Design and implementation of mobile applications

POLITECNICO
MILANO 1863

MyMeals

Design Document

Jeffar Farouk Riccardo Medici

Professor:

Luciano Baresi

A.Y. 2022/2023

Table of Contents

1 Introduction

1.1 Project Description

The application, called **MyMeals**, is aimed at users that want to keep track of what they eat and want recommendations for what to cook. Its main functionalities consist in keeping track of the calories and the nutrients ingested and it suggests recipes and meals based on food availability and macro categories.

Going more into details:

- User can manually insert the meal by typing the name and the nutrients.
- User can insert the meal and the nutrients by scanning the barcode present on the product.
- Each meal eaten during the day is displayed on the main screen as a card showing calories, fats, proteins and carbohydrates.
- Users can insert a specific caloric target and monitor the progress during the day.
- In the data section, the user can visualize how many calories and nutrients were ingested both during the week and the month. This data is also visualized in charts.
- In the recipe section, the user can choose recipes from different macro groups: Healthy recipes, Random recipe and by country of origin.
- The user can also get recommendations inserting a main ingredient and the required mealtime (breakfast, lunch, dinner and snack).

1.2 Features

1.2.1 Sign in using different services.

Users have the possibility to sign in by using their email address and a password (with confirmation) or using the Single-Sign-On functionality with supported identity providers.

1.2.2 Dashboard

The dashboard is the main screen of the apps where the meals inserted during the day and their Nutritional values are displayed. On top of the screen a card shows the caloric objective, the number of calories eaten and the difference between those two values.

1.2.3 Insert a new meal:

The application keeps tracks of the nutrients and the meal eaten. To do so the user needs to insert the different meals into the app. The user can click the plus button in the bottom right corner of the screen. They then can choose between two options:

- Manual insert: the user manually inserts the meal name and its nutritional values (calories, carbohydrates, proteins and fat)
- Automatic insert with Barcode scanner: the user can use the camera of his device to scan the barcode of the selected product to automatically insert a meal. The user is then asked to insert the amount of product (in grams).

1.2.4 Look for a recipe:

In the recipe screen the user is provided with a scroll down menu with five different options:

- **Casual**: the casual function allows the user to look for a recipe by inserting the main ingredients and the corresponding time of the day (breakfast, lunch, dinner, snack).
- **Healthy**: identical to the casual function, it allows the user to look for a recipe by inserting the main ingredients and the corresponding time of the day. The result will comprehend only recipes considered to be healthy (relative lower number of calories and fat).
- **Random**: It returns the user a receipt depending on the time of the day selected
- **Analyzer**: The analyzer allows the user to know the nutritional values of a given recipe. The user inputs the ingredients with and their weight in grams (example: flour 200g, chicken 300g, bull peppers 100g). The app returns nutritional values, and the user can add the results on his list of eaten meals.
- **Explore**: The explore function allows the user to look for a recipe choosing the time of the day and the country of origin of the recipe.

1.2.5 Look for a recipe ingredients and instructions

Once a user has found a recipe he is interested in, the app shows a window with the complete ingredients and nutrients list. The user can also add the recipe to the meals he has eaten, selecting the correct number of portions. A link to recipe instructions is also provided.

1.2.6 Analyze data and trend.

The app gives the user the possibility to keep track of his nutritional intakes. The app provides cards showing the nutrient eaten in a daily, weekly, and monthly basis. The app also provides weekly and monthly charts showing the number of calories, carbohydrates, proteins and fats eaten at the specific the time of the day the meals were inserted.

1.2.7 Manage app settings:

A user can manage the application theme (system theme, light theme, dark theme) and insert the caloric objective.

