



POLITECNICO
MILANO 1863

**SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE**

Favor

DESIGN AND IMPLEMENTATION OF MOBILE APPLICATION

TEAM MEMBERS:

Nicholas Nicolis - 10867841

Andrea Paparella - 10701904

Academic Year: 2022-23

Contents

Contents	i
1 INTRODUCTION	1
1.1 Project Idea	1
1.2 Goals	1
1.3 Definitions	1
1.4 Acronyms and Abbreviations	2
2 Overall description	3
2.1 Functional requirement	3
2.2 App Functionalities	3
2.3 App Constraints	4
2.4 Assumptions	5
3 Architectural design	6
3.1 General deployment design	6
3.2 Database - MongoDB	6
3.3 Server Logic - NodeJS	6
3.4 Client Logic - Flutter	7
3.5 External APIs	7
3.5.1 Google Auth	7
3.5.2 Google Calendar	8
3.5.3 Google Maps	8
3.5.4 Sightengine	9
4 User Interface	10
4.1 Cupertino app and Material app	10
4.2 Responsiveness and Adaptive	10
4.3 Screen flow	10

4.4	Relevant independent Screens	16
4.4.1	Other multiple-layout screens	17
5	Testing	19
5.1	Unit testing	19
5.2	Widget testing	19
5.3	Integration testing	20
5.4	Test Coverage	21
6	Development considerations	22

1 | INTRODUCTION

1.1. Project Idea

We supposed the existence of a group of researchers with the following needs: monitoring how much a selected population will freely share its time to help others.

The purpose of the app is to easily allow people to interact with each other through some ordinary activities and stimulate them to connect even more. Participants in the experiment have the opportunity to meet new individuals and potentially enhance their daytime experiences. The researchers can then analyze the exchange of skills and time and draw conclusions.

1.2. Goals

1. Users will be able to post a help request on a shared feed.
2. Users will be able to post when they are available to spend some time helping other users.
3. Users will be able to see other people's requests.
4. Users will be able to connect with other users and book their posts. In this way, they can decide to help or to be helped, depending on the type of request.
5. Users can see how well they are doing during the entire experiment.
6. Scientists will have the possibility to see the overall progress of the experiment, with some custom metrics.

1.3. Definitions

- **Caller:** is a user that needs help in some activities, it needs a favor.
- **Provider:** is a user that is allowed to share its skills with other users, it does a favor.

- **Favor:** is a key term in the entire project and it represents a public claim. It can be created by a *Caller*, meaning it needs that favor, or it can be created by a *Provider*, to manifest its availability to help.
- **Post:** is the actual materialization of a *Favor* in order to be shared and posted online.
- **UserMode:** is the single status of a user at a given time. It can either be equal to *Caller* or *Provider*, but not both simultaneously. If the user is a researcher the system assigns the status of *admin*.
- **MongoDB:** is a popular NoSQL database that uses a document-oriented data model to store, manage, and retrieve semi-structured data.
- **NoSQL:** is a term used to describe a non-relational database management system that allows for the storage and retrieval of data in a flexible and scalable manner.
- **NodeJS:** is a JavaScript runtime that allows for server-side execution of JavaScript code and the building of fast, scalable network applications.
- **Flutter:** is an open-source UI software development kit for building high-performance, beautiful, and responsive cross-platform mobile apps for iOS and Android.

1.4. Acronyms and Abbreviations

- **UX:** User Experience, refers to the overall experience a person has while interacting with the app.
- **UI:** User Interface, is the area where human-device interactions take place.
- **API:** Application Programming Interface, is a set of protocols and tools for building software applications.
- **SSO:** Single-Sign-On, is an authentication scheme that allows a user to log in with a single ID to any of several related, yet independent, software systems.

2 | Overall description

This chapter covers the functionality the application must offer and its implementation plan. The second part of the chapter focuses on the user cases supported, as well as any assumptions or constraints considered during the design process.

2.1. Functional requirement

This section outlines all the functional requirements the app must meet to fulfill the goals described in the previous chapter.

- The app must distinguish users and recognize if they are looking for help or if they want to help someone else.
- The app must provide an easy way to declare a favor.
- The app must provide an easy way to participate in a favor created by another user.
- The app should gamify the experience of users in order to keep them motivated to continue the experiment.
- The app should allow scientists to visualize the data collected.

2.2. App Functionalities

This section presents the solution and implementation designed to meet the functional requirements.

- The app enables users to access the home page even if they have not registered for Favor yet.
- The app allows users to register and sign in using either internal credentials or through SSO services such as Google.
- The app allow users to become *Provider* or *Caller* in every screen, thanks to specific interfaces.

- The app allow user to create *Favors* after they have declared in which UserMode they want to create it.
- The app allow users to inspect a specific *Favor* looking for the other person information and the *Favor* description. It's also possible to visualize the location of the *Favor* on a Google map.
- The app allow users to book an existing *Favor* and reserve it for themselves.
- The app allow users to add their booked favor to their Google Calendar.
- The app allow users to mark the favor as completed and review their experience with the *Caller* or *Provider* they have interacted, through a simple rating interface.
- The app provides a search page that allows users to navigate through posts and search for specific favors.
- The app provides recommended *Favors* to users.
- The app features a public leader-board that ranks users who have completed favors, aimed at motivating greater participation in the experience.
- The app allow users to review the personal information given in the registration phase on the the account screen.
- The app provides a custom view for the admin, allowing them to have a general view of the experiment.

2.3. App Constraints

To ensure a high-quality, successful, and functional application, certain standards must be addressed. The key ones are listed here:

Adaptive layout: the app must change page layout based on the screen size and orientation, in order to improve the UX.

Responsive layout: if no specific layouts are designed for a given screen size, the app must dynamically adjust its dimensions to fit the available space optimally.

Public navigation: the app can be used in read-only mode by users that haven't an account yet.

