Design and Implementation of Mobile Application - 2021/2022

Run With Me

A mobile application built in Flutter to find friends to run with.

Team Members:

Federico Dei Cas - federico.deicas@mail.polimi.it
Daniele V. De Vincenti - danielevalentino.devincenti@mail.polimi.it

Politecnico di Milano

Contents

1 Introduction

1.1 Objectives

The objective of the app is to provide a platform in which users can find or propose running events, giving a starting place and time, and the expected distance and time to complete.

Users can subscribe to other events or create their own, choosing the maximum number of allowed participants. The backend will calculate a difficulty level between 0-5 for each proposed run, to suggest event based on the users' fitness level, which is calculated by requiring some questions during registration.

Even if a user is not registered, he/she can still browse events. Registration is then required if he/she wants to subscribe. We made this choice because many application do not allow to interact without registering first, which can be a deterrent.

1.2 Overview

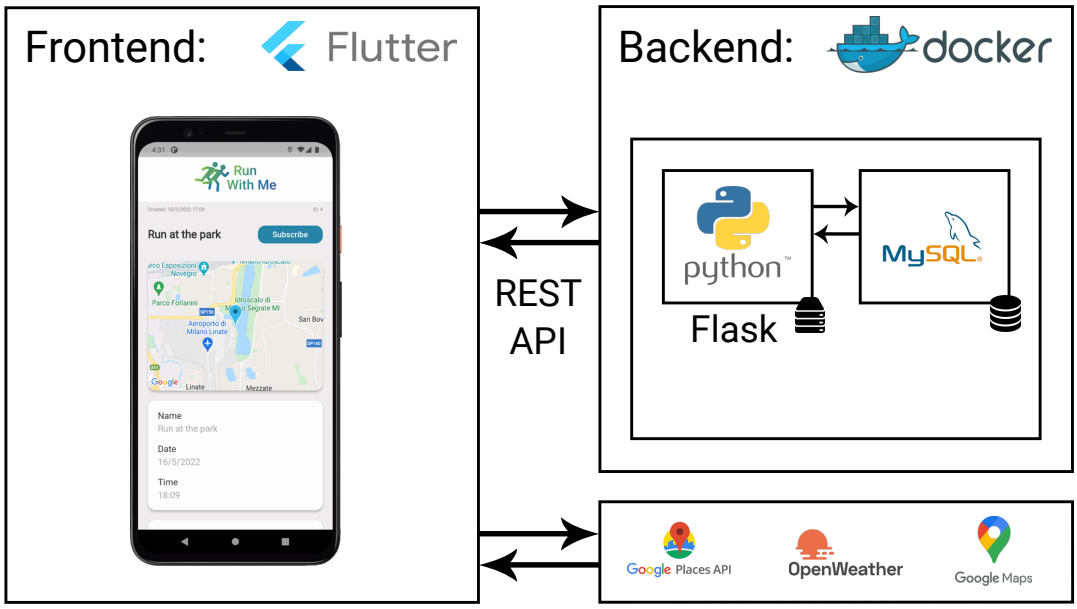


Figure 1.1: High level overview of the app architecture

As shown in figure 1.1, the app is composed by a frontend, a custom backend and some external services integrations. For the frontend, our choice was flutter mainly because of the performance and stability it offers, and the possibility of having a single codebase for different platforms.

For the backend, we chose a custom solution, building our own API and database, to have more versatility and full control over our application, also allowing us to integrate our backend with some external services.

2 Requirements

2.1 Functional Requirements

Given the number of functionalities chosen, we decided to leave low priority feature for future developments, but still plan the app to support these features.

ID	STATEMENT	PRIORITY
FR1	Users should be able to register to the application.	High
FR2	Users should be able to authenticate to the application and log out.	High
FR4	Users should be able to edit their info.	Low
FR5	Users should be able to view nearby events based on theirs location, or a chosen location, with some filter parameters like max distance and show also events that are full.	High
FR6	Non-registered users should be able to view nearby events based on theirs location.	Medium
FR7	Users should be able to create new events.	High
FR8	Users should be able to subscribe to events.	High
FR9	Registered users should be able to un-subscribe to events.	High
FR10	Users should be able to view event details, like starting place, time, date, and the number of current subscribed participants.	High
FR11	User who created the event should be able to edit it afterwards.	Low
FR12	User who created the event should be able to delete it afterwards.	Low
FR13	The application shall have a home screen.	High
FR14	The application shall display weather forecasts, and today's temperature, based on user's location.	Medium
FR15	The application shall display user's stats for the last 7 days (total distance, total time, average pace)	Medium
FR16	Users should be able to search events with an interactive map view.	High
FR17	Users should be able to see their distance from the events.	High
FR18	Users should be able to choose between grid or list view for the events.	Medium
FR19	Users should be able to see events they attended as history.	Medium
FR20	Users should be able to sort events by distance, date, difficulty, length, duration.	Medium

ID	STATEMENT	PRIORITY
FR21	Users should be able to see incoming events they subscribed to.	High
FR22	Users should be able to switch between dark/light theme, and persisting the choice.	High
FR23	Users should be able to have a default location in case the GPS is not working/disabled.	High
FR24	The application should be able to display recently viewed events.	High
FR25	Users should be able to see other user details.	Low
FR26	The application should be able to calculate an approximate difficulty level of an event given distance and time	High
FR27	The application should be able to calculate the user fitness level	High
FR28	The application should be able to recommend events based on the users fitness level and distance from them	High
FR29	Users should be able to choose a maximum number of participants to their events	High
FR30	Users should not be able to subscribe to an event which has reached the maximum number of participants	High
FR31	Once an event is passed, it should be moved automatically from the booked events section to the past event section.	High

2.2 Non Functional Requirements

ID	STATEMENT	PRIORITY
NFR1	Secure communication between client app and server.	High
NFR2	User's password shouldn't be saved in clear in the database.	High
NFR3	Auth token should be stored securely on the client persistent storage.	High
NFR4	The application should be scalable as the user base grows.	Medium
NFR5	Each request should be processed within 10 seconds.	Medium

2.3 Constraints and Assumptions

ID	STATEMENT
C1	User should have a working internet connection.
C2	User should have a working GPS system and allow app permission (but can revert to default user location in case not allowed/not working).
C3	Android 4 KITKAT WATCH SDK / API level 20

3 Design System

3.1 Overview

When the project features have been finalized and after the functional requirements of the application have been decided, a complete design system case study has been carried out in order to define all User Interface (UI) elements that were going to be used in the development of actual application. Along with the UI elements, a study has been done to decide an overall color palette and a font to be used in the app. Finally, after having finalized everything that was needed for the UI, full renders of every page have been made to aid the developers in the coding phase.

3.2 UI Elements

The UI for this app is strongly based on the Material design principles and all major UI elements such as cards, buttons, toggles and sliders, have been taken from it.

Elements in this applications are flat, with no gradients in the UI except for the Application Bar (AppBar) in some pages as it will be seen later in this document. Corners of almost all elements are rounded with a radius between 10 and 15 px depending on the specific element.

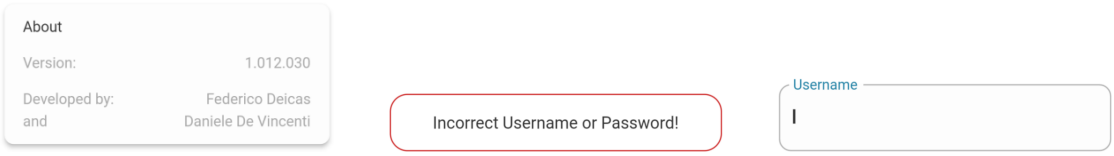
To better show key piece of information to the user and to improve the overall readability, cards element (Figure 3.1a) are given a boxed shadow at the bottom. Other flat elements with rounded corners that are used throughout the app are alert dialogs, feedback messages and input boxes:

1. *Alert Dialogs*: they take the same aesthetic of a card and are not shown in this chapter.

2. *Feedback Messages*: as in Figure 3.1b, they are shown as rounded rectangle that dynamically adjust depending on the amount of text that needs to be displayed. They can have color coded borders depending on the type of message that they are displaying.

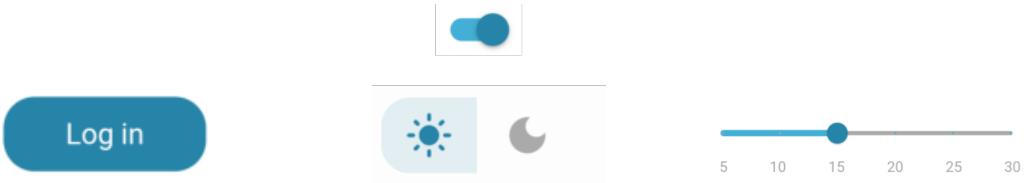
3. *Input Boxes*: as in Figure 3.1c, they are roughly the same shape of a feedback message, except that they have a tighter rounded corner radius, they have no colored border and when active, they show the description of the input on the top left corner.

Other UI elements that have been used extensively throughout the app are buttons, toggles and sliders.



(a) Cards (b) Feedbacks (c) Text inputs

Figure 3.1: Some graphic elements



(a) Buttons (b) Toggles (c) Sliders

Figure 3.2: Some other graphic elements

1. *Buttons*: as in Figure 3.10c, all buttons have a flat background color, rounded corners and no borders. The description of the button functionality are either inside the button itself or on top of it.
2. *Toggles*: as in Figure 3.10d, they can appear both as default material design toggles with a description text on top and as custom toggles with icons inside.
3. *Sliders*: as in Figure 3.2c, they are default sliders from the material design, with a numbered scale beneath.

3.3 Multi Theme Support

Before going into the next chapters and exploring more the different color choices, it is important to denote that, already from the early stages of development, the app was designed to support different themes: more specifically a light theme based on light backgrounds with dark text and a dark theme with mainly darker backgrounds and white text. From now on these two theme modes will be defined as *Light Theme* and *Dark Theme*.

3.4 Color Palette

The next phase in the design of the application UI was to define a color palette. In this chapter, multiple palettes for different elements and in different theme modes are going to be presented.

Firstly, a set of primary and secondary colors have been chosen, along with some variations. These colors are used both in the *Light Theme* and in the *Dark Theme* and can be seen in Figure 3.3

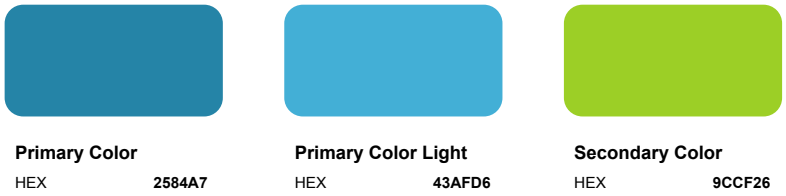


Figure 3.3: Palette for primary and secondary colors in both theme modes

3.4.1 Light Theme

As said before, for the *Light Theme* a set of light backgrounds and darker texts has been used. In Figure 3.4 are shown the chosen set of colors for the light theme backgrounds, as well as the error color.

Figure 3.4: Palette for background colors in light theme mode

In Figure 3.5 are shown the dark colors for all the text elements in the app. As can be seen from this figure, the text color gets darker and darker the more important a piece of text is. This color difference between different text parts in the app is used to convey a different importance to a particular string of text and to guide the user towards the most important elements of a page. This color difference in text also improve readability.

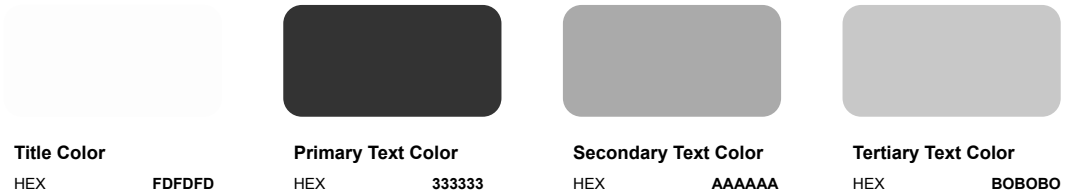


Figure 3.5: Palette for text colors in light theme mode

3.4.2 Dark Theme

In the *Dark Theme* the same main colors have been used, just for different things. In Figure 3.6 all background colors for the dark mode are shown. Here a new color was needed for the background as no other color was dark enough.

