POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

MacroTracker

Design Document
Design and Implementation of Mobile Applications

Alessandro Sironi

Tommaso Sprocati

Students: 10680296-232828
10666918 - 232789
Professor: Luciano Baresi
Academic Year: 2022-23

Contents

1 | Introduction

1.1. Project Description

The mobile application developed is called **MacroTracker**. The application consists in a fitness-focused project that allows athletes to track and log food intake and macronutrients in order to compile statistics of one's eating habits. Therefore, to help the user achieve their fitness goals by collecting quantitative data on their diet. The main functionalities are:

- The ability for the user to enter their own **nutritional goals**.
- The **tracking** of the daily diet, aided by the possibility to insert the consumed food through either a **manual insertion**, or the **scan of a product's barcode**. Regarding this function, we identified an open-source API service called ***OpenFoodFacts*** (`https://it.openfoodfacts.org`) that contains a large dataset of off-the-shelf food products information.
- The ability for the user to receive a **notification reminder** to help them remember to log their food intake of the day.
- Showing useful information through **graphs** to monitor progress.
- The integration with **Apple HealthKit** in order to take advantage of the ecosystem to store data, so that it is available from other apps.

The project has been developed in **Flutter**. The supported devices are phones and tablets, both iOS and Android devices. The main focus has been placed on iOS and iPadOS devices.

1.2. Features

1.2.1. Diary Summary of the day

The user can see at a glance the day's situation regarding their food intake and diet and, via a calendar view, can change the date to inspect other days' situations.

1.2.2. Quick add for water and coffee assumption

Through easy and quick buttons, the user can add or remove a water or coffee assumption for the day.

1.2.3. Goal setting

The user can easily modify their diet goals to reflect the changes in their training and lifestyle. The Goal page presents to the user also a few information and tips to help the athlete take conscious decision about their diet goals.

1.2.4. Saving common foods for quick adds

The user can create ad-hoc common foods that are recurrent in their diet for a quicker add in the daily assumption.

1.2.5. Modify an insertion

The user can always modify or delete an insertion, whether if they decide to change the values or if they input data by mistake.

1.2.6. Multiple Device Synchronization

By logging in with the same account in multiple devices, the user has all the data synchronized via the cloud, so that they can update their data from multiple devices and have them synchronized.

1.2.7. Notification reminder

The user will be reminded to use the app through a scheduled notification that takes place near the end of the day. The notification is scheduled only if the user actively uses the app, so that if they decide to pause the use of the app, they're not overloaded with notifications.

1.3. Purpose

The purpose of this document is to provide a description of the architecture and design of the application, focusing on the architecture adopted and the decisions that drove the final design.

2 | Architectural Design

2.1. Overview

MacroTracker is a Flutter application, developed for iOS and iPadOS devices, with support for Android devices. Its architecture relies on Firebase as a safe and secure **backend** for authenticating the users and store the users' data.
The food database to which the application requests the food nutritional values is **OpenFoodFacts** - `https://it.openfoodfacts.org`. The *open data* and *open source* database offers APIs that let developers take advantage of the vast amount of labeled food products.
At last, the app has been thought out to interact with Apple Health's ecosystem. The values saved in the app are made available as Apple Health entries on supported devices.

2.1.1. Authentication

MacroTracker logins and signs up users via Firebase Authentication (`https://firebase.google.com/docs/auth`). The supported sign-in method is **Email/Password**. Users are assigned an unique identifier, that will be used in the Firestore Database for identifying the user's collection.

2.1.2. Data Management

The application relies on an online and in-cloud database: Cloud Firestore (`https://firebase.google.com/docs/firestore`). The database contains data related to the user's basic information, the user's nutritional goals, the food entries for each day, as well as the diet foods saved for quicker adds.

2.1.3. Data Structure

The database structure is defined as follows.

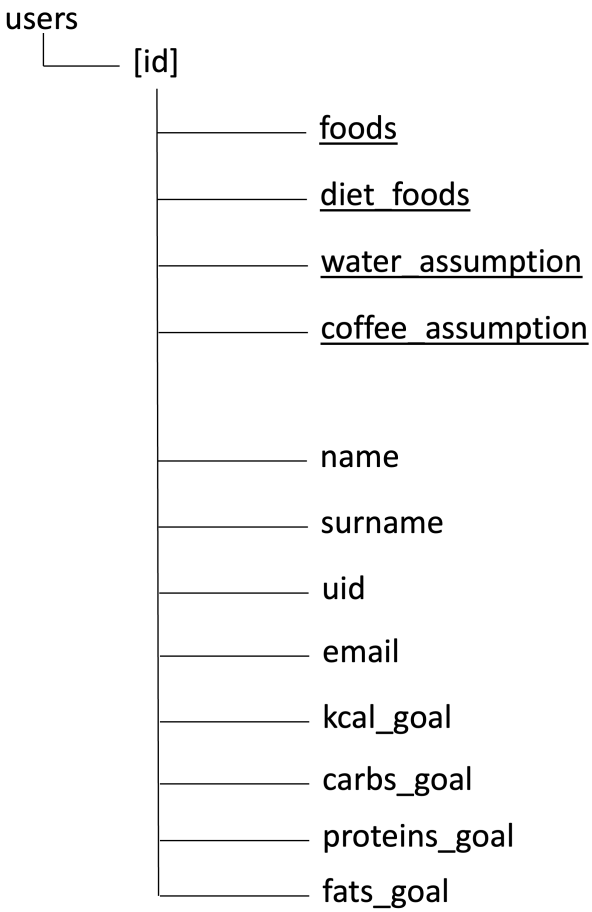


Figure 2.1: Database structure

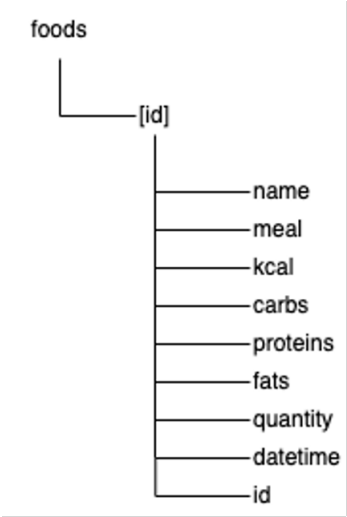


Figure 2.2: Foods Database structure

Note that underlined elements in the structure represent sub-collections, whereas the non-

underlined entries are fields. Each user has access to its own collection, identifiable by the UID assigned by Firebase Authentication. Every user has the ability to log its daily diet, favorite foods, the amount of glasses of water and coffee drunk each day. The user can then retrieve old data to check the progress made, and-or to adjust mistaken entries.

2.1.4. Food Data retrieval

MacroTracker takes advantage of the largest open-source and open-data food database, that contains more than 2.7 million of products from 150 countries. The user has the ability to find in the database nutrition facts of off-the-shelf products via a barcode scan. The database then returns the nutritional info of the desired product. Open Food Facts provides an API to interact with the database (`https://pub.dev/packages/openfoodfacts`).

2.2. Architectural Style and Patterns

2.2.1. Two-Tier Architecture, Model and View-Controller

The application has been thought out to use a two-tier architecture. The client portion is the app itself whereas the "Server" is itself the cloud-provided back-end of Firebase. The pattern chosen to organize the project itself is "View-Controller", as most of the data manipulation, creation and retrieval is handled through Firebase; widgets are built dynamically given the retrieved data.

2.3. Widgets

As a Flutter application, MacroTracker is built by several widgets that build the correct view and respond to the user's input.