Favor time: the app must avoid the creation of *favors* in the past or in a future that is far from the actual date.

2.4. Assumptions

During the design phase, certain assumptions were made regarding external user interactions and data processing to simplify the internal logic. These assumptions remain reasonable given the context of the application and the project's emphasis on the general design rather than the logic of data processing.

It follows the list of assumptions:

- The interactions are designed to be 1 to 1.
- The recommendation system is based on the "most recent" post, for the sake of simplicity.
- The variety of tasks that you can choose as a *favor* and the number of locations is limited, but enough to showcase how the app behaves.
- The locations presented in the app are relative to the city of Milan.
- The user is asked to provide quite a lot of information during the registration phase because this app is meant to be as a support tool for research, and the data are useful for social analysis.
- Users have all freely expressed their desire to participate in the experiment.
- Users should meet in person to complete the *favours* and have in-presence interactions. To encourage personal connections and build meaningful relationships, an internal chat system has not been implemented to avoid directing attention away from this important aspect of the platform.

3 | Architectural design

3.1. General deployment design

Favor is a mobile application supported by iOS devices and android phones and tablets. The front-end is built using the Flutter framework and interacts with a local server built with NodeJS and a MongoDB database cloud to provide a seamless user experience.

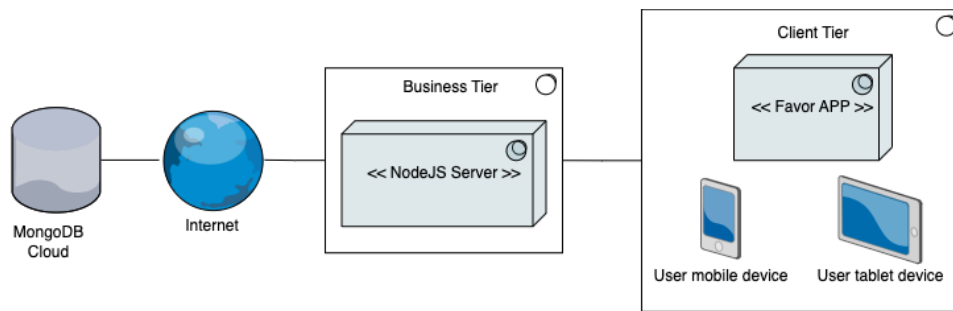


Figure 3.1: deployment view

3.2. Database - MongoDB

We utilized **MongoDB** as the **database management system** to store the application's data. This NoSQL database was chosen for its scalability and flexibility in handling large amounts of unstructured data. The database is connected to the server side of the application, which was built using JavaScript and NodeJS.

3.3. Server Logic - NodeJS

NodeJS is used as the backend for the mobile application due to its fast performance and ability to handle multiple requests efficiently. NodeJS provides APIs for the Flutter front-end to access the database and perform server-side logic. This ensures a smooth user experience and improved app performance.

3.4. Client Logic - Flutter

The choice of Flutter as the framework was driven by the desire for a cross-platform solution, allowing for the deployment of the app on both Android and iOS using a single codebase. This results in a significant reduction in development and maintenance efforts. Although cross-platform development may limit some native capabilities, this is not a major issue as the app will not require the usage of specific OS features. Flutter was selected over other cross-platform frameworks due to its ease of use and the availability of robust libraries in its ecosystem.

For authorization token management, the application utilizes **Shared Preferences** plugin, which wraps platform-specific persistent storage, to store tokens in persistent memory. Tokens are stored securely and are accessible throughout the application to ensure that users remain authenticated.

The application utilizes the **Provider pattern** for state management, allowing for easy access and modification of data throughout the entire application, updating UI elements whenever the underlying data changes.

The graphical elements of the application are implemented using either **stateless** or **stateful widgets**, depending on whether a state is required. We also utilized **hierarchical widgets** to access shared data throughout the application, such as the User Mode. While this could have been achieved through the use of providers, we chose to explore different techniques within the framework for the purpose of learning and expanding our understanding.

3.5. External APIs

3.5.1. Google Auth

Google Sign-In API for Flutter allows users to sign in to our application using their existing Google account, making the sign-up process quick.

Implementation: We leveraged the Google Sign-In Flutter package, which provides an easy-to-use wrapper for the Google Sign-In API. In our application's SignIn screen, we added a Google Sign-In button, which, when clicked, triggers the Google Sign-In process.

API Response: This process involves sending a request to the Google Sign-In API, which returns a Google Sign-In token that can be used to authenticate the user. Once the user is successfully signed in, we store the user's profile information, such as the name,

email, and profile picture, in our database.

3.5.2. Google Calendar

Google Calendar API for Flutter provides functions for inserting events inside existing Google account owners' calendars.

Implementation: We leveraged this API providing users with an "Add to Google Calendar" functionality after the booking of a favor. To integrate it into our system we implemented a Google Calendar API Wrapper, this allowed us to minimize boilerplate code during the insertion of events. In our application's booked favor screen, we designed an "Add to Google Calendar" button.

3.5.3. Google Maps

Google Maps API for Flutter is a set of APIs that allow developers to display Google Maps in the application and interact with it by adding markers, polylines, and more. It provides features such as geocoding, places, and navigation services to add location-based functionality to the app.

Implementation: These functionalities are exploited in the following screens:

- **FavorInformationScreen:** once a *Favor* has been created and a location has been appointed, this screen is built containing the post's information and a map that localizes the place where to execute the *Favor*.
If the user is interacting with a standard smartphone, the map functionality can be accessed by rotating the screen; otherwise, if it is operating with a tablet, it is always present. This decision has been made to don't overwhelm the standard portrait-mobile visualization.
- **BookedFavorScreen:** once a *Favor* has been booked from a user, this screen is built containing the booking information and a map that localizes the place where to execute the *Favor*. Here the API works the same way as the *FavorInformationScreen* since the two share a similar layout.
- **AdminScreen:** this screen makes extensive use of the API and showcases the city of Milan centered around the Duomo square at a high-level zoom (zoom=12). Admins have the ability to navigate around the city and through the use of custom buttons can view markers on locations where *Favors* have been completed. These markers display relevant information gathered from the database.

