POLITECNICO
MILANO 1863

Computer Science and Engineering

Design and Implementation of Mobile Applications

Academic year 2021-2022

Design Document

Expire App

Authors:
Arslan Ali 10807090
Alessandro Sorrentino 10746269

Version 1.0

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Purpose

The goal of DD (Design Document) is to provide a complete description of the design of Expire App's system, within which the main component is the Flutter application developed for the course of Design and Implementation of Mobile Applications at Politecnico di Milano (Italy). In particular this document will describe the architectural decisions, their communication interfaces, runtime views and user interaction flow. In the last chapter (5) it will also present implementation, integration and test plan.

1.1.1 Goals

The goals of the *Expire App* are the following:

G.1 User can login and register to the application

G.2 User can register food products

G.3 User can visualize inserted products

G.4 User can create and visualize shopping lists

G.5 User can view statistics

G.6 User can view and modify personal data

G.7 User can belong to a family group

G.8 User can visualize food recipes

1.2 Scope

It often happens that after you've been to the grocery store, you put products in the fridge, in the cupboard, etc. And forget about it, until they expires. Expire App has

mainly been developed for users who want to track the expiration date of the products. It also allows to create a shared list of products with members of the same household. If you're running out of ideas, and want to avoid waste, Expire App allows you to find recipes with the products you have on hand and it also allows users to view the nutritional information of foods. The architecture must be designed to be maintainable and extensible, while the individual components must have high cohesion and decoupling. For this purpose, the various components must not incorporate several unrelated functionalities and should decrease interdependency between them. Furthermore, design patterns and architectural styles will be used for the same reason.

1.3 Definitions, Acronyms, Abbreviations

In the following section is clarified the meaning of some definitions, acronyms and abbreviations which will be use in the DD, in order to help the general understanding of the document.

1.3.1 Definitions

Table 1.1: Definitions

The system	The whole application to be developed
User	A generic person who uses the application
Unregistered user	A generic person who is using the application without having registered
Application service	Functionality offered by the application for certain users
Family group	a group of users belonging to the same household
Family ID	Unique code belonging to each family group
Layer	Separate functional component that interact with other layers in some sequential and hierarchical way
Framework	Reusable set of libraries or classes for a software system
Mockup	Full-size model of a design or device, used for product presentations or other purposes

1.3.2 Acronyms

Table 1.2: Acronyms

DD	Design Document
DBMS	Database Management System
DB	Database

UML	Unified Modelling Language
API	Application Programming Interface
UI	User Interface
HTTP	Hypertext Transfer Protocol
ID	Identification
JSON	JavaScript Object Notation
OCR	Optical Character Recognition

1.3.3 Abbreviations

Table 1.3: Abbreviations

G.i	i-th goal
R.i	i-th requirement
D.i	i-th domain assumption
C.i	i-th constraint

1.4 Document Structure

The DD is structured in the following five chapters:

- **Chapter 1 - *Introduction*:** Brief description of the document that will be presented, it gives a general information about what this document is going to explain.
- **Chapter 2 - *Overall description*:** It presents an overall description of the system's features and constraints. Furthermore, in this section are outlined the assumption related to users and surrounding behaviours that we considered as valid in the next chapters.

- **Chapter 3 - *Architectural design*:** This chapter describes the architectural choices adopted to build Expire App, with an overview of the main components and their interactions, in the final section it also describes some deployment choices.
- **Chapter 4 - *User interface design*:** Mockups of Mobile UIs of system
- **Chapter 5 - *Implementation and Testing strategy*:** The last chapter describes some guidelines that will be used to implement Expire App, in which order components and sub-components are integrated and how they are tested.
- **Chapter 6 - *Future development*:** Future developments and improvements to make an even better app

Chapter 2

Overall description

2.1 Product perspective

Expire App is a system that provides all the functionalities described in the product functions section. It includes all the subsystems needed to fulfil these software requirements. User can access the service through the application, which should be downloadable and runnable both on iOS and Android devices either phone or tablet. All interfaces must conform to a uniform, intuitive and user-friendly design, that doesn't require the reading of detailed documentation to be used. The mobile application can run on devices that comply with the minimum requirements in terms of memory and screen size, computational power and other relevant parameters specified in the subsection 2.5. Furthermore, the back-end services are all hosted by Google's Firebase.

2.2 External services

In the next subsections, the main external services used in the application will be illustrated and briefly explained.

2.2.1 Firebase

Firebase is a Backend-as-a-Service (Baas). It provides a variety of tools and services.It is built on Google's infrastructure. Firebase is categorized as a NoSQLdatabase program, which stores data in JSON-like documents. Firebase offers multiple backend services, of which Expire App uses:

- Firebase Authentication: is an extensible token-based auth system and provides out-of-the-box integrations with the most common providers such as Google, Facebook, and Twitter, among others.
- Firestore Database: cloud-based NoSQL database server that stores and sync data, that supports automatic scaling. It stores data as documents that are logically classified into collections. The Firestore document offers support for multiple file types, numbers, strings, and nested objects.
- Cloud Storage: is an object storage service. In Expire App it's used for storing pictures of products

2.2.2 OpenFoodFacts

Open Food Facts is a free, online and crowdsourced database of food products from around the world licensed under the Open Database License (ODBL). In Expire App it's involved when a user add a product by scanning barcode. The database contains mostly food products and their details, such as nutritional values. Those data is also used in the statistics screen, to create charts regarding nutritional values.

2.2.3 Spoonacular API

Spoonacular is a recipe search engine and social cooking platform. In Expire App this service is used whenever the user tap on the recipe section. This API provides a list of recipes based on the ingredients that the user has already saved, it's also possible to get detailed information about a recipe by tapping on it.

2.2.4 Google Cloud Vision API

This service is a OCR offered by Google. It permits the conversion of handwritten/printed texts into machine-encoded text, in Expire App it used when a user adds a new product manually, and instead of writing the list of ingredients manually, he/she can take a picture of the label, and the service will translate it into text.

2.2.5 Authentication

Firebase Authentication provides backend services. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.

2.2.5.1 Google authentication

Authentication into Expire App using a Google account, 3rd party library is required to trigger the authentication flow, called *google_ sign_ in*.

2.2.5.2 Facebook authentication

Like Google authentication, also Facebook authentication
required a 3rd party library, called *flutter_facebook _auth* and some manual adjustment with native settings for android and iOS. To use Facebook authentication it's needed a Facebook account and access the Facebook Developer Platform. Facebook provides an authentication token that can be used via OAuth for authorization and log in into Firebase.

2.2.5.3 Apple authentication

Authentication into Expire App using a Apple account (possible only for iOS and iPadOS users).

2.2.6 Scandit

Scandit is a platform for mobile computer vision and augmented reality, it offers a variety of products, in Expire App we used the barcode scanner for reading:

- barcode: when a user want to insert a new product to the list
- qrcode: when a user want to join a family group

2.3 Assumptions and dependencies

D.1 To authenticate the user needs an internet connection

D.2 User must have access to a camera

D.3 All the family members live together and trust each other

2.4 Functional requirements

In this section are outlined the various the functional requirements of the application dived by goals. If two goals share a requirement, it is cited twice and referenced with the same number. Furthermore, domain assumptions are referenced if pertinent to the context.

- **[G1] User can login and register to the application**
  - [D1] To authenticate the user needs an internet connection
  - [R1] User can login and register using email and password
  - [R2] User can login using Google authentication service
  - [R3] User can login using Facebook authentication service
  - [R4] User can login using Apple authentication service
  - [R5] User can access the application without registering
- **[G2] User can register food products**
  - [D2] User must have access to a camera
  - [R6] User can retrieve products infos by scanning barcode
  - [R7] User can set expiration of a product
  - [R8] User can take and upload a picture of a product
  - [R9] User can manually insert nutritional values of a product
  - [R10] User can retrieve a text from a label by taking a picture
  - [R11] User can manually register a product
- **[G3] User can visualize inserted products**
  - [R12] User can sort products by their expiration date
  - [R13] User can filter products by categories
  - [R14] User can filter products by search keywords
  - [R15] User can view detailed information about a product
  - [R16] User can delete registered products

  - [R17] User can add a product to a shopping list
  - [R18] User can hide expired products

- **[G4] User can create and visualize shopping lists**
  - [R19] User can create shopping lists with custom label
  - [R20] For each shopping list the user can consult the contained products
  - [R21] User can mark a shopping list as completed
  - [R22] User can delete a shopping list
  - [R23] User can check a product inside a shopping list
  - [R24] User can increment/decrement the quantity of a product inside a shopping list
  - [R25] User can delete a product from a shopping list
  - [R26] User can add a product to a shopping list using a custom label and quantity
  - [R27] User can add a product to a shopping list from an already inserted product (see **G2**)

- **[G5] User can view statistics**
  - [C1] User must be authenticated
  - [R28] User can view statics about nutritional values of inserted products

- **[G6] User can view and modify personal data**
  - [R29] User can insert/update the display name
  - [R30] User can delete his/her account
  - [R31] User can upload a personal avatar

- **[G7] User can belong to a family**
  - [C2] User belongs to a family
  - [D3] All the family members live together and trust each other
  - [R32] User can join a family using a unique family ID
  - [R33] User can share his/her family
  - [R34] User can leave his/her family
  - [R35] User can consult members belonging to his/her family
  - [R36] User can visualize and edit all the products his/her family

- **[G8] User can visualize food recipes**
  - [C3] User must be authenticated
  - [R37] User can visualize recipes based on inserted products (see **G2**)
  - [R38] User can view the detail of a recipe
  - [R39] User can mark a recipe as favourite

2.5 User characteristics

We consider users every person who downloaded the application on his/her device (Android, iOS or iPadOS). The actors of the applications are the following:

- *Unregistered user*: a person who has not registered and is allowed to use the application with limited functionalities.
- *User*: a person who is already registered into the application.

2.6 Constraints

2.6.1 Regulatory policies

Access to confidential and sensitive data must be restricted to protect it from being lost or compromised in order to avoid adversely impacting users. At the same time, users must be able to access data as required for them to work effectively. All sensitive data and user information are acquired by the company under the accepted terms and conditions. Email addresses won't be used for commercial uses. Conditioned on compliance, *Expire App* grants to the user a personal, nonexclusive and nontransferable license to download, install and use the mobile or to access the system through the web app. Furthermore the system is capable to delete personal data upon user request.

2.6.2 Hardware constraints

These are the minimum hardware requirements needed to access the system:

- **Device requirements:**
  - Android OS (version $\geq$ 5.00 Lollipop)
  - iOS and iPadOS (version $\geq$ 12.00)
  - 193 MB of available space on disk
  - 3G or Wi-Fi (needed for some functionalities)

2.6.3 Availability

It refers to the system's ability to be available for use, especially after a failure has occurred. The fault must be recognized and then the system must respond in some way. The system should be available for 99.97% of the time, which implies a maximum of 3.65 day of downtime per year and 7.31 hours of downtime per month.