2.3.1. Main screens and Navigation

The application has some main screens that are accessible from each other via a bottom navigation bar, present in (almost) all the screens:

- **Diary**: This is the main view for the application, and - provided the user is logged in - is the first screen shown at the start of the app.
- **Goal**: This page allows the user to set their kcal, carbs, proteins and fats goals. It also provides some basic information to guide the user through their decision.
- **Add**: Represented by a plus icon, this page lets the user make a new food entry by scanning a barcode, adding manually or from the diet.
- **Diet**: This view shows all the foods that the user has saved, to be logged more easily. The foods can be filtered, modified and added from this view.
- **Profile**: In this view, the user can modify their personal data, switch to dark/light

mode and log out.

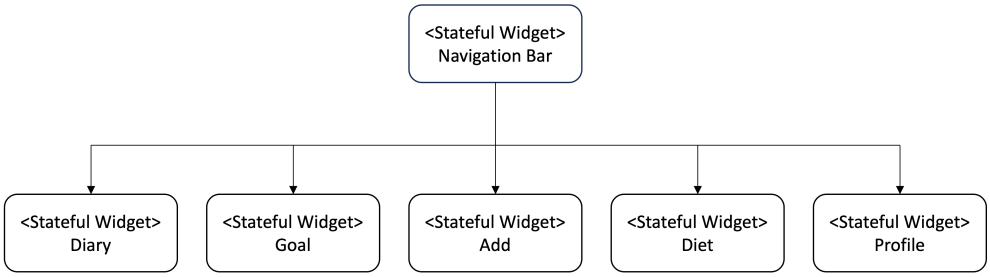


Figure 2.3: Navigation from the Bottom Bar

2.4. Barcode Scanner

A fundamental feature of MacroTracker is the ability to log commercial products quickly. This is achieved by scanning the product's barcode. The barcode number is then used as input with the OpenFoodFacts APIs described above, and the product's information is retrieved from the database. The plugin used by the app is **Flutter Barcode Scanner** - `https://github.com/FlutterFlow/flutter_barcode_scanner`. The plugin provides the logic code and the UI for the scan. The device's camera is activated and the user is guided to frame the barcode in the space. Upon a scan, the app registers the barcode and converts the numbers to a string. The scan works in different orientations and is flexible, so that the scan is seamless and easy to achieve.

2.5. Apple Health

The application has been developed from the start with Apple's Health ecosystem in mind. Apple Health is a comprehensive health and fitness platform developed by Apple Inc. It is an app available on Apple devices, such as iPhones and Apple Watches, and is designed to help individuals monitor and manage various aspects of their health and well-being. Apple Health serves as a centralized hub for collecting and organizing health-related data from various sources, including built-in sensors, third-party fitness trackers, and compatible health and wellness apps. The app can track a wide range of health metrics, such as daily steps, distance walked or run, heart rate, sleep patterns, calories burned, and more. In addition to tracking physical activity, Apple Health allows users to input and monitor other health data manually, such as weight, body measurements, and *dietary intake*. This enables users to have a holistic view of their health and fitness progress over time. MacroTracker seamlessly integrates with Apple Health, ensuring that the data captured by the app is automatically saved and contributes to the user's overall health profile. By storing this information in Apple Health, MacroTracker empowers users to gain a comprehensive understanding of their health, enabling them to analyze their macronutrient intake alongside other vital health metrics. This holistic approach enables individuals to make informed decisions about their diet, identify areas for improvement, and strive for a balanced and nourishing lifestyle. To achieve this, the apps uses the plugin **Health** - `https://pub.dev/packages/health` - a wrapper for HealthKit that:

- Handles the permissions to access health data.
- Writes health data to Apple Health's data store.

When a food is logged by the user, MacroTracker automatically writes to Health the calorie intake and the macronutrients intake. Obviously, a removal or a modification of an insertion is reflected in Health.

2.6. Notifications

The app provides the user the ability to receive a daily notification that reminds them to log their diet. The notification is useful to help developing healthy habits by keeping up to date the food intake, to let the athlete observe their progress. The notification has been thought out to not be intrusive, so the user will receive the daily notification only if they actively use the app. It empowers individuals to establish a consistent routine for logging their diet without feeling overwhelmed or burdened by intrusive alerts. The app's non-intrusive daily notification feature plays a vital role in assisting users to develop healthy habits, maintain consistency in tracking their diet, and stay motivated towards their dietary goals. The app uses the **Flutter Local Notifications** plugin - `https://pub.dev/packages/flutter_local_notifications` - to schedule and deliver the notification to the user. A local approach has been chosen, as there is no need for the backend to deliver this notification as it is static. The notification's component is called `Local Notification Service`, and the code is executed while using the app, scheduling a notification at the next evening (19:00). This hour has been chosen as it is close to dinner and should not interfere with social events.

3 | Dependencies

3.1. Plugins

The table provided below enumerates the noteworthy plugins that have been incorporated into the application as dependencies. Each plugin is accompanied by a rationale for its inclusion, outlining the purpose and justification behind its integration.

Plugin	**Use**
Cloud Firestore	Necessary for the usage of Cloud Firestore API.
Firebase Auth	Necessary for the handling of the user's credentials, login and sign up.
Flutter Local Notifications	Allows to receive a daily notification to remind the user to log their data.
Open Food Facts	APIs that handle the search of a commercial product through its barcode. Retrieves the product's information.
Health	Allows MacroTracker to interact with Apple Health's HealthKit. Used to save dietary intake to Health's datastore.

Table 3.1: Most significant plugins used and related description.

4 User Interface

4.1. Design of the app

The design of the mobile and tablet application has been realized with the help of FlutterFlow platform, in order to create the pages and choosing light mode and dark mode palette colours.

4.2. Mobile version

In the following sections are shown the screens of the **mobile** application. Every page will be shown respectively in dark and light mode.

4.2.1. Authentication Pages

These are the two initial pages that are displayed once the user opens the app the first time or if they're not logged in. The app firstly shows to the user the login page, then when clicking on the *Register* button, it displays the other one.

