POLITECNICO MILANO 1863

Department of Computer Science and Engineering

# BADDY

Design Document

| | |
|---|---|
| **Morreale** Federico | 10561624 |
| **Marini** Gabriel Raul | 10575543 |

# Contents

Chapter 1

# Introduction

## 1.1 Purpose

In this section we will briefly introduce the application. The application is developed for the course of *Design and Implementation of Mobile Applications* of professor Luciano Baresi at Politecnico di Milano. The goal of the course is to design and implement a mobile application on a platform of our choice. This document illustrates the decisions we made in order to accomplish this goal.

This Software Design Document is a document that provides documentation that will be used as overall guidance to the architecture of the software project. Moreover, I will provide documentation of the software design of the project, including use case models, class and sequence diagrams

The purpose of this document is to provide a full description of the design of *Baddy*, a cross-platform application, providing insights into the structure and design of each component.

## 1.2 Scope

The idea of this application comes from a need for the elders to find someone that can take care of them, if they do not have children, or they are not available. We are sure that the next generation of old people will be able to use applications smoothly without any trouble. We will now briefly describe all the functionalities.

The user will be able to register and search for a person who is going to take care of him. The caregiver too needs to register and set its city, name and phone number in order to be reached by all the other users. Elders can also write reviews and rate the caregiver. Caregivers can update their profile with additional information like age, birth, nationality, gender and set their state to be available for a job or not.

## 1.3 Stakeholders

The main stakeholders of this app are the elders and people that wants to earn something extra as caregiver, but not limited to children of the elder that want to find someone when they are absent for example.

Moreover, we have to consider, as a stakeholder, professor Luciano Baresi that holds the *Design and Implementation of Mobile Applications* course and Giovanni Quattrocchi, teaching assistant of the course.

## 1.4 Time Constraints

We have no precise deadline for this application, our idea is to ultimate it for one of the 2 calls in Jan/Feb. We already know the technologies to develop the application so we have to spend less time for that part. We are starting the development in November together with this DD, then we will do all the necessary testing.

## 1.5 Overview

This document is structured as follows:

1. **Introduction.** A general introduction of the Design and Technology Document. It aims giving general but exhaustive information about what this document is going explain.
2. **General Overview.** A general overview of the project. In this section, the reader could find the core features of the application and the requirements of the system.
3. **Architectural Design.** This section contains an overview of the high-level components of the system and then a more detailed description of three architecture views: Component view, Deployment View and Runtime View. Finally, it shows the Component Interfaces, and the chosen architecture styles and patterns.
4. **User Interface Design.** This section contains the screenshots of the application with some comments to give to the reader a general overview of the user interfaces.
5. **Frameworks, External services and Libraries.** This section aims to explain the main frameworks, external services and libraries used pointing out their advantages.
6. **Development and testing.** This section identifies the development and testing approaches. their results.

7. **References** A summary of worked time and cost estimation of the work.

At the end, there is an Appendix where software and tools used are reported.

Chapter 2

# General Overview

## 2.1 Concept

*Baddy* is a cross-platform application that allows people to find a caregiver for the elders. The application requires both end users and caregivers to authenticate with their email, filling a customized form based on the role of the user. Those data are mandatory in order to use the application. The main components of the project are:

- A client that queries the database to fetch data of users, but also send data to it, like pictures or new user data.
- A server that listens to requests and links the client to the database with all the needed logic, like the authentication part for example.

## 2.2 Core Features

This section details a list of the core functionalities of the application. The features are divided by the screen where they are present, allowing the reader to easily understand them and indicating where they could be found in the application.

### 2.2.1 Splash Screen

This is a welcome page screen that pops up only for the very first time a new user opens the application. It briefly describes all the functionalities contained in the app.

### 2.2.2 Authentication Screens

- Allows the user to authenticate using his/her email and password.
- Allows the user to register his/her-self in the application.

### 2.2.3 Homepage

- Allows the user to search for a caregiver, by specifying the city.

- Shows all the information of the caregivers, including name, city, price/h and average rating
- Allows to navigate to the profile page if logged in as caregiver
- Allows to log out the user.

### 2.2.4 Update Profile Screen

- Allows the caregiver to change his personal information, such as birth, gender, nationality, city, availability, name, surname, phone number.
- Allows the caregiver to update his profile photo.

### 2.2.5 Profile Screen

- Allows users to check for specific information about a caregiver, such as birth, gender, nationality, city, availability, name, surname, phone number.
- Allows users to check for reviews written by other users about the specific caregiver.
- Allows users to write a review about the specific caregiver.

## 2.3 Functional Requirements

### 2.3.1 General Requirements

- The app can be used by everyone who needs to find a caregiver, and also all caregivers can use tha application in order to find new clients.
- The app needs user auth, so it has also to provide a registration form.
- The app needs internet connection in order to communicate with the server.
- The server must accept requests from the client app.

### 2.3.2 Splash Requirements

- Splash screen should be visible only the first time the user opens the application.
- Splash screen should contain a text followed by an image that explains the purpose of the app.
- A swipe left/right should be present in order to switch between 3 different splash screens.

### 2.3.3 Authentication Requirements

- Login screen should be accessible only to users not currently logged in.
- Authentication screens should allow users to register or login using email and password. If the user wants to sign up as a caregiver, additional information is required: phone number and city.
- Authentication screens should redirect the user to the Homepage if the authentication is successful.

### 2.3.4 Homepage Requirements

- *The homepage* should contain a search box for the city, and a list of caregivers.
- Each list tile should contain name, surname, city, photo, price/h, and average rating of the caregiver.
- As soon as the user type something in the search box, list tiles that do not match the current string should disappear.
- Whenever the user taps on the list tile, the *Profile Screen* should appear.
- The app bar should contain the name of the logged user plus 2 action buttons, the first to update the profile (only if logged in as caregiver, otherwise is hidden), and the second to logout.

### 2.3.5 Update Profile Requirements

- *Update Profile Page* should be visible only by the caregiver.
- *Update Profile Page* should allow the caregiver to update his information, so that all the other users can see the changes.
- The date should be inserted in the right format.
- Gender can be chosen between 3 options only: male, female, transgender.
- The caregiver must be adult in order to use the application, so the birthdate must be compliant with that.
- The caregiver must be able to update his photo, either by using the camera or the gallery.

### 2.3.6 Profile Requirements

- *The profile* should display the photo of the selected caregiver.
- *The profile* should display all the information that can be inserted in the *Update Profile Screen.*
- A bottom bar must be visible in order to swap tabs, the first tab contains information about the caregiver, the second tab contains a list of reviews written for the current caregiver, and the third one contains a form to write the review with the possibility to choose how many stars give to the caregiver.
- Caregiver are not allowed to write reviews.

## 2.4 Non-Functional Requirements

These are the non-functional requirements that specify criteria that can be used to judge the operations of the application.

- **Availability** the services must be always up and running. In case of failure it must be restored as soon as possible.
- **Extensibility** the application was developed trying to keep a simple structure in order to allow further extensions easily.
- **Maintainability** the code must be clear, readable and with explicative comments to allow future maintenance.
- **Eye catching UI** nowadays is very important to have a cleaned and simple design. The application was developed following the Apple Human Interface Guidelines.
- **Portability** cross-platform implementation allows to use the application both in Android and iOS systems;
- **Reliability** all the data must be trusted, and they could not be modified by anyone. The remote database must be protected and the connection between client and server must be handled with an HTTPS protocol.
- **Scalability** the system (considering client and server) should always be available to be used. System failures and server-side crashes must be avoided;
- **Usability** the application was developed to make the user interface as simple as possible keeping all the functionalities needed to provide the best user experience.

Chapter 3

# Architectural Design

## 3.1 Overview

From an external point of view, users using their smartphones exploit the different services offered by *Baddy* that are different for clients and caregivers. The application to be developed is based on one application component (application logic) which manages the interactions between different components of the system. Application logic is present both in the mobile view and in the back-end logic. In the mobile view there is the management of the GUI and the connection with other useful components present in the smartphone like GPS and camera; in the back-end logic instead there is the most of the logic and also the interfaces to external services: Database provider, Storage provider and Push notification provider.

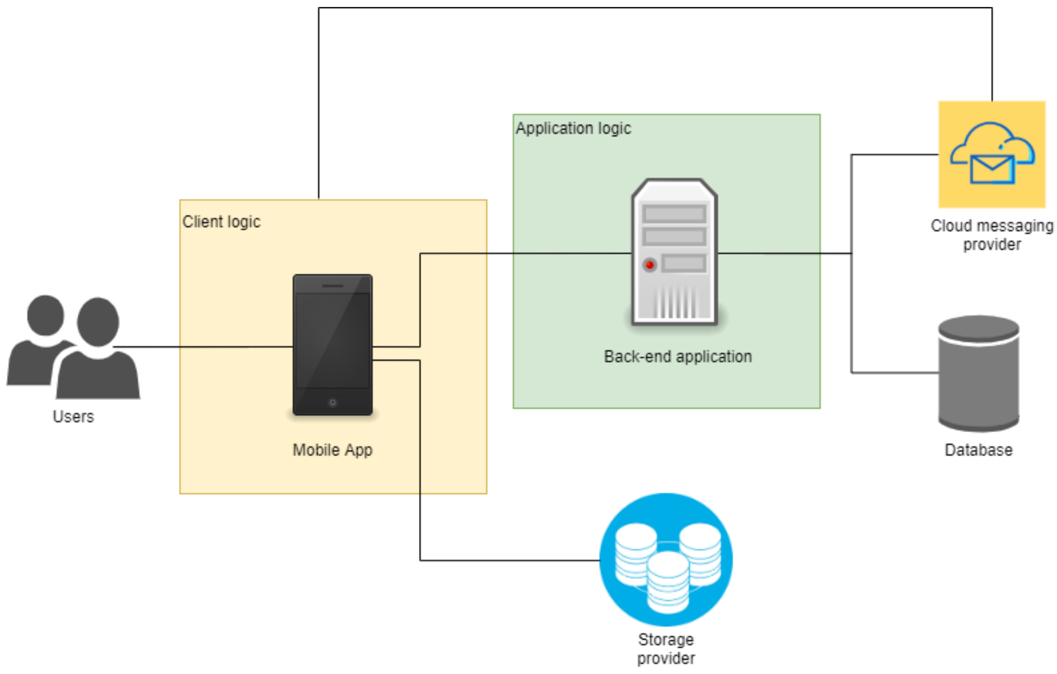


Figure 3.1: Overview of the application components

## 3.2 Component View

In this section the system will be described in terms of its components: their functionalities will be discussed and detailed. Moreover, interfaces among components and among external systems will be shown.

