends page



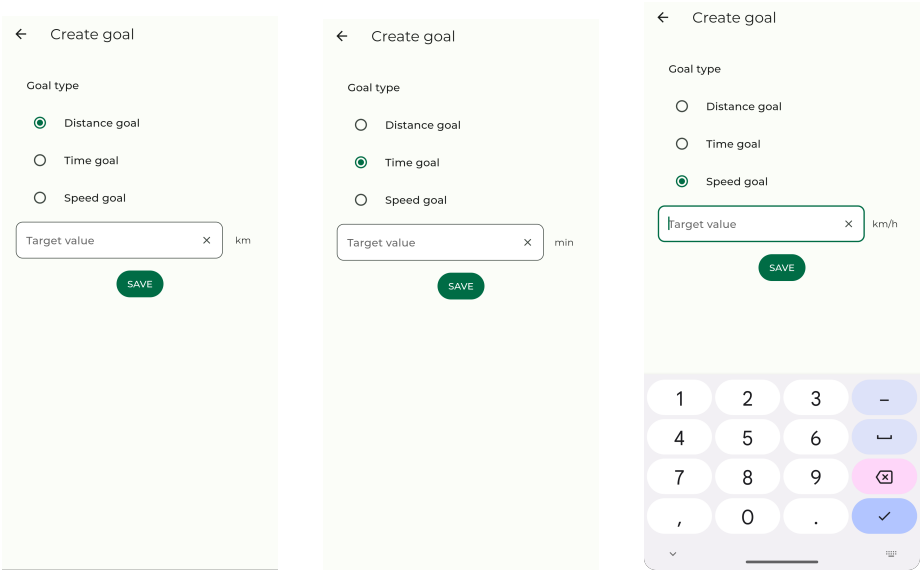
(a) Starting screen (b) After place selection (c) Date selection

Figure 13: Proposal creation, part 1



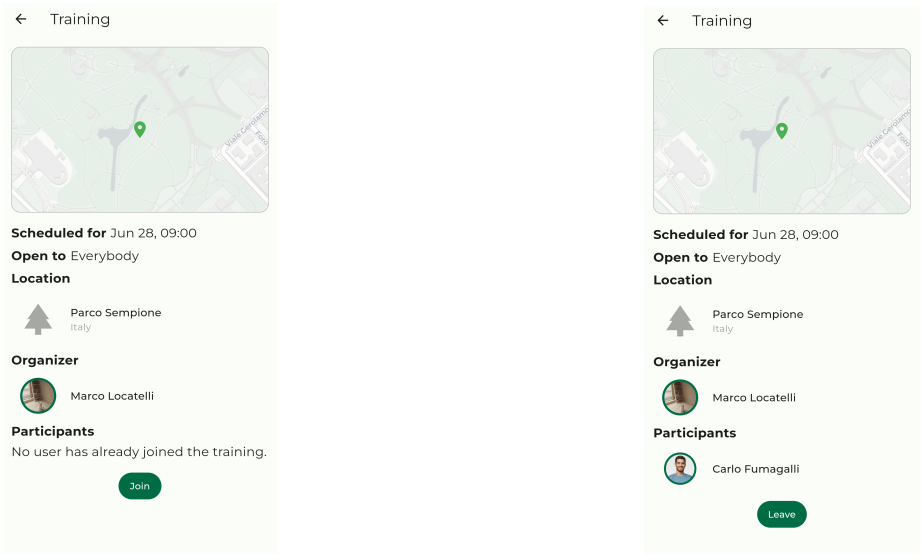
(a) Time selection (b) Privacy level selection (c) Everything selected