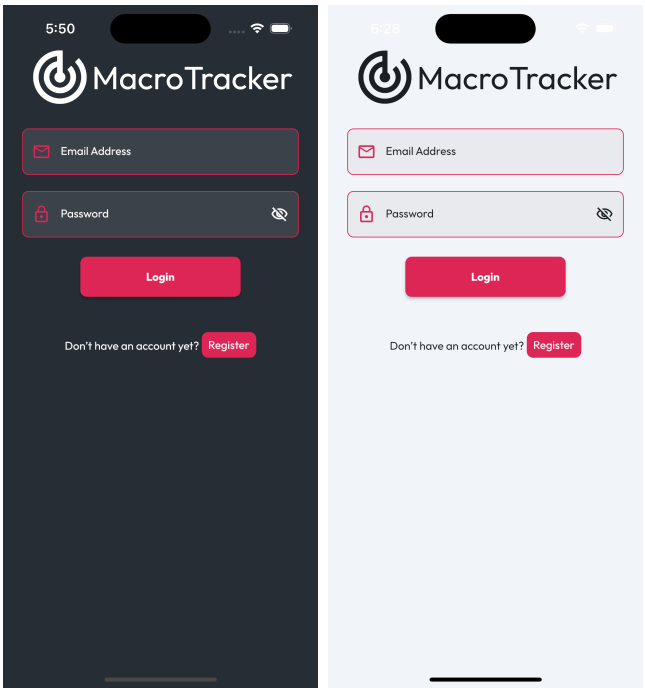


Figure 4.1: Login page

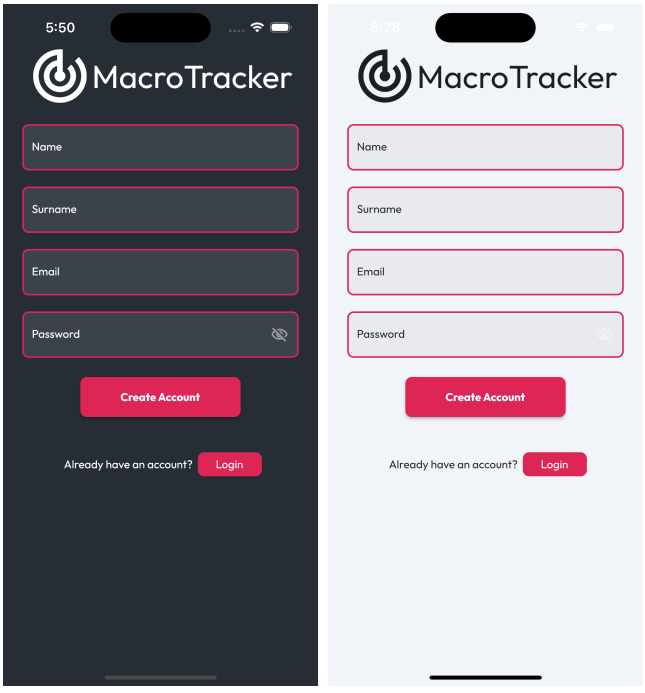


Figure 4.2: Register page

4.2.2. Diary page

This screen shows the main page of the app where it is displayed the current daily progress of the user diet over the fixed goals. It permits the user to add/remove glasses of water and cups of coffee drank in the day. Then it shows the foods inserted in the selected day. The calendar on the top enables the user to see the history of their diet assumptions; by clicking on the previous days the page is updated with the data corresponding to the day selected.

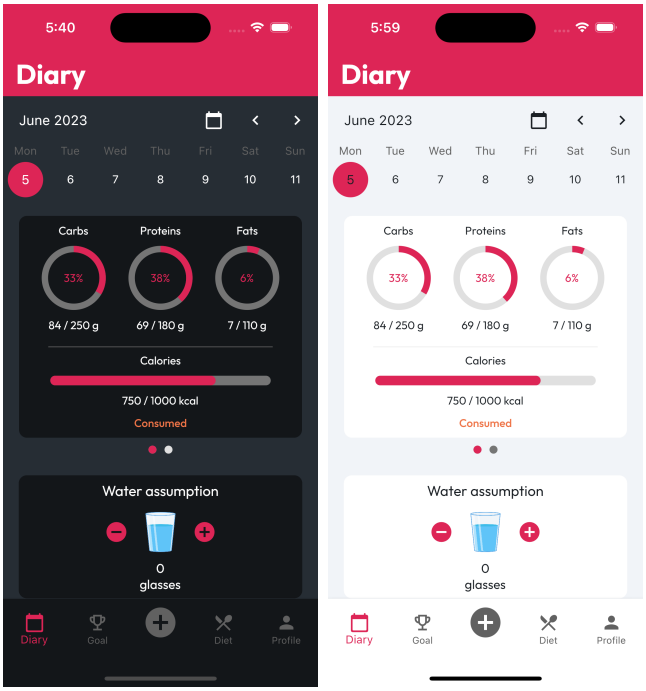


Figure 4.3: Diary page (1 of 2)

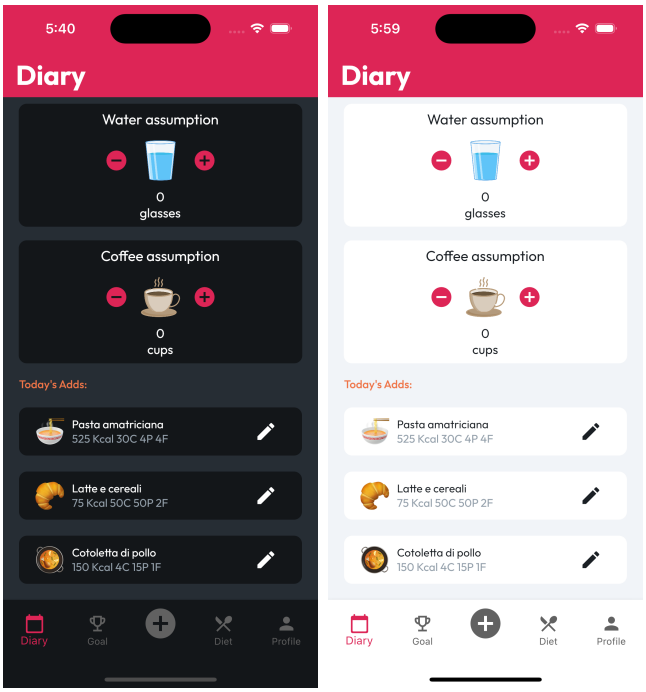


Figure 4.4: Diary page (2 of 2)

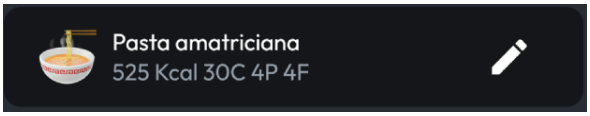


Figure 4.5: The single tile of displayed food shows the name of the food, the meal category (icon) and the macro nutrients of the food in a synthetic way.

4.2.3. Edit food page

This screen is shown when clicking on the *pencil* icon on the single food and permits the user to edit all the food properties or to delete the food clicked.

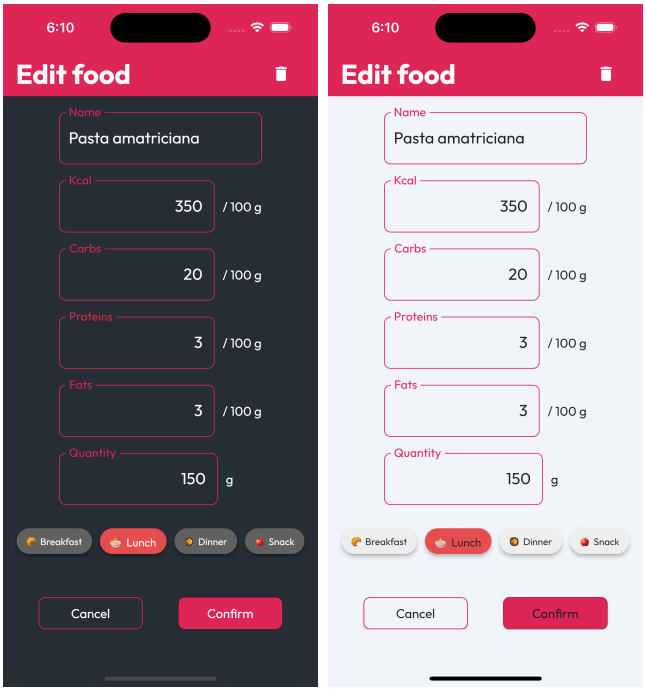


Figure 4.6: Edit food page

4.2.4. Goal page

This screen shows the user's fixed goal and permits the user to set them based on their necessities. Below there are some tips related to some typical athlete diet types.