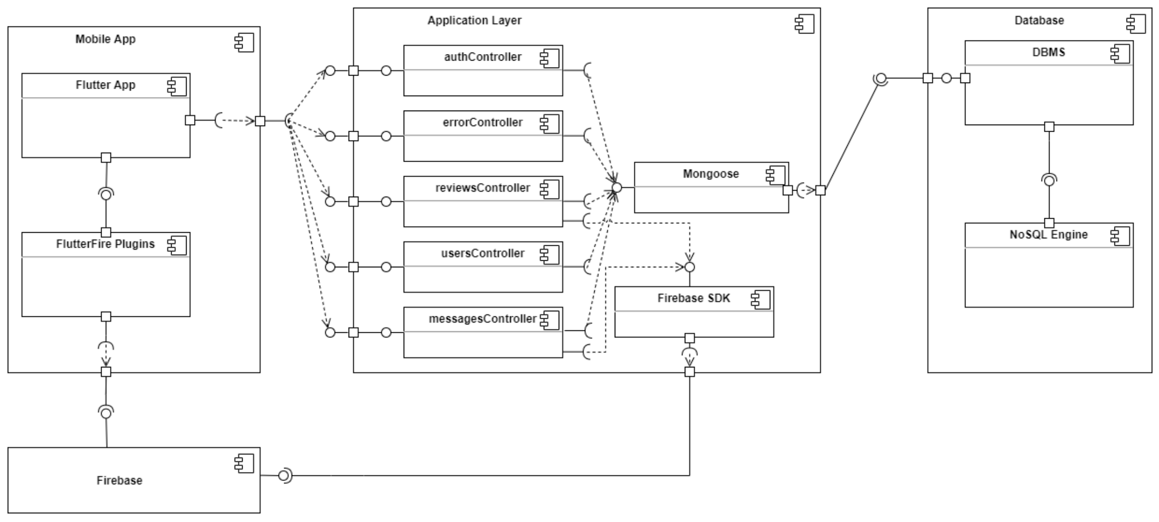


Figure 3.2: Component diagram

### 3.2.1 Database

The application database is managed using a Non-Relational DBMS. It allows the reading of data, ensuring to the users the possibility to log in and check the stored data. It is also used for data manipulation (insertion, modification and deletion). The database offers to the Application Server an interface that it can use to interact with it. Particular attention must be paid to the encryption of passwords used to the user access to the system.

### 3.2.2 Application Server

The main feature of the Application Server is to define rules and work-flows of all the functionalities defined by the RestfulAPI. The Application Server must have interfaces to communicate with the Mobile Application and also with the DBMS; the communication must be done using the HTTPS protocol. In the brief introduction below, logic modules and their descriptions are presented, while all the connections among them can be seen in the Global Component View.

- **authController:** This module handles the authentication and authorization process. It generates authorization tokens and checks their validity.
- **errorController:** It manages all the routes that are not available and all the bad formatted requests.
- **handlerFactory:** General service that implements all CRUD operations.
- **reviewsController:** It manages all the newly created reviews with all the rankings.
- **usersController:** It manages users operations, like insertion of a new offer and updating profile.
- **messagesController:** It manages all the messages sent by clients to caregivers' profiles

### 3.2.3 Mobile Application

The mobile application communicates with the Application Server through RestfulAPIs that are defined in order to describe the interactions between the two layers and that must be independent of the two implementations. More details about the implementation will be provided in the following sections.

## 3.3 Deployment View

In this section a deployment diagram details the execution architecture of the system, including nodes such as hardware or software execution environments, and the middleware connecting them.

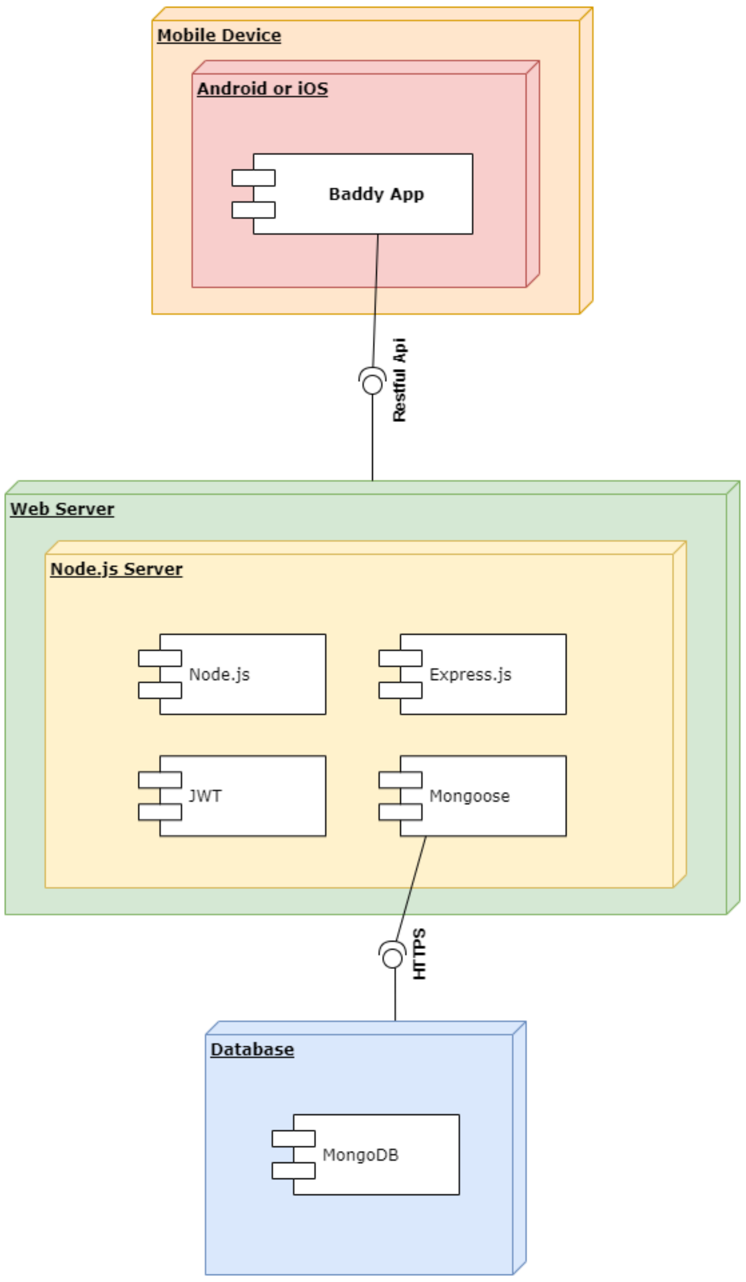


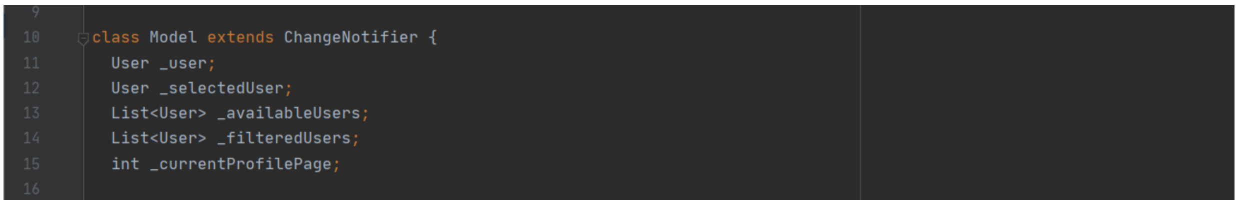
Figure 3.3: Deployment diagram

## 3.4 Selected architectural styles and Patterns

This section details the architectural styles and patterns used to design the system architecture. In the following sections, the different patterns will be explained in terms of styles, principles and why we chose them.

### 3.4.1 State Management with Provider

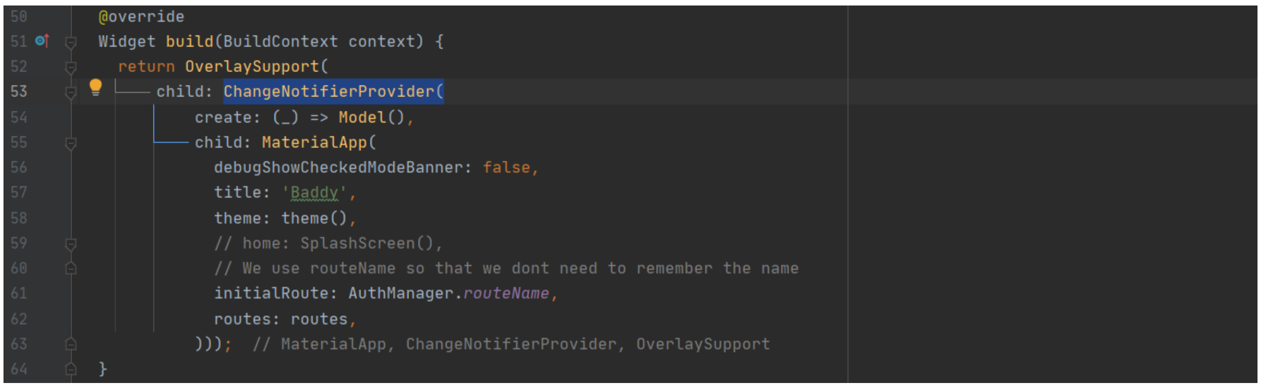
Managing State using setState() starts becoming heavy as the code grows, because whenever you need to change the widget's UI, you have to call setState() inside the changing widget, so that it gets rebuilt, and since application is composed of hundreds of different widgets, there could be hundred different points where you have to take care of calling setState(), and managing state. Thus, using this raw technique to manage state is not a good option, but thankfully we have a better approach, which is not just easy but effective, called Provider State Management. The Provider approach consists in transforming the model of the application in a notifier by simply extending the class **ChangeNotifier** (this is one way to encapsulate the application state and the one we are currently using).



```
class Model extends ChangeNotifier {
  User _user;
  User _selectedUser;
  List<User> _availableUsers;
  List<User> _filteredUsers;
  int _currentProfilePage;
```