2.6.4 Reliability

It is the ability of a system or component to function under the conditions established for a given period of time, that is, continuity of the correct service. Therefore, the system must be fault tolerant and robust.

2.6.5 Security

Attacks against a system can compromise the confidentiality, integrity or availability of a system or its data. System security includes the development and implementation of security countermeasures. Some security measures are adopted in the code, for example the use of information hiding techniques (usage of private classes, creation of proper getters and setters methods, etc.). On the server side, some Firebase security rules are exploited, these rules are hosted on Firebase servers and are applied automatically at all times and administrator can change the rules of the database in Firebase console.

2.6.6 Parallel operations

The system must support concurrent operations from different users, for example the insertion of a product in the same family group, or managing the registration in the application.

Chapter 3

Architectural design

3.1 Overview

This chapter presents a general description of the system components, both at a physical and at a logical level. In the following sections are presented the chosen paradigm of the system and its components, followed by the description of how the system will be deployed. Moreover, components' sequence and interface diagrams are listed to describe the way components interact to accomplish specific tasks of the application.

Expire App is built using Flutter. Flutter is an open source framework developed by Google, for building natively and multiplatform applications from a single codebase. Flutter is powered by Dart, a programming language optimized for fast apps on every platform. The software to be developed is a distributed application with a client-server multi-tiered architecture. The general architecture of the system can be subdivided into the three logic layers:

- *Presentation Layer:* It provides an interface to users accessing the mobile application allowing them to exploit Expire App's services in the most efficient and intuitive way.
- *Application Layer:* it contains business and data access logic. It handles the functions to be provided for the users. It also manages communication between the end user interface and the database.
- *Data Layer:* it manages the persistent storage and retrieval of data accessing to the database.

3.2 Component view

The purpose of this section is to show the component diagram which is intended to represent the internal structure of the modeled software system in terms of its main components and the relationships among them.

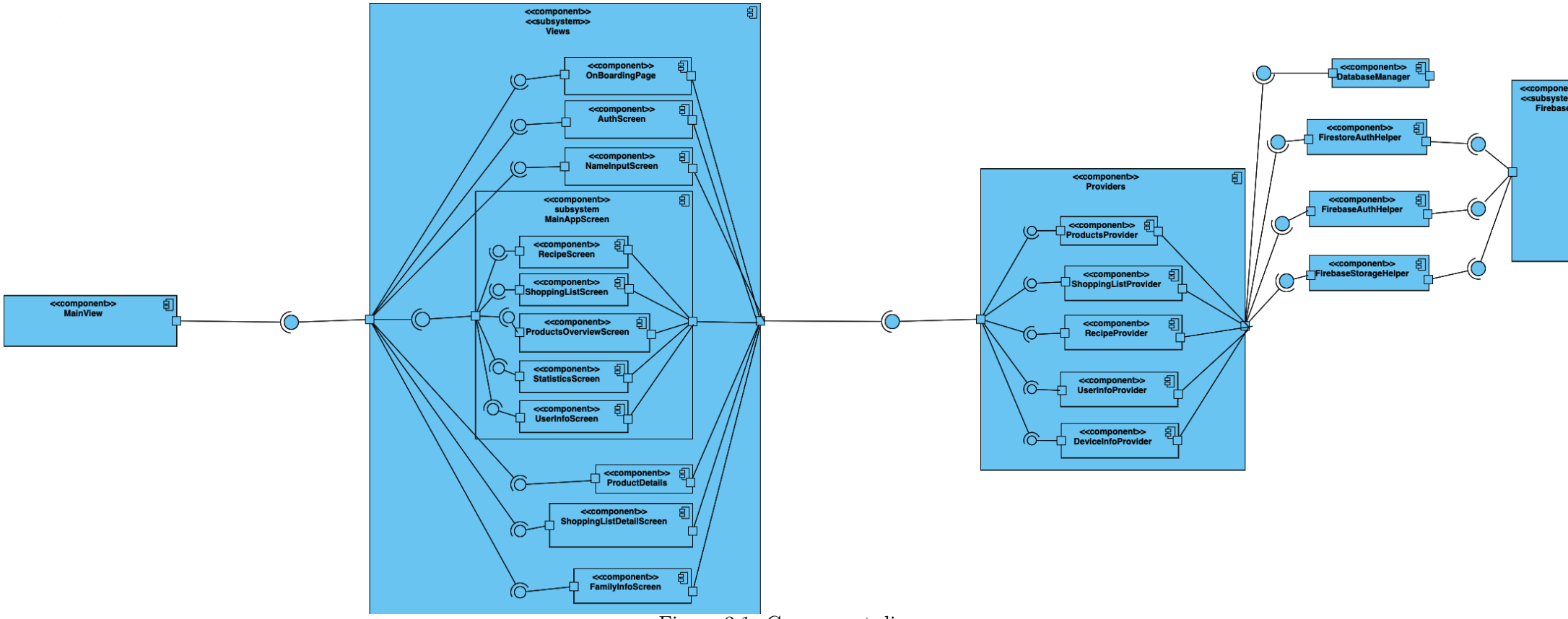


Figure 3.1: Component diagram

3.2.1 Mobile Application

The application server must handle the business logic, the connections with the data layer, and also have to manage the different ways of accessing the services from different clients. To comply with all these requirements, it must provide interfaces to connect with all the previously mentioned components. In particular the application rely on Google's Firebase. Firebase is a Backend-as-a-Service (Baas). It provides a variety of tools and services. It is built on Google's infrastructure. Firebase is categorized as a NoSQL database program, which stores data in JSON-like documents.

3.2.2 Application server

The implementation of the mobile application must be autonomous from the structure of the application server. The UI must respect the design guidelines provided by Google's Material Design. As stated before, application is developed using Flutter, which is a framework base on the Dart programming language.

To separate application states and UI, an external package called **Provider** is used. It does mainly two jobs:

- Separates state from UI
- Manages rebuilding UI based on state changes

Which makes loads of things simpler, from reasoning about state, to testing to refactoring. It makes codebase scalable. It can be considered a low boiler-plate way to separate business logic from widgets in apps. In addition to using Providers, other external packages have been used(non-exhaustive list):

- **sqflite:** SQLite plugin for Flutter
- **http:** contains a set of high-level functions and classes that make it easy to consume HTTP resources
- **flutter scandit:** barcode and qrcode scanner
- **shared preferences:** Wraps platform-specific persistent storage for simple data
- **syncfusion flutter charts:** data visualization library charts
- **firebase:**
  - **cloud firestore:** cloud-hosted, NoSQL database accessible directly via native SDKs
  - **firebase storage:** is a service used to store and download files generated directly by clients
  - **firebase auth:** firebase authentication API
- **google sign in:** API to manage Google login and registration

- **flutter facebook auth:** API to manage Facebook login and registration
- **openfoodfacts:** package for the Open Food Facts API
- **testing:**
  - **mockito:** testing framework
  - **firebase auth mocks:** unit tests involving Firebase Authentication
  - **build runner:** provides general-purpose commands for generating files, and for optionally testing the generated files or serving both source and generated files
  - **fake cloud firestore:** Fakes to write unit tests for Cloud Firestore
  - **google sign in mocks:** mocks for google sign in package, used for testing purposes

Furthermore to get data about products and *recipes*, also external services are used. **OpenFoodFacts** is used for retrieving information about a product given a barcode. **Spoonacular API** is used for retrieving recipes and detail about a recipe.

3.2.3 Database and Storage

The application mainly makes use of two databases: one internal, managed through the external *sqflite* library, the other external, mentioned before, *Firebase*. The first one is used when a user use the application without signing up, Firebase instead hosts the database and storage when user is logged in. The data layer comprises a DBMS and a NoSql Database Engine to manage insertions, modifications, deletions of data inside the storage memory. The advantage of the NoSql engine it indexes queries by default: this means that the performance is proportional to the size of the result set and not of data set. Despite the implementation, for which we choose to rely on the service hosted by Firebase, this layer must respect the ACID proprieties while executing transactions.

3.3 Deployment view

This section shows the physical distribution of the system, that is, the environment in which the system runs. Specifically, Figure xxx, shows the structure of the hardware component that execute the software.

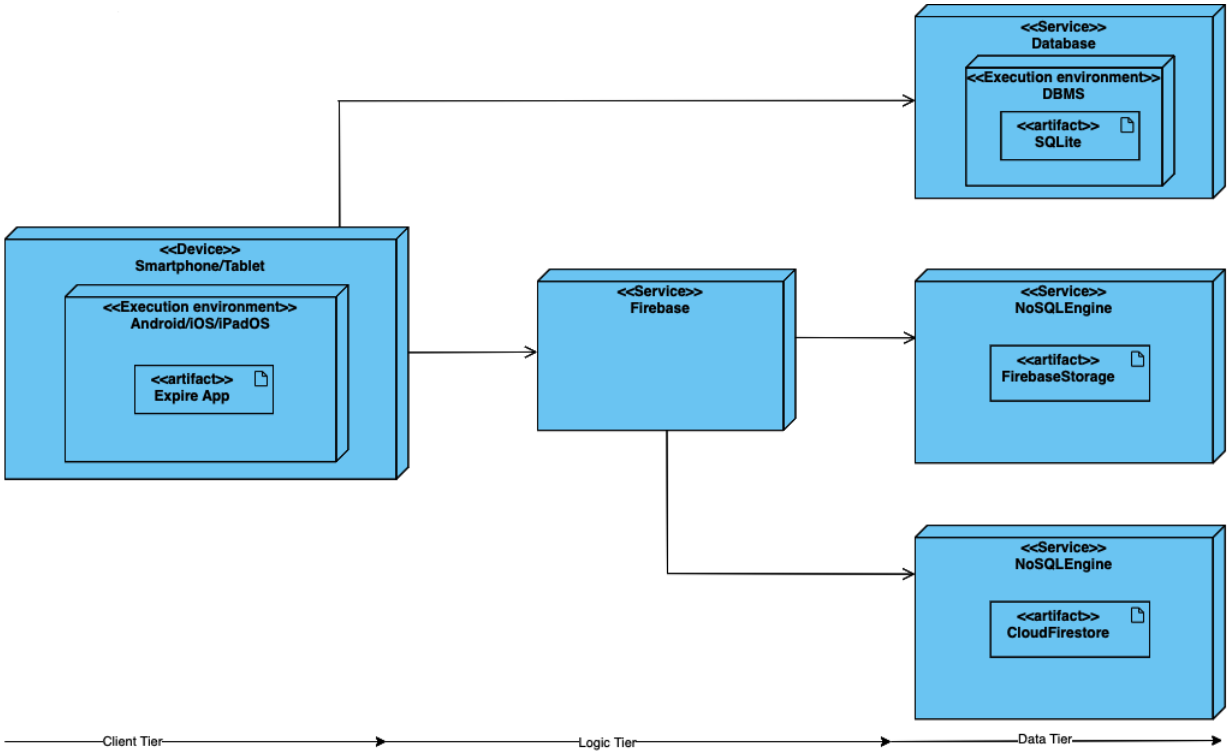


Figure 3.2: Deployment diagram

3.4 Runtime view

The purpose of this section is to describe the runtime behaviour of the most meaningful functionalities of the system by highlighting the interactions between different components. These interactions are described, howsoever, maintaining a certain level of abstraction because we don't know precisely how the application server will instantiate its processes.