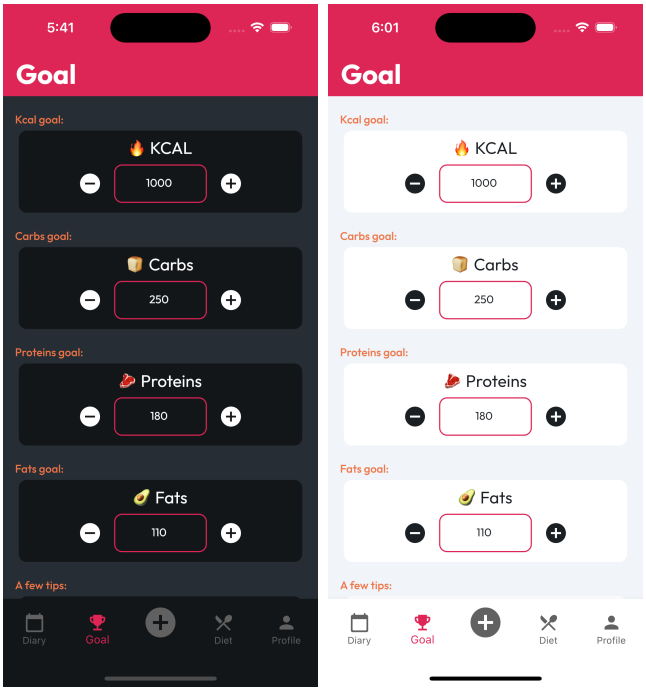


Figure 4.7: Goal page (1 of 2)

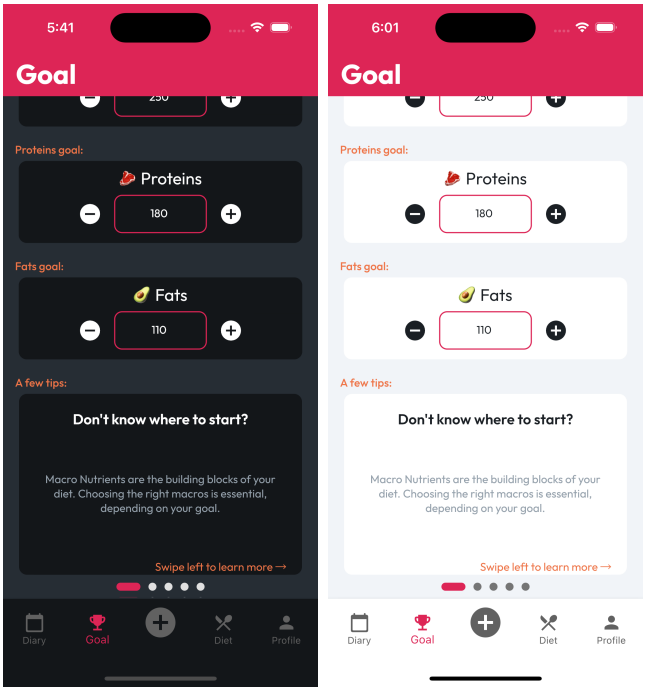


Figure 4.8: Goal page (2 of 2)

4.2.5. Add page

This page shows the main methods to add a new food to the user daily diary or to the list of saved foods in the diet. Below there is a list of the foods registered the previous day in order to speed up the UX add procedure if the user wants to add a food inserted the day before.

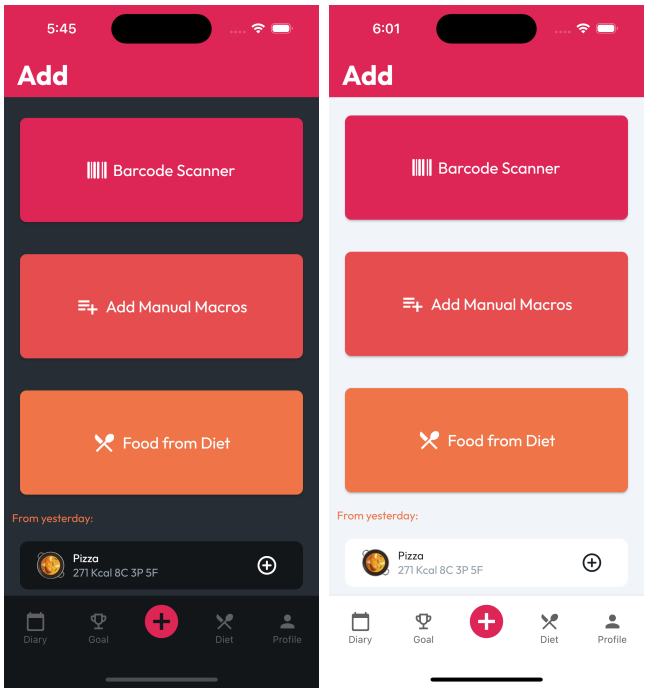


Figure 4.9: Add page

4.2.6. Scan barcode page

Page for scanning foods' barcode. It let the user choose whether to use the front or the retro camera and using or not the flash.

Figure 4.10: Scan barcode page

Note: after the scan the app let the user insert the quantity of the food with the *Add food* page, or displays an alert if it doesn't find the food.

4.2.7. Diet page

This page displays the list of foods inserted in the user diet. For each food (tile) the user can edit the food already inserted or add it to the daily diary.
By clicking on the meal types (e.g. Breakfast, Lunch, Dinner, Snack), the user can filter the foods to show a restricted list.
On the top right there is the *Add custom food* button that enables the user to insert a new food.

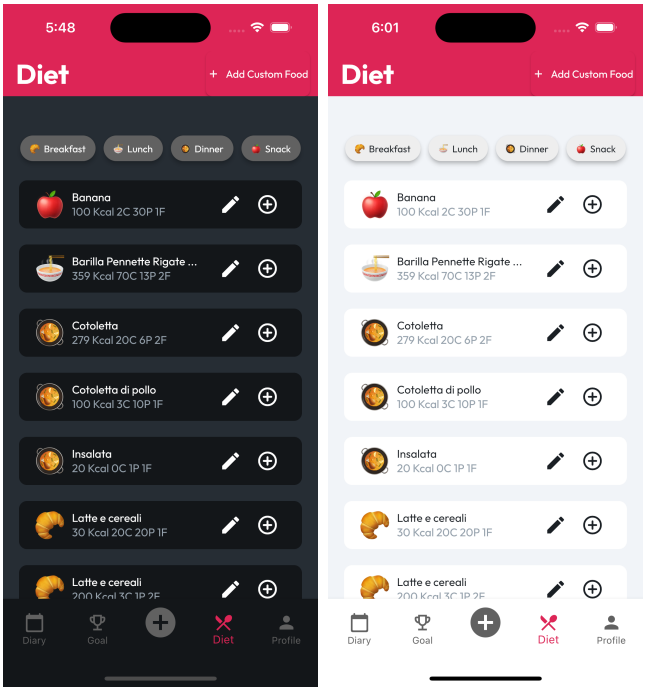


Figure 4.11: Diet page

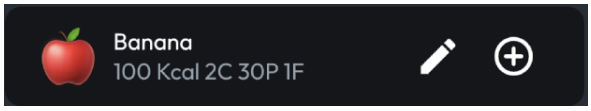


Figure 4.12: Single tile of diet food.

4.2.8. Add food page

This page is shown when clicking on the *Add manual macros* or *Add custom food* buttons. From there the user can submit all the custom properties of a food he wants to register. The *Add to Diet* button adds the food to the user's diet; while the *Log Food* adds it to the diary foods.