API Configuration: The visual design of the map has been altered to enhance the accessibility of crucial information, particularly in the AdminScreen. Thus, all the default google labels have been removed, keeping only the ones relative to the city's districts and roads. To easily achieve this, we used an external Google service called Styling Wizard.

Related information We have geolocated the districts of Milan by referring to the city's official documentation to synchronize the database's district information with the actual map location.

3.5.4. Sightengine

The Sightengine API is a content moderation tool that allows us to automatically check posts for inappropriate or offensive content before they are stored in the database and shared online. By integrating the Sightengine API into our server-side code, we can ensure that all post descriptions are filtered for inappropriate or offensive content before they are published. This will help maintain a safe and respectful community for all users.

Implementation: The Sightengine API is integrated into the server-side code responsible for processing and storing the post descriptions. The API performs a scan of the post description and returns a result indicating whether the content is safe or if it contains potentially offensive material. If the content is deemed unsafe, it will not be stored or published, and the user will be notified of the issue.

API Configuration: The Sightengine API will be configured to check for a variety of potential issues, including but not limited to:

- Nudity and sexual content
- Hate speech
- Offensive language

API Response: The Sightengine API will return a result indicating the level of safety of the post description. The API response will include details about the specific types of content found in the post description that led to its rejection. Eventually, the client is notified by a server response, with a status code of 409, signaling inappropriate content.

4 | User Interface

4.1. Cupertino app and Material app

The entire App has been designed starting from a **CupertinoApp**, to reproduce the use and feel of an iOS device. However, we decided to introduce also some material elements, heavily customized, to add some specific functionalities.

This has been possible thanks to the **localizationsDelegates** function, instantiated in the `main()`, that allows using all the material libraries without errors.

4.2. Responsiveness and Adaptive

The responsiveness of the application has been achieved with the use of some customized **LayoutBuilder** widgets. With these internal structural widgets, the application decide which layout to use based on the available screen dimensions. For some specific screens, an ad-hoc alternative layout has been designed. Some examples are the `SignIn`, the `FavorInformationPage`, or the `AccountPage`.

In general, even if some alternative layouts have not been implemented, the application remains fully adaptive through the use of **MediaQuery** function.

4.3. Screen flow

The following section displays the primary flow of the application using a combination of UML diagrams and example screenshots.

First Time in App

If the user enters the application for the first time, some `IntroductionScreens` are shown.

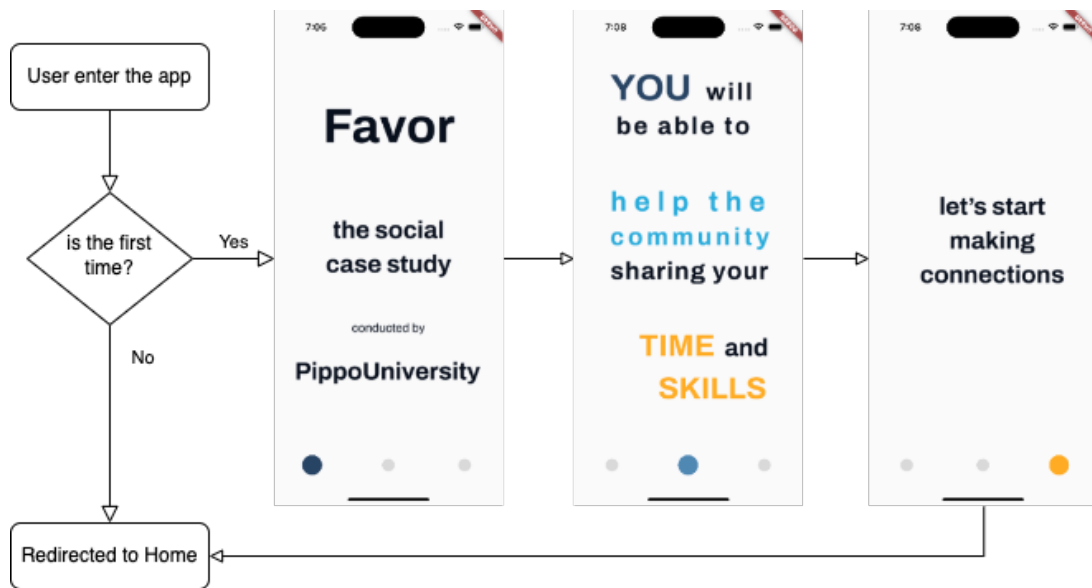


Figure 4.1: First Time in App

Is the user already registered?

Regardless of authentication status, users are directed to the Feed for navigation within the app. Upon interaction with functionalities that require authentication, such as booking a *favor*, they are then redirected to the SignInScreen.