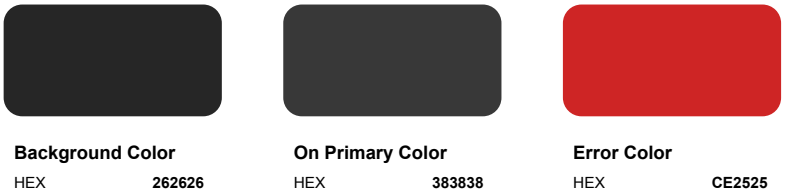


Figure 3.6: Palette for background colors in dark theme mode

Text colors shown in Figure 3.7 are used for all the different text styles within the dark mode. As for the *Light Theme*, the same principle regarding different color gradients in text strings applies. The lighter a certain string is, the more important it is for the user.

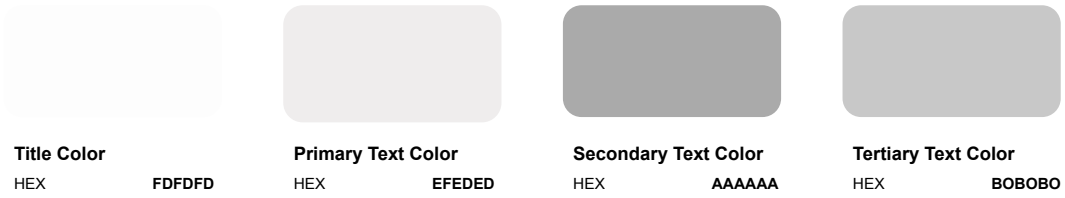


Figure 3.7: Palette for text colors in dark theme mode

As can be seen, titles are displayed in a very light color because are always on top of a colored banner. In Figure 3.8 it can be seen a title in a page AppBar, displayed in *Title Color* over a gradient background.

Figure 3.8: Sample glyphs for the chosen Roboto font

3.5 Fonts

In order to improve readability on multiple themes and on all devices, from possibly very small devices such as older phones, to big screens such as tablets, the font choice was very important. Thus, Roboto from Google Fonts was used. As stated in the official font page: *Roboto has a dual nature. It has a mechanical skeleton and the forms are largely geometric. At the same time, the font features friendly and open curves. While some grotesks distort their letterforms to force a rigid rhythm, Roboto doesn't compromise, allowing letters to be settled into their natural width. This makes for a more natural reading rhythm more commonly found in humanist and serif types.*

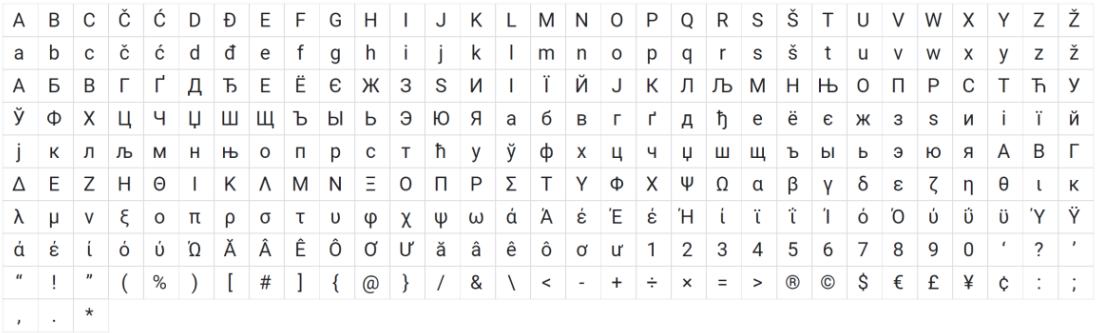
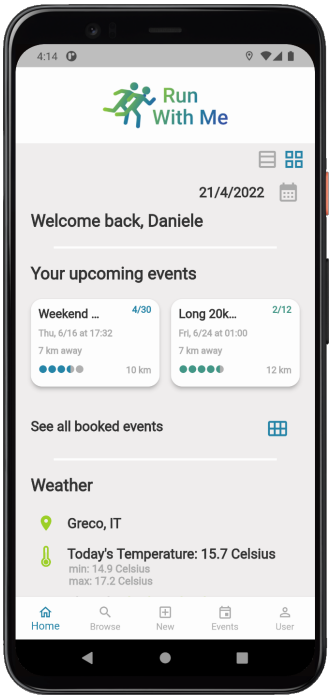
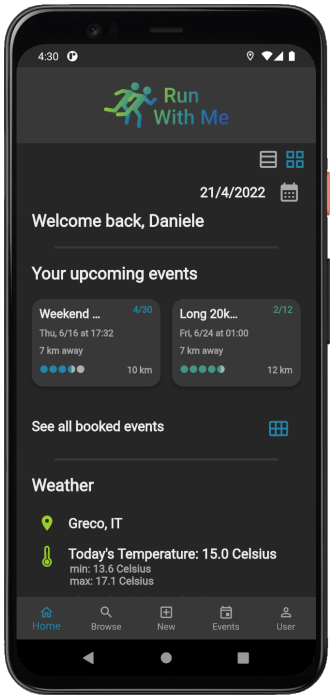


Figure 3.9: Sample glyphs for the chosen Roboto font

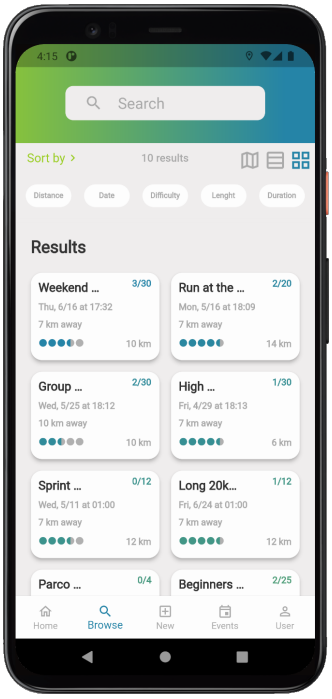
Finally, in Figure 3.10 are shown some example pages both in light and dark mode.



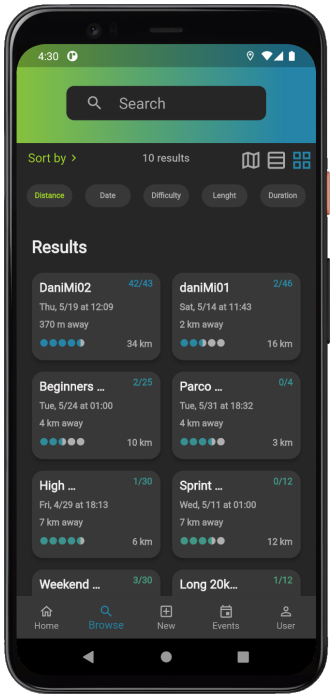
(a) *Light Mode*



(b) *Dark Mode*



(c) *Light Mode*



(d) *Dark Mode*

Figure 3.10: Screens in different modes

4 Architectural Design

4.1 Overview

The Architecture of the app is a client/server. We decided to implement a self-hosted backend solution with a separated application server and database server, for versatility purposes. The client and the application server communicate through REpresentational State Transfer (REST) Application Programming Interface (API), which is then connected to the database tier, holding our persistent data safely. We decided not to have persistent local storage for the data (except for Authentication (auth) token and user settings, like the chosen theme mode). The data requested from the database throught API are then temporarily stored in non-persistent dart classes.

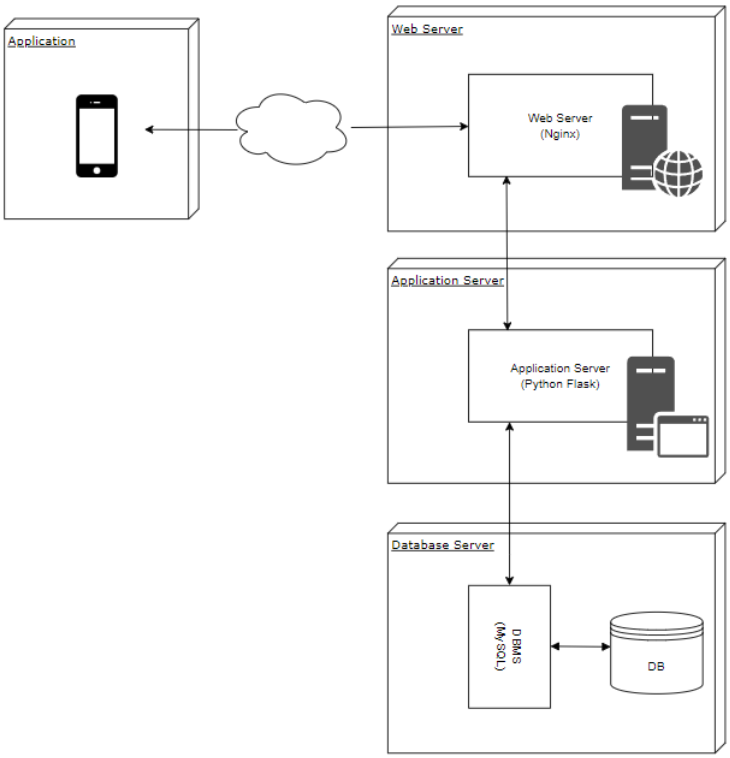


Figure 4.1

4.2 Backend

The backend is divided into an application server and a database server, which communicate to each other through Transmission Control Protocol (TCP) connection.

4.2.1 API Specification

For request /events and /event/event_id we provided both an authenticated and unauthenticated version accessible through different routes, to let the non registered users also see events nearby without having to register beforehand. All other requests (except for register and login) require the auth token. The auth token is provided through a bearer token inside the Authorization header of the Hypetext Transfer Protocol (HTTP) request. The API documentation with examples of requests is available here: https://documenter.getpostman.com/view/18483918/UyrAFxGx

events

REQ	PATH	PARAMS	DESC
GET	/events	long, lat, max_dist_km	Returns a subset of the events that are in range from the (lat, long) starting point of the request to the maximum distance, specified in kilometers.
GET	/events/auth	long, lat, max_dist_km	Returns a subset of the events that are in range from the (lat, long) starting point of the request to the maximum distance, specified in kilometers.
GET	/event/event_id		Returns the event specified by its id.
GET	/event/auth/event_id		Returns the event specified by its id.
GET	/events/user/user_id		Returns the subset of the events to which the user with id equal to user_id is subscribed.
GET	/events/admin/user_id		Returns the subset of the events to which the user with id equal to user_id is administrator.
POST	/event/add		Add a new event with info specified in the request body.
POST	/event/event_id		Update a new event with info specified in the request body.
DELETE	/event/event_id		Delete the event with specified event_id

bookings

REQ	PATH	PARAMS	DESC
GET	/bookings/event	event_id	Returns all the bookings done for the event with specified event_id .
GET	/bookings/user	user_id	Returns all the bookings done by the user with specified user_id .
POST	/booking/add	event_id	Add a new booking for user requesting to the event with id event_id . The user id is inferred from the auth token.
DELETE	/booking	event_id	Delete the booking for user requesting to the event with id event_id . The user id is inferred from the auth token.