Figure 14: Proposal creation, part 2



(a) Distance goal (b) Time goal (c) Speed goal

Figure 15: Page to create goals



(a) User hasn't joined (b) User has joined

Figure 16: Proposal page

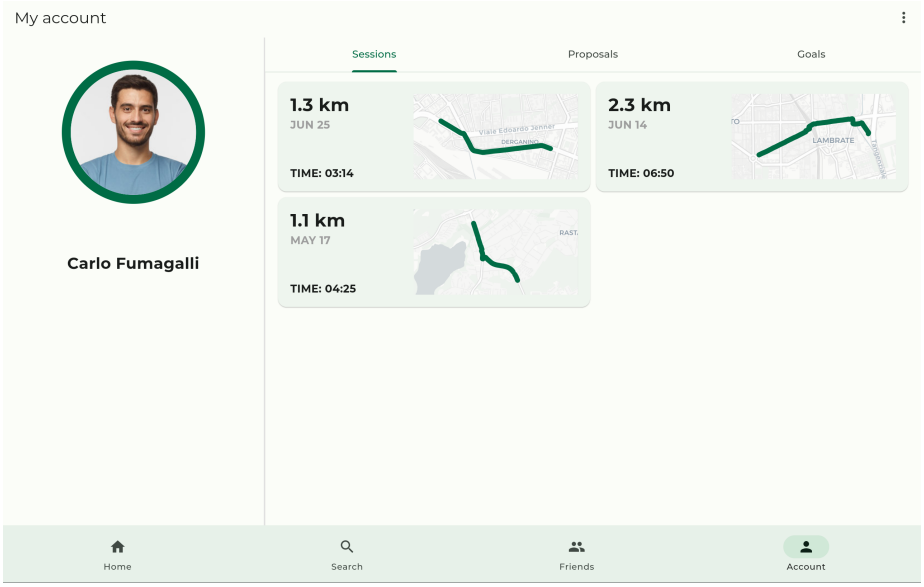


Figure 17: Account page, tablet version

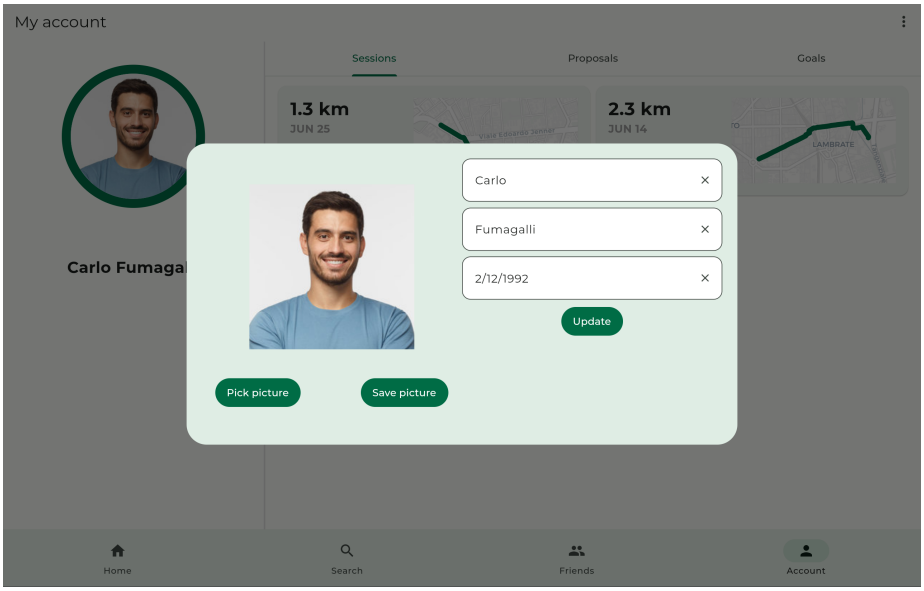


Figure 18: Profile editing, tablet version

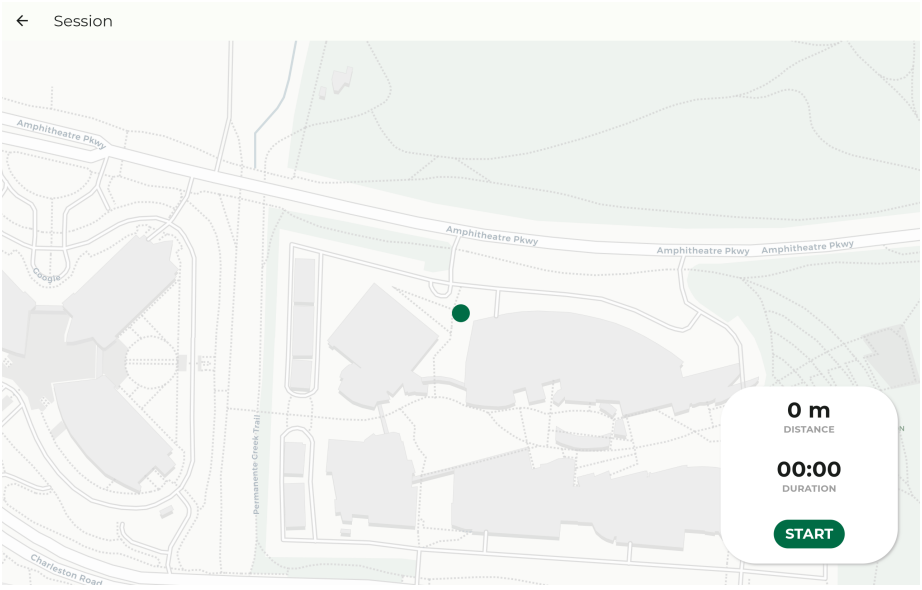


Figure 19: Session page, tablet version

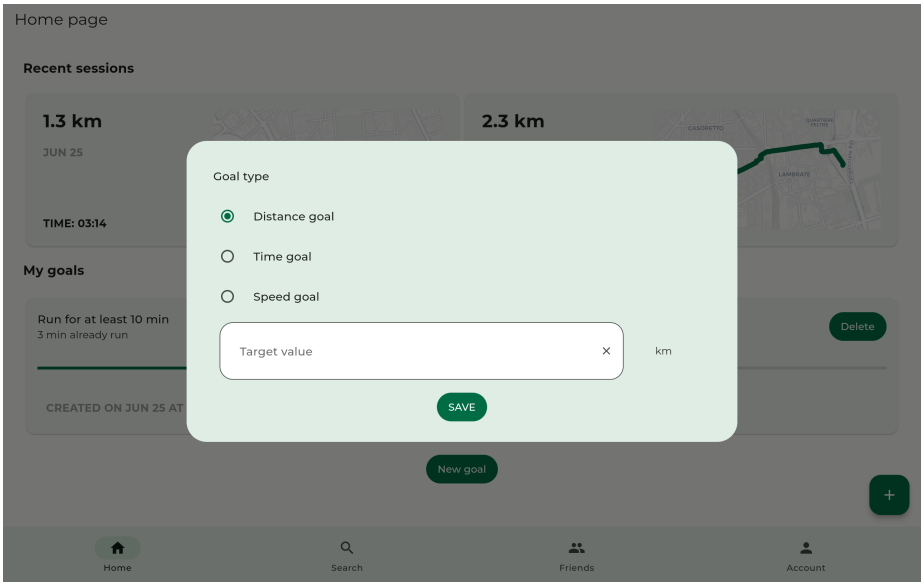


Figure 20: Form creation, tablet version

4 Implementation and testing

4.1 Implementation

The implementation process composed of the following phases:

Backend setup and Model development The first step of the implementation process consisted in setting up the backend, in particular the Firebase Cloud Firestore database, after an analysis of the real world environment to describe in code. In this moment, also the authentication methods were discussed and implemented by setting up the Firebase Auth service. At this point, the application could be set up itself and attempts were made to connect it to the just-setup Firebase services. The application has been developed with the Flutter SDK, which eased the integration with the backend thanks to the common developing enterprise: both Firebase and Flutter, indeed, are released and maintained by Google. Once the application has been integrated with the basic backend services, the database structure was introduced on the client side by being mapped in the Model classes.

Isolated features At this the implementation could proceed towards the core of the development process; the actual functionalities started being implemented in isolation and they were merged in the overall application only when they had been verified to work correctly on emulators.

Testing When the development process had reached a phase in which stability was proven, testing began and was used to improve the application by removing some just-discovered bugs. The followed testing procedure is described in detail in the following section.

Extension to tablet devices At the end of this process, given extension to tablet devices was considered a mainly graphical update, implementation proceeded towards supporting wider screens.