1.3 Definitions, Acronyms, Abbreviations

Definitions

- **User:** an account registered in the application with username and password
- **Logged user:** a user that is currently logged inside the application using that device.
- **Android:** it is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets.
- **Dart:** it is a programming language designed for client development, such as for the web and mobile apps. It is developed by Google and can also be used to build server and desktop application.
- **Flutter:** Flutter is a mobile app development platform created by Google. It allows developers to create web, desktop, and cross-platform apps that run on Android and iOS devices. Flutter uses a reactive programming language called Dart, making development faster and easier than traditional methods.
- **Framework:** it is an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing application-specific software.
- **Firebase Auth:** Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more. Firebase Authentication integrates tightly with other Firebase services, and it leverages industry standards like OAuth 2.0 and OpenID Connect, so it can be easily integrated with your custom backend.
- **Firebase Firestore:** Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. it keeps data in sync across client apps through real-time listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity.

Acronyms:

- **API:** Application Programming interface, it indicates on demand procedure which supply a specific task.
- **DBMS:** Database Management System.
- **UI:** User Interface.
- **DD:** Design Document
- **SDK:** Software Development Kit, it provides a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allows developers to create software applications on a specific platform.
- **MVC + S:** Model View Controller with a separate Server layer.

1.4 Reference Documents and Websites

- Flutter
- Firebase
- Itnext.io

1.5 Document Structure

- **Section 1: Introduction**
  This section offers a brief description of the application and its main functionalities. Definitions, acronym and abbreviations are also provided.
- **Section 2: User Interface Design**
  This section contains different screenshots of the application, together with charts to provide and understanding of the flow of functioning of the application. The presented mockup refers to the client-side experience.
  Section 3: Requirements
  This section contains the main requirements that the application must satisfy, a brief description is also provided.
- **Section 4: Architecture and Implementation.**
  This section contains an overview of the architecture of the system and the description of the components present in the application.
- **Section 5: Testing**
  This section provides a list of the different typologies of tests performed as well as the code coverage and a more detailed summary on the complete testing part.
- **Section 6: Future developments**
  In this section future functionalities and improvements of possible future versions of the app are presented.

2 User Interface Design

2.1 Introduction

The objective of this section is to show the design of the main screen of the user application in order to describe the flow of the main functionalities. The flow is created according to specific and illustrated input of the end user.