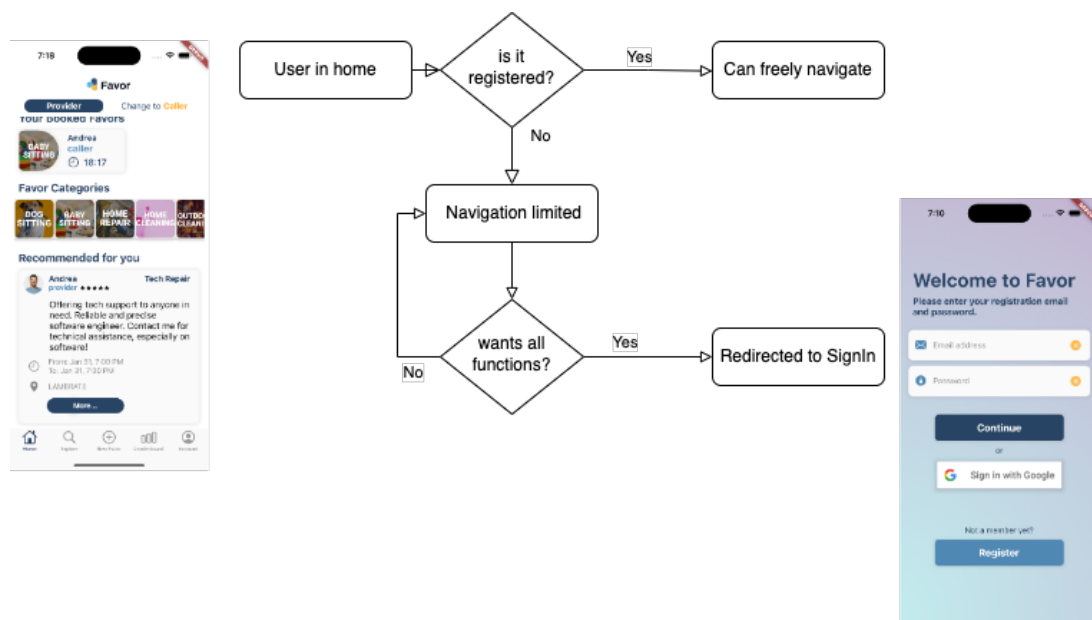


Figure 4.2: Is the user already registered?

User registration

Users have the option to proceed with either Google or the default registration. After en-

tering basic information such as name, surname, email, and password, they are prompted for personal details such as age, gender, occupation, city of residence, and a brief bio.

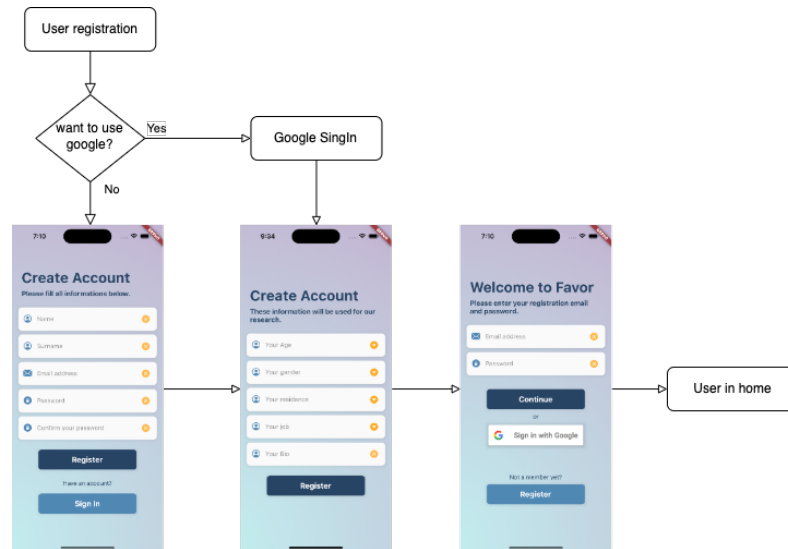


Figure 4.3: User registration

Home possible flow

After signing in, users are directed to the page below, where they can navigate the app using the bottom navigation bar. Specifically, they can access the Feed by tapping the Home button, search for a *favor* by tapping the Explore button, create a *favor* by tapping the New Favor button, view the leaderboard by tapping the leaderboard button, or access their personal information by tapping the Account button. They can also be redirected to a specific FavorInformationPage or FavorBookedPage, by tapping on their relative buttons.

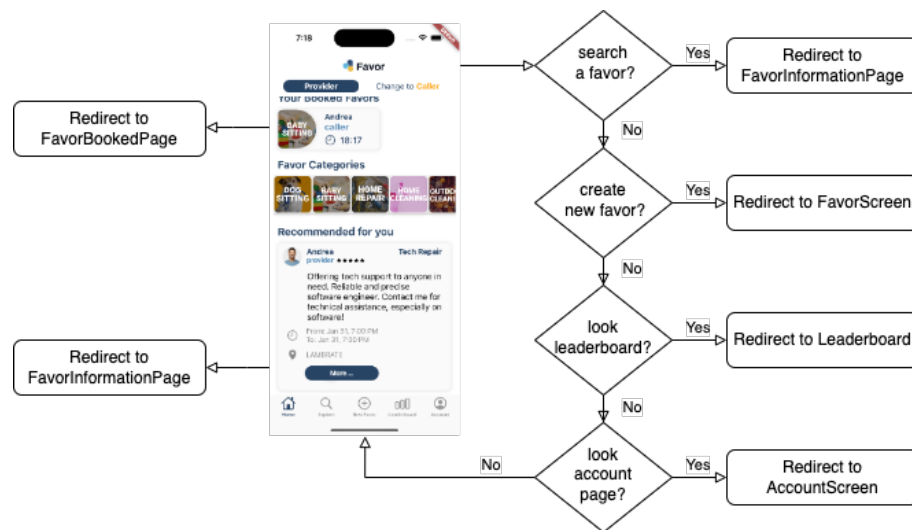


Figure 4.4: Home possible flow

Life Cycle of a *favor*, from creation to completion

Users can create and post a *favor* by inserting category, location, timing, and description. Once shared, these can be viewed on the Feed.