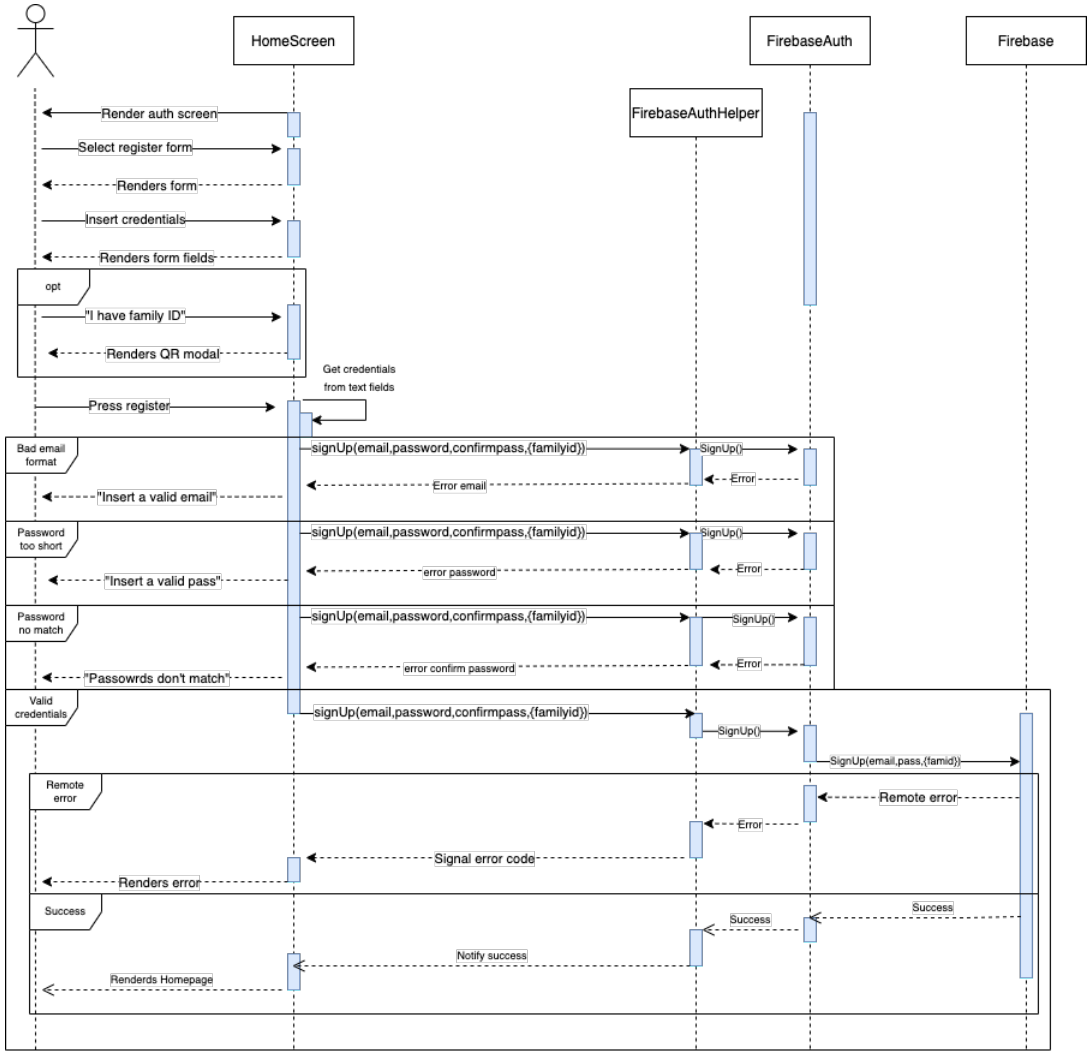


Figure 3.3: User signs up

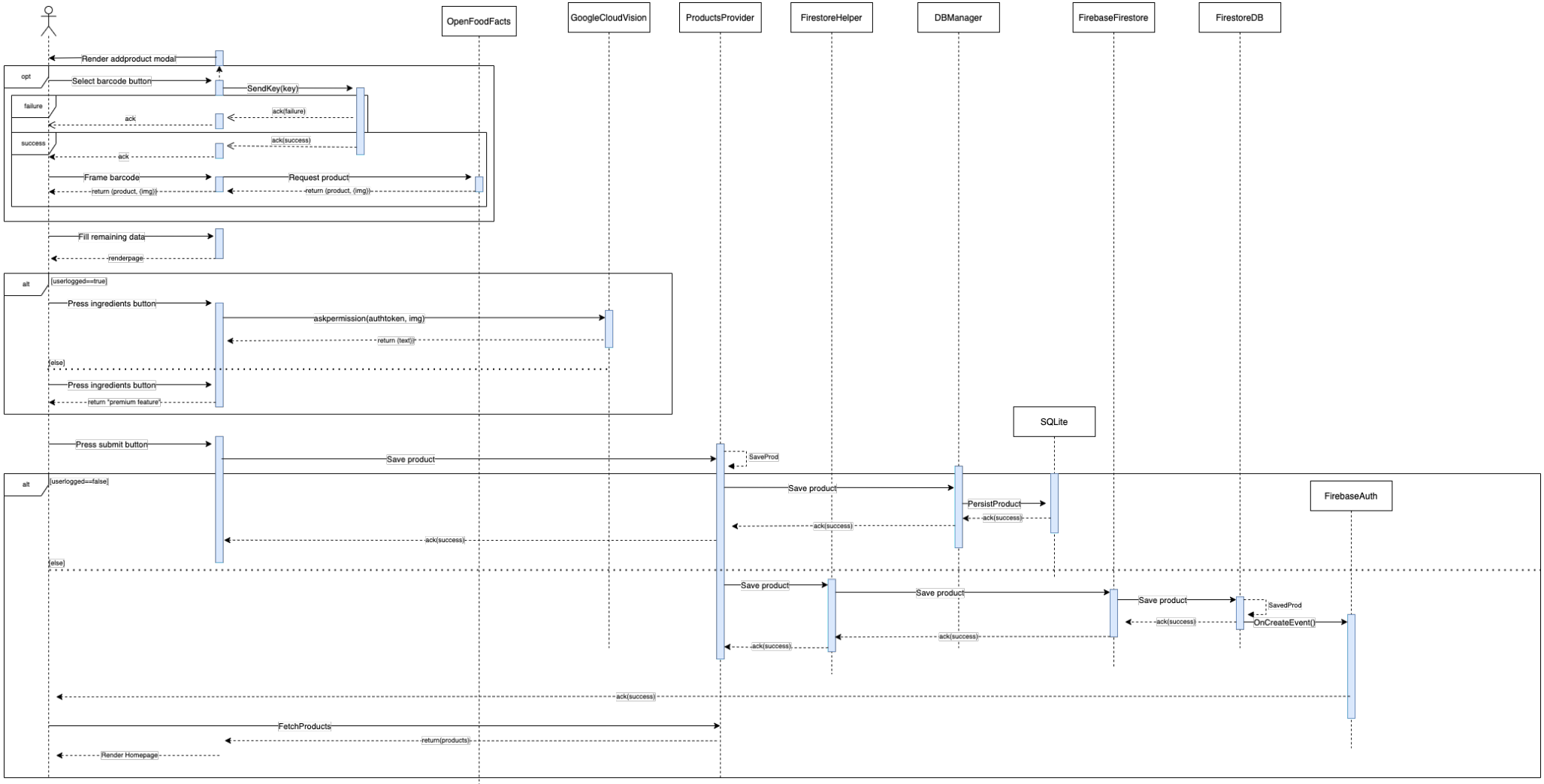


Figure 3.4: User adds a product

3.5 Selected architectural styles and patterns

This section clarifies the architectural style and architectural pattern adopted in the design of the Expire App system. The main difference between the two is that an architectural style is the application design at the highest level of abstraction, it is a name given to a recurrent architectural design; while an architectural pattern is a way to implement an architectural style, it is a way to solve a recurring architectural problem related to it.

As stated before to design the application a three tier architecture has been used. It is a client-server architecture in which the functional process logic, data access, computer data storage and user interface are developed and maintained as independent modules on three levels:

- **Presentation tier**;
- **Logic tier**;
- **Data tier**;

In this type of architecture there is the thin client, that is the device that requests the resource, equipped with a user interface in charge of the presentation-level functions. Then, there is the application server, also called middleware, in charge of providing the resource and communicating with the server database, which stores the data used by the application. The main advantage of the three-tier architecture is the logical and physical separation of functionality. This allows each tier to run on a separate operating system and server platform that best suit its functional requirements. Compared to one or two tier architecture, this allows for **faster development** as each tier can be developed simultaneously by several teams. Furthermore, this separation of levels allows programmers to use the latest and best languages and tools for each level. Moreover, any tier can be scaled independently of the others as needed and if one tier is interrupted, it will be less likely to impact the availability or performance of the other tiers, so this architecture also has a positive impact on **scalability** and **reliability** as well. Since the presentation layer and the data layer cannot communicate directly, a well-designed application layer can function as a kind of internal firewall, preventing some malicious attacks and ensuring **greater security**.

3.6 Other design decisions

In order to store informations, when a user uses Expire App without registering, a relational DB(Sqlite) has been used. In the figure below the ER diagram and the Relational model.