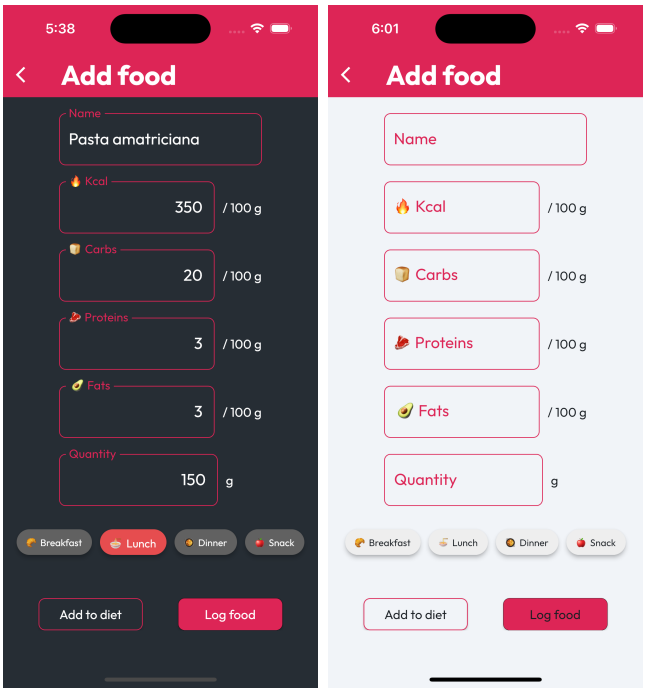


Figure 4.13: Add food page

4.2.9. Profile page

This screen contains the settings of the app and the user information. In detail it greets the user with a "*Welcome*" followed by the user name, and a set of settings for update user's information.
It gives the possibility to switch to light/dark mode and to log out from the app with a button.

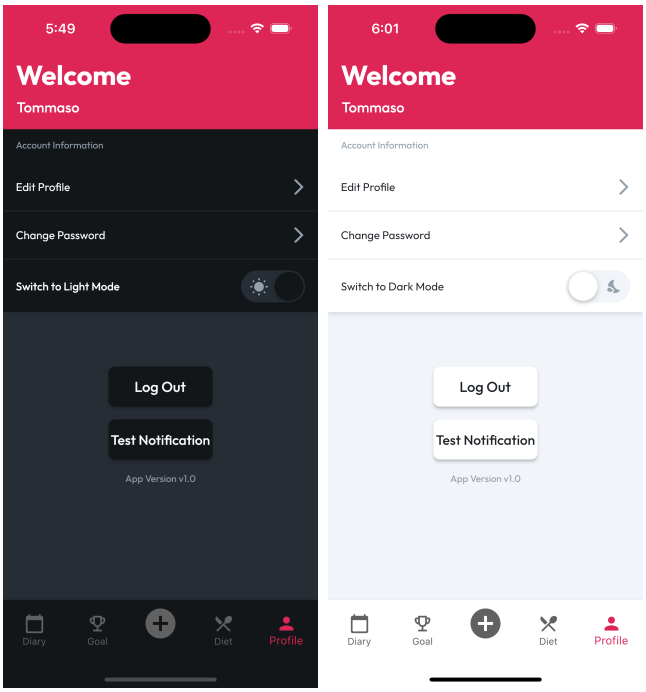


Figure 4.14: Profile page

4.2.10. Edit profile page

Forms for updating user's information.

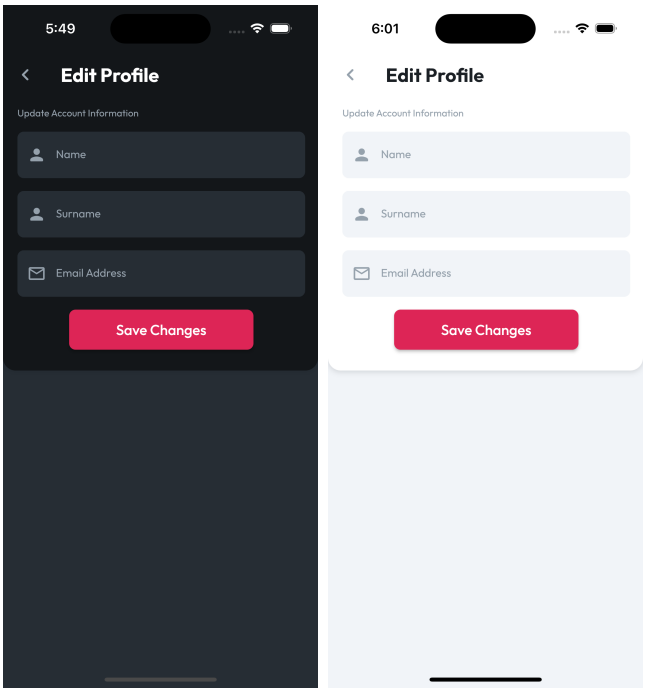


Figure 4.15: Edit profile page

4.2.11. Change password page

Form for updating the user's password.

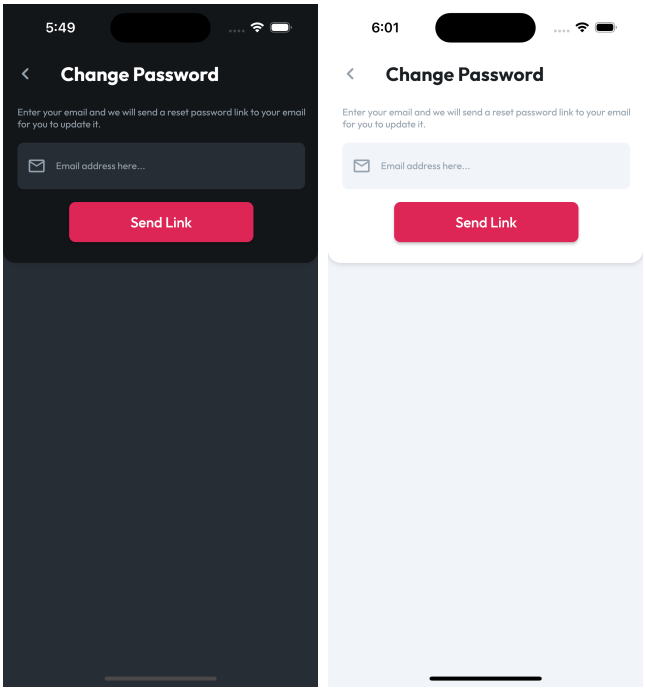


Figure 4.16: Change password page

4.3. Tablet version

In the following sections are shown the screens of the **tablet** application. Every page will be shown respectively in dark and light mode.
The content and functionalities of the pages are equals to the mobile version.