4.2 Testing

This section describes the procedures followed during the testing campaign which was carried out to verify the correct behavior of the application and, specifically, of all its components.

The campaign consisted in a thorough implementation of the first two phases of those described on the official Flutter Documentation [1], i.e., Unit Testing and Widget Testing. Each of these phases is described in detail in the following sections.

In order to complete the most accurate campaign, testing was supported by the following libraries:

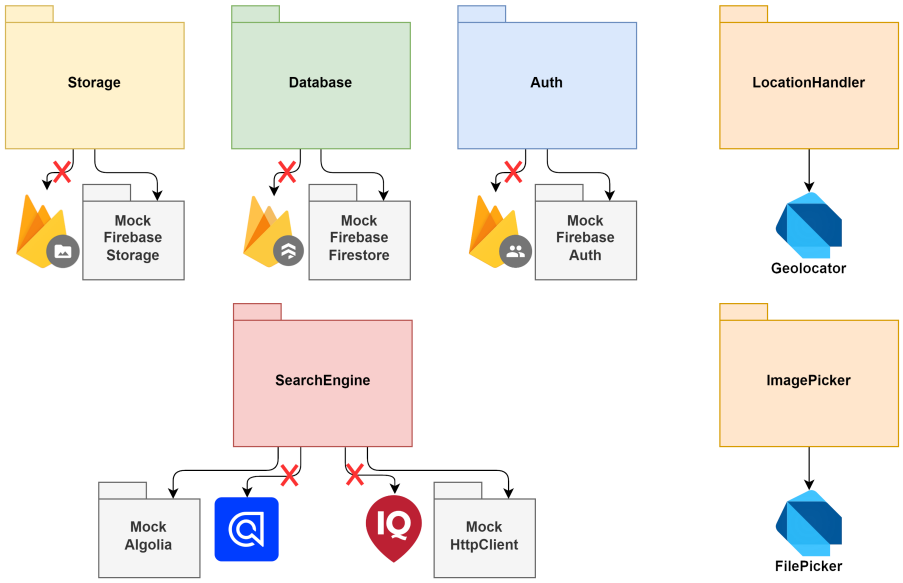


Figure 21: Unit testing strategy for Interface classes

- `flutter_test`: the testing package officially provided with the Flutter SDK. This package provides all the basic functionalities for testing in the Flutter environment;
- `mockito`: a library for the creation of mock objects, which are meant to substitute actual components with the aim of providing a full controllability of the testing environment. Within this library, two packages resulted in being of paramount importance: the mockito package, with all the basic functionalities to interact with mocks, and the annotations package, which allowed the automatic generation of mocks when used in conjunction with the following library;
- `build_runner`: a library to allow the automatic generation of mocks by leveraginng the *mockito* annotations;
- `network_image_mock`: a library to simulate correct calls to a server to lay out network images.

4.2.1 Unit Testing

This part of the campaign aimed at verifying the correct behavior of the basic Interface and Model classes by testing them in isolation.

Given that the latter type of classes represent the core objects used in the application, those classes and their methods could be easily tested without the introduction of mocks. Instead, since most of the Interface classes leverage

external components such as the Firebase ones, in order to complete tests for them it is necessary to mock such components with aid from the *mockito* library. Figure 21 shows the intended testing process for these classes. As it is visible, the set of Interface classes also includes elements that do not rely on external components but, rather, on libraries such as the *Geolocator* and the *File Picker*. Given that such classes were created only as wrappers for the static methods of these libraries and that *mockito* does not allow the stubbing of static methods, such classes are the only ones which were deemed acceptable not to be tested.

4.2.2 Widget Testing

This phase, which was intended to follow the one just presented, aims at testing all the remaining classes from the graphical perspective. In widget testing, the aim is to verify the UI and its components are rendered as expected.