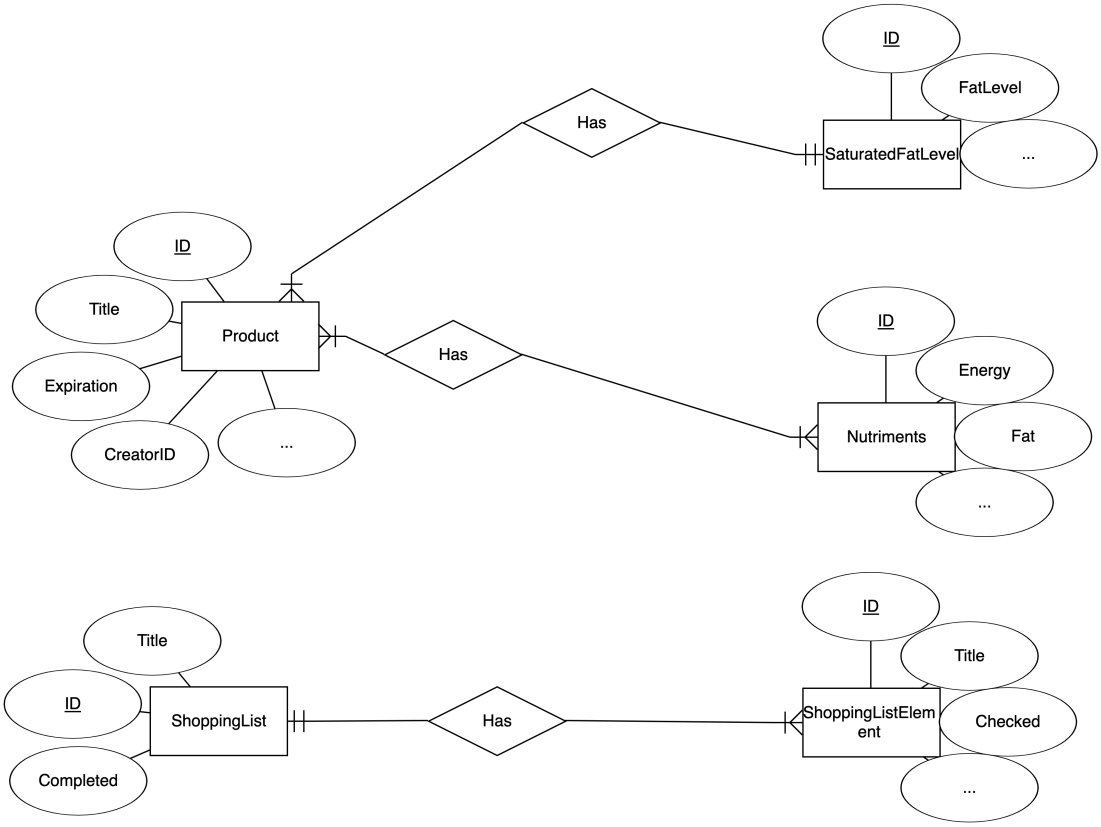


Figure 3.5: ER diagram

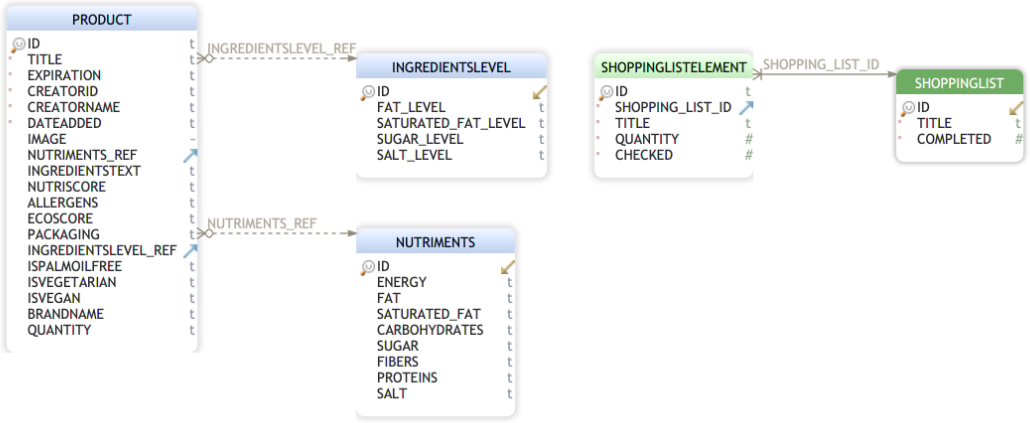


Figure 3.6: Relational Model

Chapter 4

User interface design

4.1 General Design

In this section are mentioned further details on the design of the various UIs. User interfaces must conform to a uniform, intuitive and user-friendly design, that doesn't require the reading of detailed documentation to be used. They should comply with the guidelines provided by Google Material Design to unifies the user experience across platforms, devices, and input methods. Text and important elements, like icons, should meet legibility standards when appearing on colored backgrounds, across all screen and device types.

4.2 Mobile Interface

Design is a crucial topic in developing a mobile application. A key issue when conceiving a design is thinking that having a "nice-looking" app is enough to achieve it. The first error while approaching the design phase is not to put the user at the centre. When software is intuitive, users have a more pleasant experience and tend to use it again while, if they find difficulties, their painful experience will make them abandon the application.

The first things to understand during the design phase was how users would have moved inside Expire App. We wanted to group the various elements in characterizing our offer by domain, to make as most intuitive as possible for users to add for eg., a product, or view a recipe. To achieve our objective, we divided the application into five main sections:

- **Recipe screen:** containing the list of recipes based on the products that the user has
- **Shopping List screen:** containing lists of products that the user can insert
- **Home screen:** containing the list of products that the user has, it gives also the possibility to add new products

- **Statistics screen:** containing some statistical infos about the products that the user has (for eg. the quantity of salt in products, quantity of fat, etc..)
- **User info screen:** containing profile informations, and gives to the user the possibility to change his/her display name, leave a family, join a family, and logout.

To let the user navigate among these sections, we used a Tab selector, which provides some buttons on the bottom of the display, where each button corresponds to a single screen.

Interactive components: One of the main goal in developing Expire App, is to create a user friendly interface, so our team decided to put particular focus on how interactive elements, like buttons and cards, should behave. To achieve this, as already stated Expire App is developed as well as for different platforms, but also for different screen resolutions and sizes, including landscape versions. Next sections will show the screens related to each of the views listed above.

4.2.1 Authentication screen

The authentication screen, containing the forms to login and register in the service, is the one that appears when the user opens for the first time the application (after an initial onboard screen). In the login section, as shown in the below figure, if a user is using an iOS device, he/she could also login/register with Apple ID, or even continue without registration (possible also for Android OS users). In the registration screen, there is also the possibility to register with already having a family ID, in this way once the user enters in the application, on the homepage he/she will have could view the products of all family members.

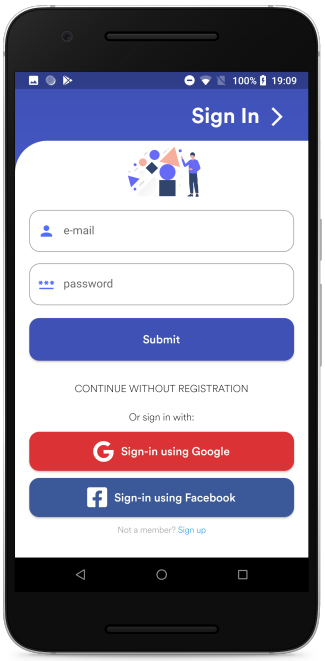


Figure 4.1: Mobile Login page

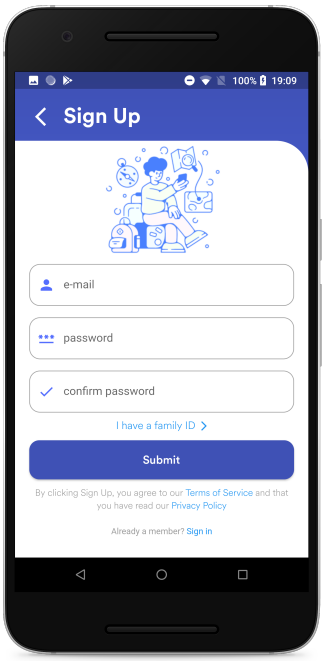


Figure 4.2: Mobile Registration page

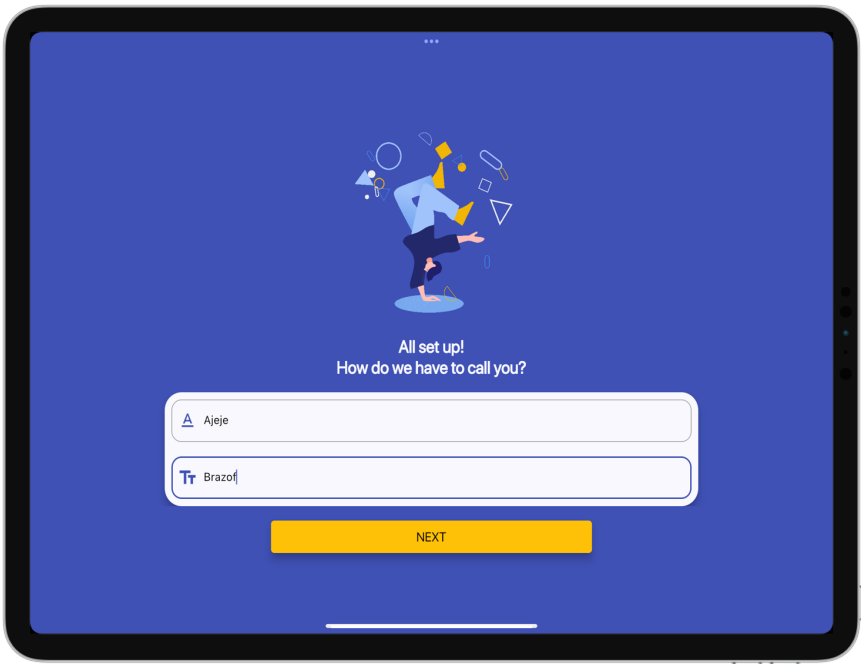


Figure 4.3: Tablet set display name page

4.2.2 Home screen

The homepage is designed to display all the products of a user (if he/she has any), or the products also from other family members if he/she belongs to a family group. The screen also displays some useful information about the product, for example the expiration date. There is also the possibility to add new products, by pressing the "+" button, adding them by using barcode or by simply adding them by name. In addition, by pressing on a product, present in the product list, there is the possibility to view detailed information about that product, such as nutritional values, ingredients, etc.