2.2 Mobile Application Interface

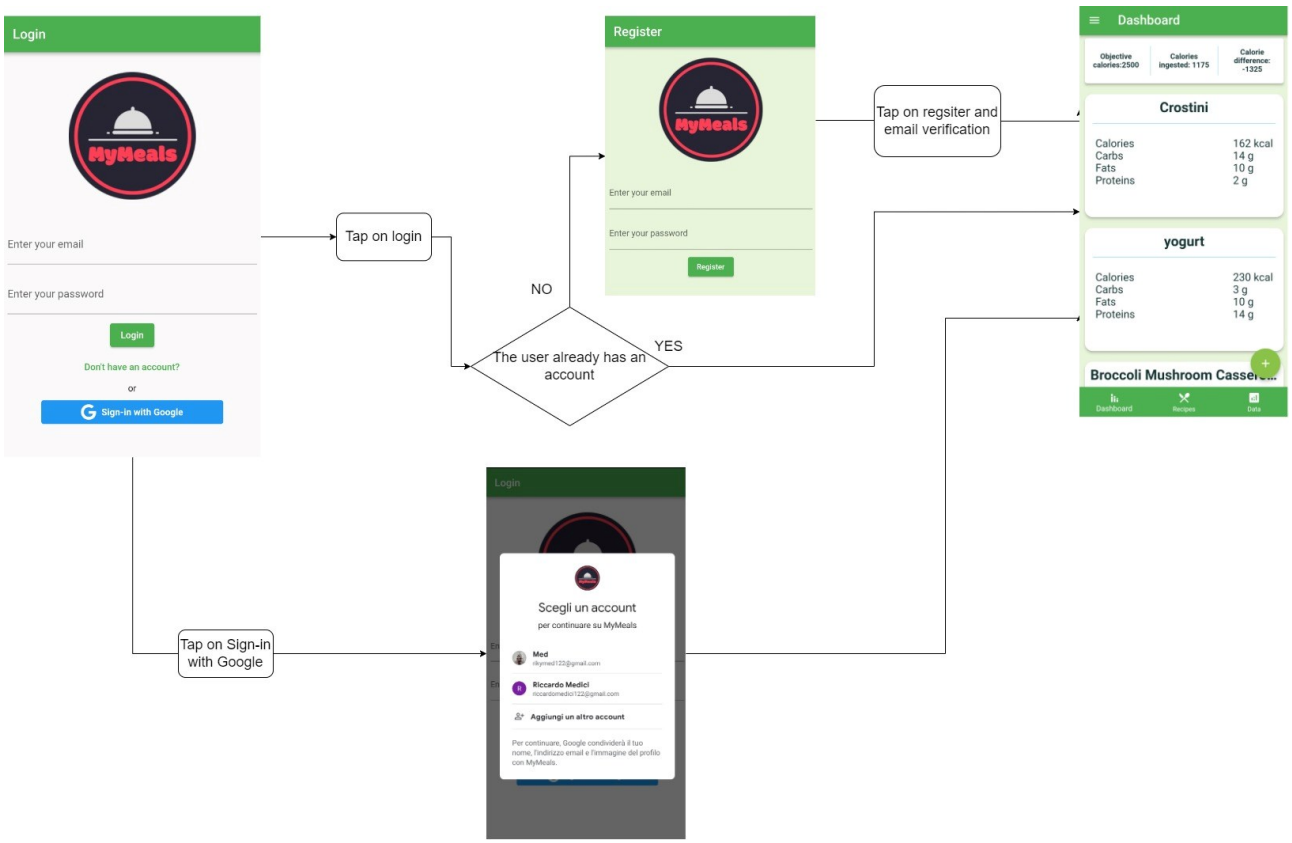

Figure 1 UI for login/sign-in.

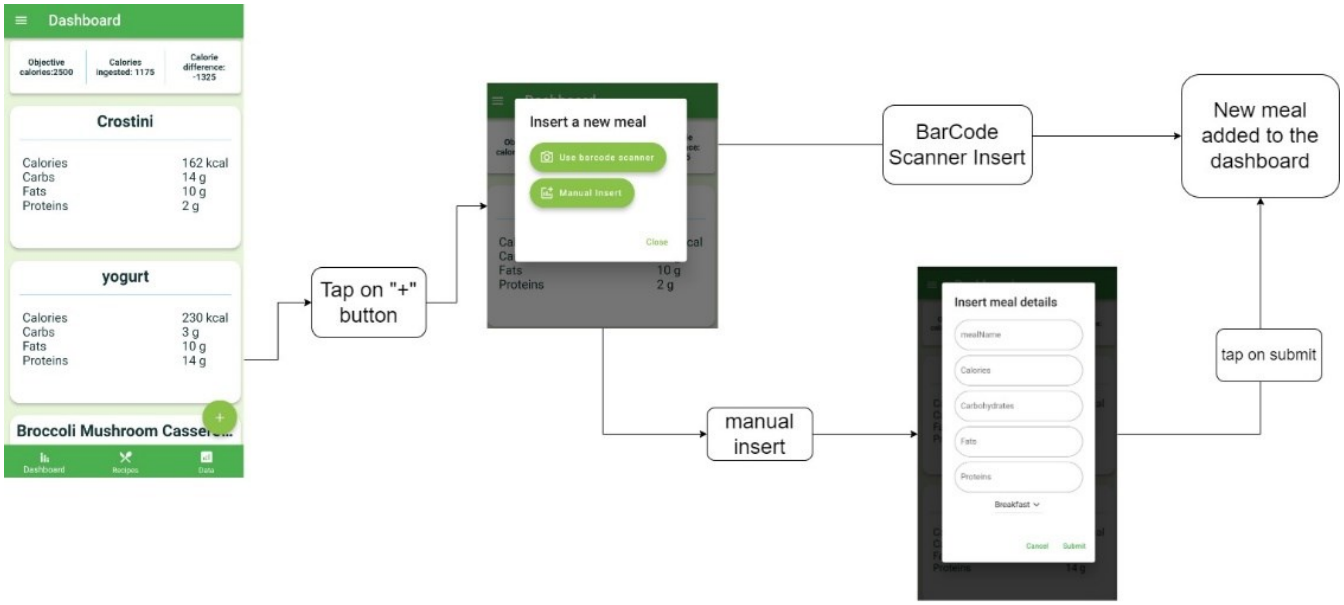

Figure 2 Adding a new meal.