Given the tree structure which is typical of a mobile and, specifically, of a Flutter application, the testing process for this phase proceeded accordingly to such structure. Therefore, the process started with the check of the classes that would have been laid out as leaves of the tree, the Components, and proceeded with sequential testing towards the root of the tree, with the Pages. A fragmented representation of such structure is provided in Figures 22 and 23, which also show the mocks used for each class. Each of the used mocks, which mainly substitute the Interface classes tested previously, is shown only for the classes at the leaf-level, but it was used also for all the parent classes of the leaf depending on the mock. This choice was motivated by the need, for most of the UI classes, to interact with the Interface ones and by the aim of avoiding repeated cumbersome testing with mocks on external components.

4.2.3 Results

This section shows the results of the aforementioned testing phases. The coverage tables displayed below were obtained after running the `flutter test` command with options for line coverage and then converting the output to html format. As it is visible from the aggregated data, the overall coverage achieved in this campaign is 98%.

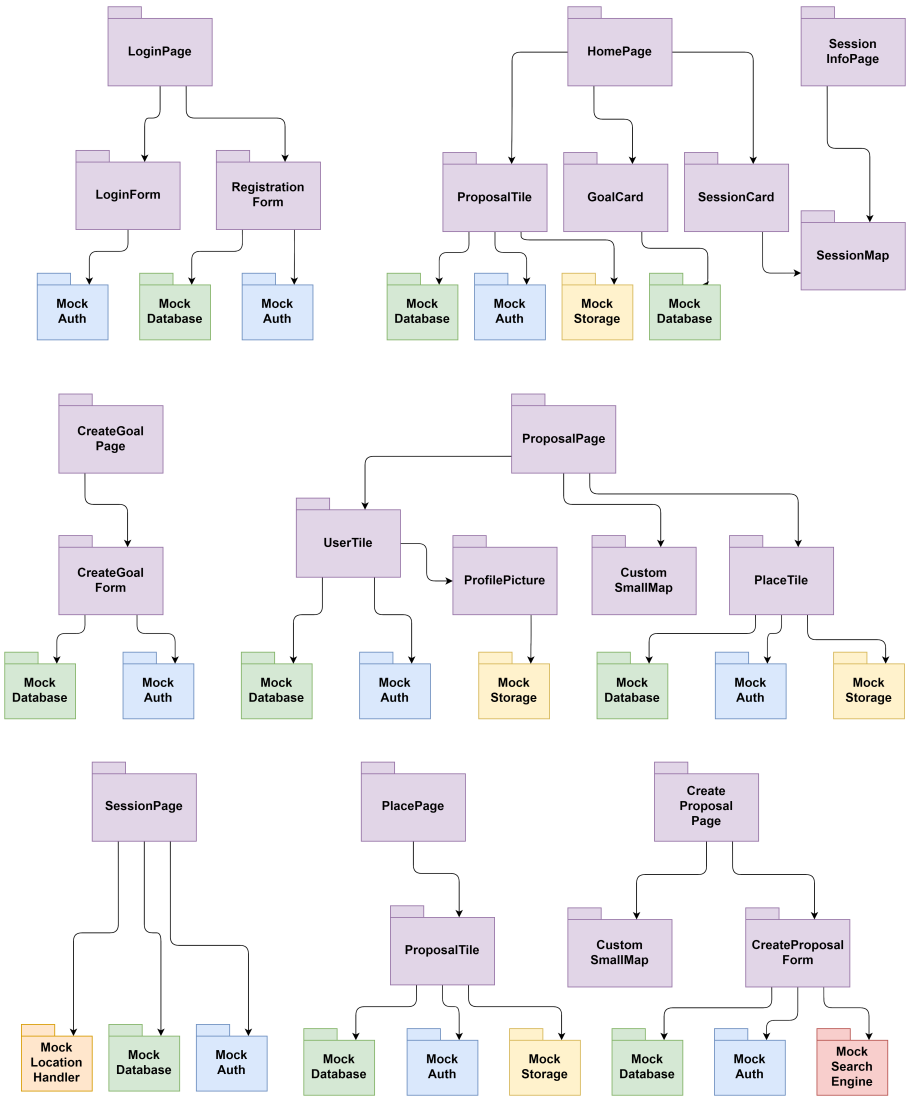


Figure 22: Testing strategy for pages, part 1

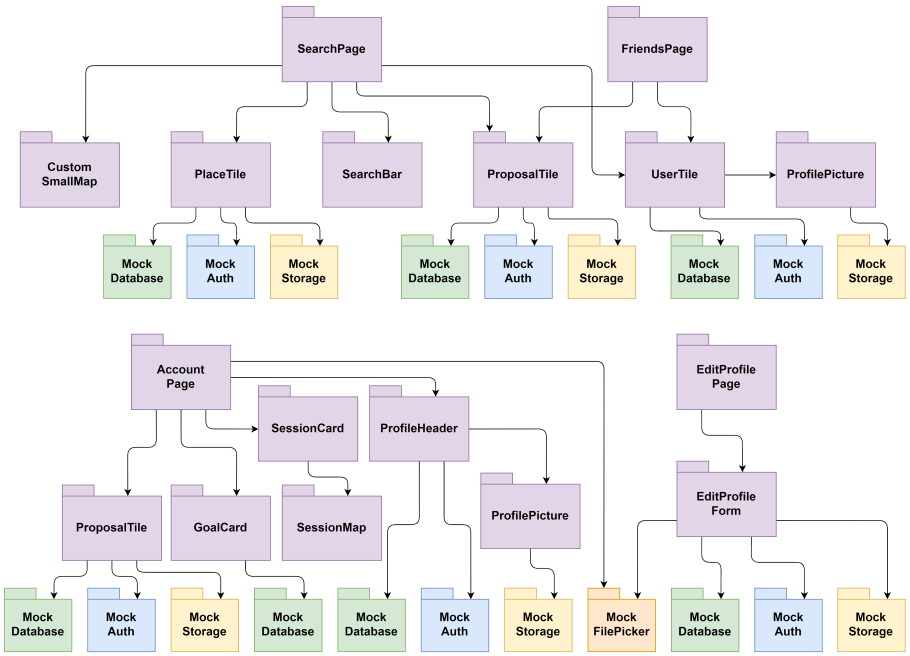
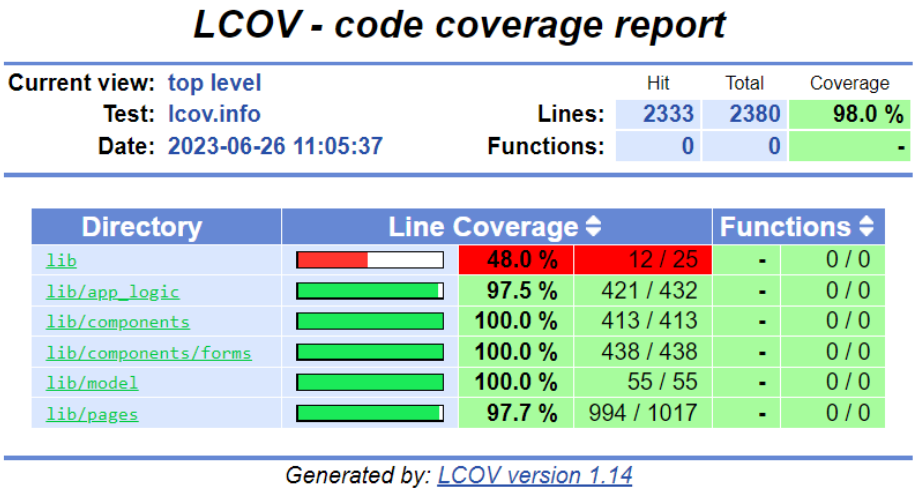


Figure 23: Testing strategy for pages, part 2



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	**Lines:**	2333	2380	98.0 %
Test:	lcov.info	**Functions:**	0	0	-
Date:	2023-06-26 11:05:37				

Directory	Line Coverage		Functions	
lib	48.0 %	12 / 25	-	0 / 0
lib/app_logic	97.5 %	421 / 432	-	0 / 0
lib/components	100.0 %	413 / 413	-	0 / 0
lib/components/forms	100.0 %	438 / 438	-	0 / 0
lib/model	100.0 %	55 / 55	-	0 / 0
lib/pages	97.7 %	994 / 1017	-	0 / 0