Figure 3.4: Change notifier usage example

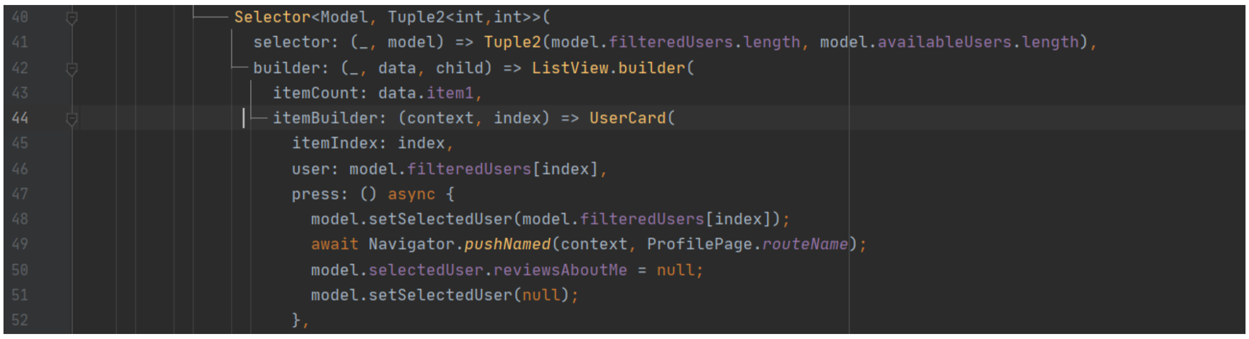
Then with **ChangeNotifierProvider** widget we provide an instance of a ChangeNotifier to its descendants. We should put ChangeNotifierProvider above the widgets that need to access it.



```
@override
Widget build(BuildContext context) {
  return OverlaySupport(
    child: ChangeNotifierProvider(
      create: (_) => Model(),
      child: MaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Baddy',
        theme: theme(),
        // home: SplashScreen(),
        // We use routeName so that we dont need to remember the name
        initialRoute: AuthManager.routeName,
        routes: routes,
      )));  // MaterialApp, ChangeNotifierProvider, OverlaySupport
}
```

Figure 3.5: Change notifier provider usage example

At this point to access the model is enough to extends the **Consumer** class, which allows us to retrieve the model data by its type. However the Consumer approach could be not optimal because every change in the model cause the rebuild of all the Consumer widgets. This is the main reason we used the **Selector** class, a component that can filter updates by selecting a limited amount of values and prevent rebuild if they don't change.



```
Selector<Model, Tuple2<int,int>>(
  selector: (_, model) => Tuple2(model.filteredUsers.length, model.availableUsers.length),
  builder: (_, data, child) => ListView.builder(
    itemCount: data.item1,
    itemBuilder: (context, index) => UserCard(
      itemIndex: index,
      user: model.filteredUsers[index],
      press: () async {
        model.setSelectedUser(model.filteredUsers[index]);
        await Navigator.pushNamed(context, ProfilePage.routeName);
        model.selectedUser.reviewsAboutMe = null;
        model.setSelectedUser(null);
      },
```

Figure 3.6: Use of Selector example

Thanks to provider, we can access the model easily with a single line of code, without the need of passing the instance down to the tree of the widgets that would make the code difficult to understand and maintain.

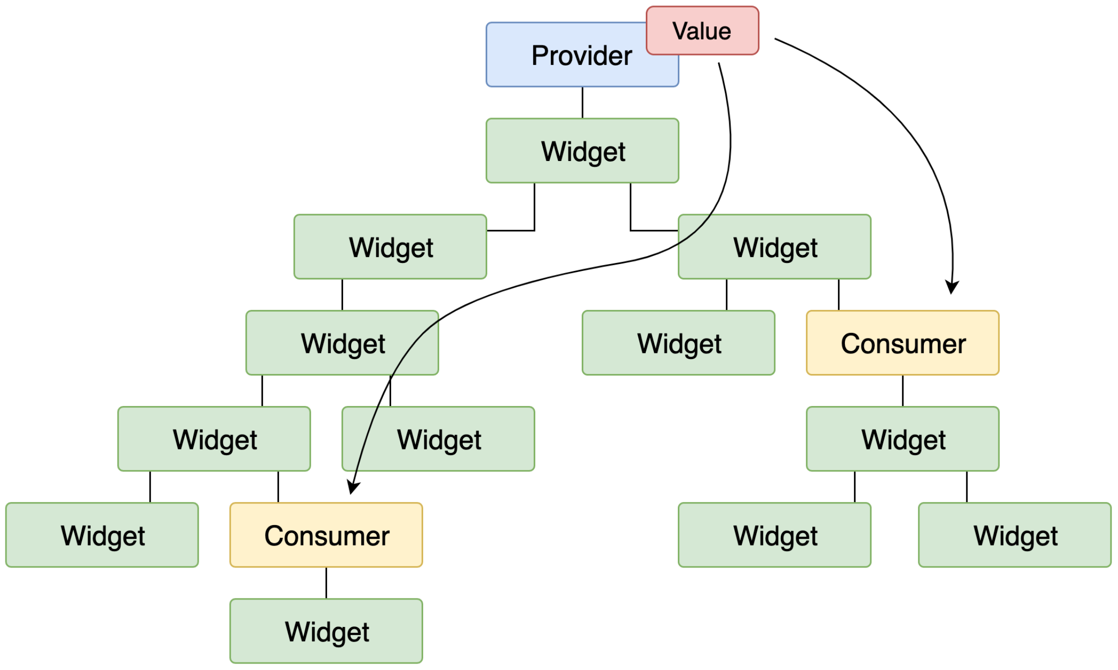


Figure 3.7: Provider components in widget tree

### 3.4.2 Representational state transfer

REST is a software architectural style that defines a set of constraints to be used for creating Web services. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. In order for an API to be considered RESTful, it has to conform to these main criteria:

- A client-server architecture made up of with requests managed through HTTP.
- Stateless client-server communication, i.e. each request is separate and unconnected.
- A uniform interface between components so that information is transferred in a standard form.

Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. This improves the independence of the various tiers of the system. This pattern is used to develop APIs for communications between the application server with the mobile client and the application server.

### 3.4.3 Multi-tier architecture

In the Multi-tier Architecture pattern, the components of the system are placed on different physical layers, which are not isolated, but they communicate through interfaces with at least another layer of the system. The main benefits of a multi-tier architecture are:

- different tiers of an application gives development teams the ability to develop and enhance a product with higher speed than developing a unique code base because a specific layer can be upgraded with minimal impact on the other layers;
- scalability is another great benefit of a multi-tier architecture. By distributing the different layers you can scale each autonomously depending on the need at any given time;
- having different layers can also increase the system's reliability and availability by hosting different parts of your application on different servers.

### 3.4.4 Thin client

A thin client is a lightweight component that has been optimized for establishing a remote connection with a server-based computing environment. A thin client is an advantage because all the application logic is allocated on the application server, which has enough computational power to manage concurrency efficiently.

### 3.4.5 Other design decisions

### 3.4.6 Security

All the information regarding users are sensitive. The system must prevent any attack that could steal data or make the service unavailable. The data exchanged will be encrypted by the mean of the HTTPs protocol that relies on SSL/TLS. The system will never expose any sensible data with external actors without the user's consent.

Chapter 4

# User Interface Design

## 4.1 Overview

In this section are present some screenshots of *Baddy* mobile application. During the design and development, the attention was focused on the mobile application, even if the application is developed to adapt itself to different screen sizes.

## 4.2 Screens

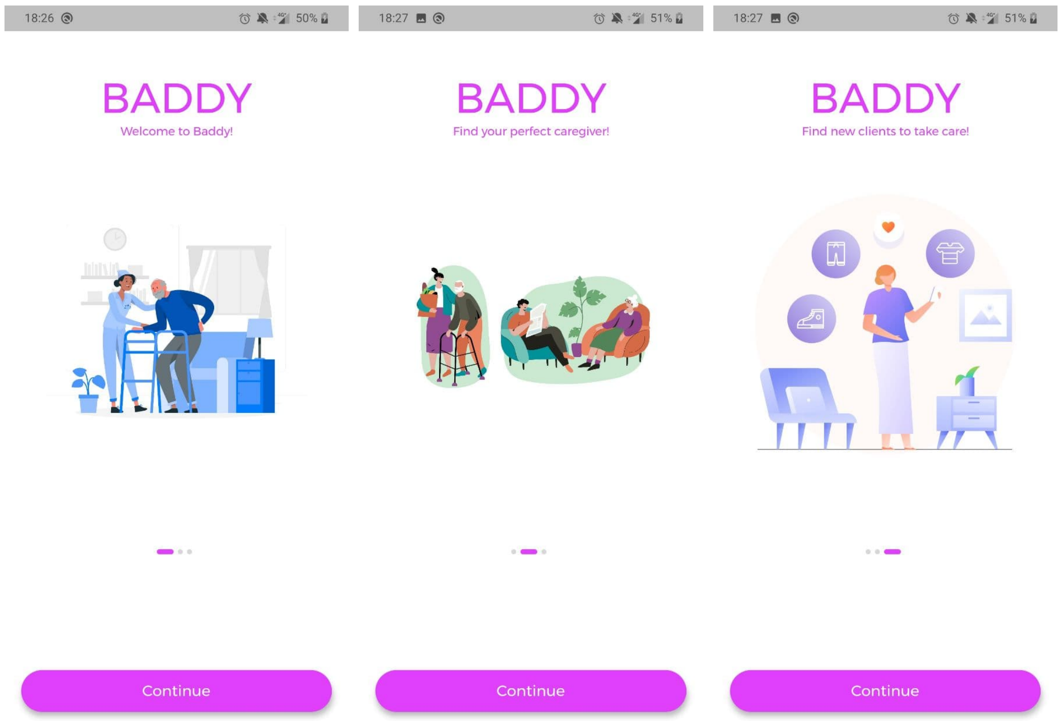


Figure 4.1: **Splash Screen**

The **Splash Screen** is the first screen that will show up if you open the app for the first time.