users

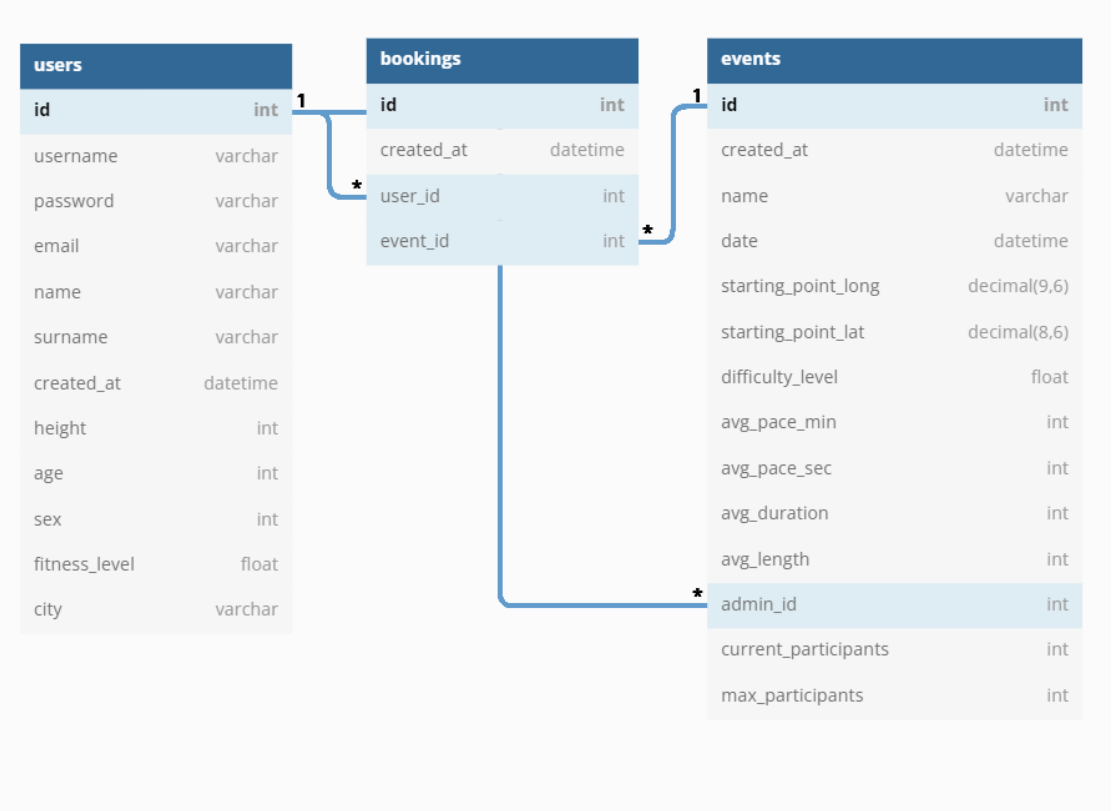
REQ	PATH	PARAMS	DESC
GET	/user/id/user_id		Returns user info (except for password) for the user specified by user_id.
GET	/user/username/user_id		Returns user info (except for password) for the user specified by username.
POST	/user/add/	user_info in req body	Add a new user. After adding auth the conventional way to add a new user is auth-register and then users-update user.
POST	/user/user_id	user_info in req body	Update an existing user.
DELETE	/user/user_id		Delete the user specified by user_id.

auth

REQ	PATH	PARAMS	DESC
POST	/login	username, password	Login the user returning the access token and the user_id if successful, or 401 UNAUTHORIZED if not.
POST	/register	username, password, email	Register the user returning its user_id if successful, or 409 CONFLICT if not.

4.2.2 Database

The relational scheme of the db is schematized as follows:



4.3 Application

To manage the data in the client application we used an implementation of the observer pattern, a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

With this solution, every class can easily access data and be notified of updates (for example events, selected theme colors, location) when needed, thus providing cleaner architecture, easier to use and maintain.

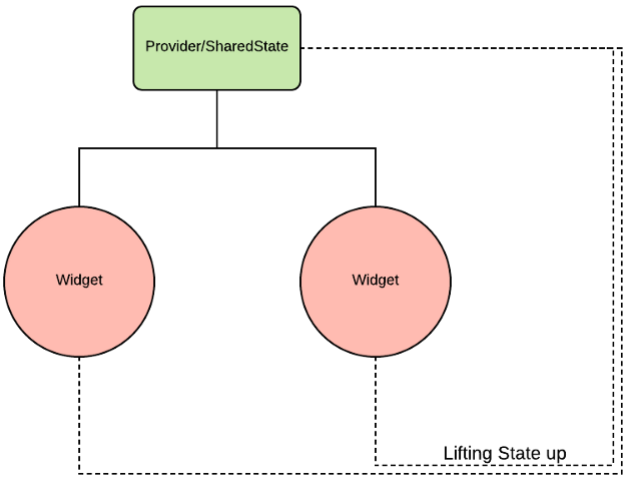


Figure 4.2: High level architectural overview of the Provider package, used to implement the observer pattern to manage data inside the client.

An example of use case of this pattern is the events class, used to store the data retrieved from the API.

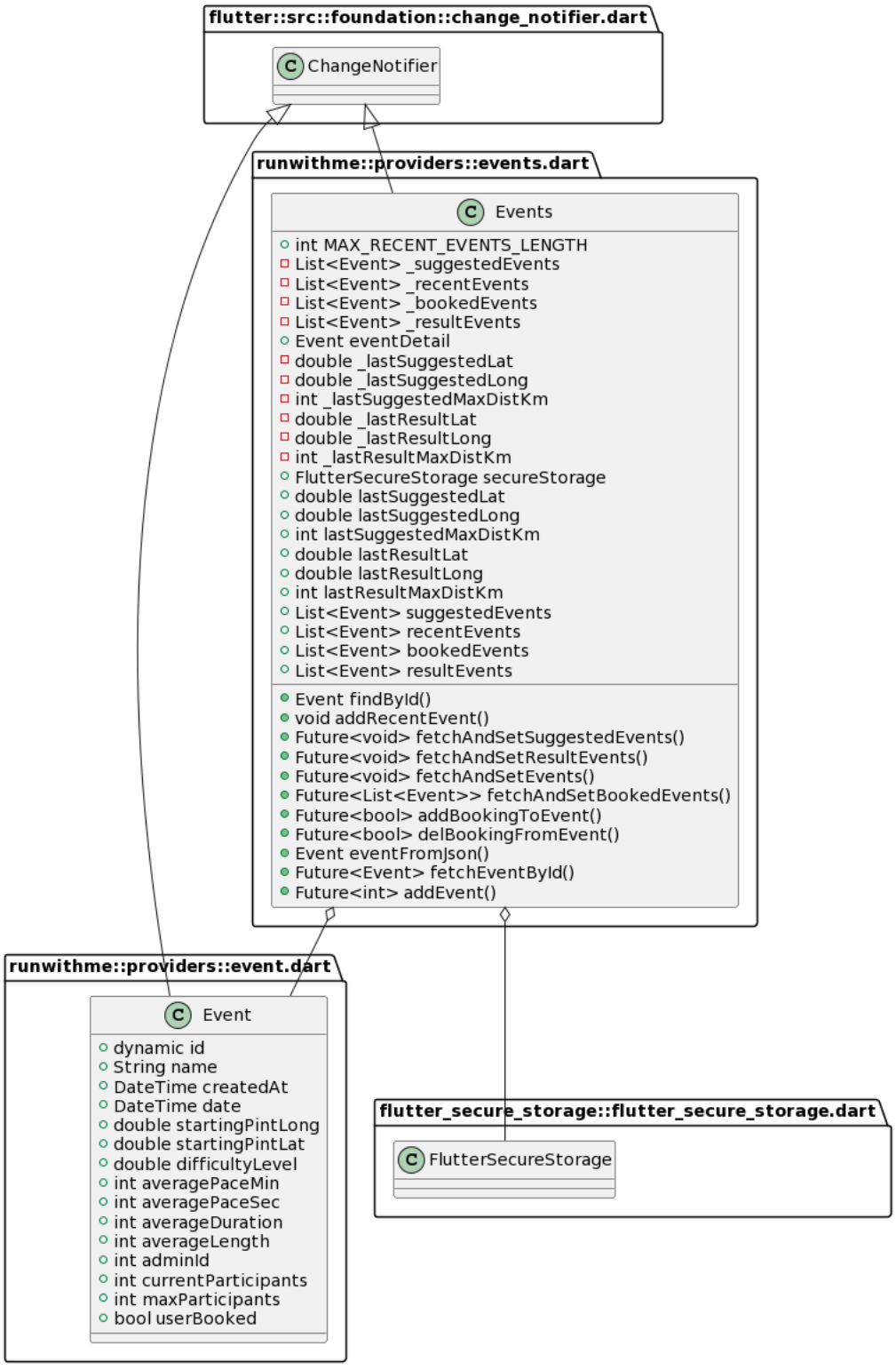


Figure 4.3: UML diagram of the events class, implementing the observer pattern. Here FlutterSecureStorage is used to retrieve the auth token to attach it withing HTTP requests.

4.3.1 Device Storage

In order to function properly, the application needs to store a set of parameters and configuration settings, such as the user preferred theme mode and the selected map style. To do so, a custom *SettingsManager* provider has been developed, as well as a *FileManager* class to manipulate files on the device local storage. Throughout the app, the Settings manager reads or writes a JavaScript Object Notation (JSON) file and parses it to and from a settings object that can be used by any other widget.

4.3.2 Storage Security

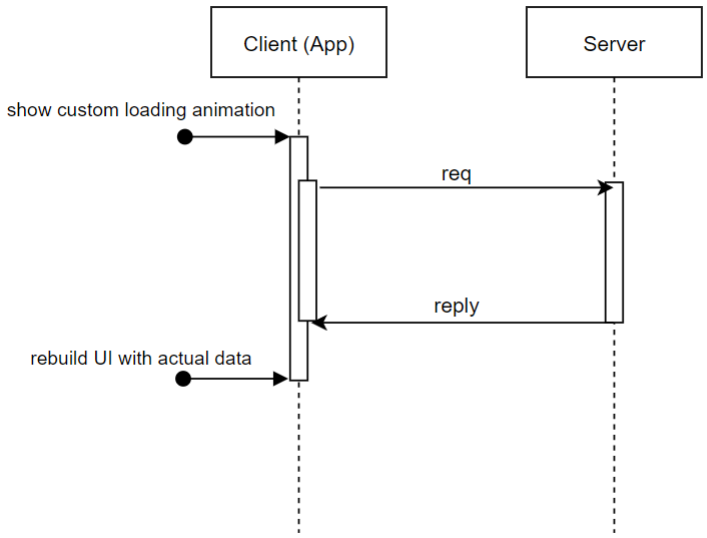
Similarly, the *SettingsManager* is used also to store the user login credentials on the device storage. To do so safely, the settings manager uses an external flutter module called *FlutterSecureStorage* to encrypt the credentials using AES encryption for Android. AES secret key is encrypted with RSA and RSA key is stored in KeyStore.

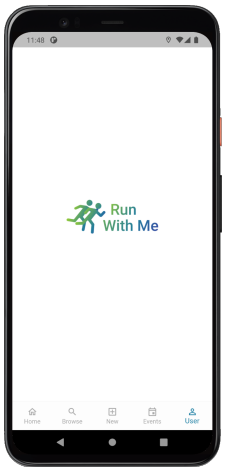
4.3.3 Multithreading

The app requires to be designed to support multithreading, because we must support the interaction with external services and guarantee an interactive while making requests. We also need multithreading to keep updating the current position of the user by updating the Global Positioning System (GPS) position with a thread continuously running in the background.

API

For the HTTP requests, we designed the app to dispatch in an asynchronous way, showing a customized loading animation while waiting for the request to complete. When the reply is received, we update theUI showing the data to the user.





GPS

To keep the current user position updated, we have a thread running in the background that checks for user position updates every 30 seconds.

```
Future<void> start() async {
  //Starting background location thread
  keepBackgroundThreadAlive = true;
  while (keepBackgroundThreadAlive) {
    //Searching for user location in the background
    await Future.delayed(Duration(seconds: 30));
    determinePosition(LocationAccuracy.best, context);
  }
  //Closing background location thread
}
```

4.4 Sequence Diagram

In this section we provide 2 examples of sequence diagrams to describe add event and search events scenarios.