Generated by: LCOV version 1.14

Figure 24: Coverage report - Overall

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - lib				
Test:	lcov.info	**Lines:**	12	25	48.0 %
Date:	2023-06-26 11:05:37	**Functions:**	0	0	-

Filename	Line Coverage			Functions	
main.dart		48.0 %	12 / 25	-	0 / 0

Generated by: LCOV version 1.14

Figure 25: Coverage report - Lib folder

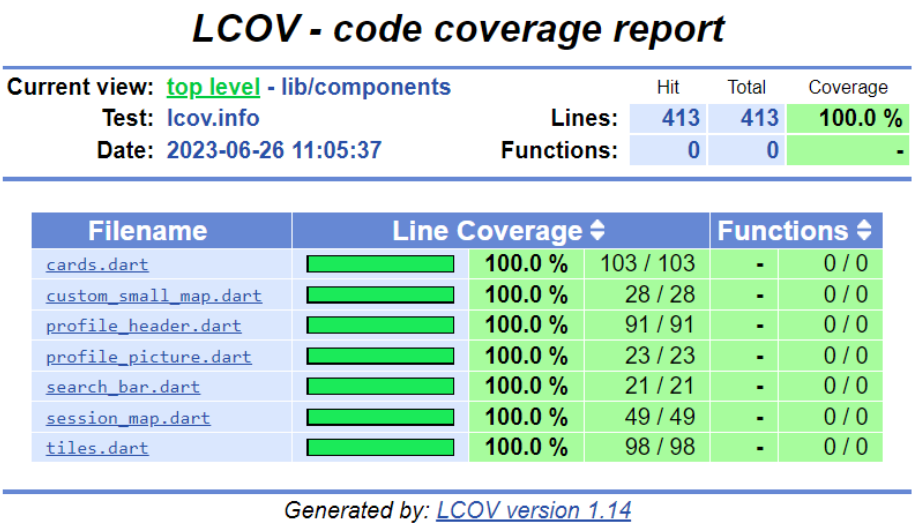
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - lib/app_logic				
Test:	lcov.info	**Lines:**	421	432	97.5 %
Date:	2023-06-26 11:05:37	**Functions:**	0	0	-

Filename	Line Coverage			Functions	
auth.dart		100.0 %	25 / 25	-	0 / 0
database.dart		100.0 %	321 / 321	-	0 / 0
exceptions.dart		100.0 %	4 / 4	-	0 / 0
image_picker.dart		0.0 %	0 / 4	-	0 / 0
location_handler.dart		53.3 %	8 / 15	-	0 / 0
search_engine.dart		100.0 %	46 / 46	-	0 / 0
storage.dart		100.0 %	17 / 17	-	0 / 0

Generated by: LCOV version 1.14

Figure 26: Coverage report - Interfaces



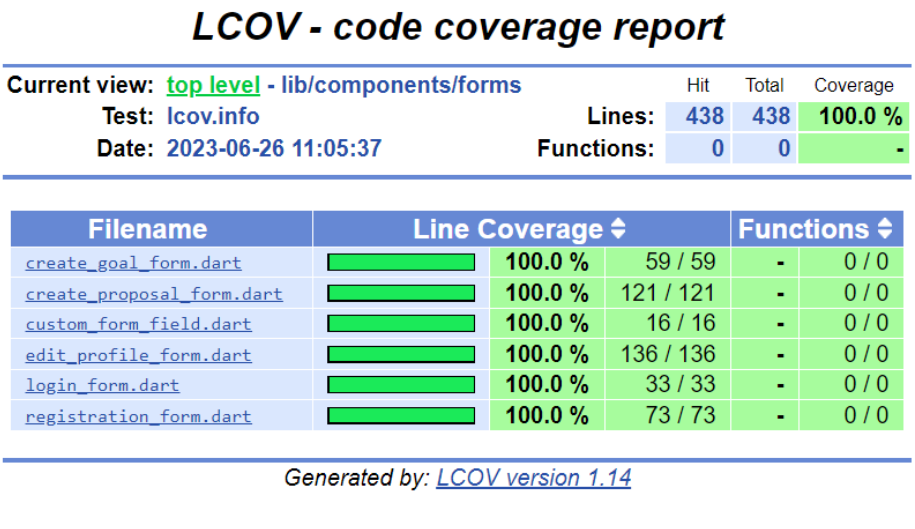
LCOV - code coverage report

Current view: top level - lib/components | Hit | Total | Coverage
Test: lcov.info | Lines: | 413 | 413 | 100.0 %
Date: 2023-06-26 11:05:37 | Functions: | 0 | 0 | -