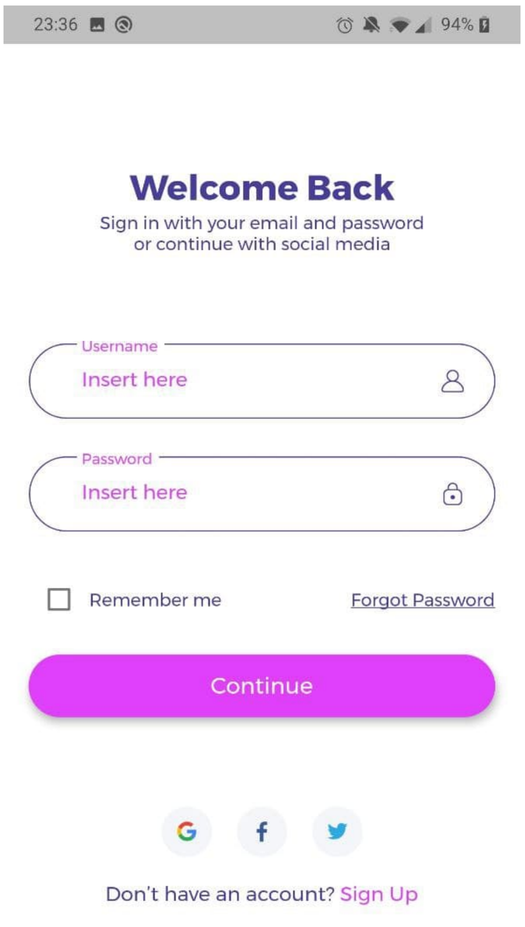

23:36
94%
Welcome Back
Sign in with your email and password
or continue with social media
Username
Insert here
Password
Insert here
Remember me
Forgot Password
Continue
Don't have an account? Sign Up

Figure 4.2: **Login Screen**

The **Login Screen** is the first screen that will show up when the application is loaded if not already logged in.

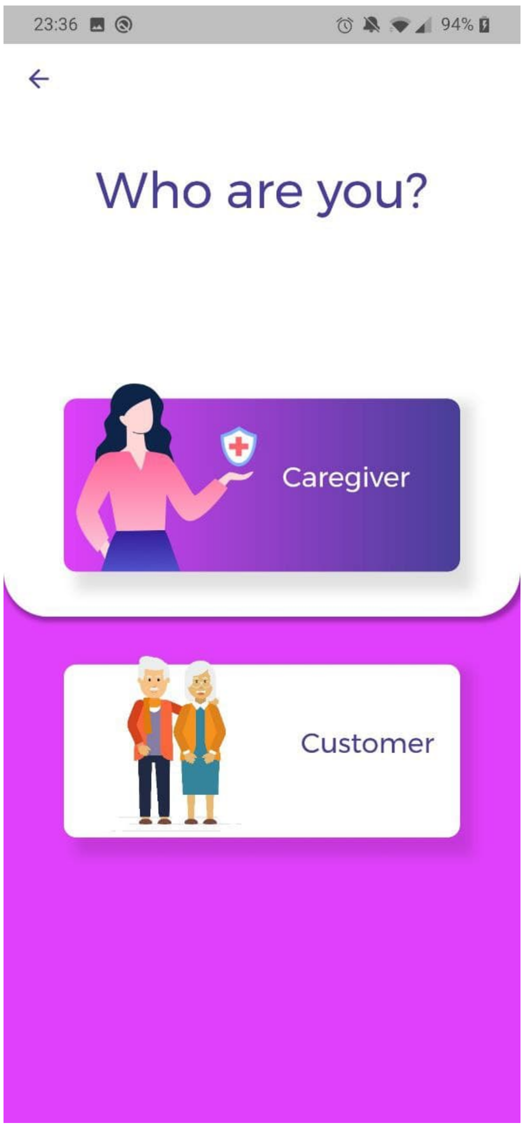


Figure 4.3: **Choose Role Screen**

The **Choose Role Screen** shows up as soon as the user taps on ‘sign up‘ in the *Login Screen*. Here the user must choose between standard user and caregiver.

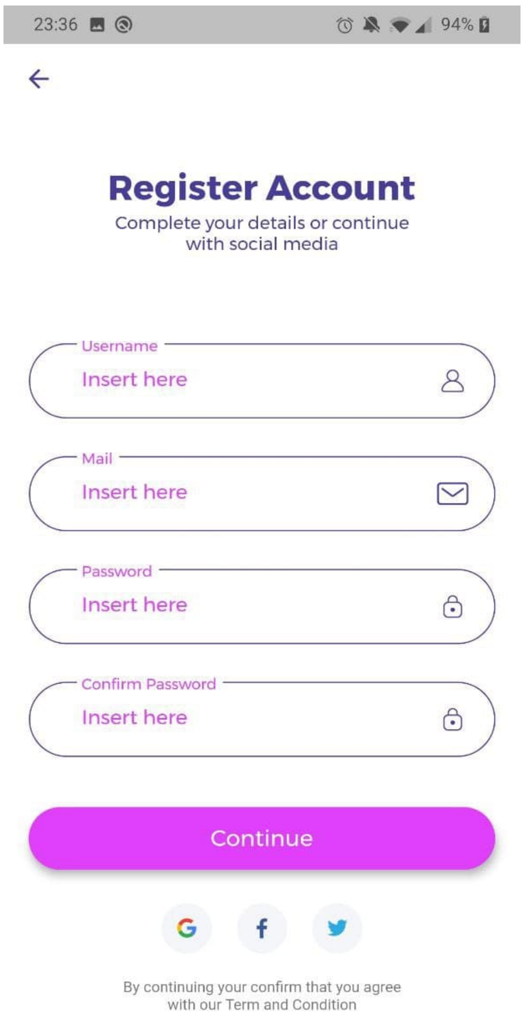
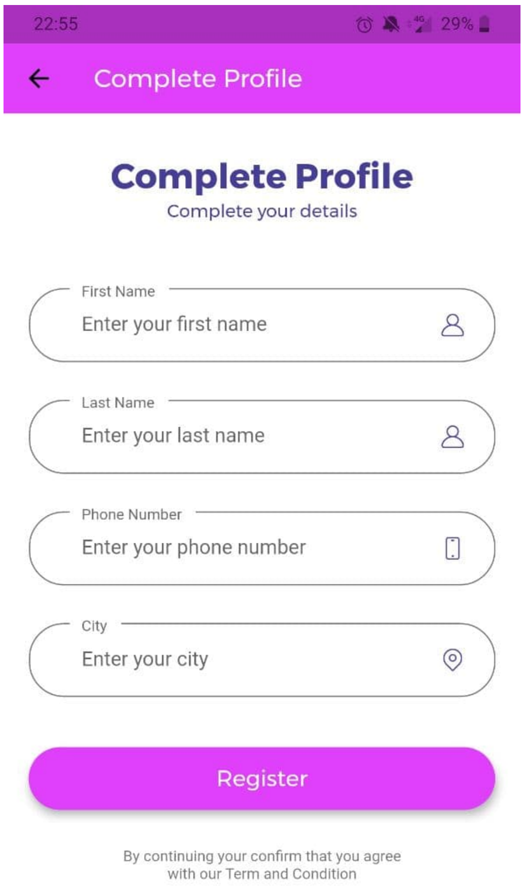

23:36
94%
Register Account
Complete your details or continue with social media
Username
Insert here
Mail
Insert here
Password
Insert here
Confirm Password
Insert here
Continue
By continuing your confirm that you agree with our Term and Condition

Figure 4.4: **Sign Up Screen**

The **Sign Up Screen** appears after choosing the role. Here the user must insert mandatory data in order to proceed. This step is the same for both standard user and caregivers.



Complete Profile

Complete your details

First Name: Enter your first name

Last Name: Enter your last name

Phone Number: Enter your phone number

City: Enter your city

By continuing your confirm that you agree with our Term and Condition

Figure 4.5: **Complete Registration Screen**

The **Complete Registration Screen** shows up only if the chosen role is the caregiver. Here more mandatory info are required in order to complete the registration.

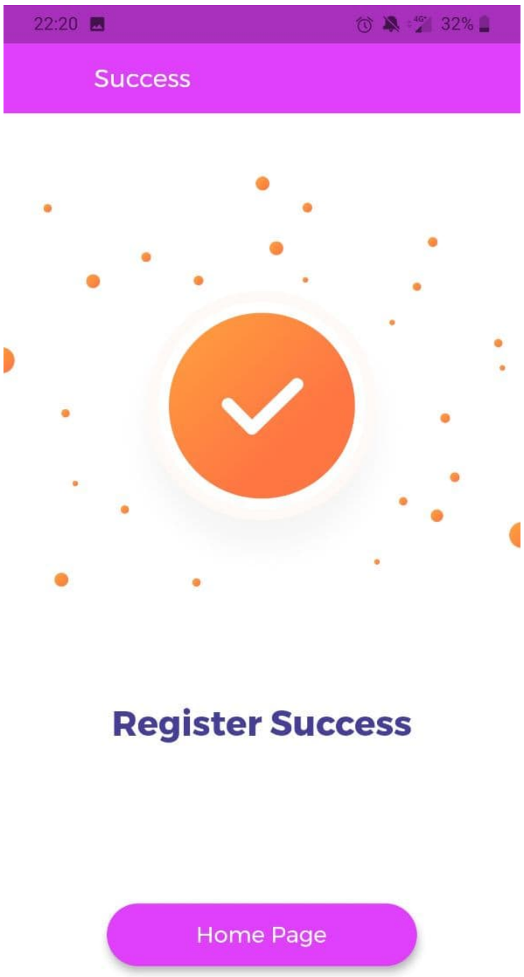


Figure 4.6: **Success Screen**

The **Success Screen** appears if all the steps for the registration are done correctly. From here you can easily access the homepage by clicking the button.