4.4.1 Add Event

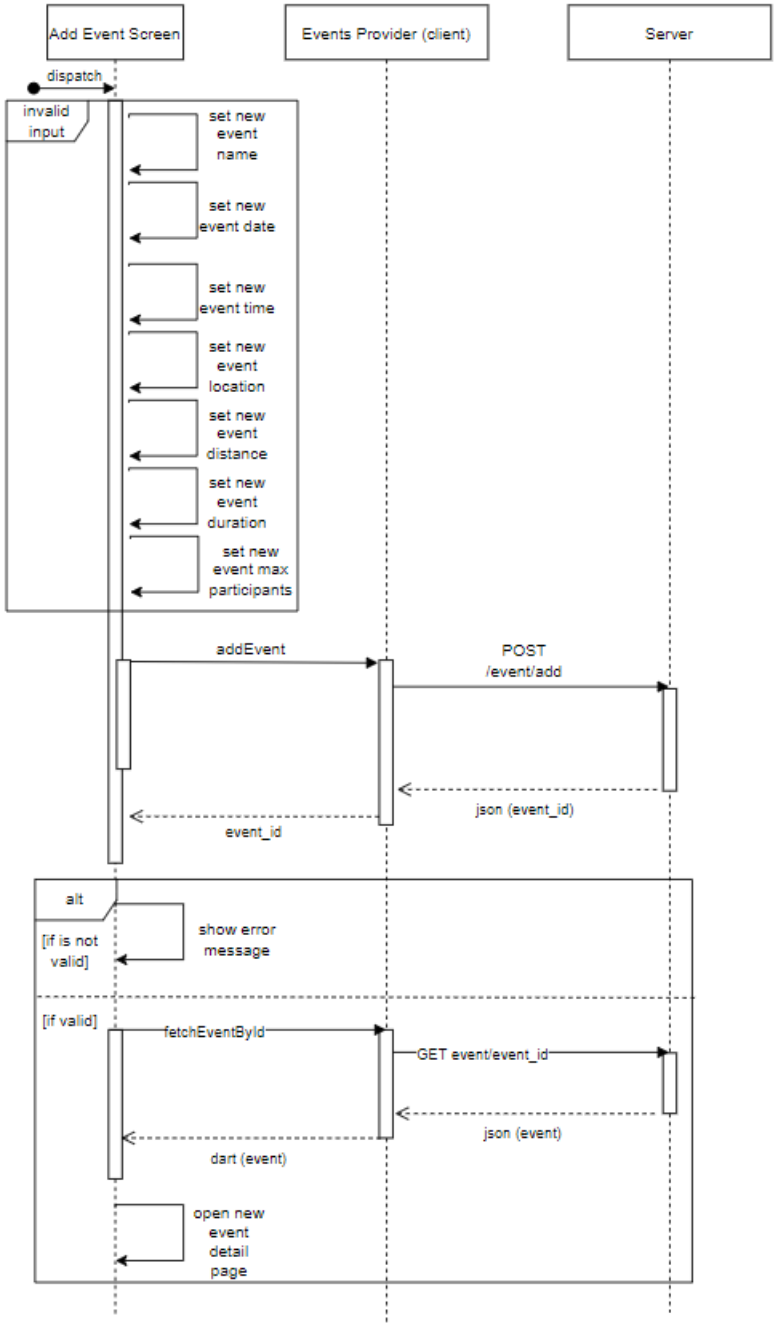


Figure 4.4

4.4.2 Search Events

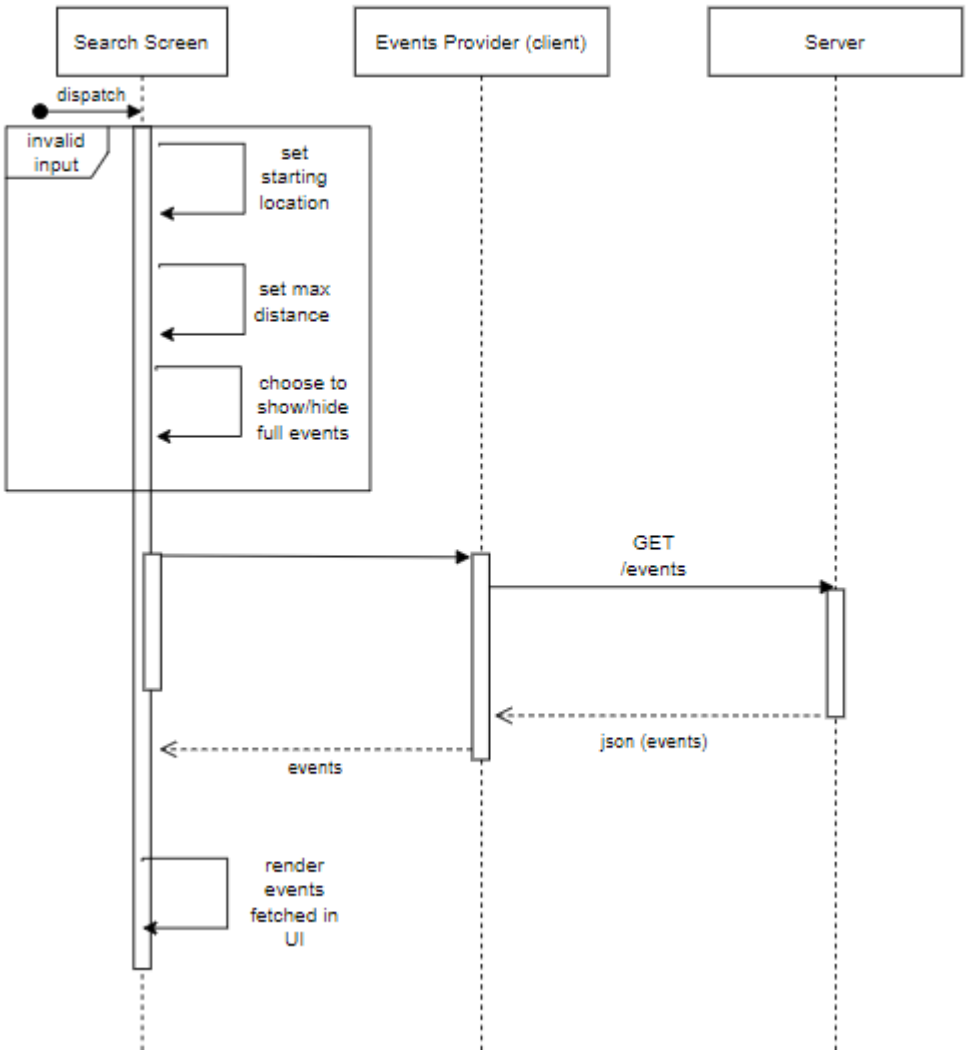


Figure 4.5

5 User Interface

5.1 Overview

In this section we will discuss the general layout of the application, that will remain mostly untouched, except for some overlay pages that can appear in fullscreen. This was done in order to aid the user navigating through the app: as a matter of fact, most of the interactive elements in the UI are always in the same positions so that the user can get accustomed to them and get quicker at using the app.

As can be seen in Figure 5.1 the app is essentially divided in three sections: the AppBar, the Body and the Navigation Bar (NavBar):

1. *AppBar*: the upper region of the screen, it can be flat or with a gradient depending on the current page. It is used to display the app logo for aesthetic purposes or to convey information about the current page at a glance.

2. *Body*: this is the main region of the screen, all the important information is displayed here.

3. *NavBar*: this is the menu of the app. From here the user can move between most of the pages, except for the Event Detail page and the register page that are accessible from specific locations.

5.2 Pages

In this section all the application pages will be presented and described from the UI perspective.

5.2.1 Home Page

The home page shown in Figure 5.2 is the default page that is opened upon startup of the application. Here the user is greeted, either by name of with a general message depending on whether it is logged in or not. Then, depending on the login status of the user, some basic information is displayed: some upcoming events that the user has subscribed to, some weather information for the user location, his current position and finally some of its past events.

5.2.2 Browse Page

The browse page in Figure 5.3a is used to search for new events nearby or at any given location. By tapping the search bar in the AppBar a bottom sheet is shown (Figure

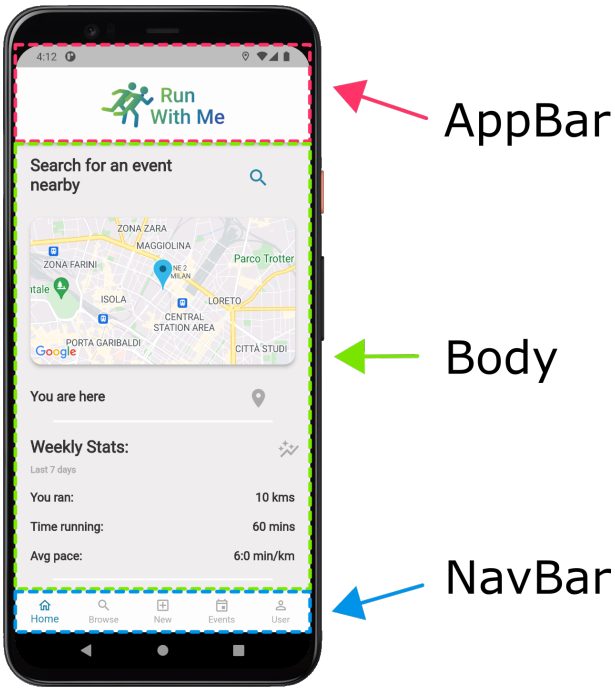


Figure 5.1: General application structure

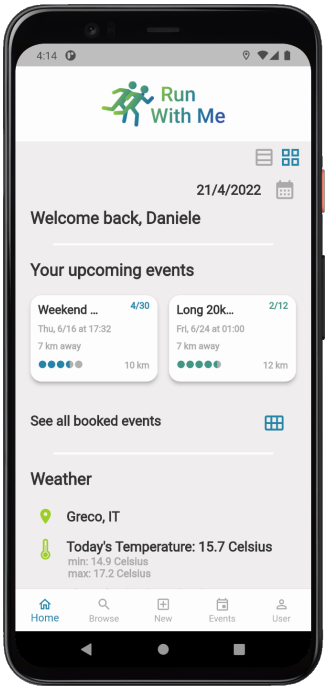
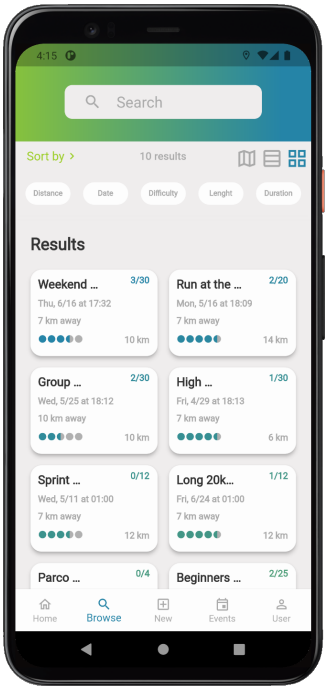
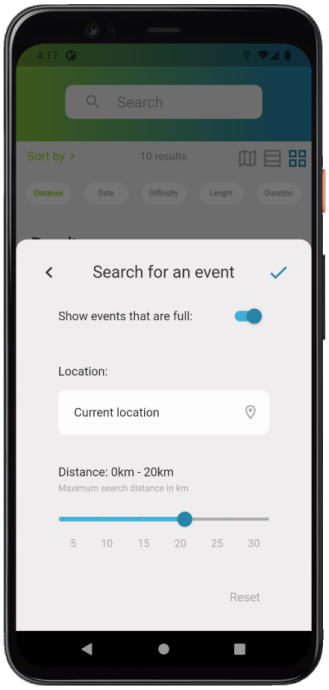


Figure 5.2: Home page screenshot



(a) *Event list screenshot*



(b) *Search bottom sheet screenshot*

Figure 5.3: Browse page screenshots

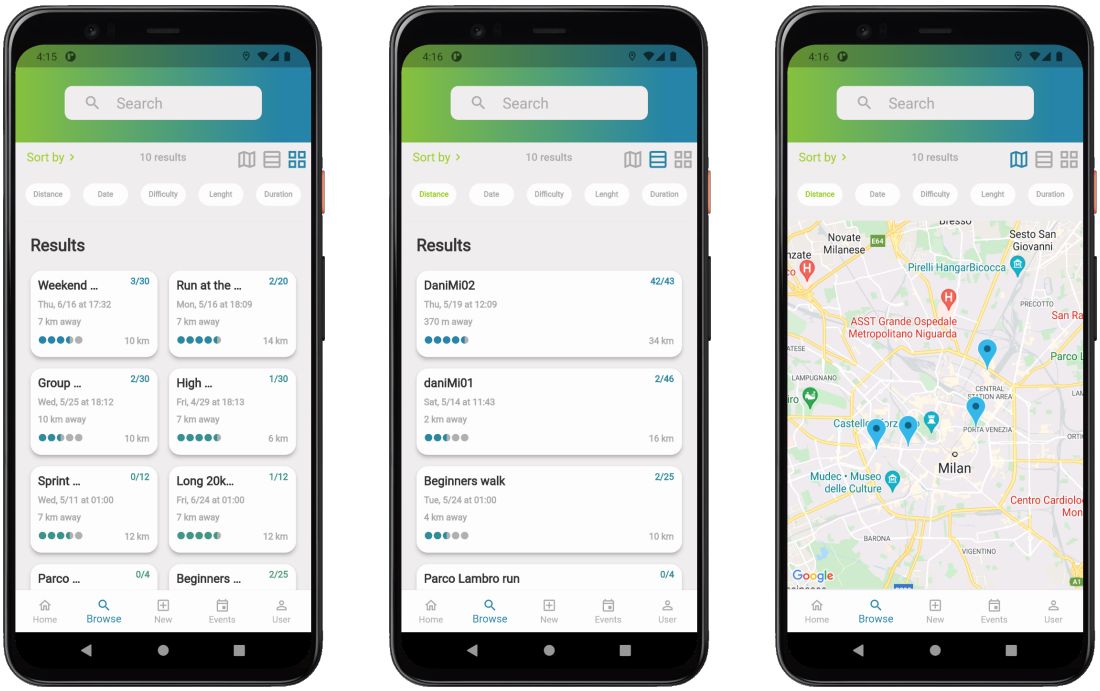
5.3b) and the user can initiate a search for future events. To do this, it is asked for a center location and search distance in kilometers. Finally the user can ask to receive all existing events or just those that are not already full.