4.3.1. Diary page

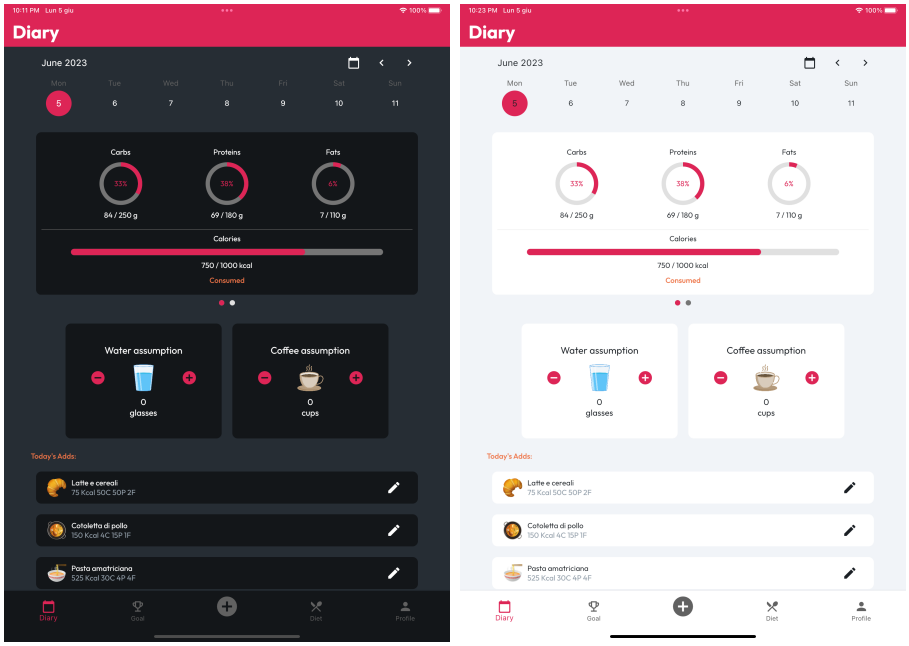


Figure 4.17: Diary tablet page

4.3.2. Edit food page

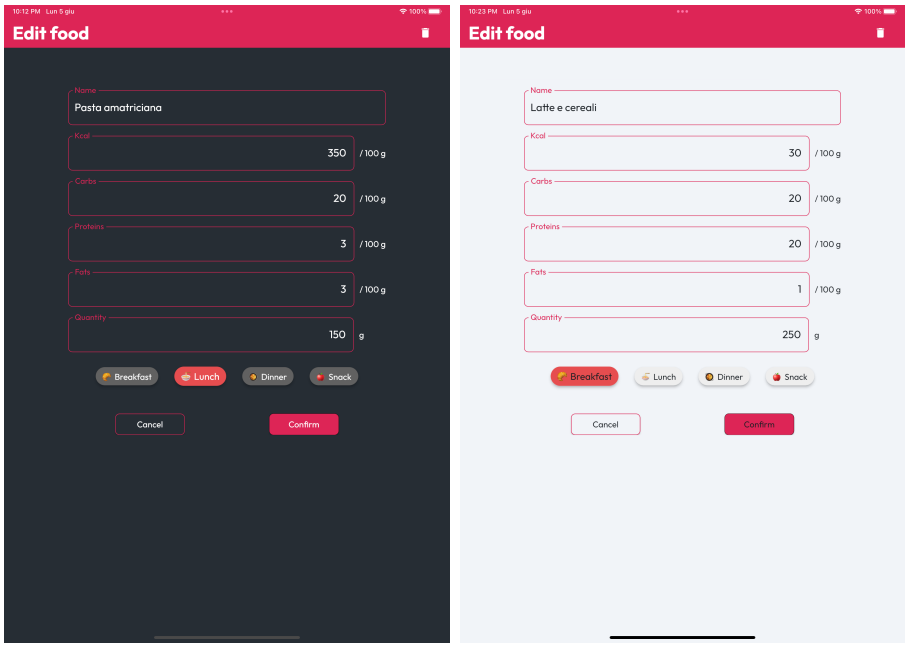


Figure 4.18: Edit food tablet page

4.3.3. Goal page

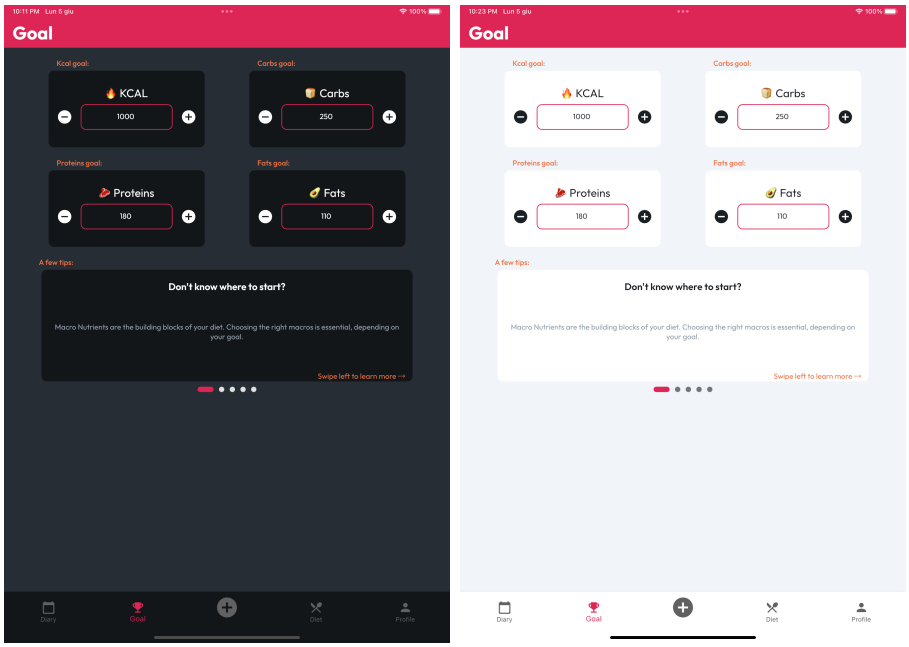


Figure 4.19: Goal tablet page

4.3.4. Add page

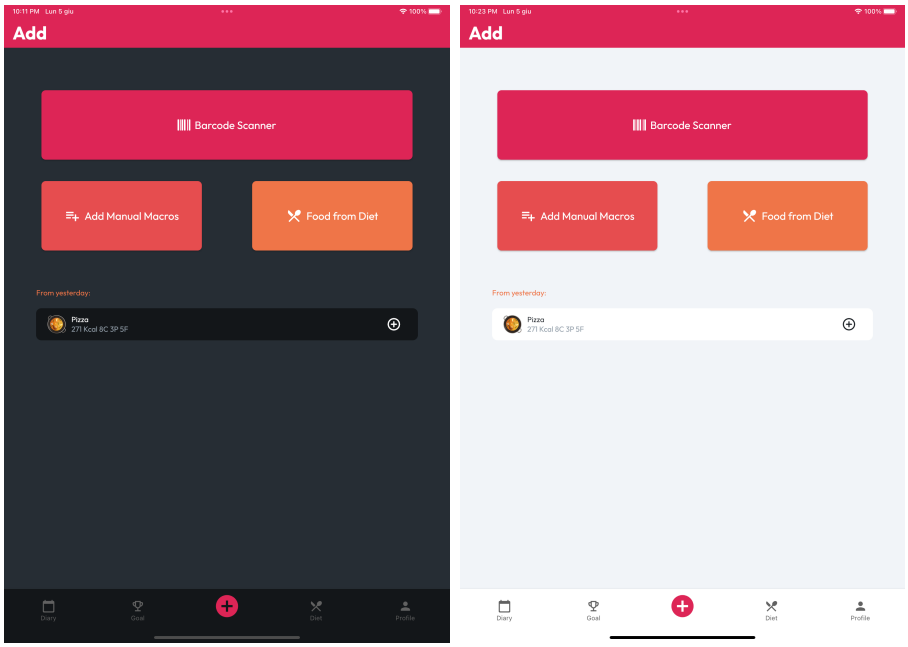


Figure 4.20: Add tablet page

4.3.5. Add food page

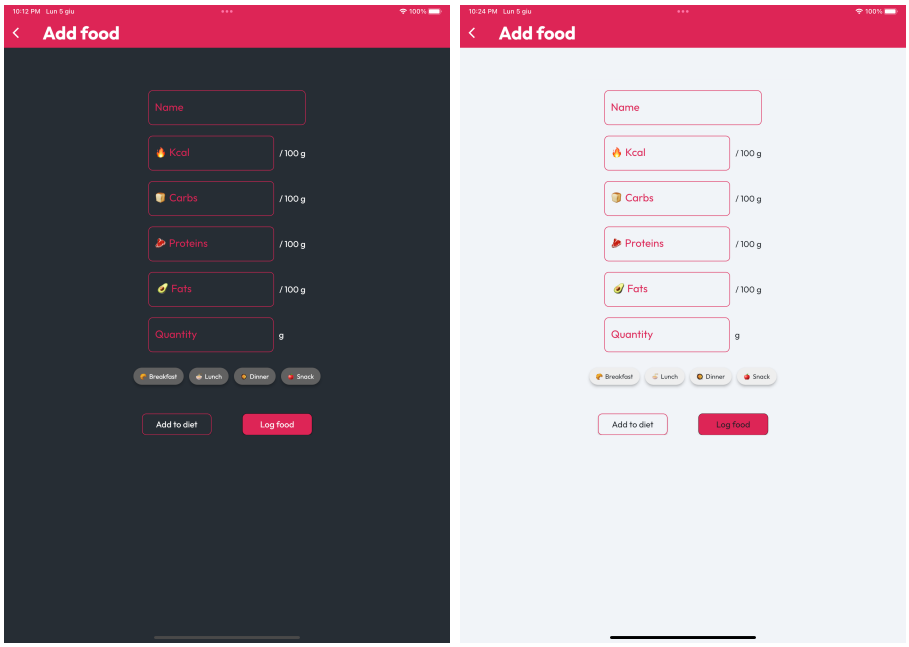