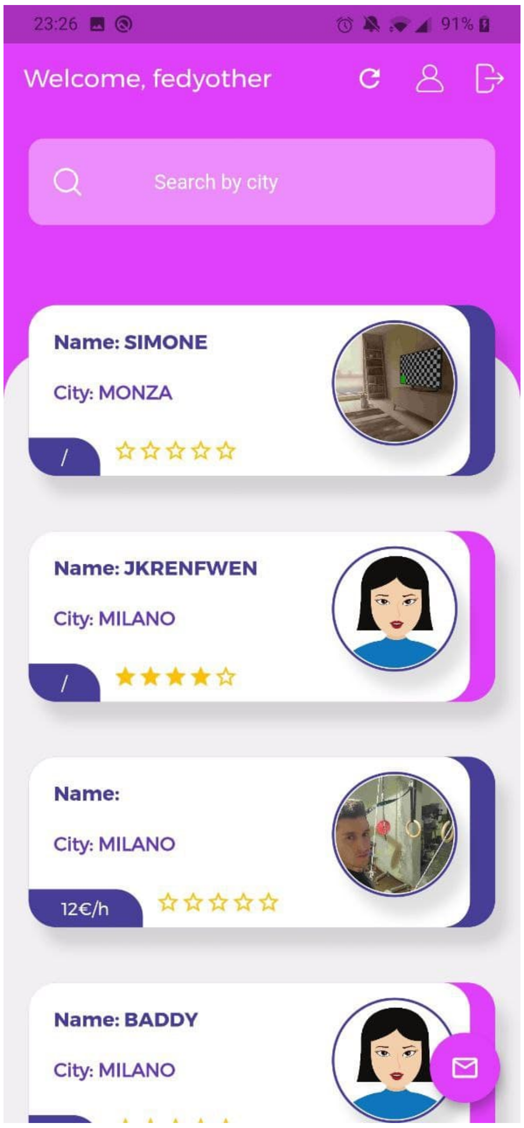


Figure 4.7: **Homepage**

The **Homepage** appears only if the user is correctly authenticated. Here a list of available caregivers is shown with all the relevant information. You can also search for a specific city.

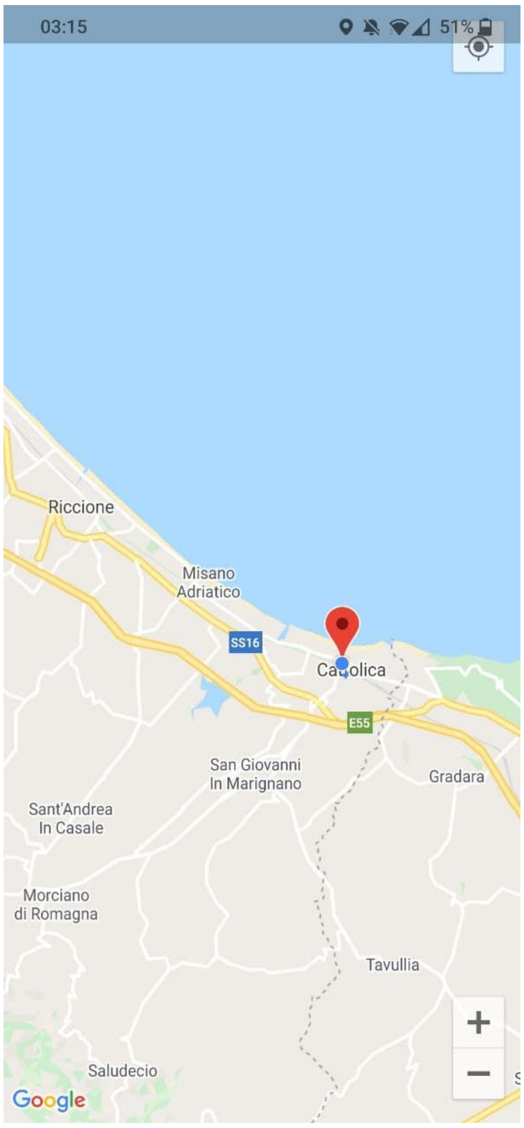
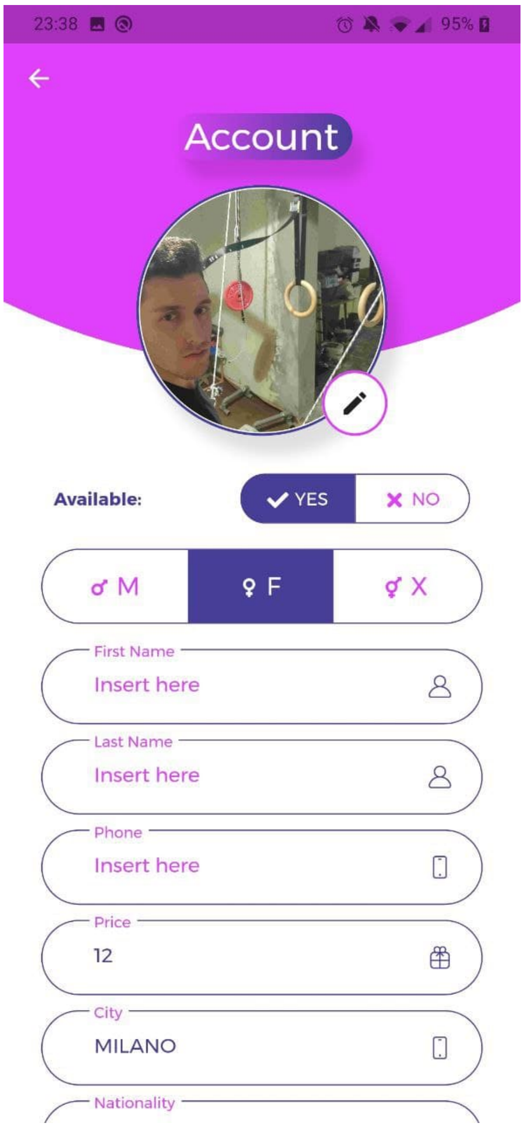


Figure 4.8: **Map Screen**

The **Map Screen** shows all the markers of available caregivers near the user.



23:38 95%

Account

Available: [✓ YES] NO

M [✓ F] X

First Name: Insert here

Last Name: Insert here

Phone: Insert here

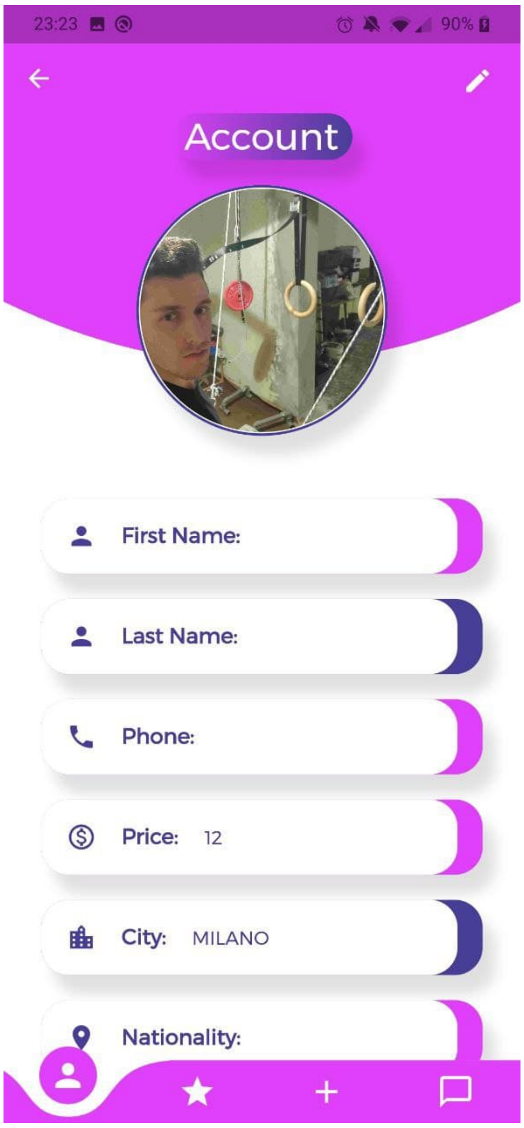
Price: 12

City: MILANO

Nationality

Figure 4.9: **Update Profile Screen**

The **Update Profile Screen** is accessible only by caregivers, using the top right icon. Here the caregiver can update all the information he wants to be visible from others.


23:23 90%

Account

First Name:

Last Name:

Phone:

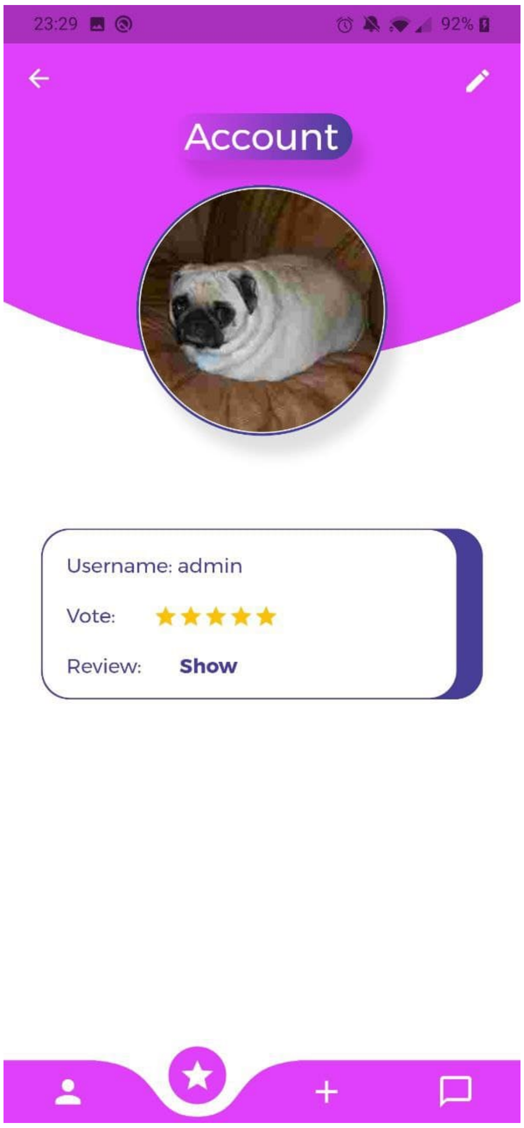
Price: 12

City: MILANO

Nationality:

Figure 4.10: **Profile Screen**

The **Profile Screen** shows all the information of the selected user from the homepage.



Username: admin

Vote: ★★★★★

Review:

Figure 4.11: **Reviews Screen**

The **Reviews Screen** shows all the reviews a user got from other users.