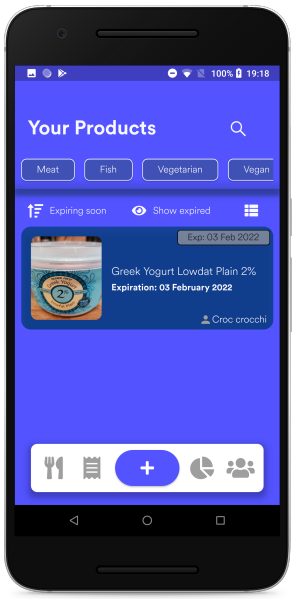


Figure 4.4: Mobile product list screen

Figure 4.5: Mobile add product screen

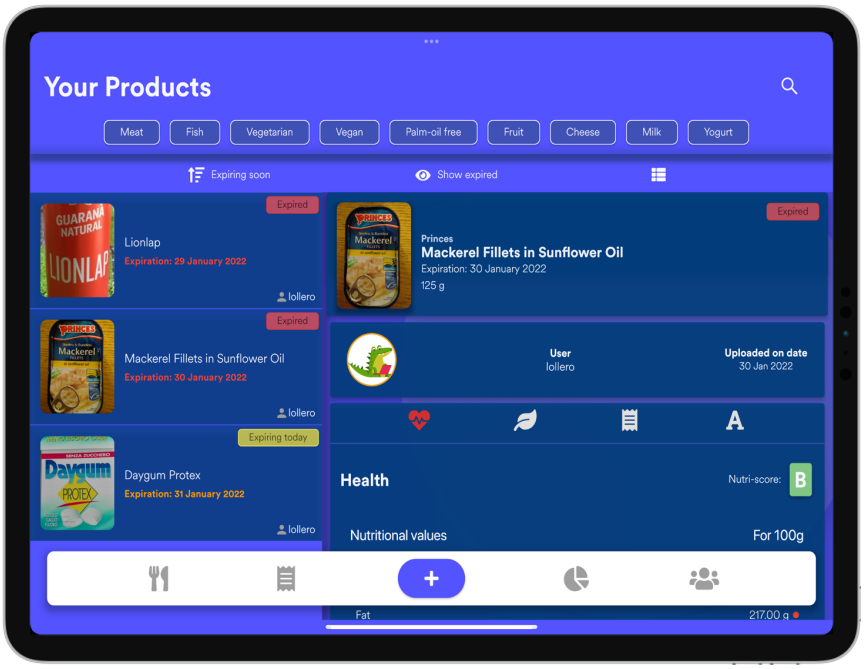


Figure 4.6: Tablet display product and details screen

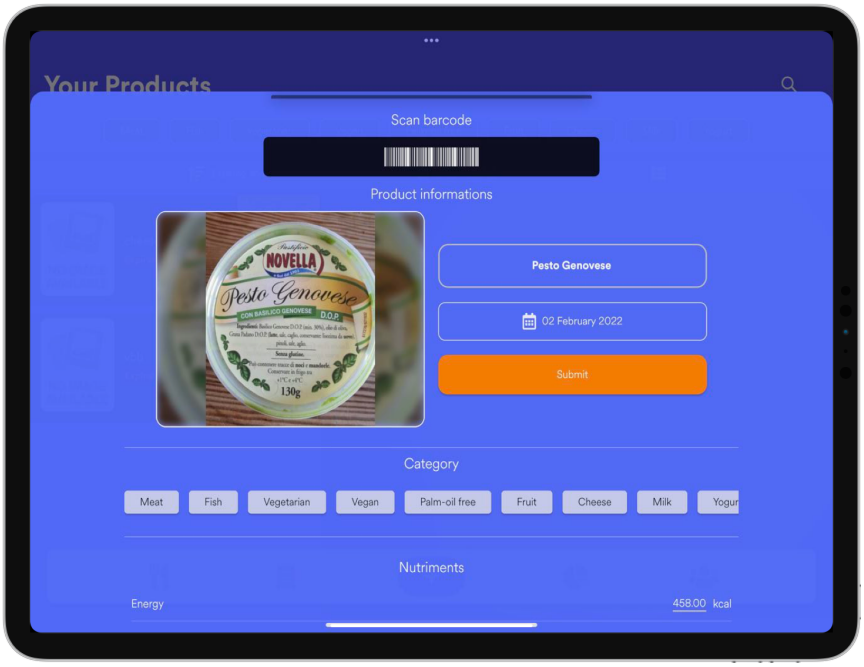


Figure 4.7: Add product view from ipad landscape

4.2.3 Recipe Screen

This section shows a list of recipes based on the products that users have. By scrolling down the page, users can explore more and more recipes. By tapping one of those recipes another screen is visualized, with the detail of the recipe: ingredients, preparation steps, preparing time, and servings.

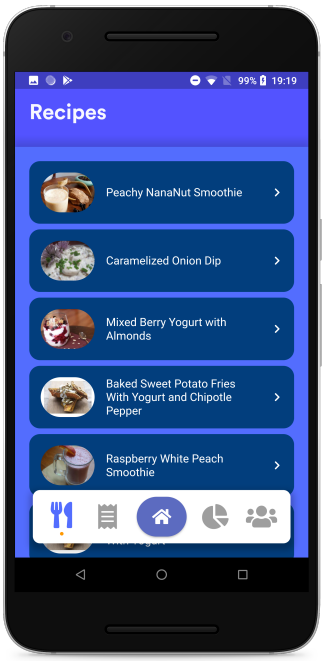


Figure 4.8: Mobile recipe screen

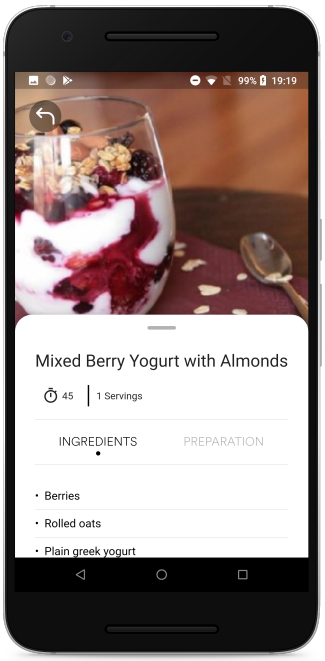


Figure 4.9: Mobile recipe detail screen

4.2.4 Shopping Lists Screen

The shopping lists screen manage, as the name suggests, the possibility to create lists with products. Those lists are useful when a user goes for grocery, and they're helpful to remind about products to buy. In top right part of the screen, there's a button named "Add List +", once tapped create a new list, where users can add products.

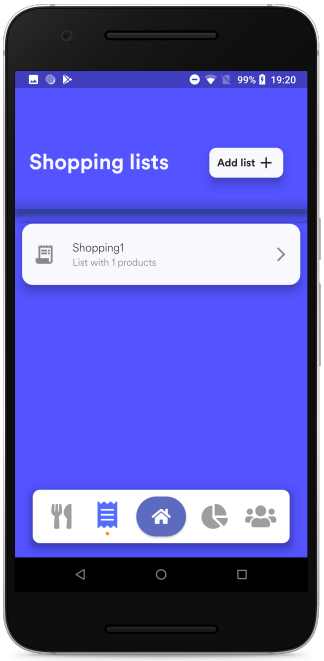


Figure 4.10: Mobile shopping list screen

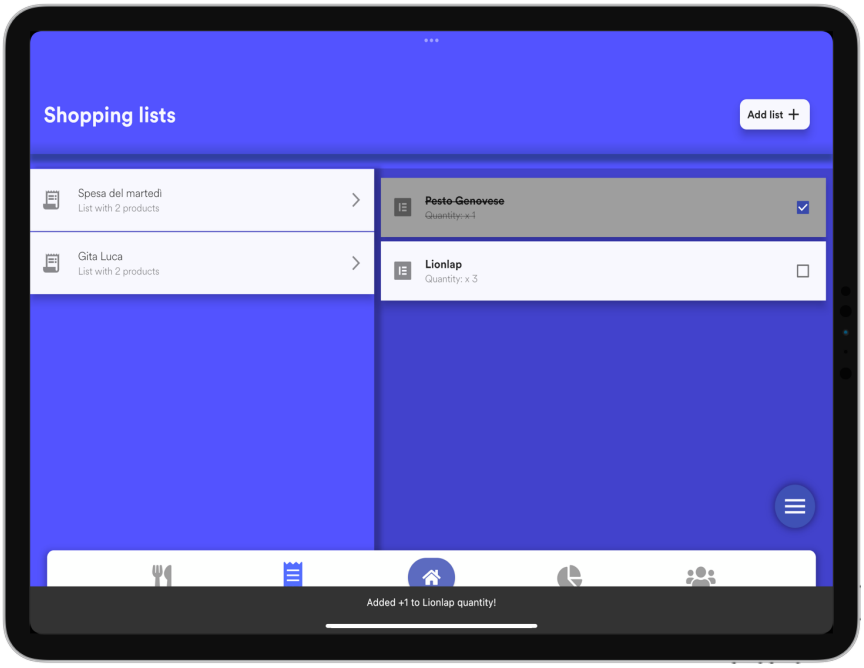


Figure 4.11: Tablet shopping list screen

4.2.5 Statistics Screen

This screen displays statistical data regarding products nutritional values. In particular there are 4 donut charts, representing the overall quantity of sugar, fat, saturated-fat and salt in the products. Every chart represents, for each value, three scores: "High", "Moderate" and "Low".

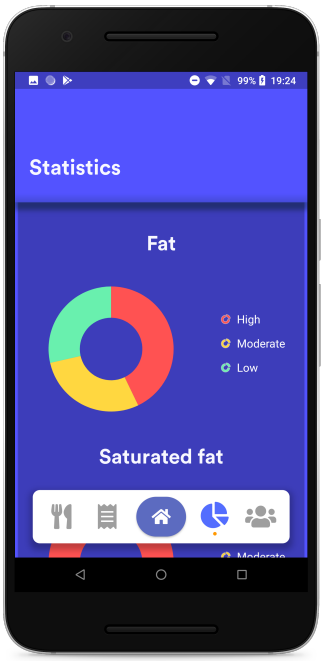


Figure 4.12: Mobile statistics screen

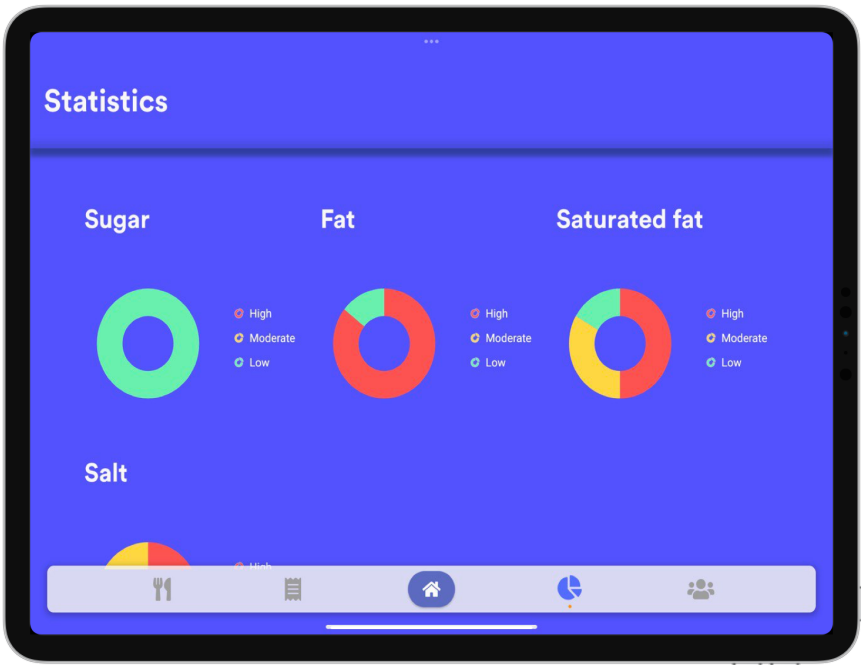


Figure 4.13: Tablet statistics screen

4.2.6 User Info Screen

The user info screen is the last that users can access through the tab selector. It shows the profile image of the user at the top, with his/her nickname and email. Below the screen is divided in two main subsections:

- Account: it contains two buttons, "change name" and "delete account", that gives to the users the possibility to change their display name and delete their account.
- Family: this subsection manage the synchronization with family members, and contains 4 buttons: "share family" that give to the users the possibility to generate a qr code and give to the other users the possibility to join their family; "leave family", once pressed, pop up an alert dialogue to ask users if they really want to leave the family group (whether they're a part of it); "family members" open a new screen and shows the list of users of a family; "join family" pop up a dialog where users have the possibility to join a family group by scanning a qr code or inserting manually the id of a family group.