Other than the results from searches, the browse page also shows the user some suggested events, based on parameters such as location and event difficulty compared to the user skill level. Finally this page also shows the recently visited events, so that a user can easily go back to a recent event while deciding which one he wants to subscribe to. All the events shown in this page can be viewed in different modes based on the active icon in the top right corner of the body of the screen. The three supported mode are *Grid Mode* (Figure 5.4a), *List Mode*(Figure 5.4b) and *Map Mode* (Figure 5.4c).

Finally, all events in this page can be sorted by *Distance*, *Date*, *Difficulty*, *Length* or by *Duration*. By tapping the *Sort By* button in the top left corner of the body of the app, as in Figures 5.4 5.3 a drop down menu with all the possible sorting parameters is shown and by tapping one of them, all events in the page are immediately sorted.

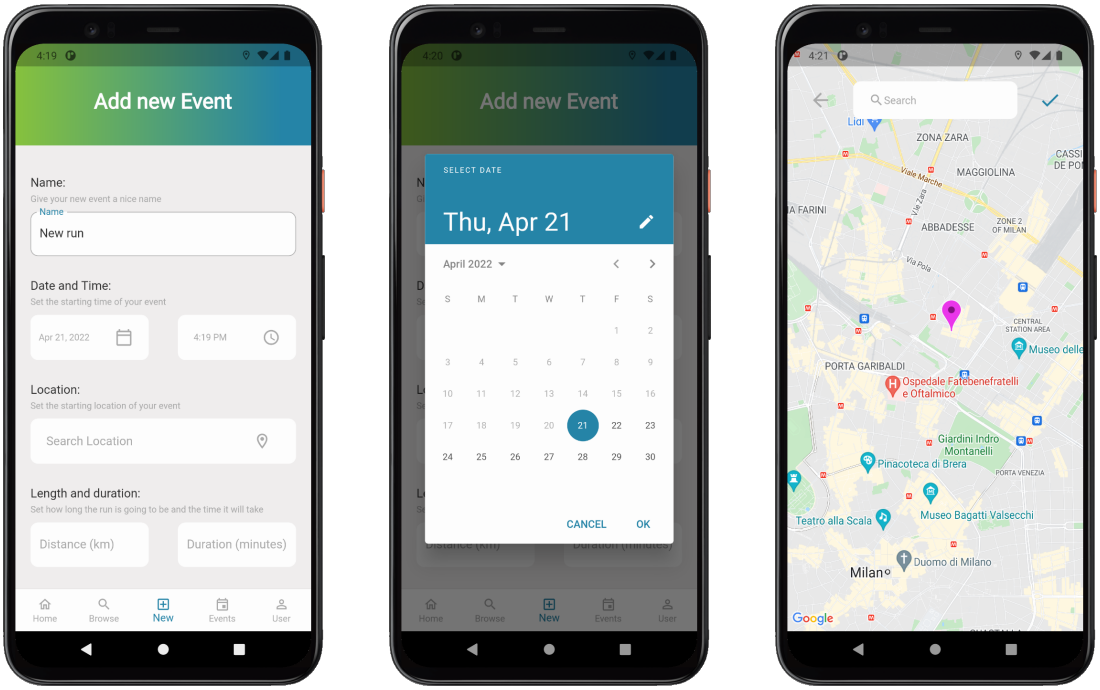
5.2.3 New Page

In this page the user can create new events for other users to see and to subscribe to. In order to create new events, the user have to fill a form (Figure 5.10a) with important



(a) *Grid mode* (b) *List mode* (c) *Map mode*

Figure 5.4: Browse page screenshots



(a) *Grid mode* (b) *List mode* (c) *Map mode*

Figure 5.5: Browse page screenshots

data such as the event name, the date, location, number of participants and others. The default android date picker (Figure 5.10b) has been used to allow the user to choose a date for the event. A similar time picker has been used for the time of day. For the event location a full screen map overlay, as in Figure 5.10c, has been used. From here the user can either search a location from the upper search bar, or directly input a location by long pressing the map.

5.2.4 Events Page

In the Booked Events page (Figure 5.6) the user can check all the events that he has subscribed to, both in the past and in the future. As in the Browse page, the user can sort events by multiple parameters by tapping the *Sort By* button and selecting the correct parameter.

5.2.5 User Page

The User page in Figure 5.7 is used to show the current user all its information. Also, at the bottom of the page the user can find some settings and some general information about the app. From the settings section, the user can select the theme mode of the app and can also log out of the application.

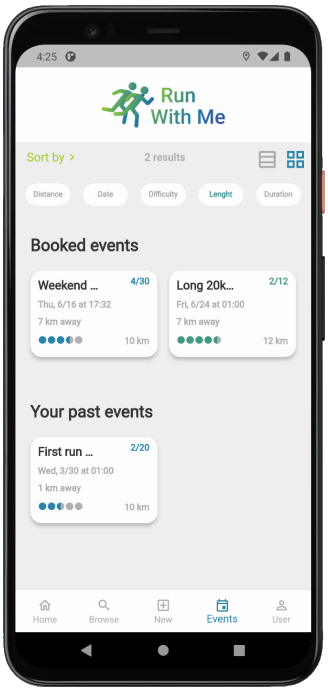


Figure 5.6: Events page screenshot

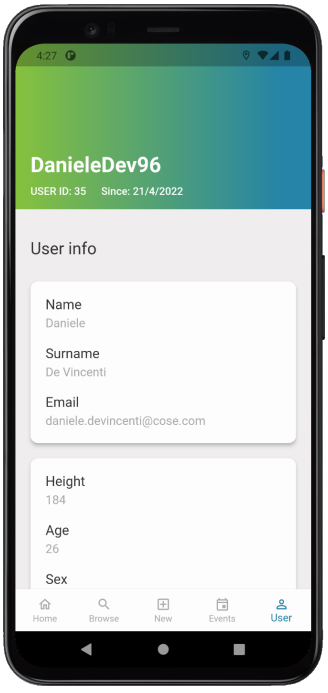


Figure 5.7: User page screenshot

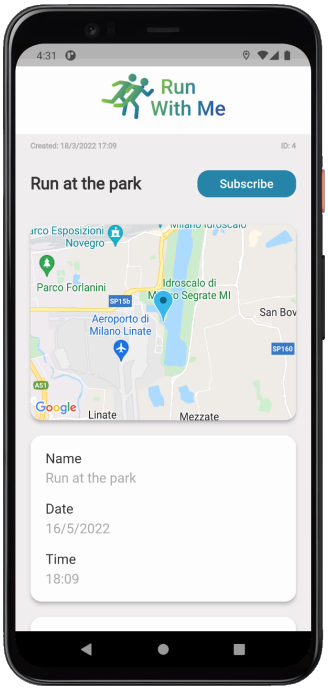


Figure 5.8: Event detail page screenshot

5.2.6 Event detail Page

This page is used to show the user all the details of a particular event that he has selected. As can be seen in Figure 5.8, this page renders full screen and it has no bottom navbar. To open this page the user simply touches an event card from any other page and the app spawns the new page. In the same way the user can then go back one step, thus closing this page and returning to the previous one. In the event detail page, the user can see all the information regarding a specific event, such as its location, name, date and so on. The user can also subscribe or unsubscribe to such event, if he is logged in. In case the user is not logged in, a log in button is shown instead.

5.2.7 Login Page

This page is shown to the user in place of the normal user page when he is not logged in. From here the user is prompted for the *username* and *password* and a *log in* button is shown. There is also a text button, clearly visible yet not distracting, that guides the user through the registration process. Tapping this button will open up a *bottomsheet* that will be explored in the next section.

5.2.8 Register Page

The register page, which is technically not a page and more of a *bottomsheet* that opens up in the user page, is used by a new user to register into the RunWithMe app. To do

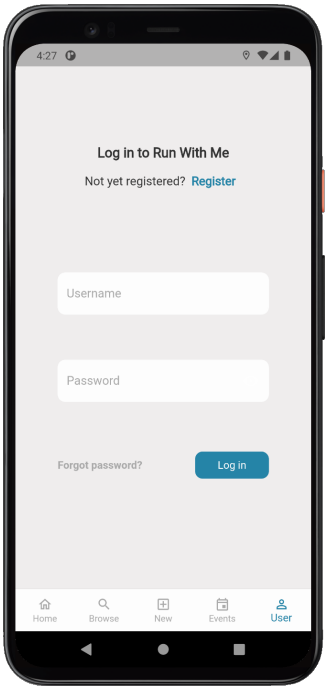


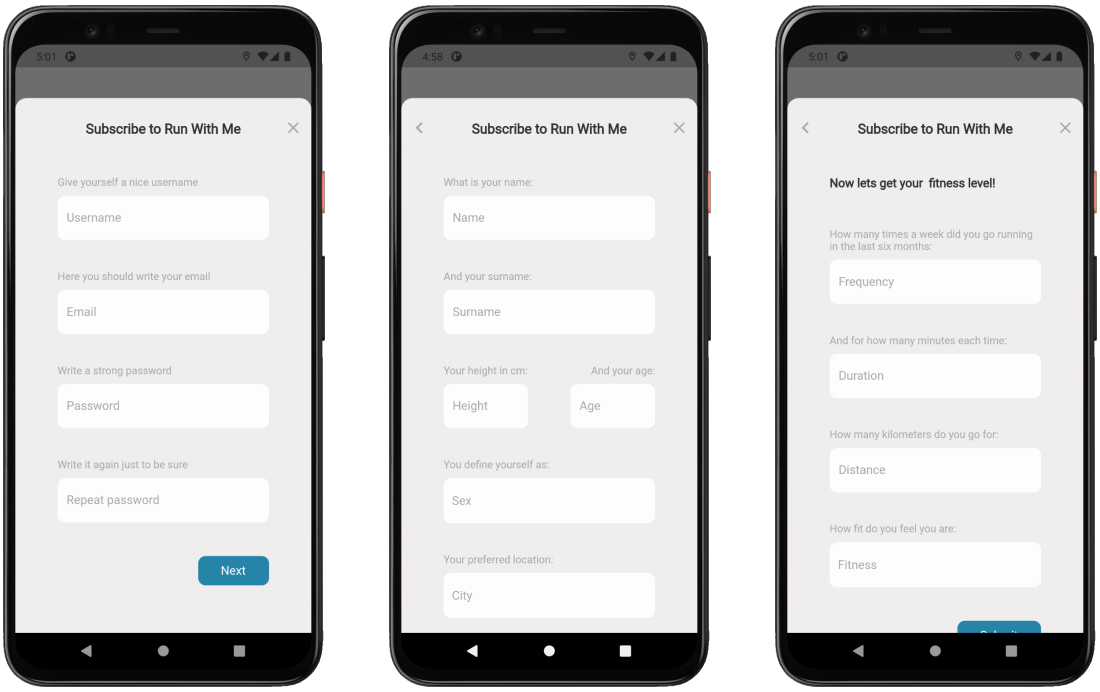
Figure 5.9: Login page screenshot

so, the user is asked a series of questions and is prompted for some information regarding himself in a three step process. At first, the user is asked for a new *username*, a valid *email*, an a strong *password*. In the second page, some more personal information are requested, such as *name*, *surname*, *sex* and default *location*. To compute the user fitness score used throughout the app, the third and last page of the registration process asks the user for some general behavioral information, that are then used in a proprietary algorithm to establish the user's fitness level from *one* to *five*.

At any point during the registration, the user can freely move through the three pages and change an answer without losing any previously inserted data. Only when submitting the registration or when closing the *bottomsheet*, the inserted data is lost.