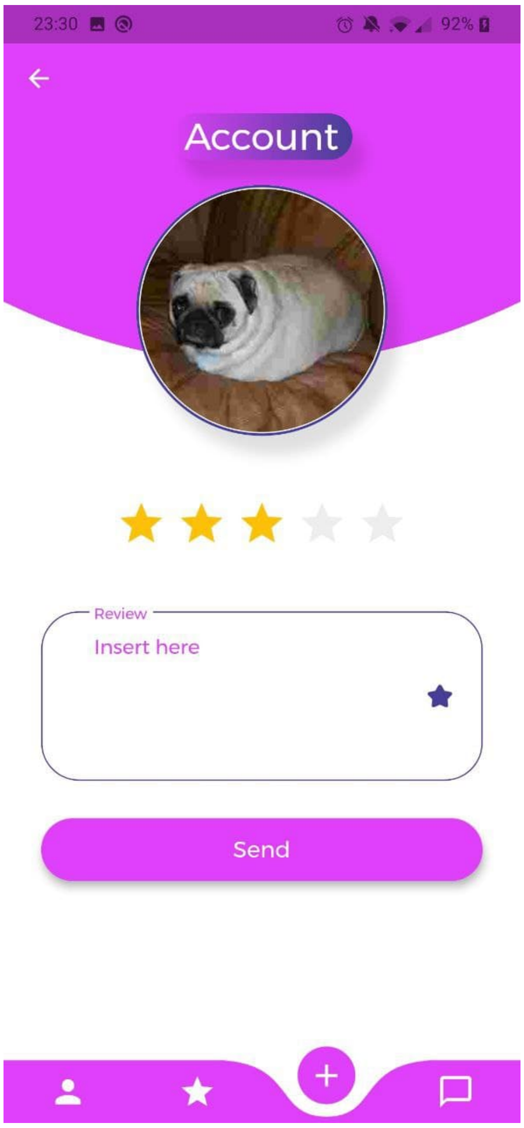


Figure 4.12: **Write Review Screen**

The **Write Review Screen** allows only the standard user to write a review and give stars to the selected caregiver.

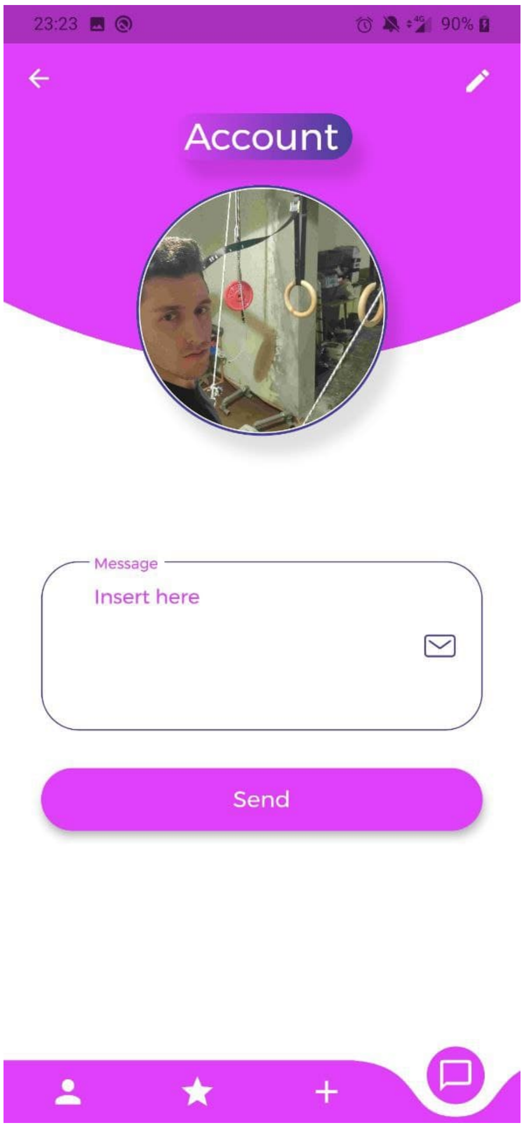
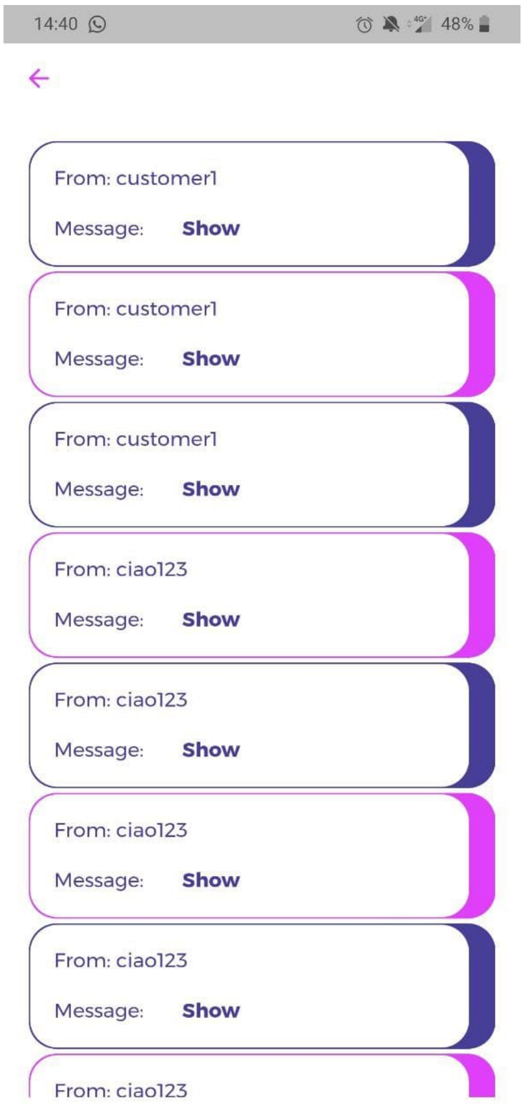


Figure 4.13: **Write Message Screen**

The **Write Message Screen** allows only the standard user to write a private message to the caregiver.


14:40 48%

From: customer1
Message: **Show**

From: customer1
Message: **Show**

From: customer1
Message: **Show**

From: ciao123
Message: **Show**

From: ciao123
Message: **Show**

From: ciao123
Message: **Show**

From: ciao123
Message: **Show**

From: ciao123

Figure 4.14: **Messages List**

The **Messages List** allows only the logged caregiver to see all the inbox messages received by the standard users.

Chapter 5

# Frameworks, External services and Libraries

## 5.1 Framework

Figure 5.1: Technological stack

### 5.1.1 Flutter

Baddy mobile application was developed using **Flutter**, an open-source development kit created by Google used to develop applications for Android, iOS, Linux, Mac, Windows, Google from a single codebase. The main reasons of this choice are:

- **Portability**: with a single code base is possible to create a product that would fit both iOS and Android devices. Furthermore, it is possible to use it to create web and desktop apps, but these features are still under testing;
- **Rich in components**: Flutter has a lot of UI components that come together with the framework, without the need for installing a lot of external libraries to design interfaces;
- **High performance**: Flutter has very high performances thanks to how it works under the hood. It never uses web views or bridge to render the UI elements, but it uses his rendering engine, which is built mostly in C++.

### 5.1.2 Node.js

The server side application relies on **Node.js**, an environment for server-side scripting based on JavaScript. It's a light, scalable, and cross-platform way to execute code. It uses an event-driven I/O model which makes it extremely efficient and makes scalable network application possible. The most important advantages we took into account for this choice are:

- code executes faster than in any other language;
- the ever-growing NPM (Node Package Manager) gives developers multiple tools and modules to use, thus further boosting their productivity;
- the single-threaded, event-driven architecture of Node.js allows it to handle multiple simultaneous connections efficiently;

### 5.1.3 MongoDB

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. *Baddy* relies on it in order to store all the information about users, reviews and messages.

### 5.1.4 Firebase

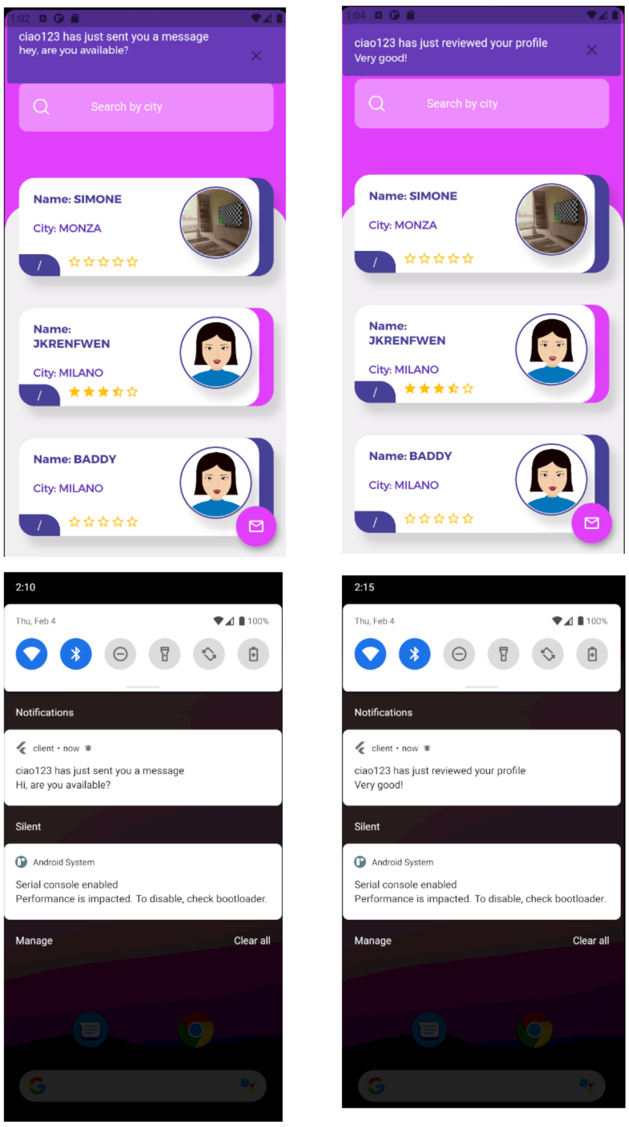
Some particular features and functionalities are based instead on external service providers, in particular **Firebase Cloud Messaging** was used to send notifications to users' smartphones and **Firebase Cloud Storage** to upload and easily retrieve users' profile image. Firebase lets the developer to focus on developing the best app possible for the business by simplifying a lot of operations. Since the operations and internal functions are so solid, and taken care of by the Firebase interface, there is more time available for developing high quality app that users are going to want to use.