Finally at the bottom there are some other buttons and finally the log out button.

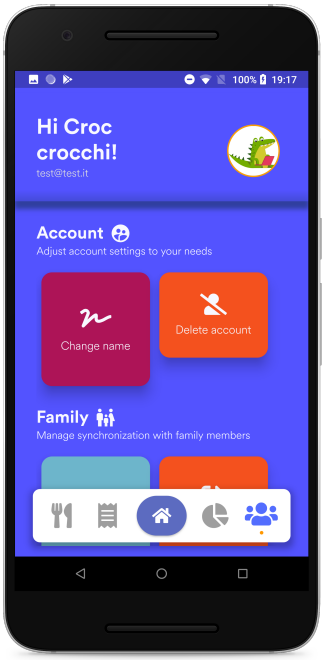


Figure 4.14: Mobile user info screen 1

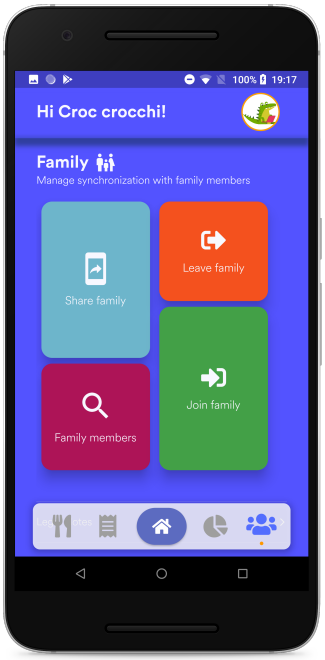


Figure 4.15: Mobile user info screen 2

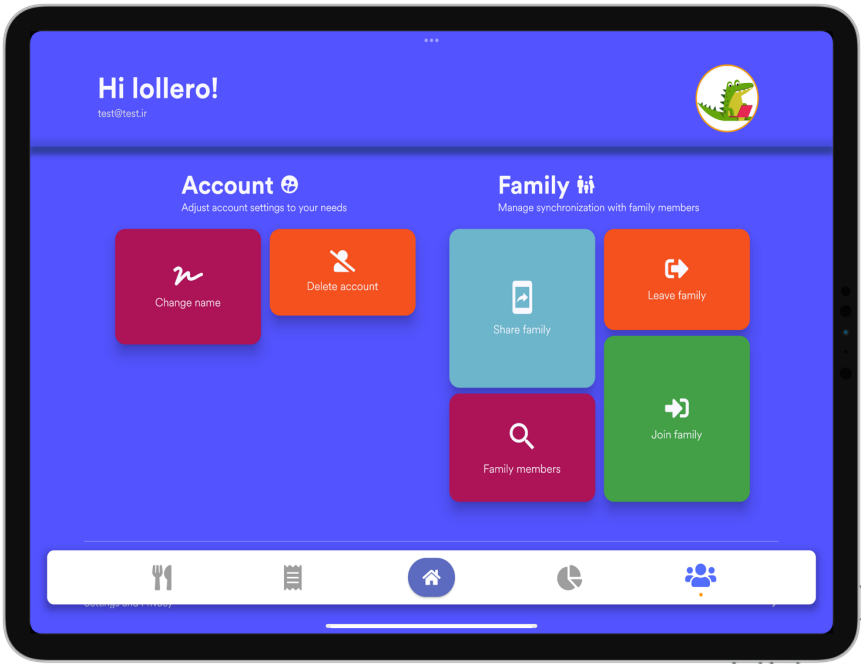


Figure 4.16: Tablet user info screen

4.3 Mobile Visual Flow Charts

Following figures display action flows, called *mobile app visual flow charts*, that represents the main action flows that a user can make.