5.3 Multi Device Support

A key feature of any modern application is the ability to scale properly on a multitude of devices, each with different screen sizes. To achieve this, all major UI elements for this app have been designed to be scalable so that they can dynamically resize to fit as many different screen sizes as possible. After the implementation phase, the app was tested with a selection of virtual devices to ensure that it was working properly on all screen sizes. As can be seen in Figures 5.11, 5.12, 5.14, the app renders correctly both in a smartphone screen size and in a tablet size. Moreover, different smartphone screen sizes and aspect ratios were tested to a positive result. The devices in those screenshots



(a) *First page* (b) *Second page* (c) *Third page*

Figure 5.10: Browse page screenshots

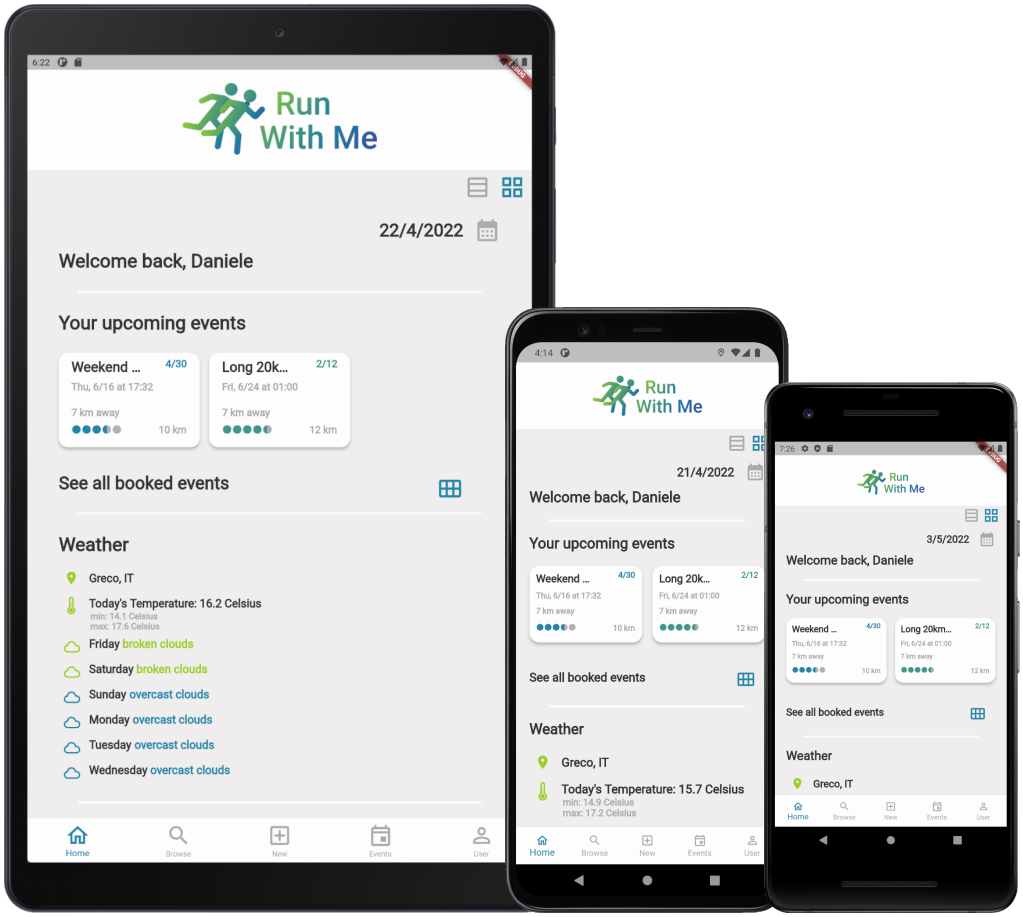


Figure 5.11: Screens on different screen sizes

are respectively: a *Samsung Galaxy Tab A 2016*, a *Pixel 5* and a *Pixel 2*.

Other than just scaling UI elements and text sizes according to the screen dimensions, a particular effort has been made in differentiating the layout between tablets and smartphones. In fact, with bigger screen sizes, the app dynamically adjust spaces between elements to make a better use of the bigger screen space available. Also, the app can decide to show more than two columns of events if the screen size is big enough, as it's shown in Figure 5.12.

We also added the landscape mode only for tablets since it would be useful as many tablet users ues it always in landscape mode. We did not implemented landscape mode for mobile phone's screens since it would not be feasible with our application.

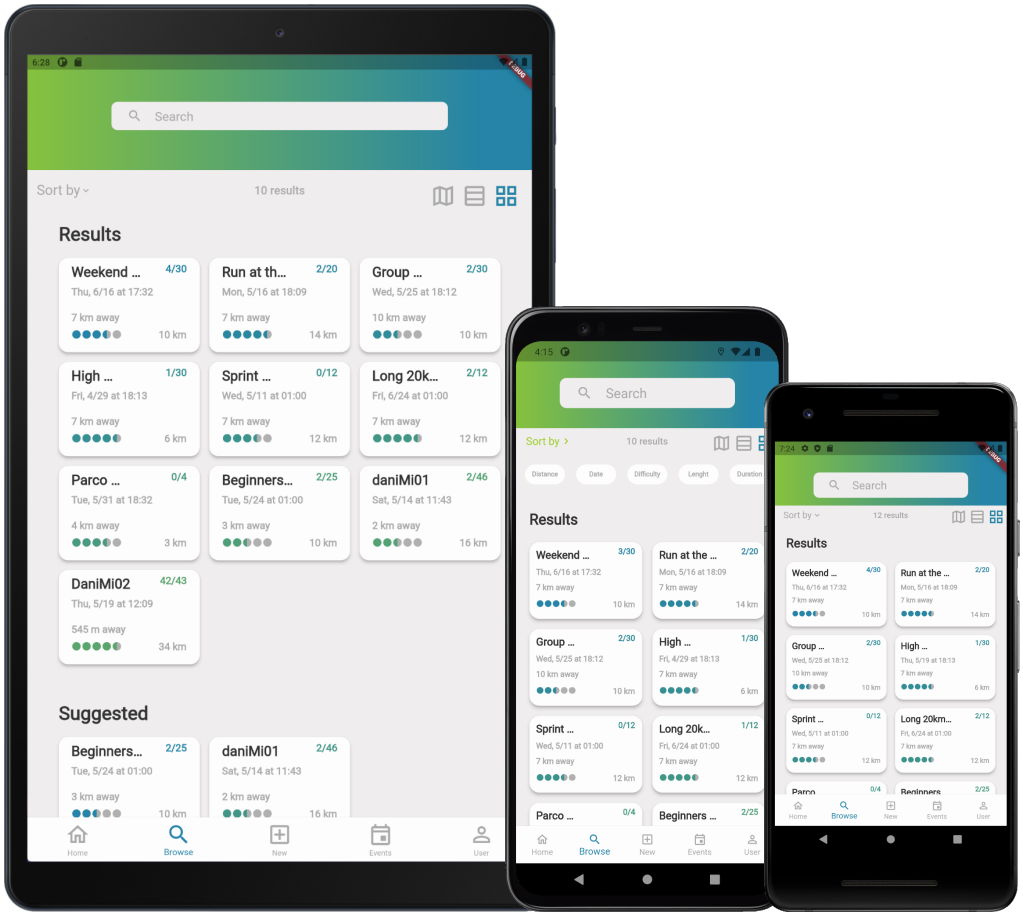


Figure 5.12: Screens on different screen sizes

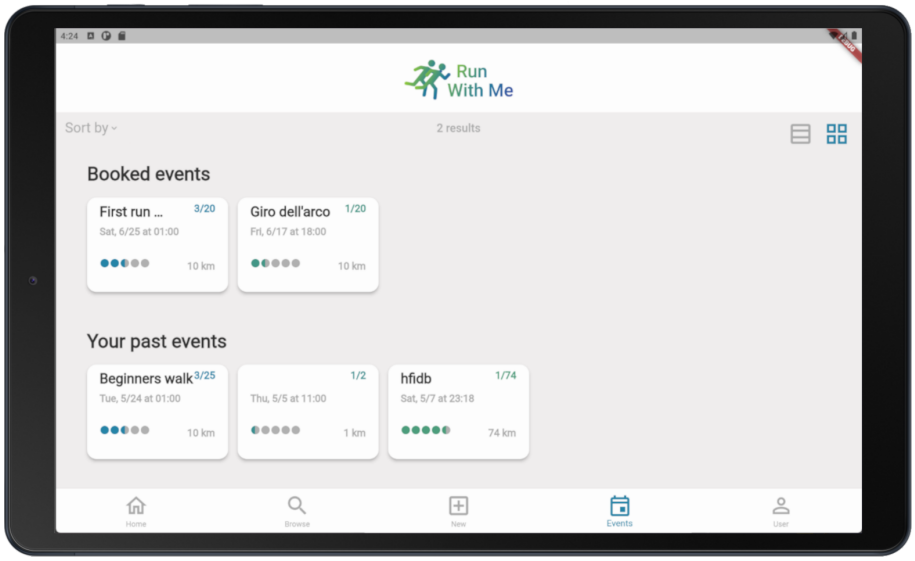


Figure 5.13: Landscape mode on Tablet

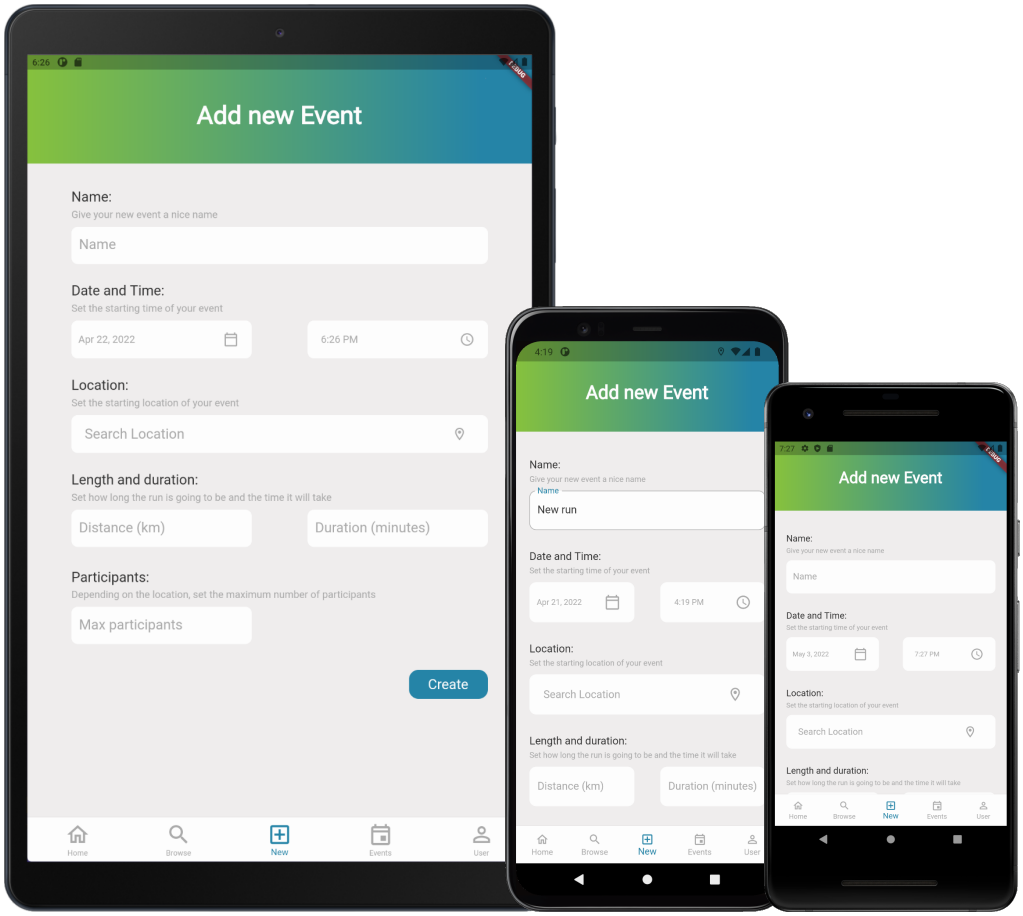


Figure 5.14: Screens on different screen sizes

6 User Experience

An important phase in the development of any application is the design of the user experience, so that the users of the app can get the best possible experience from the start and are not tempted to delete the app out of frustration.