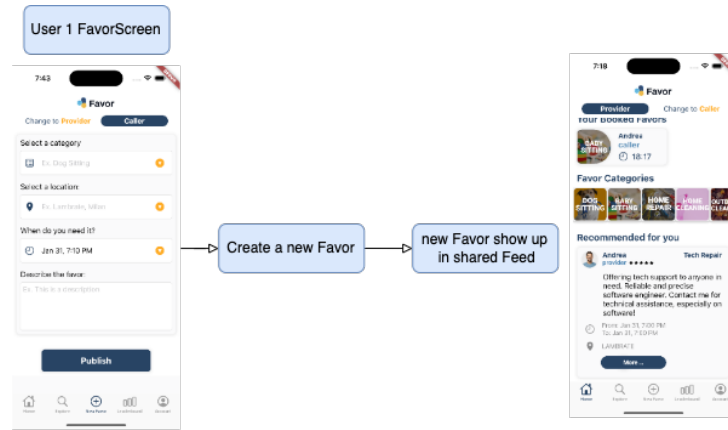


Figure 4.5: Life Cycle of a *favor* created by user 1

In the Home, users can see the previously booked *favours* by horizontal scroll, tappable categories redirecting to Explore screen by horizontal scroll, and recommended posts by vertical scroll. Users can access more information about a recommended post by tapping the "More..." button and can book it. The third image from the left also shows the appearance of the BookedFavor screen, which includes a button to add the event to the Google Calendar and a "Mark as Completed" button. Upon tapping the latter, users are asked to provide a review for the *caller* or *provider* they interacted with. This review will play a key role in updating the leaderboard.

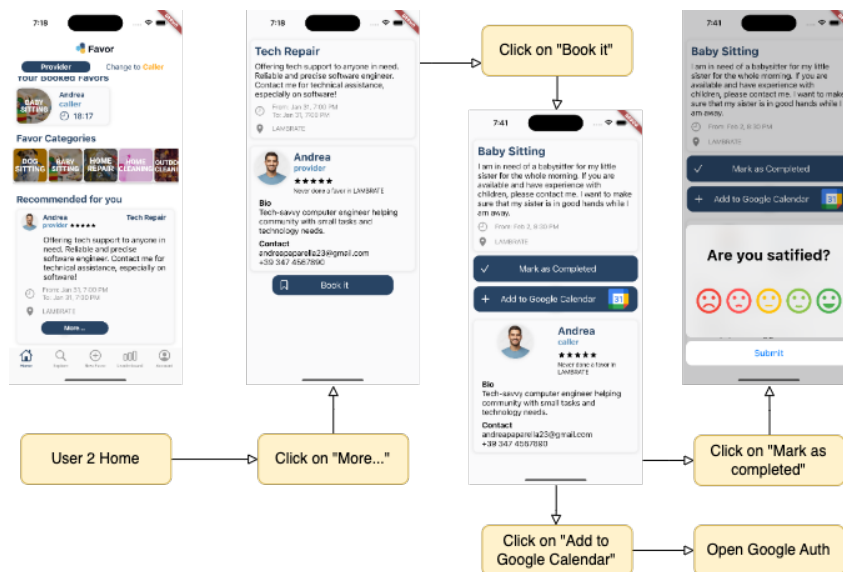


Figure 4.6: Life Cycle of the same favor booked by user 2

Search

By tapping on a *favor* category on the Home screen, users are redirected to the Explore screen where there will be shown *favours* regarding the clicked category. Moreover, users can clear the search input bar and search for *favours* by location or a different category. The upper navigation bar helps the user to search for *provider* or *caller* *favours* by simply tapping the "Change to Provider/Caller" buttons. In the picture below, the User Mode is *caller*, so it can only see *provider* *favours*. In the same way, if the User Mode was *provider*, the system would show only *caller* *favours*.

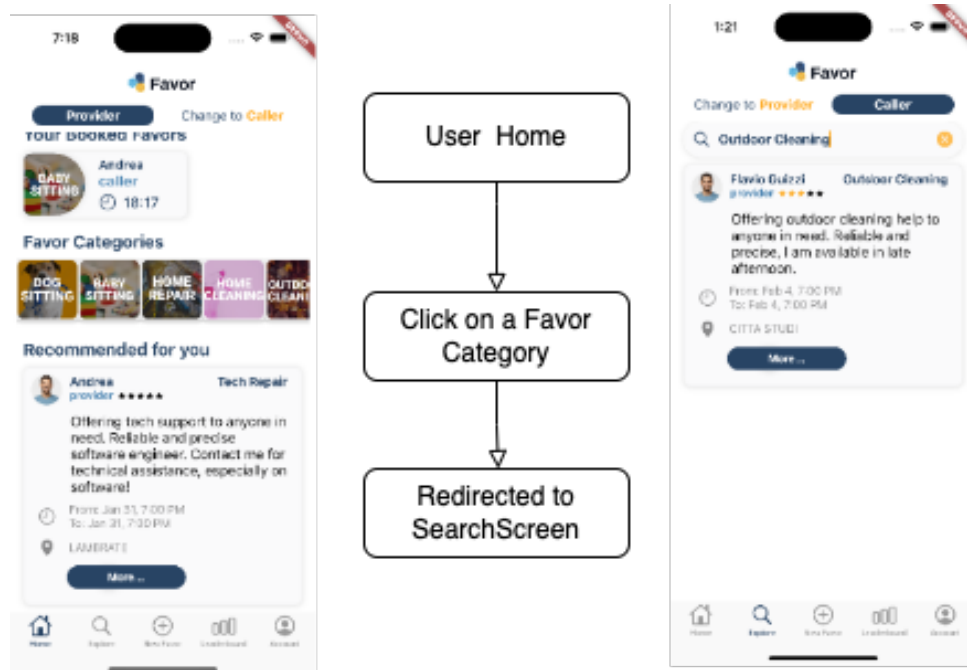


Figure 4.7: Search

Leaderboard

Users can see how well they are performing as *provider* or *caller* by selecting a location and the User Mode in the upper navigation bar. The leaderboard is ranked based on a formula that takes into account how many *favours* have been completed and the ratings received.