Filename	Line Coverage			Functions	
cards.dart		100.0 %	103 / 103	-	0 / 0
custom_small_map.dart		100.0 %	28 / 28	-	0 / 0
profile_header.dart		100.0 %	91 / 91	-	0 / 0
profile_picture.dart		100.0 %	23 / 23	-	0 / 0
search_bar.dart		100.0 %	21 / 21	-	0 / 0
session_map.dart		100.0 %	49 / 49	-	0 / 0
tiles.dart		100.0 %	98 / 98	-	0 / 0

Generated by: LCOV version 1.14

Figure 27: Coverage report - Components



LCOV - code coverage report

Current view: top level - lib/components/forms | Hit | Total | Coverage
Test: lcov.info | Lines: | 438 | 438 | 100.0 %
Date: 2023-06-26 11:05:37 | Functions: | 0 | 0 | -

Filename	Line Coverage			Functions	
create_goal_form.dart		100.0 %	59 / 59	-	0 / 0
create_proposal_form.dart		100.0 %	121 / 121	-	0 / 0
custom_form_field.dart		100.0 %	16 / 16	-	0 / 0
edit_profile_form.dart		100.0 %	136 / 136	-	0 / 0
login_form.dart		100.0 %	33 / 33	-	0 / 0
registration_form.dart		100.0 %	73 / 73	-	0 / 0

Generated by: LCOV version 1.14

Figure 28: Coverage report - Forms

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - lib/model	Lines:	55	55	100.0 %
Test:	lcov.info	Functions:	0	0	-
Date:	2023-06-26 11:05:37				

Filename	Line Coverage			Functions	
goal.dart		100.0 %	10 / 10	-	0 / 0
place.dart		100.0 %	9 / 9	-	0 / 0
proposal.dart		100.0 %	12 / 12	-	0 / 0
session.dart		100.0 %	17 / 17	-	0 / 0
user.dart		100.0 %	7 / 7	-	0 / 0

Generated by: LCOV version 1.14

Figure 29: Coverage report - Model

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - lib/pages	Lines:	994	1017	97.7 %
Test:	lcov.info	Functions:	0	0	-
Date:	2023-06-26 11:05:37				

Filename	Line Coverage			Functions	
account_page.dart		99.5 %	199 / 200	-	0 / 0
create_goal_page.dart		100.0 %	9 / 9	-	0 / 0
create_proposal_page.dart		100.0 %	65 / 65	-	0 / 0
edit_profile_page.dart		100.0 %	17 / 17	-	0 / 0
friends_page.dart		62.2 %	23 / 37	-	0 / 0
home_page.dart		93.3 %	111 / 119	-	0 / 0
login_page.dart		100.0 %	29 / 29	-	0 / 0
main_screens.dart		100.0 %	20 / 20	-	0 / 0
place_page.dart		100.0 %	47 / 47	-	0 / 0
proposal_page.dart		100.0 %	146 / 146	-	0 / 0
search_page.dart		100.0 %	89 / 89	-	0 / 0
session_info_page.dart		100.0 %	33 / 33	-	0 / 0
session_page.dart		100.0 %	206 / 206	-	0 / 0

Generated by: LCOV version 1.14

Figure 30: Coverage report - Pages

5 References

5.1 Support software

- **Diagrams.net**: online drawing service used for the realization of all the diagrams in this document.
- **Android Studio**: IDE for the phases of development and testing of the application.
- **Overleaf**: online text editor for the composition of Latex documents.

References

[1] Flutter.dev. Testing flutter apps. https://docs.flutter.dev/testing. Accessed on: 25/06/2023.