Figure 4.21: Add food tablet page

4.3.6. Diet page

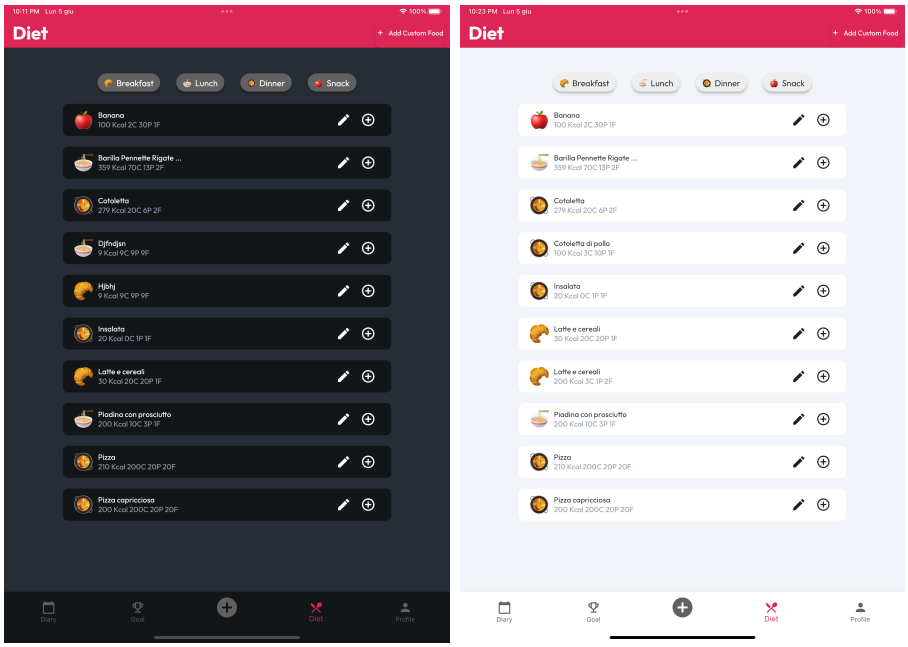


Figure 4.22: Diet tablet page

4.3.7. Profile page

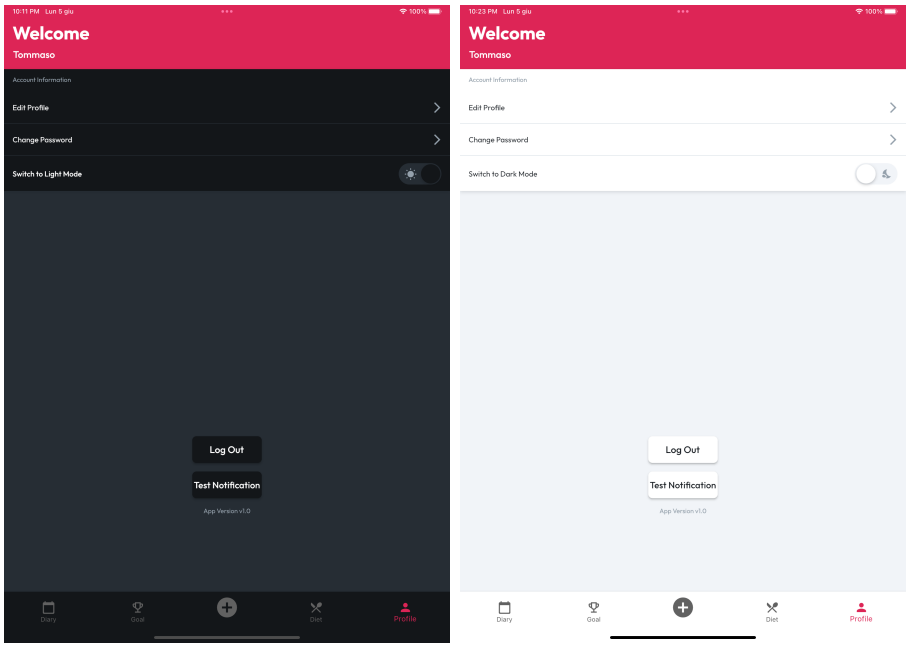
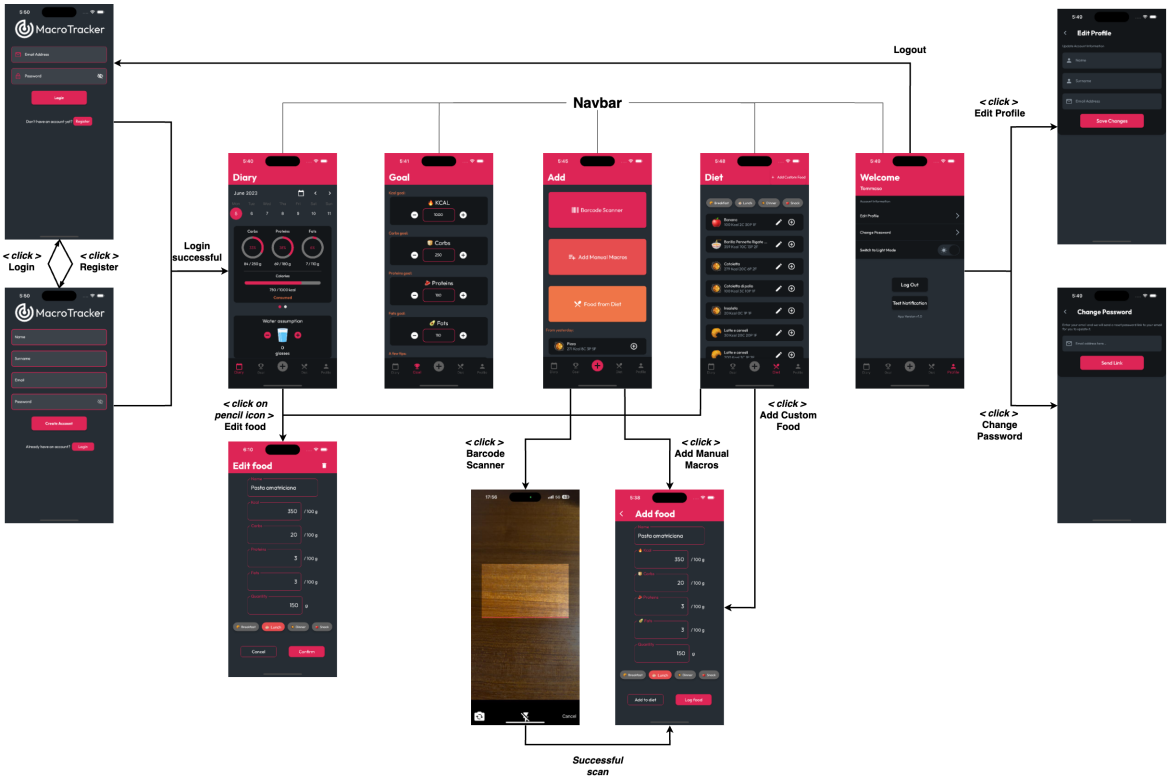


Figure 4.23: Profile tablet page

4.4. UX Flow

This diagram shows the actions that the user can perform to navigate through the screens of the app.
(For convenience it has been used mobile screens with dark mode.)