## 5.2 External Services

As already mentioned, some features of *Baddy* have been implemented with the help of a series of external services. This choice comes from some simple considerations that take into account that each provider offers a consolidated platform supported by a detailed documentation and a large community of developers. Here are reported the external services that the application use for some of its functionalities.

### 5.2.1 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a platform that provides a reliable and efficient connection between servers and devices that allows to deliver and receive messages and notifications on iOS, Android, and the web at no cost. As part of the application it is used to send notifications to a caregiver's account when someone writes a review about him/her or a customer sends a message of interest after viewing his/her profile.


ciao123 has just sent you a message
hey, are you available?

ciao123 has just reviewed your profile
Very good!

ciao123 has just sent you a message
Hi, are you available?

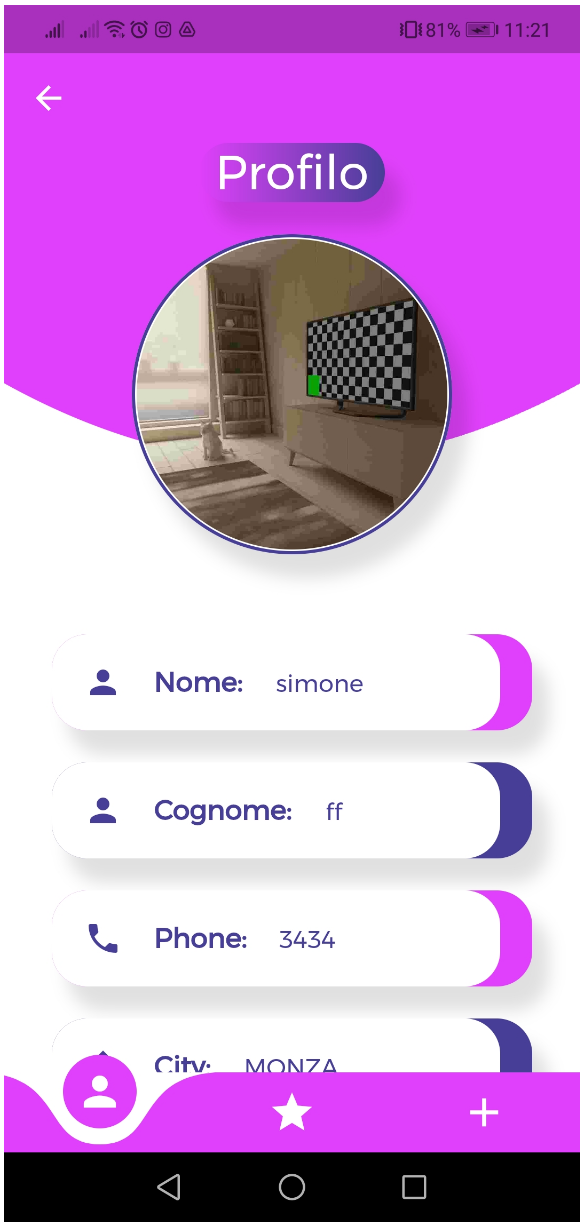
ciao123 has just reviewed your profile
Very good!

Serial console enabled
Performance is impacted. To disable, check bootloader.

Figure 5.2: Background and Foreground notification of a new review

### 5.2.2 Firebase Cloud Storage

Firebase Cloud Storage gives the possibility to upload and share user generated content, such as images and video, in order to build in the easiest way media content for applications. The Cloud storage is available also for Flutter; that means that files can be uploaded directly from mobile devices and web browsers. In the context of the application the service is used to upload the profile image in order to display it whenever is required.



Nome: simone

Cognome: ff

Phone: 3434

City: MONZA

Figure 5.3: Profile image previously uploaded by the user via Firebase Cloud Storage

## 5.3 Libraries

For the development of the application the majority of the libraries used are standard packages of Flutter. There are however some components that use platform SDKs to interact with external services mentioned above.

### 5.3.1 Firebase Messaging SDK

This client side component manages all the events that come with a push notification, allowing to define the behaviour of the application according to its state when a new notification is received. The fragment of code below shows how the SDK is used to define the actions to take when the app is in foreground.

```
...
import 'package:firebase_core/firebase_core.dart';
import 'package:firebase_messaging/firebase_messaging.dart';
...

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();

  FirebaseMessaging.onMessage.listen((RemoteMessage message) {
    print('Got␣a␣message␣whilst␣in␣the␣foreground!');
    print('Message␣data:␣${message.notification.body}');

    showOverlayNotification((context) {
      return Card(
        color: Colors.deepPurple,
        margin: const EdgeInsets.symmetric(horizontal: 4),
        child: SafeArea(
          child: ListTile(
            title: Text(message.notification.title,
                style: TextStyle(
                  color: Colors.white,
                )),
            subtitle: Text(message.notification.body,
                style: TextStyle(
                  color: Colors.white,
                )),
                 trailing : IconButton(
                      icon: Icon(Icons. close ),
                      onPressed: () {
                        OverlaySupportEntry.of(context).dismiss();
                      }),
              ),
           ),
         );
      }, duration: Duration(milliseconds: 4000));
   });

   runApp(MyApp());
}

...
```

### 5.3.2 Firebase Storage SDK

In this case the SDK of Firebase is use to upload the profile image of the user and to get its reference in order to display it by simply referring to a URL. In only three lines of code we have done all the stuffs.

```
...
import 'package:firebase_storage/firebase_storage.dart';
...

class Apis {
  ...
  static Future<String> uploadImage(
       {@required PickedFile image,
       @required String username,
       @required int timestamp,
       @required String jwt}) async {
     try {
       File imageFile = File(image.path);
       String path = _createPath(username, timestamp.toString());
       final storageRef = FirebaseStorage.instance.ref().child(path);
       await storageRef.putFile(imageFile);
```

```
        final url = await storageRef.getDownloadURL();
        var dio = Dio();
        await dio.patch(URL + usersRoute + "/updateDetails",
            data: {"photo": url},
            options: Options(headers: {'Authorization': 'Bearer␣$jwt'}));
        return url;
      } on DioError catch (e) {
        print(e.response);
        return "default.jpg";
      } catch (e) {
        print(e);
        return "default.jpg";
      }
    }

    static String _createPath(String username, String timestamp) {
      //mail/timestampRecord/imagesList
      return "$username/$timestamp/" + "photo.jpg";
    }
    ...
}
```

### 5.3.3 Other dependencies

In the development process of the application, we also employed the following external libraries of Flutter:

- **dio[3.0.10]**: a powerful Http client for Dart, which supports Interceptors, Global configuration and other useful features used to invoke the REST service of the server-side components;
- **flutter_secure_storage[3.3.5]**: a Flutter plugin to store data in secure storage, used to store the jwt and other sensitive information;
- **cupertino_icons[1.0.0]**: asset repo containing the default set of icon assets used by Flutter's Cupertino widgets;
- **provider[4.3.2+2]**: a wrapper around InheritedWidget to make them easier to use and more reusable, used for the **state management**;

- **flutter_spinkit[4.1.2+1]**: a collection of loading indicators animated with Flutter;
- **image_picker[0.6.7+12]**: a Flutter plugin for picking images from the image library, and taking new pictures with the camera, used to upload the profile image;
- **awesome_dialog[1.2.1]**: a Flutter package project for simple and awesome dialogs;
- **flushbar[1.10.4]**: a flexible widget for user notification that allows to customize text, button, duration and animations;
- **cached_network_image[2.3.3]**: a flutter library to show images from the internet and keep them in the cache directory, used for the profile image;
- **argon_buttons_flutter[1.0.6]**: package used to create the buttons of the application;
- **liquid_swipe[1.5.0]**: a Flutter plugin to implement liquid Swipe effect to provided containers, used for the onboarding page;
- **google_fonts[1.1.1]**: with google_fonts package, .ttf or .otf files do not need to be stored in your assets. Instead, they can be fetched once via http at runtime, and cached in the app's file system;
- **curved_navigation_bar[0.3.4]**: package used for the navigation bar;
- **url_launcher[5.7.10]**: used to open the dial when tapping on profile's contact details;
- **toggle_switch[0.1.8]**: used to render toggle switch widget;
- **font_awesome_flutter[8.10.0]**: icon pack available as set of Flutter Icons;
- **overlay_support[1.0.5]**: support for overlay, easy to build toast and internal notification, used to render foreground notifications.

Chapter 6

# Development and testing

## 6.1 Overview

The purpose of this chapter is to detail the approach and the techniques used during the development process of the application. Given the nature of the application, it is important to clarify the plan that guided all the phases of the software development in order to achieve the final result.

## 6.2 Development approach

The development of the application has been carried out following a **test-driven** approach. In particular the components have been developed in parallel to a series of **widget tests** written with the purpose of matching the developed components with the expected behaviour. The order of the components has been established considering a **bottom-up** development plan, which starts from simple and atomic components used to model more complex widgets and views. In particular basic widgets like buttons, boxes, forms and other frequently used components have been developed first and immediately **unit-tested**.

## 6.3 Testing

To implement the system, we decided to test the application mainly through debug on both emulators and real devices. The only exception is the **mobile application**, for which we also performed unit testing on the main UI components and subsequent integration testing to verify the compatibility among them. In particular separable widgets used by multiple components of the application have been tested using a particular type of test supported by Flutter, the widget tests. We then developed a special integration test suite using the Flutter driver to verify the correct behavior of the application as a whole.

### 6.3.1 Widget tests

Widget tests are simply **unit-tests** performed in an environment that allows to test the behaviour of a widget class. The goal of this approach is to verify that the tested widget's

**UI** looks and interacts as expected. This operations involves multiple classes and requires a test environment able to support the entire widget lifecycle. In the specific context of the application, the tests carried out verified the behavior of the widget by injecting an input and verifying the presence of the expected elements through the finder. A **finder** is a class provided by the test environment used to locate widgets and elements inside the pumped widget. If the finder is able to locate the expected element with the expected features and properties the test is passed.