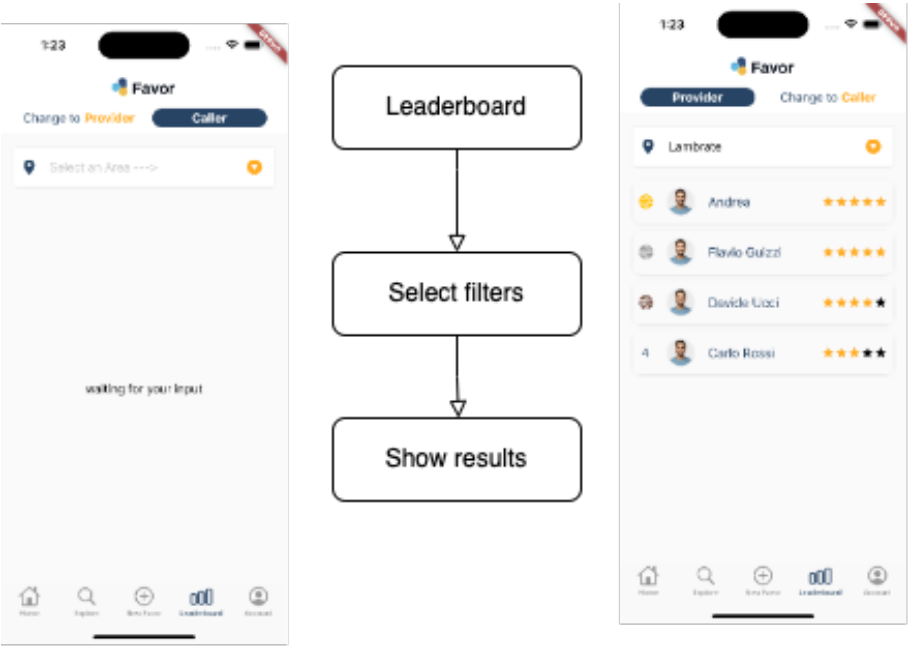


Figure 4.8: Leaderboard

4.4. Relevant independent Screens

In this section are listed a series of fundamental Screens that are on the core of the app.

Favor Information This screen shows all the relevant information concerning a specific favor. This screen offers an alternative layout for landscape views or tablet screens.

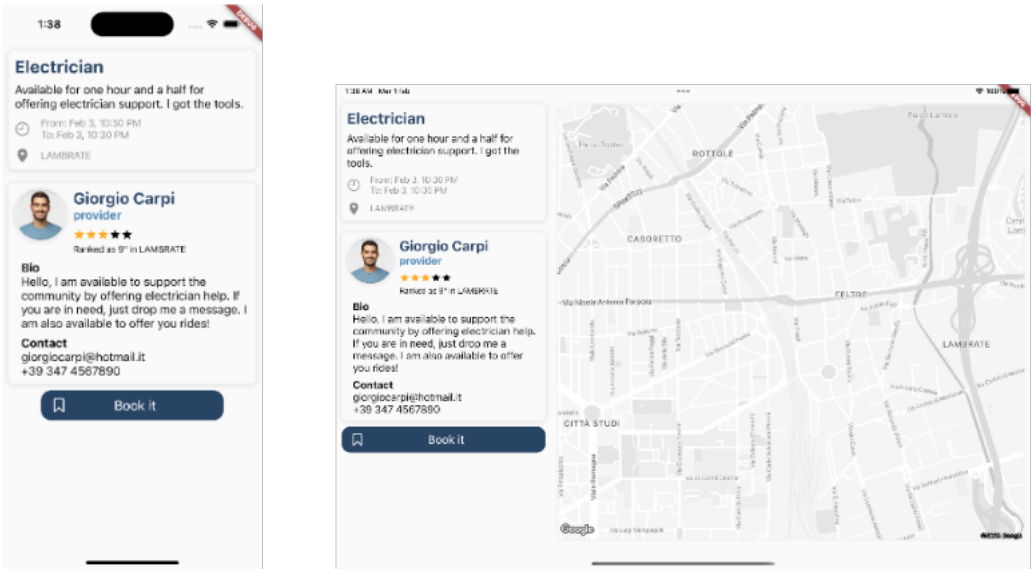


Figure 4.9: Favor Information screen. Left: iPhone 14 Pro. Right: iPad Pro

Admin This screen features a dynamic map that displays data collected by the application. It's also possible to navigate through locations (where *favors* have been executed) thanks to the buttons labeled with the districts' names.

This screen offers an alternative layout for landscape view or tablet screens.