5 | App Testing

5.1. Unit testing

Most of the methods in the application involve either making API calls, to Firebase or to the previously described services, or organizing the screen's layout, making it inappropriate for unit testing.

5.2. Widget testing

To ensure successful testing, a *mocked* version was developed for each widget in the app. These versions eliminate any Firestore queries and API calls that may lead to test failures, replacing them with static values. The primary objective of this testing approach is to confirm the presence of all expected sub-widgets within each widget's layout and validate that the displayed texts accurately match the statically inserted content.

Widget tested	**Description**	**Problems and notes**
AddBarcodeFood Widget	The test verifies that the widgets contain all the necessary *TextFields* for inserting food, as well as the parent widgets that encapsulate them, along with the appropriate title in the *AppBar*.	-
Add Widget	Asserts the presence of the three buttons that enable the user to add food, along with the *ListView* and the texts related to the foods inserted yesterday. Additionally, it verifies the correctness of the *AppBar* and the parent widgets.	Since FlutterFlow was utilized in the implementation, certain widgets were not captured by the test, thereby preventing us from asserting their presence through this testing approach.

Widget tested	Description	Problems and notes
ChangePassword Widget	The test case verifies the presence of a form specifically designed for users to enter their email address for password reset purposes. It further ensures the existence of the necessary text components that accompany the form. Additionally, the test validates the accuracy of the title widget and confirms that its displayed text content aligns with the expected values.	The presence of the button could not be asserted due to it being implemented as a FlutterFlow (FF) widget.
Diary Widget	Asserts the presence of the Calendar, while also meticulously verifying the exact number of *LinearPercentIndicator* and *CircularPercentIndicator* elements representing the user's progress. Additionally, it ensures the availability and functionality of the adding/removing buttons for water glasses and coffee cups. Moreover, it examines and confirms the correct display of manually inserted texts across all pages, assuring the accuracy and consistency of the displayed content.	Several overflow size exceptions were encountered during the testing phase, primarily attributed to the boxes that encapsulate the percentage indicators. These exceptions arose due to the content exceeding the available space allocated for display, leading to visual anomalies and irregularities in the layout.
Diet Widget	The test case asserts the presence of the parent sub-widgets and verifies the existence of the *ListView*, which displays a mock list of foods for testing purposes. Furthermore, it ensures the correct rendering of the title in the *AppBar*.	The verification of the *ChoiceFilter* widget's presence was not feasible, primarily due to its implementation as a FlutterFlow (FF) widget.

Widget tested	Description	Problems and notes
EditFood Widget	The test ensures the presence of the necessary text fields for editing the food, including the corresponding deletion icon and other relevant parent widgets. Furthermore, like for the other tests, it specifically checks for the exact matching of the page title in the *AppBar*.	-
EditProfile Widget	The test asserts the presence of the form to be compiled to edit the user profile, and the corresponding text displaying the UI of this page.	-
Goal Widget	The test verifies the presence of all the sub-widgets that displays the widget, including the precise number of text fields for modifying the goal and the corresponding buttons for increasing or decreasing the goal values. Additionally, it expects to locate the *PageView* component that pertains to the tips provided for standard goals. Furthermore, the test examines the texts positioned above the text fields and ensures that the text initially populated within the fields aligns with the expected values.	Some overflow exceptions arose with the page, which have been resolved by incorporating a *SingleChildScrollView*. This ensures that all sub-widgets are displayed correctly across various screen formats.
Login Widget	The test primarily verifies the presence of the forms for inserting the user's email and password. Furthermore, it ensures the correct display of the app's name and logo icon. Additionally, it checks for the proper rendering of other helpful UI texts.	Similar to the previously described tests, this test was unable to directly verify the presence of button widgets. However, this challenge was overcome by examining the text content within those buttons.

Widget tested	Description	Problems and notes
Profile Widget	This test verifies the presence of all the necessary widgets required for profile management. It includes checking for the presence of the *InkWell* widget responsible for creating the row that displays various options. Additionally, it ensures the presence of the sub-widgets that compose the page. The test also verifies the correctness of all statically inserted texts within the mocked widget.	-
Register Widget	Similar to the Login Widget test, this test verifies the presence of the required sub-widgets in the Register Widget. It ensures that the widget contains four forms for user input, including fields for the user's name and surname. The goal is to validate that all the necessary components are present for successful user registration.	-

Table 5.1: Widget tested in widget testing.

5.3. Integration testing

The integration testing has been performed by the usage if two flutter packages:

- flutter_test/flutter_test.dart
- integration_test/integration_test.dart

The objective of this type of testing is to simulate user behavior within the app. Specifically, the tests described below navigate through different screens starting from the main Diary Widget and simulating user interactions to verify the expected outcomes.
Note: Before proceeding with each test, the presence of the sub-widgets in the starting widget is verified. Similarly, after the navigation, the presence of the sub-widgets in the final destination is confirmed.

Screen navigation tested	**Description**
Add screen	This test consists of three sub-tests. The first sub-test focuses on navigating from the *Diary* screen to the *Add* screen. In the second sub-test, while being in the *Add* screen, it simulates a click on the *Add Manual Macros* button. This action is performed to navigate to the *AddFood* screen and verify that the user is effectively on that page. The last sub-test is similar to the second one, but it clicks on the *Food from Diet* button instead. Overall, this test ensures successful navigation to the *Add* screen and thoroughly tests all the buttons that lead to other pages.
Diet screen	This test verifies the navigation from the the *Diary* screen to the *Diet* screen. Then, by simulating a click on the "pencil" icon, it asserts the correct navigation to the *EditFood* screen.
Goal screen	This test verifies the navigation from the *Diary* screen to the *Goal* screen. Then, it statically inputs the value "2000" in the kcal text field. After that, it simulates two clicks: one on the "plus" icon next to the corresponding field to verify that the value increases and is correctly displayed as "2050", and one on the "minus" icon to decrease the value, confirming the correct behavior of all the involved widgets.

Screen navigation tested	Description
Profile screen	This test verifies the navigation from the *Diary* screen to the *Profile* screen. After that, ensuring that the app is in the default light mode, it simulates a click on the theme mode switch and asserts the expected theme change.

Table 5.2: Screens tested in navigation testing.

5.4. User testing

The app has undergone extensive testing on both student devices and emulators throughout the entire development process. During these daily tests, various graphical, logical, and backend bugs were identified and resolved on a day-to-day basis.

One notable issue that arose was related to the insertion of values into text fields, particularly those within the *Goal* widget. This problem occurred due to the simultaneous update of values in both the app and the Firestore database.

All bugs and problems discovered were promptly communicated to each other member of the group in real-time, ensuring that everyone was aware of the issues and that they were resolved as quickly as possible.

In order to replicate the user experience, the app was utilized and tested for a duration of two weeks, allowing for thorough evaluation.

6 | Effort spent

Student	Database	Design	App	Documentation	Presentation	Total
Alessandro Sironi	2h	40h	100h	5h	3h	150h
Tommaso Sprocati	4h	40h	100h	5h	1h	150h