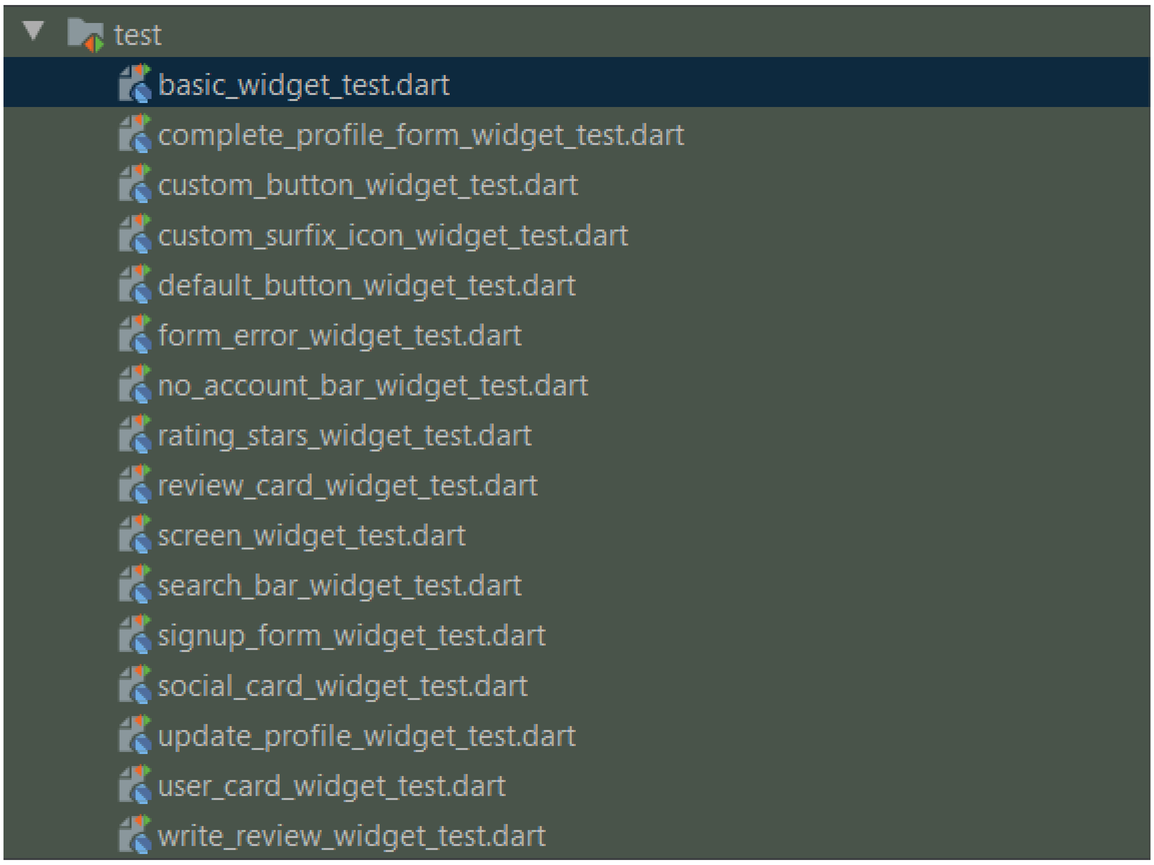
**Case study: Rating stars**

This example is provided with the purpose of clarifying the general structure of one of the developed widget tests.

The tested widget is a bar used to visualize the average rate of the reviews left to a caregiver account. This is done by the mean of series of star icons representing the average value of the user profile. In this case we are verifying if the icons representing the stars are consistent with the numeric value of the average rate. Below the code of the test.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:polimi_app/components/rating_stars.dart';

void main() {
  testWidgets('Rating stars widget test', (WidgetTester tester) async {
    //Matching parameters
    final threeStars = 3.0;
    final threeStarsAndEpsilon = 3.1;
    final threeAndHalfStars = 3.5;

    //Test three stars rendering
    await tester.pumpWidget(buildTestableWidget(RatingStars(rate: threeStars)));
    expect(
        find.byWidgetPredicate(
            (widget) =>
                widget is Row &&
                widget.children is List<Widget> &&
                ((widget.children.elementAt(0) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(1) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(2) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(3) as Icon).icon ==
                        Icons.star_border &&
                    (widget.children.elementAt(4) as Icon).icon ==
                        Icons.star_border),
            description:
                "List␣of␣icons␣with␣three␣full␣stars␣and␣two␣empty␣stars"),
        findsOneWidget);

    //Test three stars with few decimal approximation
    await tester.pumpWidget(
        buildTestableWidget(RatingStars(rate: threeStarsAndEpsilon)));
    expect(
        find.byWidgetPredicate(
            (widget) =>
                widget is Row &&
                widget.children is List<Widget> &&
                ((widget.children.elementAt(0) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(1) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(2) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(3) as Icon).icon ==
                        Icons.star_border &&
                    (widget.children.elementAt(4) as Icon).icon ==
                        Icons.star_border),
            description:
                "List␣of␣icons␣with␣three␣full␣stars␣and␣two␣empty␣stars"),
        findsOneWidget);

    //Test three and half stars rendering
    await tester
        .pumpWidget(buildTestableWidget(RatingStars(rate: threeAndHalfStars)));
    expect(
        find.byWidgetPredicate(
            (widget) =>
                widget is Row &&
                widget.children is List<Widget> &&
                ((widget.children.elementAt(0) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(1) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(2) as Icon).icon == Icons.star &&
                    (widget.children.elementAt(3) as Icon).icon ==
                        Icons.star_half_outlined &&
                    (widget.children.elementAt(4) as Icon).icon ==
                        Icons.star_border),
            description:
                "List␣of␣icons␣with␣three␣full␣stars,␣one␣half␣start␣and␣one␣empty␣stars"),
        findsOneWidget);
  });
}

Widget buildTestableWidget(Widget widget) {
  return MaterialApp(home: widget);
}
```

As it can be seen, the test has the purpose of verifying that the number and the type of the icons rendered by the widget is consistent with the decimal value representing the average rate of the caregiver. The test involves multiples scenarios that consider also the approximations carried out in presence of particular values. Following this approach we have developed a number of tests consistent with the separable components used to build the application. The other performed widget tests are those reported below.



```
test
    basic_widget_test.dart
    complete_profile_form_widget_test.dart
    custom_button_widget_test.dart
    custom_surfix_icon_widget_test.dart
    default_button_widget_test.dart
    form_error_widget_test.dart
    no_account_bar_widget_test.dart
    rating_stars_widget_test.dart
    review_card_widget_test.dart
    screen_widget_test.dart
    search_bar_widget_test.dart
    signup_form_widget_test.dart
    social_card_widget_test.dart
    update_profile_widget_test.dart
    user_card_widget_test.dart
    write_review_widget_test.dart
```

Figure 6.1: Performed widget tests

### 6.3.2 Integration tests

Widget tests generally do not check how individual components interact with each other and do not capture the performance of an application running on a device. To evaluate the correct behavior of an application based on different interacting modules we have to rely on a specific test suite that guides the application by verifying that everything works correctly along the way. With these considerations in mind we developed a suite of integration tests using the package **flutter_driver**. The basic approach is to deploy an instrumented application to a real device or emulator and then "drive" the application from a separate test suite. In our case the test suite performs the operations related to the registration of a caregiver user and then continue by using the app in order to test its main features. These process was implemented by using the core features of the driver package of Flutter. For every test, we first created the **SerializableFinders** to locate specific widgets, then we connected to the app before our tests run in the **setUpAll()** function. After that we have structured the workflow of the relevant scenarios by considering the main use cases of the application. Eventually, we disconnected from the app in the **teardownAll()** function after the tests were completed.
The code snippets below show the implementation of the initialization phase and a couple of tests performed on different screens of the application.

```dart
import 'dart:core';
import 'dart:math';

import 'package:flutter_driver/flutter_driver.dart';
import 'package:test/test.dart';



///to run self driven tests, first open your emulator or connect device,
///then open terminal, navigate to ~/dima/client directory and run:
///-->" flutter drive --target=test_driver/app.dart "<--

void main() {
  group("Baddy_self-driven_tests", () {
    String user;
    const chars = "abcdefghijklmnopqrstuvwxyz0123456789";
     final continue_splash = find.byValueKey("continue_splash");
     final login_button = find.byValueKey("login_button");
     final name_login = find.byValueKey("name_login");
     final password_login = find.byValueKey("password_login");
     final goto_signup = find.byValueKey("goto_signup");
     final customer_role_button = find.byValueKey("customer_role_button");
     final signup_button = find.byValueKey("signup_button");
     final name_textfield = find.byValueKey("name_textfield");
     final email_textfield  = find.byValueKey("email_textfield");
     final password_textfield = find.byValueKey("password_textfield");
     final confirm_psw_textfield = find.byValueKey("confirm_psw_textfield");
     final success_button = find.byValueKey("success_button");
     final homepage = find.byValueKey("homepage");
     final search_box = find.byValueKey("search_box");
     ...

    FlutterDriver driver;
      setUpAll(() async {
      driver  = await FlutterDriver.connect();
    });

    tearDownAll(() async {
      if (driver != null) {
        driver.close();
      }
    });


    ...

    test("test_continue_button_splash_screen", () async {
      await driver.runUnsynchronized(() async {
        await driver.tap(continue_splash);
        await Future.delayed(Duration(seconds: 2));
      });
    });

    ...

    test("find_caregiver_by_city", () async {
      await driver.runUnsynchronized(() async {
        await driver.waitFor(homepage);
        await Future.delayed(Duration(seconds: 1));
        await driver.tap(search_box);
```

```
          await Future.delayed(Duration(seconds: 1));
          await driver.enterText("milano");
          await Future.delayed(Duration(seconds: 1));
          await driver.tap(user_card);
          await Future.delayed(Duration(seconds: 1));
        });
      });

      ...
    });
}
```

The scenarios involved during this testing phase were:

- User follows first tutorial
- User signs up as a client
- User logs in with correct credentials
- User tries to log in without registering
- User searches a caregiver by city
- User writes a review to a caregiver
- User writes a message to a caregiver
- Caregiver visits message page
- User visits map page to search a caregiver
- Logout from application

Chapter 7

# References

- Diagrams created with Draw.io
- Flutter
- Firebase Cloud Messaging
- Firebase Cloud Storage
- Node.js
- MongoDB