6.1 Use case

First of all, the use cases of the application were determined, as shown in Figure 6.1. Three main actions were conceptualized: after a registration and login phase, the user can either *add* a new event, *search* for an event previously created, and finally *subscribe/unsubscribe* to an event.

6.2 Route Flow

Then a main use flow has been designed, in order to define all the different screens of the application and all the different options for the user to move between them. As shown in Figure 6.2, the main pages are all available at the first level, more specifically from the bottom NavBar. Then, from the *Browse* and the *booked* pages, the user can navigate to the single *Event* detail page. Also, from the *User* page, the user can be routed to the *Login* page or the *Registration* page according to the situation.

6.3 Usability

During the design phase of the application, it was clear that the user needs to be guided throughout the different application pages and all their functions and options. To do so, a number of error messages and alert dialogs have been implemented to guide the user in performing basic functions.

6.3.1 Error Messages

Error messages in particular are used to help the user understand what a certain input form is asking for. As an example, in Figure 6.3a it is shown that the user has inserted a clearly wrong email and the app throws an useful error message, telling the user to provide a valid email. Much in the same way, the application returns a meaningful error message if the two passwords are not the same (Figure 6.3a) or if the user tries to search for events without selecting a location (Figure 6.3b).

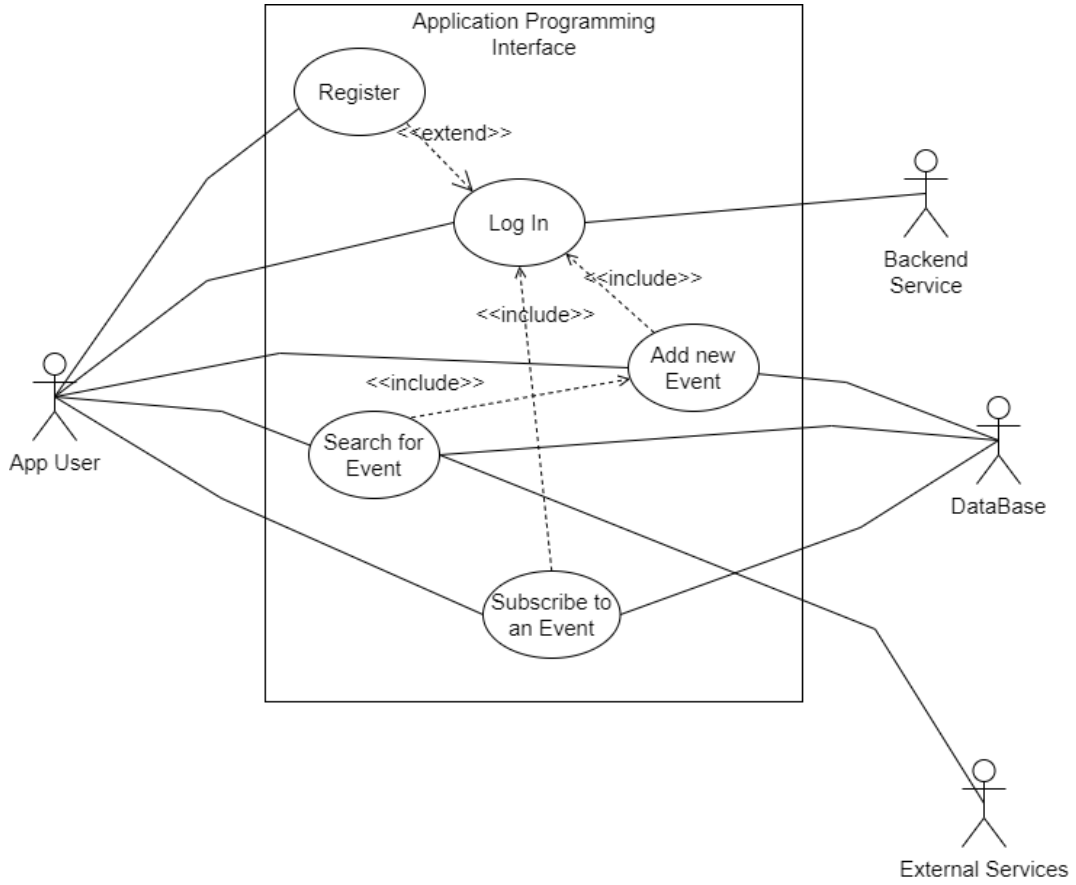


Figure 6.1: Places search dialog screenshot

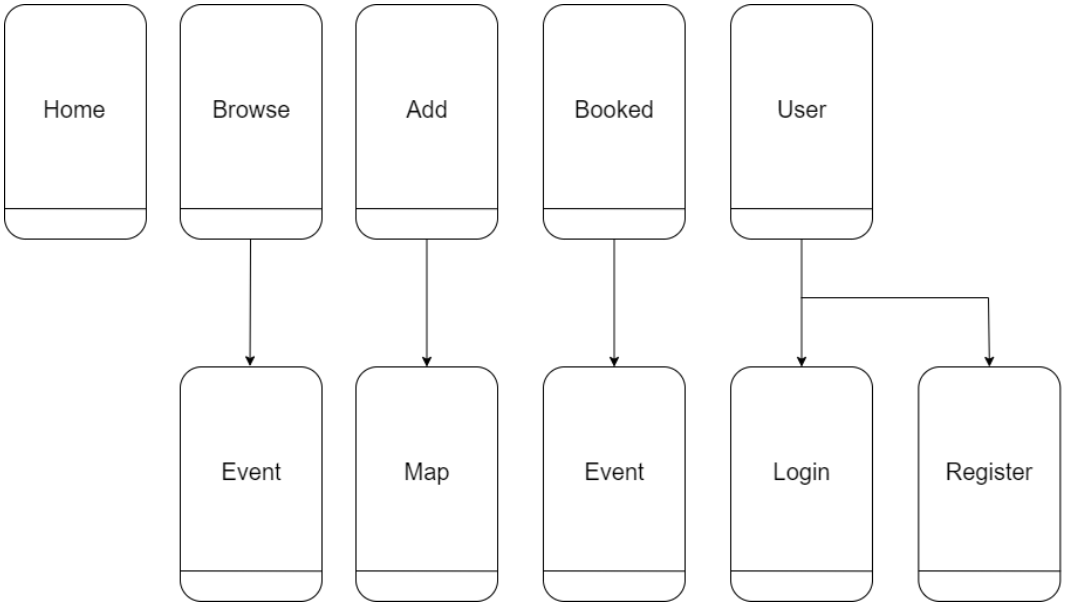
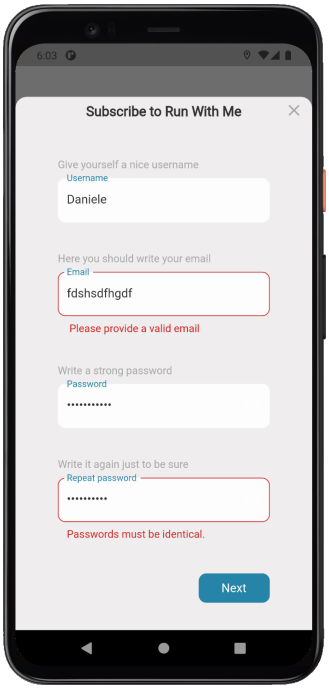
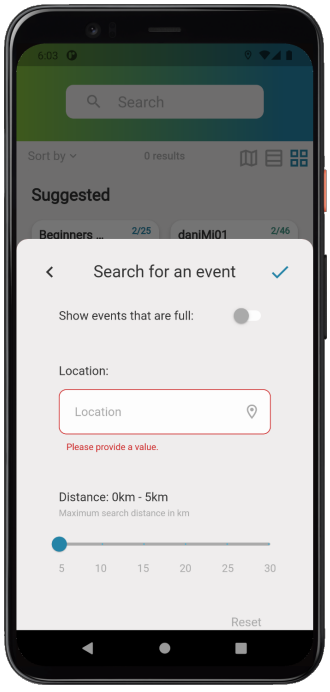


Figure 6.2: Places search dialog screenshot

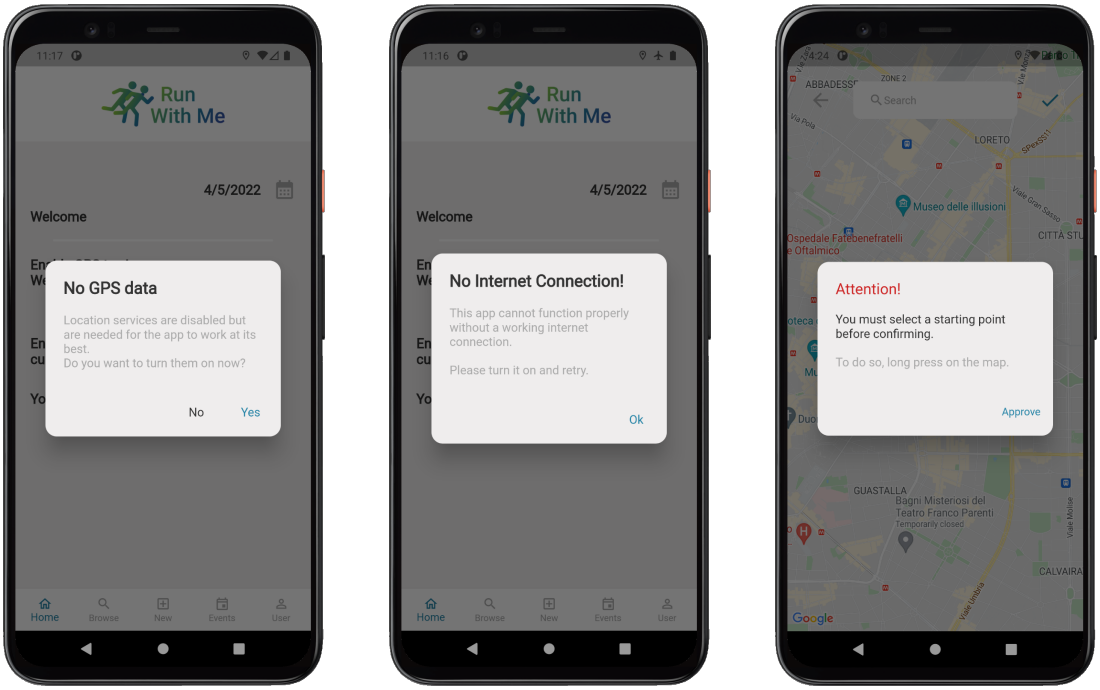


(a) *Wrong email format*



(b) *No location inserted*

Figure 6.3: Error message screenshots



(a) *No Location* (b) *No internet connection* (c) *No starting point*

Figure 6.4: Alert dialog screenshots

6.3.2 Alert Dialogs

Another tool for developers to help the user understand what the application is doing are alert dialogs. As an example, in this project are used when asking the user for permissions: in Figure 6.4a and 6.4b it is shown how the app asks for location services and data connection, redirecting the user to the respective setting page. Also, alert dialogs are used when asking for confirmation before taking a risky action, such as logging out of the application.

7 Testing Campaign

In this section we describe the most relevant testing done while implementing the application, both manual and automated.

7.1 Manual Testing

Some of our testing was done manually, for example exploratory testing while implementing the application, usability testing and some of the user interface testing, using both an Android emulator, our phones and tablet to test the application in different screen sizes.

7.1.1 UI Testing

We used SingleChildScrollView class to make the content of the body scrollable, thus avoiding content overflows especially on smaller screens. We also used relative sizes by accessing the context height and width from flutter MediaQuery class.

```
1     screenHeight = MediaQuery.of(context).size.height;
2     screenWidth = MediaQuery.of(context).size.width;
3     [...]
4     Container(
5       padding: EdgeInsets.only(left: 10),
6       height: screenHeight / 20,
7       width:
8       80 + multiDeviceSupport.tablet * 35,
9     );
```

Figure 7.1: Example of relative UI sizing based on the user's device. multiDeviceSupport.tablet is a double variable that changes based on the screen width.