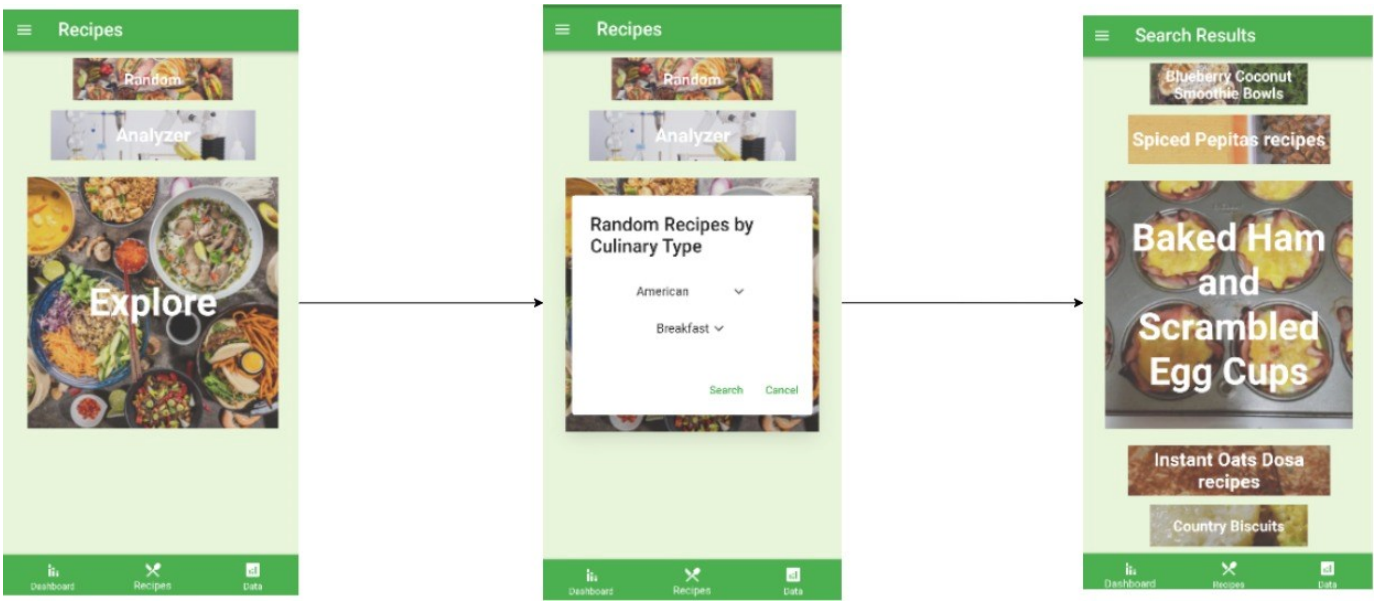

Figure 3 Finding a recipe using filtering by country of origin.