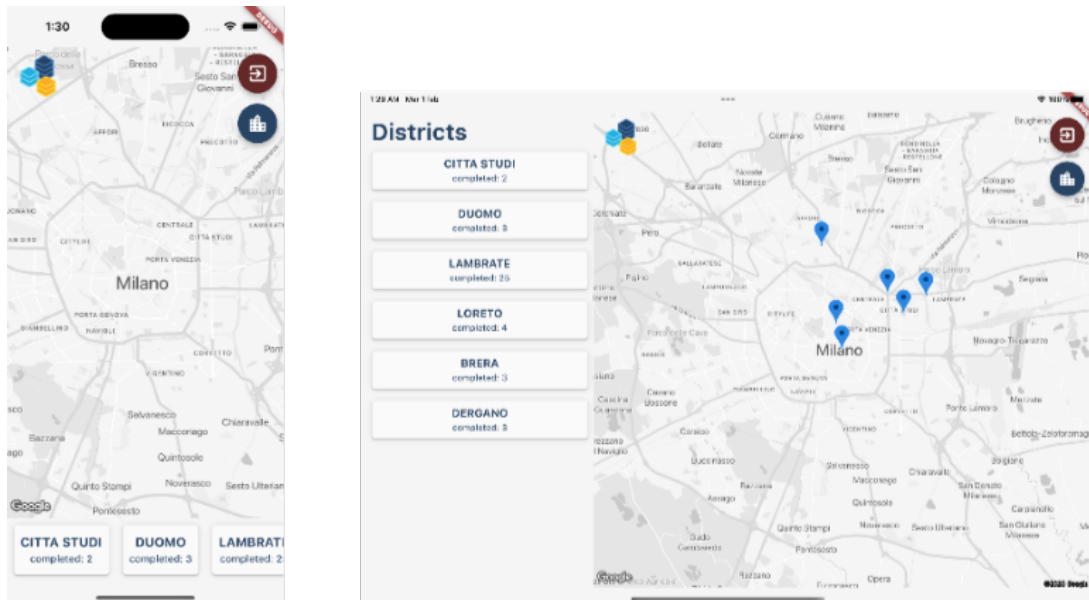


Figure 4.10: Admin screen. Left: iPhone 14 Pro. Right: iPad Pro

4.4.1. Other multiple-layout screens

This section lists all the different screen layouts available within the application.