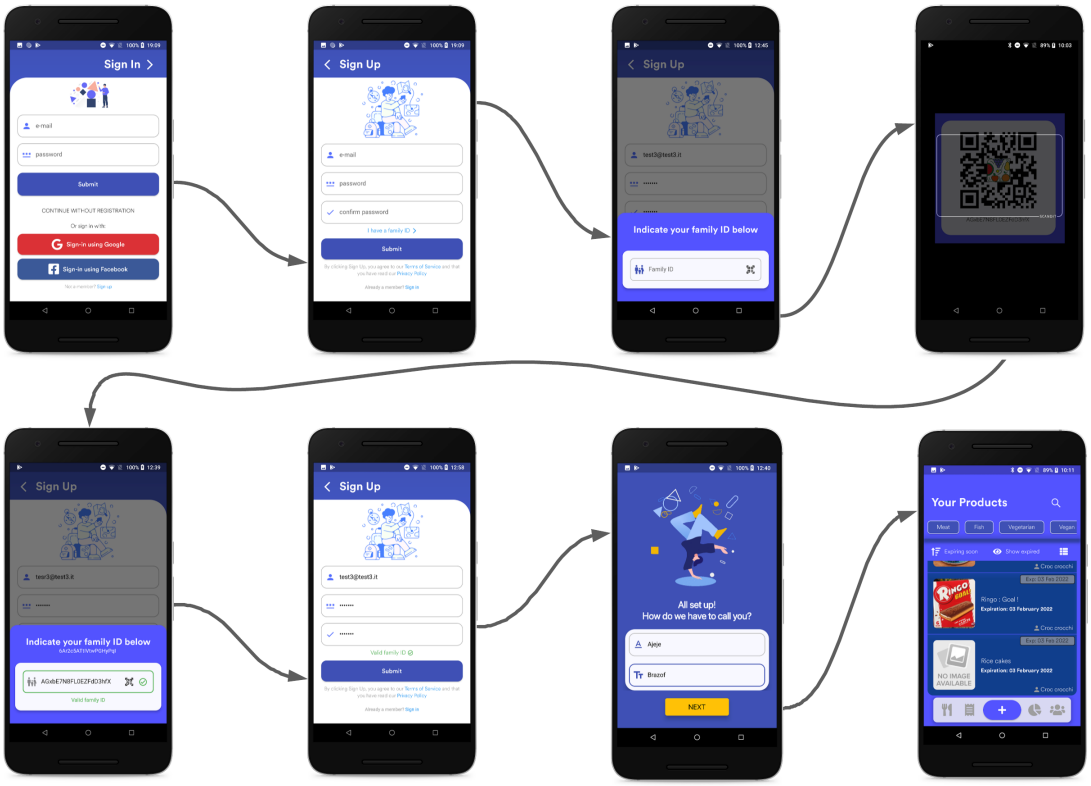


Figure 4.17: Signup flow

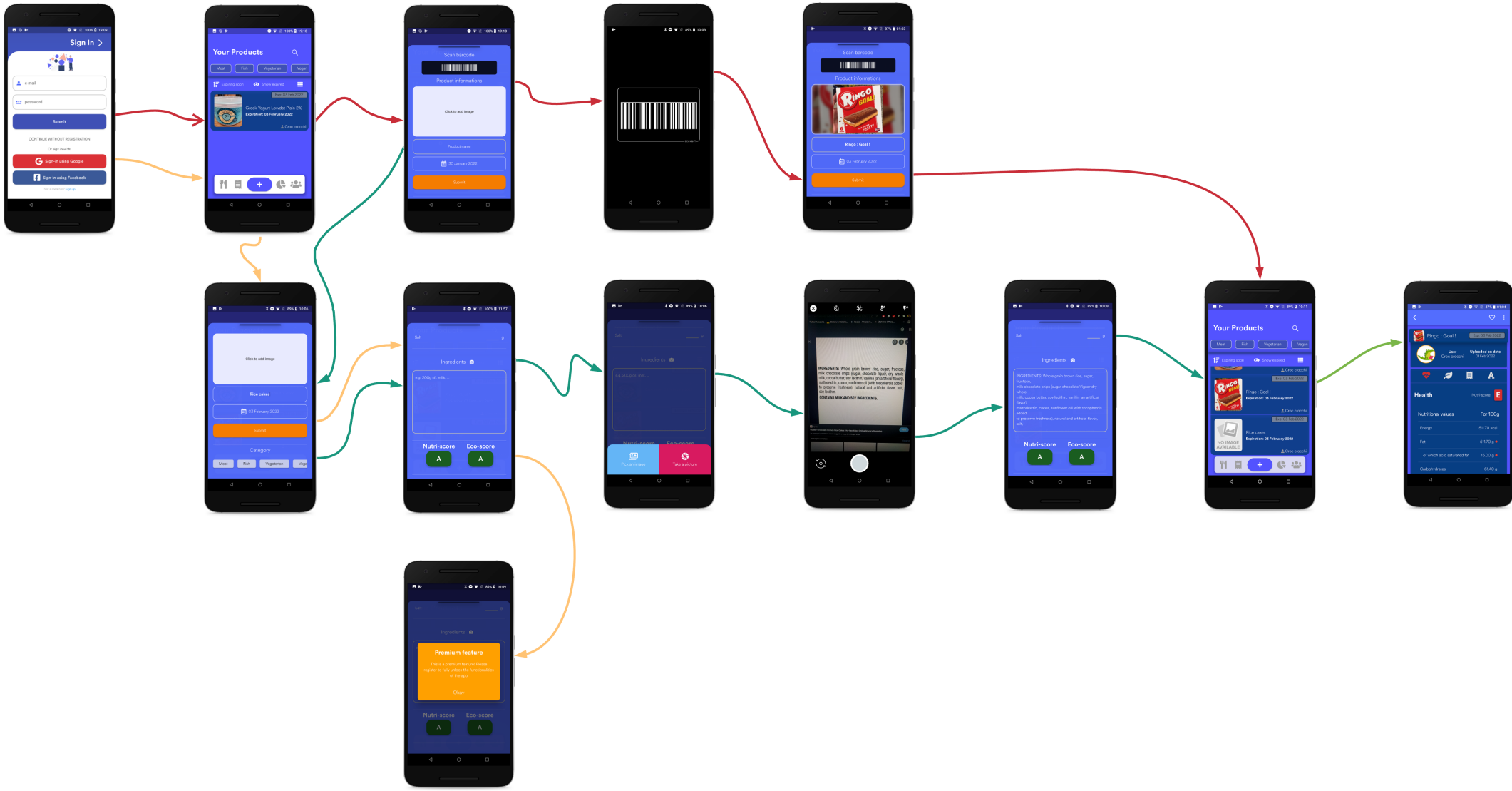


Figure 4.18: Add product flow

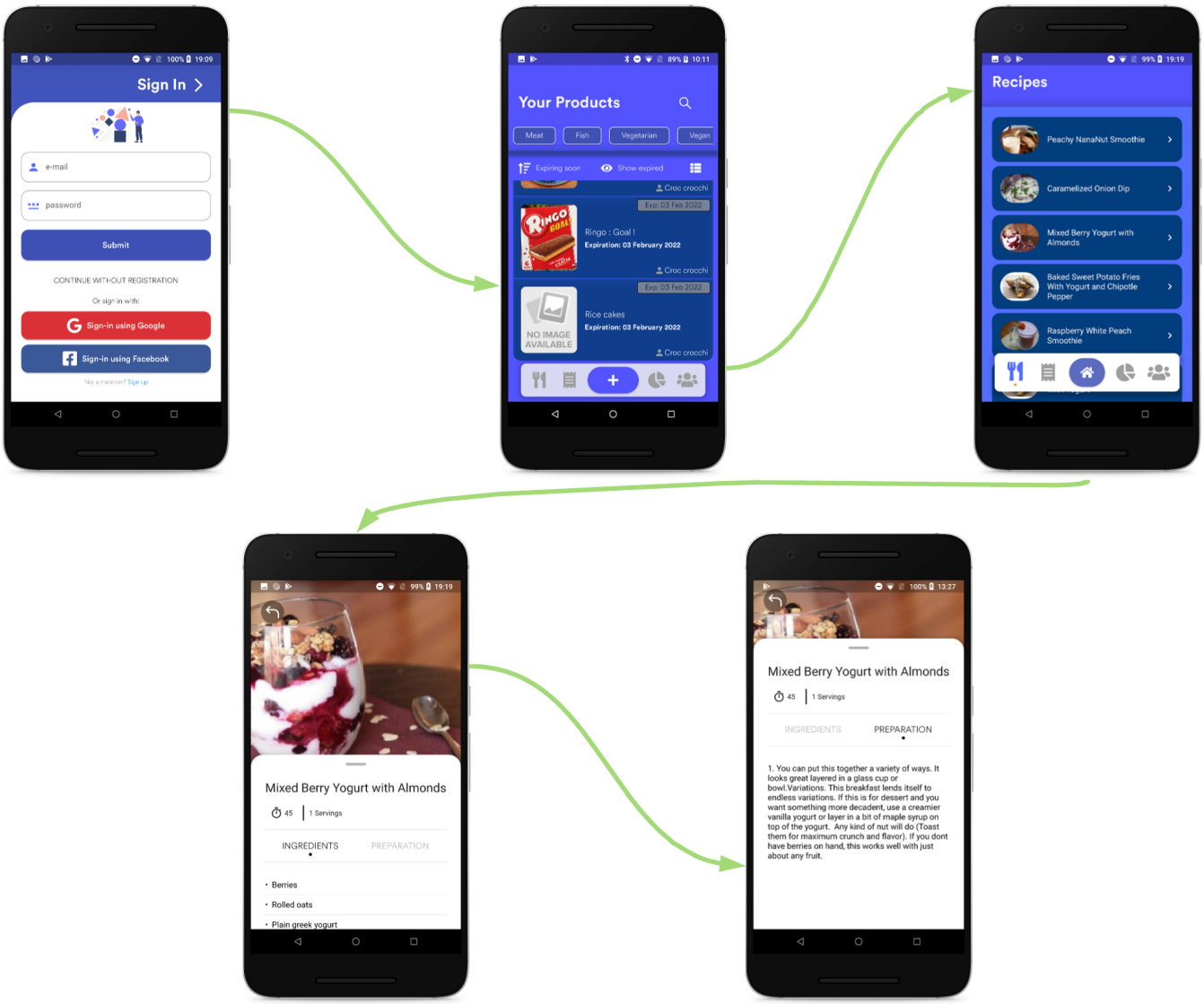


Figure 4.19: Recipes flow

Figure 4.20: Profile flow

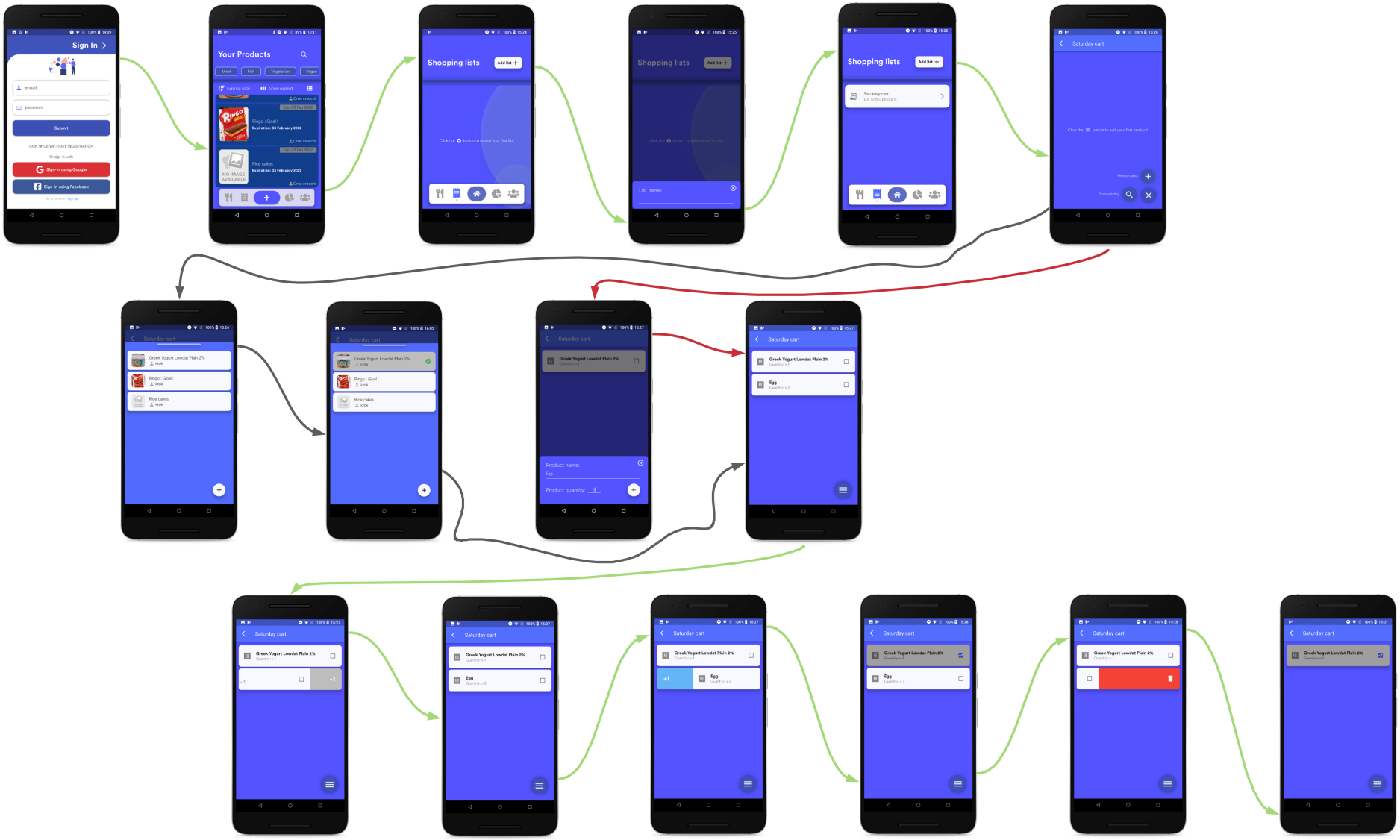


Figure 4.21: Shoppinglist flow

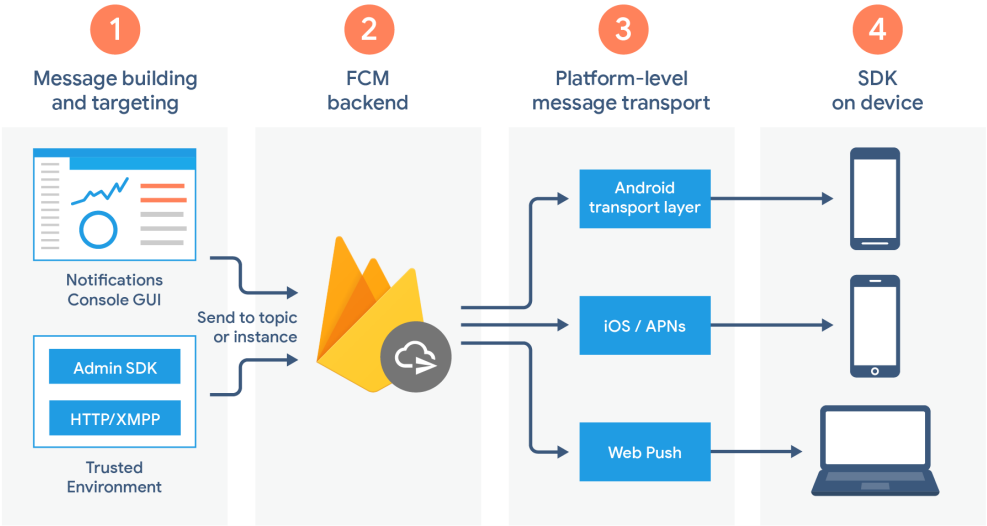
4.4 Push notifications

In our application, having push notifications is fundamental not only to remind the user that some food is expiring on the fridge but also to help communication and interactions among users belonging to the same family group. Each notification is delivered directly to all the users belonging to the same family when one of the four following actions take place:

- A user joins your family
- A user adds a product
- A user adds a shopping list
- There are some products expiring in the current day

4.4.1 Implementation

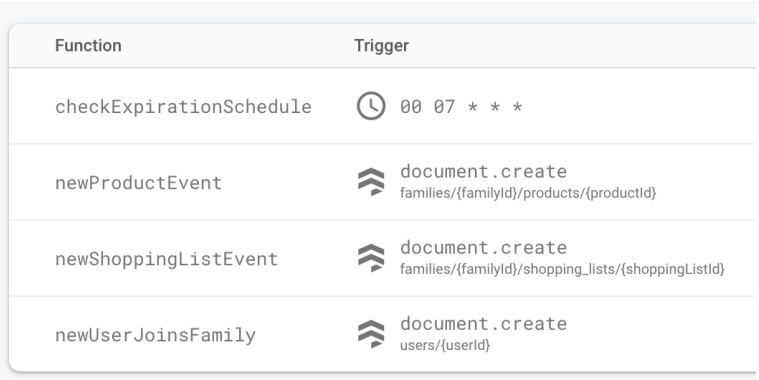
Push notifications are centralized on Google and Apple servers for security reasons. For this reason, our application makes use of the **Firebase cloud messaging** service. This allows us to avoid implementing communication towards APNs which is all handled by Firebase making the notification service almost completely transparent to the developer.



Each physical device is registered to a 'topic' which corresponds to the family id and a topic name is essentially an identifier for a pool of observers waiting for possible notifications.

To sent notifications programmatically, we made use of *Firebase functions*. We attached three different listeners on new additions on shopping lists, products and users. There

is also a scheduled function programmed to run each day at 7a.m. and inform users is a product is expiring that day (Crontab notation).