7.2 Unit Testing

We used unit testing to test mainly our dart classes that are independent from Flutter. For example, we tested our stats_helper class responsible of generating user statistics for the last 7 days, or date_helper class containing useful methods for date conversions and calculations.

```
    test('Avg Pace calc test', () {
      avgPace.addAll([5, 0]);
      expect(statsHelper.calcWeeklyAvgPaceParams(20, 100), avgPace);

    test('diff in days', () {
      expect(DateHelper.diffInDays(saturday, friday), 1);
    });
```

7.3 Widget Testing

We tested the widgets that could be tested in isolation, mainly the single widgets that then composes the screen widgets. We focused on checking that widgets were correctly displayed in the testing environment, and also some input checks, that we also done manually.

```
    await tester.enterText(find.byKey(const Key("usernameForm")), "user");
    await tester.enterText(find.byKey(const Key("passwordForm")), "password");

    GlobalKey<FormState> formState =
        (tester.state<LoginFormState>(find.byType(LoginForm)).form);
    formState.currentState!.save();

    expect(
        tester
            .state<LoginFormState>(find.byType(LoginForm))
            .initValues
            .values
            .first,
        "user");

    expect(
        tester
            .state<LoginFormState>(find.byType(LoginForm))
            .initValues
            .values
            .elementAt(1),
        "password");
  });
```

Figure 7.2: Example of widget testing automatically compiling the login form

7.4 Integration Testing

Since the screens widgets relied on HTTP requests, GPS positioning and other context data (like user settings) we needed a way to integrate this into our testing campaing. We used the Mockito package to let us implement and override our provider classes implementing the observer pattern and responsible for client data handling, returning mock data. In this way we managed to test the app screens as a whole instead of testing only their composing widgets, by removing network, and providing fake GPS data to our test.

```
1  @override
2  Future<void> fetchAndSetResultEvents(
3      double lat, double long, int max_dist_km, bool isLoggedIn) async {
4    _bookedEvents.addAll(dummyEvents);
5  }
```

Figure 7.3: Example of overriding of our events provider managing HTTP requests for testing purposes

7.5 API Testing

As we developed the backend while implementing the app with dummy data, we used Postman to test our API. We tested all our API endpoints, first within a local environment, and then with a Virtual Private Server (VPS) that was actually reachable from network. The API testing was done manually, we included also a script to attach the authentication token as an environment variable to every request needing it after logging in.

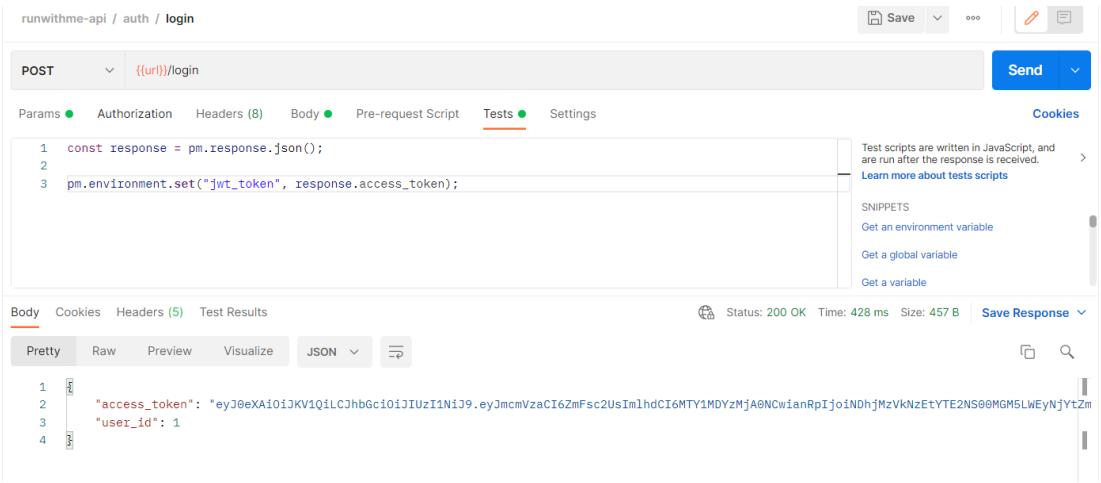


Figure 7.4: Postman login request and post-request script to assign authentication token as environment variable for other requests

To access directly the database for testing purposes we used an instance of phpMyAdmin that let us operate directly on the data. The containerized version of the backend contains everything needed to have a working version with little configuration needed: the web server (Nginx), the application server (FLask), the database server (MySQL) and the database administration tool (phpMyAdmin).

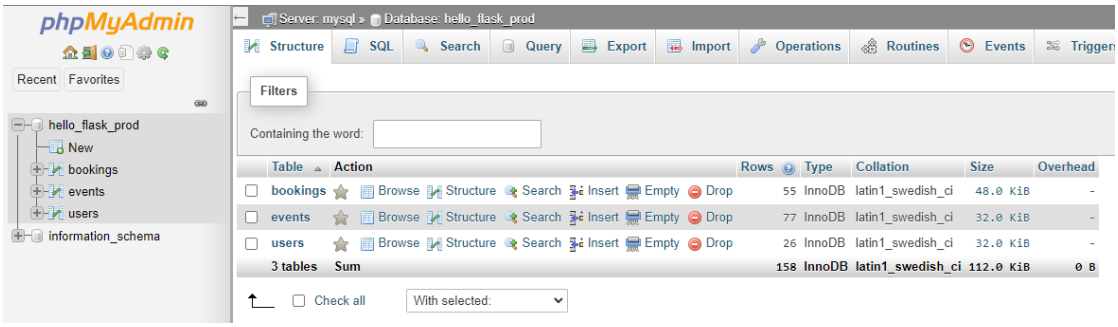


Figure 7.5: phpMyAdmin, used for testing the backend by accessing directly the database

8 External Services

With applications getting more and more complex and feature rich, often a set of external services are used to implement new features that would be difficult if not impossible for the app developers to implement on their own. It is the case for this project, where a multitude of external services have been used to achieve support of major functionalities.

8.1 Google Places APIs

In particular, this app uses *Google Places APIs* to easily store the default user location that is set at registration time. This location is used for default event searching and if the devices location functionalities are disabled. *Google places APIs* provide an easy way to correlate any position in the world with an unique identifier that can be stored in the backend as a string. This identifier can be used later on to obtain back the place coordinates and information. Much in the same way, this service is used to obtain geographical coordinates from the user input text when searching for events or when creating a new event. In Figure 8.1 a utilization example is shown: given any string of text, the *Google Places APIs* will return the top 5 most probable locations and when the user taps on one of them, all the information of the location are retrieved.

8.2 Google Maps APIs

Google Maps APIs are used in this app to show maps to the user. This is useful when selecting a location for a new event in a different way then by text, or to show the user the exact location of an event. Another use case for maps in this app is the map view of the searched events, as can be seen in Figure 8.2.

8.3 OpenWeatherMap APIs

During the requirements phase we thought it would be useful for an user to see today's weather/temperature and next days forecasts. We used OpenWeatherApi, which provides us with data. The location for the request is the current location of the user, or

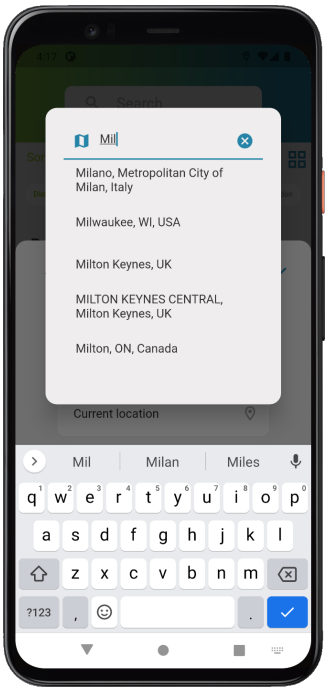


Figure 8.1: Places search dialog screenshot

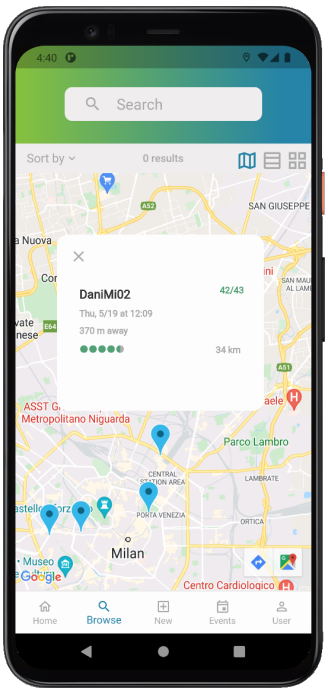


Figure 8.2: Maps overlay screenshot

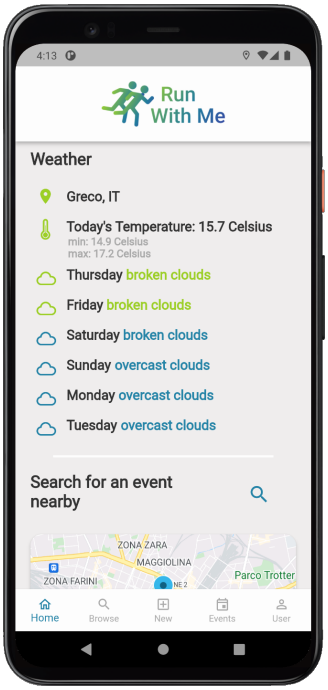
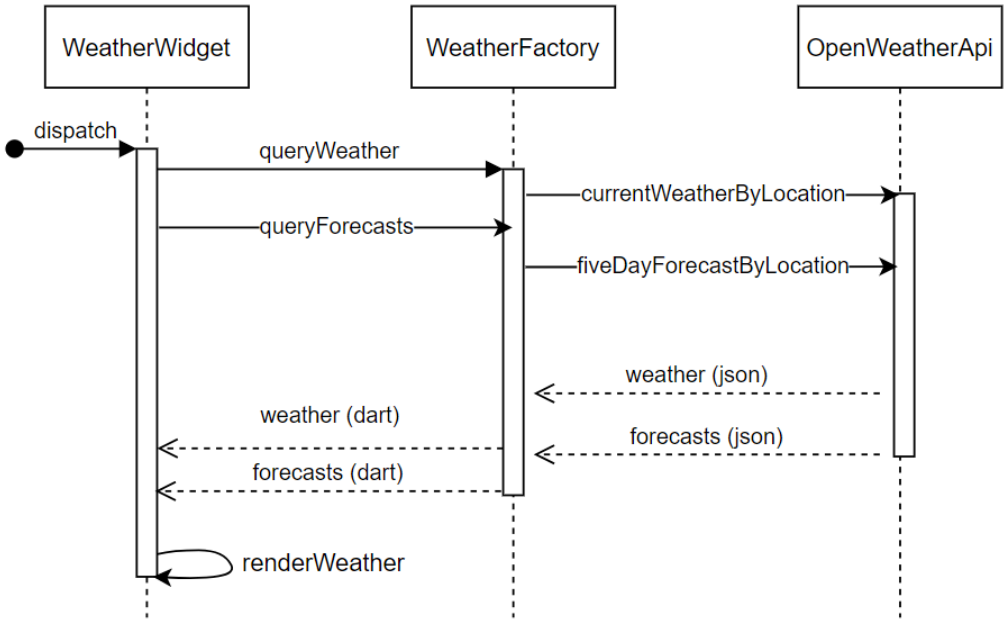


Figure 8.3: Weather section screenshot

the default user location in case GPS is disabled/not working. Data are passed throught JSON representations, then decoded into Weather objects and displayed to the user.


WeatherWidget
WeatherFactory
OpenWeatherApi
dispatch
queryWeather
currentWeatherByLocation
queryForecasts
fiveDayForecastByLocation
weather (json)
weather (dart)
forecasts (json)
forecasts (dart)
renderWeather

9 Future Developments

For future developments, we would like to implement all low priority functional requirements, add more of them and extend non functional requirements. Some of the future developments was already thought during the design phase, for which we already set up the architecture to support them.

- add more social functionalities, like being able to see users subscribe to an event and for each of them be able to open an user detail page
- add a specific chat for each event, and a direct message option to other users
- upload a profile picture
- implement a notification system
- give the possibility to export event in .ics format to add to calendars
- add social authentication

Glossary

API Application Programming Interface. 14, 18

AppBar Application Bar. 8, 11, 24

auth Authentication. 14

GPS Global Positioning System. 20, 43, 47

HTTP Hypetext Transfer Protocol. 15, 20, 43

JSON JavaScript Object Notation. 20, 47

NavBar Navigation Bar. 24, 37

REST REpresentational State Transfer. 14

TCP Transmission Control Protocol. 15

UI User Interface. 8, 9, 20, 24, 31, 33

VPS Virtual Private Server. 43