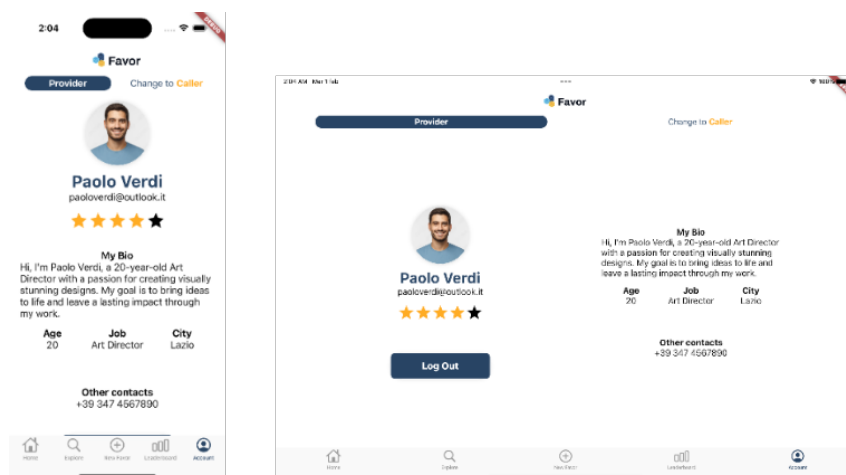


Figure 4.11: Account screen. Left: iPhone 14 Pro. Right: iPad Pro

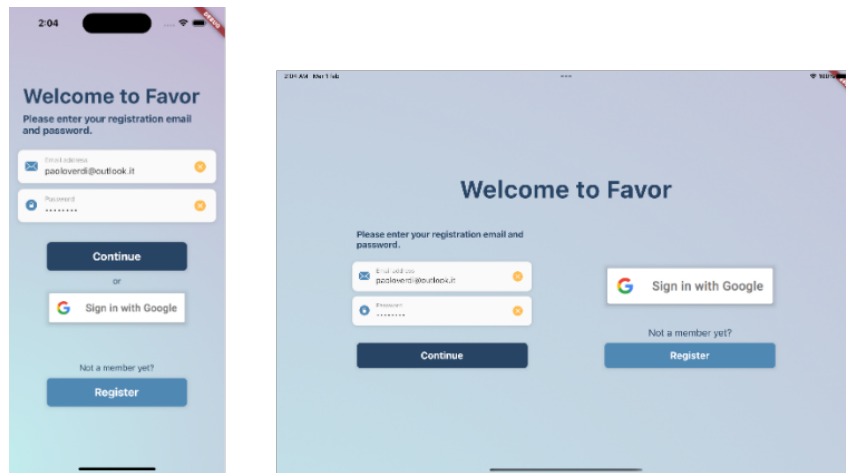


Figure 4.12: SignIn screen. Left: iPhone 14 Pro. Right: iPad Pro

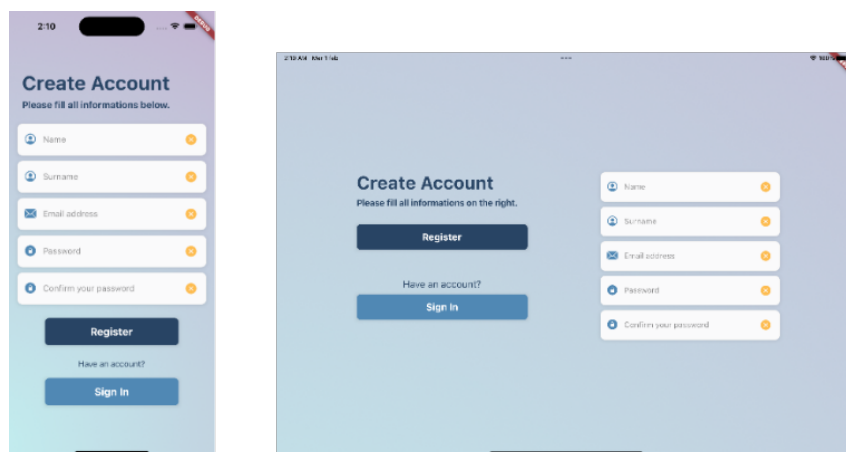


Figure 4.13: SignUp screen. Left: iPhone 14 Pro. Right: iPad Pro

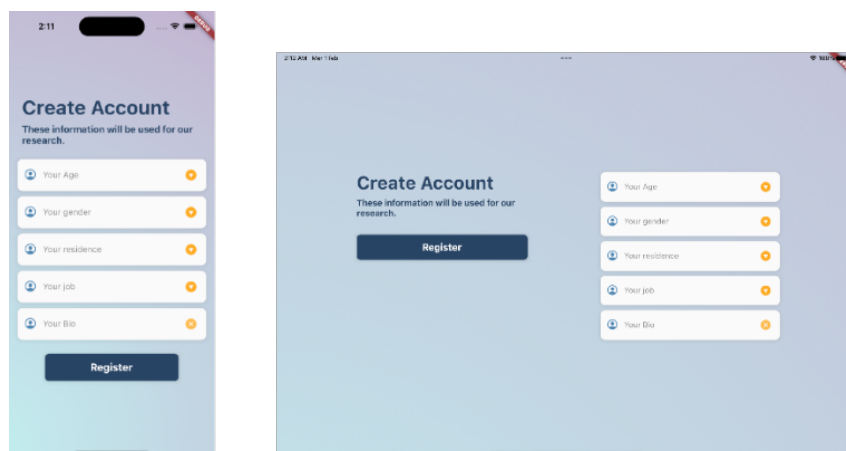


Figure 4.14: SignUp2 screen. Left: iPhone 14 Pro. Right: iPad Pro

5 | Testing

Testing is a crucial aspect of software development and ensures the smooth functioning of an application. In this chapter, we will dive into the different types of tests performed on our system, including unit tests, widget tests, and integration tests, by leveraging the "flutter test" library. We used the "Mockito" package for mocking objects that had dependencies with external systems and APIs. This process helped to catch any potential bugs early on leading to a higher quality product for end-users. Moreover, by conducting testing of various components of the application, we can ensure that the application operates as intended and meets the design specifications.

5.1. Unit testing

This section outlines the testing process for individual units of code to ensure they are functioning correctly and meeting project requirements.

We mainly focused on:

- conversion from JSON strings received from the server to Dart objects
- methods for getting and setting data into the persistent memory of the device
- utility functions for converting data into different forms

5.2. Widget testing

In order to ensure the stability and reliability of the user interface, we performed testing on individual widgets within the application. In Flutter, widget testing can be accomplished by utilizing the 'Widget Tester' class.

Here, we focused on:

- Testing our custom components created for displaying data. Such as widgets containing posts, favors, stars for the rating, and picker menus.

- Testing the widget building and display of correct data across almost every screen.
- Testing the interactions with buttons that rely on server communication.

5.3. Integration testing

The following outlines the integration testing procedure for our application, which covers the methodology, strategy, and recommended guidelines.

We tested the navigation throughout the system when using the bottom navigation bar, named CupertinoTabBar. This allowed us to confirm the correct display of mock server data on all screens. Our focus was particularly on:

- Testing that when the user has never registered before, the app shows Introduction screens and redirects the user to the Home screen
- Testing that when the user has never registered before, the app shows Introduction screens and redirects the user to the Home screen. Starting from this state, we further tested the following cases:
 - Testing that the users are redirected to the Home screen when they tap on the "Home" button in the bottom navigation bar. Here, we verified that only posts and favorite categories are displayed, not the booked favors. This is because at this stage the user is not yet registered, therefore they cannot view the booked favors.
 - Testing that the users are redirected to Explore screen when they tap on "Explore" button in the bottom navigation bar. Here, we verified that posts are correctly shown.
 - Testing that the users are redirected to the Favor screen when they tap on the "New Favor" button in the bottom navigation bar. Here, we verified the presence of the input form.
 - Testing that the users are redirected to the Leaderboard screen when they tap on the "Leaderboard" button in the bottom navigation bar. Here, we verified that users and their average ratings for a dummy leaderboard are shown.
 - Testing that the user is redirected to the Account screen when they tap on the "Account" button in the bottom navigation bar. Here, we verified that the users are eventually redirected to the "Sign In" screen since it has no credentials for viewing their account information.

- Testing that the authenticated users can see their information in the Account screen when they tap on the “Account” button in the bottom navigation bar.

We tested authentication screens by inserting dummy information and we focused on:

- Testing that when the users insert correct information in the SignUp screen, they are redirected to the SignUp2 screen.
- Testing that when the users insert correct information in the SignUp2 screen, they are redirected to the SignIn screen.
- Testing that when the users insert incorrect information in the SignIn screen, they are warned by the system.

5.4. Test Coverage

Overall, we reached a coverage percentage of over 70%. It's worth noting that "services" classes have not been tested since they rely solely on the server response and interactions. Additionally, some "helpers" functions were not tested for their ease of use.

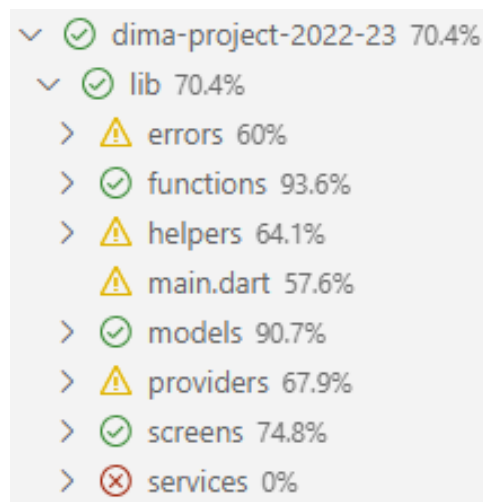


Figure 5.1: Test Coverage

6 | Development considerations

As a small development team of only two members, it was important for us to have a flexible and efficient workflow. That is why we decided to use the **agile methodology**. By using agile, we were able to:

- Quickly respond to changes in project requirements and make necessary adjustments along the way.
- Collaborate closely with each other, ensuring that we were both aligned on project goals and that there was clear communication regarding project requirements.
- Continuously improve our process by reflecting on our performance after each sprint and making changes to ensure we were always working in the most efficient and effective way possible.

In short, using agile allowed us to work together effectively and efficiently.

For the most part, we worked individually between sprints. However, there were opportunities for us to engage in **pair programming**, particularly during the testing and integration phases.

We **shared the workload** for the application logic and design implementation, dividing the tasks between us. However, Andrea focused a little more on the database and API connections, while Nicholas took the lead on the graphical aspects of the project.