4.4.2 Notifications

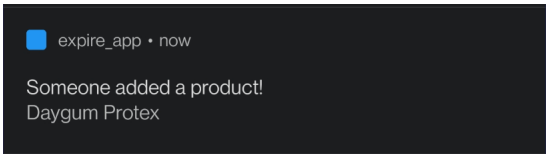


Figure 4.22: New product added

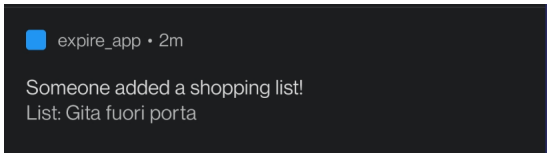


Figure 4.23: New shopping list added

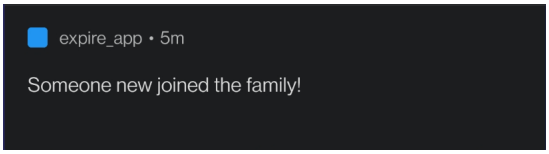


Figure 4.24: New family join

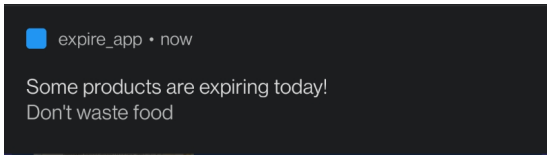


Figure 4.25: Expiration reminder

Each notification is attached to its corespondent listener (Firebase trigger function) *onCreate*.

New product

```
exports.newProductEvent = functions.firestore
    .document("families/{familyId}/products/{productId}")
    .onCreate((snap, context) => { ... }
```

Listing 4.1: My Javascript Example

New shopping list

```
exports.newShoppingListEvent = functions.firestore
    .document("families/{familyId}/shopping_lists/{shoppingListId}")
```

```
3     .onCreate((snap, context) => { ... }
```

Listing 4.2: My Javascript Example

New user joins family

```
1 exports.newUserJoinsFamily = functions.firestore
2     .document("users/{userId}")
3     .onCreate((snap, context) => { ... }
```

Listing 4.3: My Javascript Example

Scheduled timer for recurrent notification

```
1 exports.checkExpirationSchedule = functions.pubsub.schedule("00 07 * * *"
      )
2     .timeZone("Europe/Rome")
3     .onRun(async (context) => { ... }
```

Listing 4.4: My Javascript Example

Chapter 5

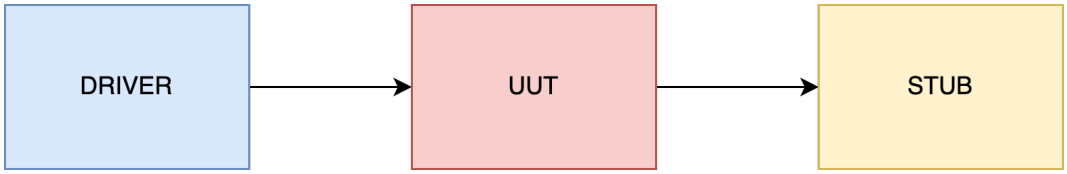
Implementation and Testing strategy

Testing is one of the most important parts of a software development. This phase allows to go through all the possible cases that can take place when the project goes live. Over 75 tests have been done, only the main components have been tested. Several external libraries are also used:

- *mockito*
- *firebase_auth_mocks*
- *build_runner*
- *fake_cloud_firestore*
- *google_sign_in_mocks*
- *firebase_storage_mocks*

5.1 Unit testing

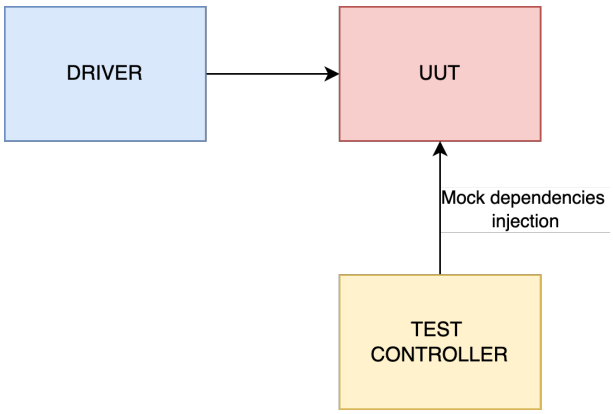
We used a typical stub-driver approach to test the functionalities of main components of our applications.



The tested components have been

- FirebaseAuthHelper
- FirestoreHelper
- ProductsProvider
- ShoppingListProvider

To mock their dependencies and create stubs, we used *Mockito*, a powerful library that helps generating stubs. In order for those components to be testable, we had to perform a refactoring to make them compliant to testable standard structure. One of the most important thing was to be able to inject dependencies via the constructor to create stubs for internal dependencies. We managed to accomplish this by dynamically resolve ad runtime the possibility to instantiate the object with some or all its dependencies mocked.



All of those components have been heavily tested in all of their use cases. We manage to obtain 60 tests (number also given from their complexity and dimension).

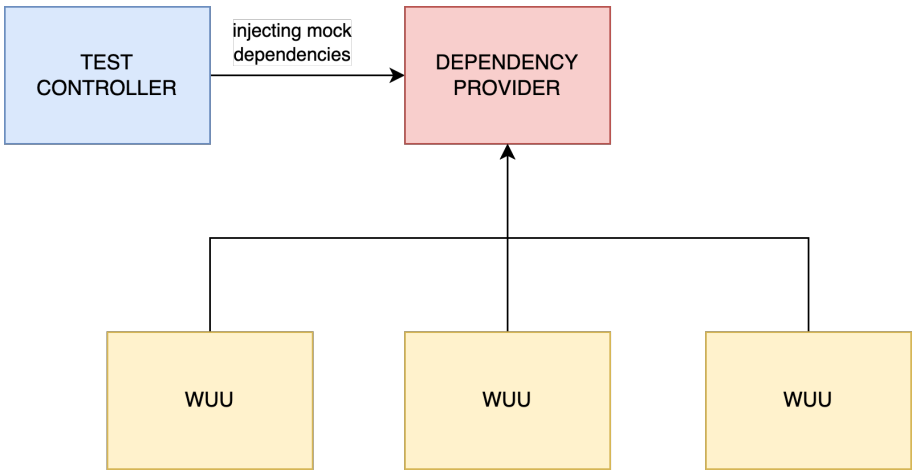
To perform assertions and evaluate tests we used the implemented flutter library called *flutter_test.dart*.

5.2 Widget testing

Widget testing aims to test UI behavior to certain actions and it's not about functionalities and logic like the unit testing. This type of testing is made available thanks to a *Widget tester* library embedded in the flutter SDK.

The main widget tested were the sign up, sign in and the add modal product.

There it presents once again the need of using stubs because of internal dependencies inside screens and widget. The difference with the unit testing is the impossibility to perform a large scale refactoring since implementing an injectable constructor would be a tedious and very long task. We solved this by using a *dependencies provider* that is responsible for providing dependencies to each widget requiring it. By doing this, each widget has dependencies fetching completely transparent while we can inject the dependencies provider with mocks to be served to widgets.



5.3 Monkey testing

Monkey testing is a technique where the user tests the application by providing random inputs and checking behaviour, or seeing whether the application or system will crash. This test has been done using *Firebase Test Lab*, that is an application-testing infrastructure offered by Google Firebase. It permits to define different device configurations' collection on which application can be tested. Every device configuration collection works on a specific range of device types, language, and orientation configuration options. Every test is performed via virtual and physical devices located at the Google data center. When a test is run against devices and configurations selected, *Test Lab* runs the test and display the results as a **text matrix**.
Devices x Test Executions = Test Matrix

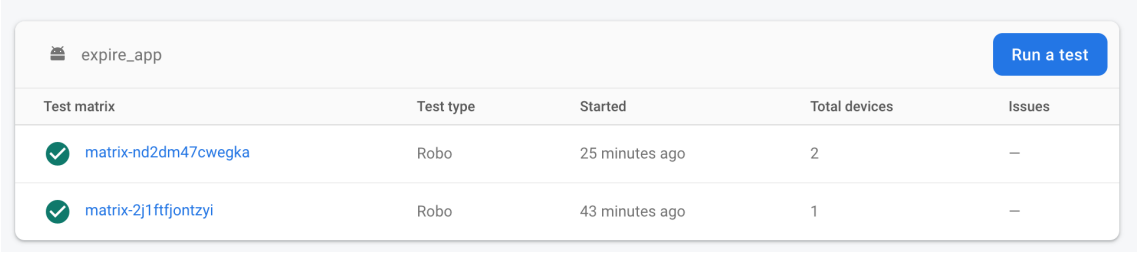


Figure 5.1: Matrix test

Another test performed with Test Lab is so called **robo test**. It analyzes the structure of app's UI and explores it methodically, automatically simulating user activities. This test captures log files, saves a series of annotated screenshots, and then creates a video form those screenshots to show the simulated operations that it performed.

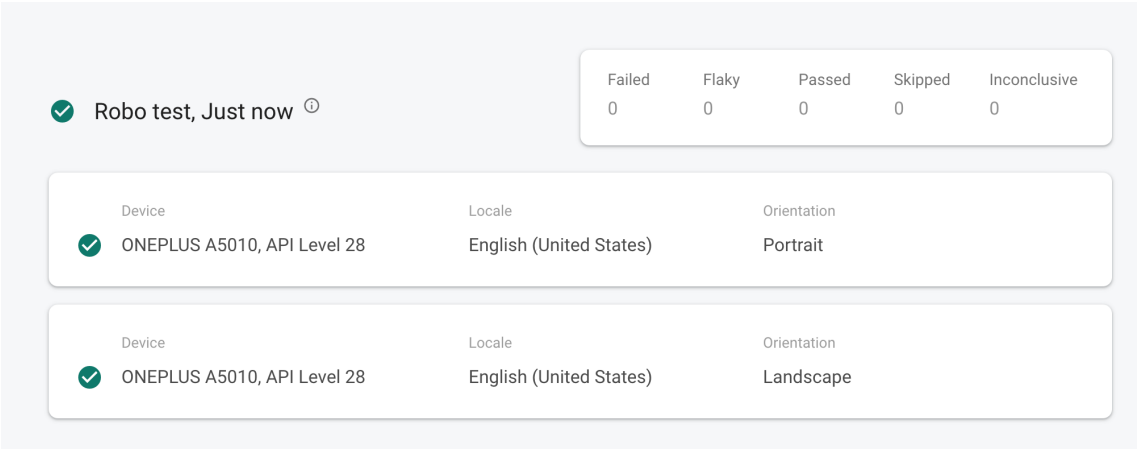


Figure 5.2: Robo test

Chapter 6

Future development

6.1 Future Plans

As future development, we would like to fully take advantage of cross platform development offered by Flutter framework, to do so implement also the notification system, explained in section 4.4, into iOS and iPadOS environments.
Moreover our team has in plan to exploit more the interaction with the final user, for eg. by implemeting a favourite section, where the user can view recipes added to favourite