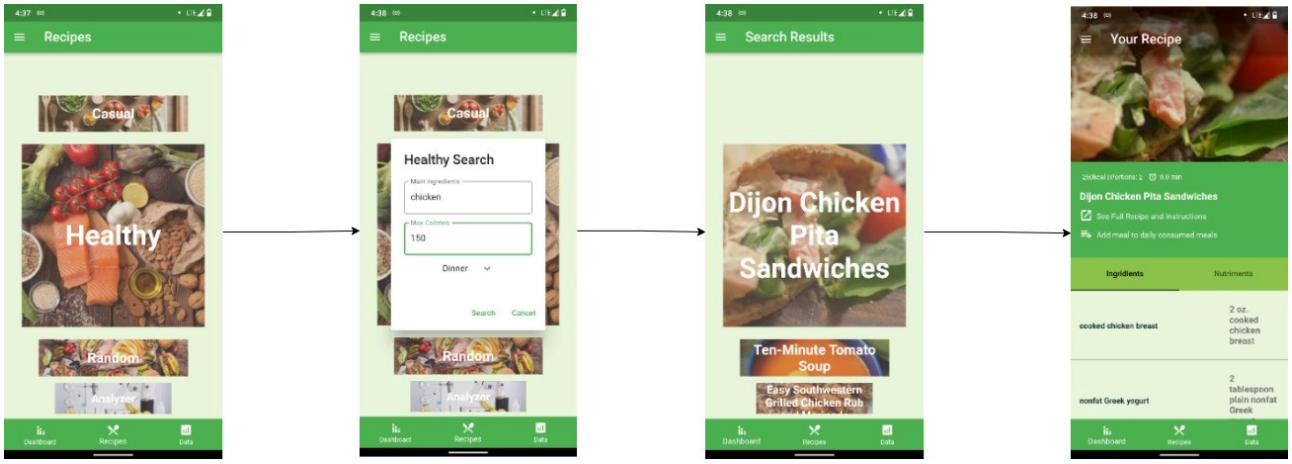

Figure 4 Finding a recipe by the macro group "Healthy"

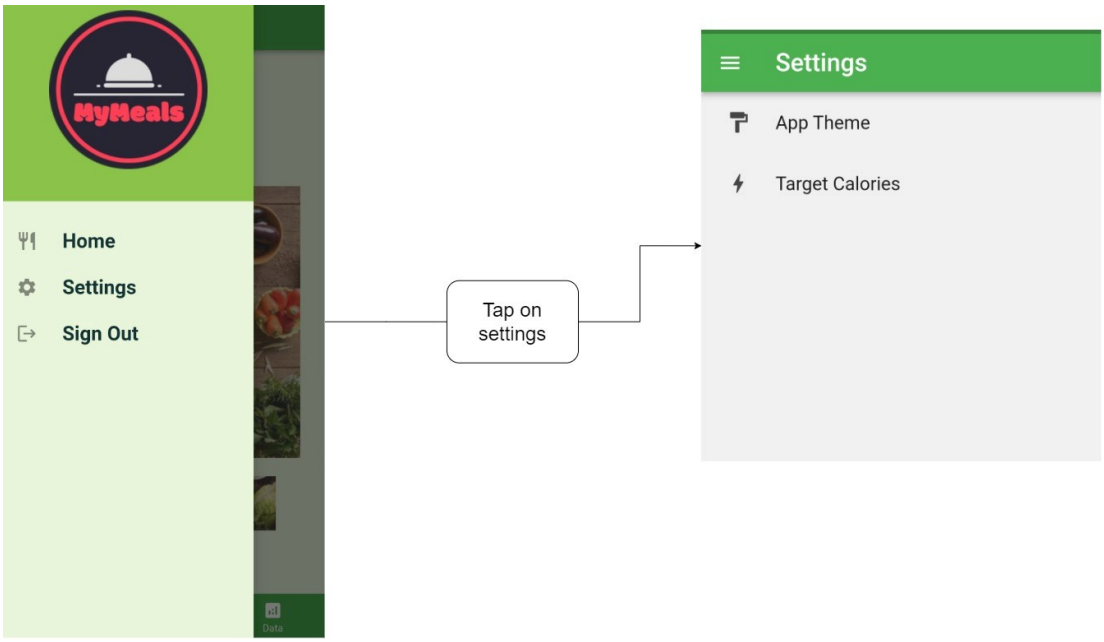

Figure 5 Settings of the application

Figure 6 Nutrients data and charts

For the tablet implementation a horizontal scroll has been, taking advantage of the landscape mode and larger screen.

For the widget that required a numerical value as a parameter for the height, a dynamic value was introduced based on the screen size by leveraging the **MediaQuery** class. In this way the positioning of the widget is appropriate independently of the screen size.

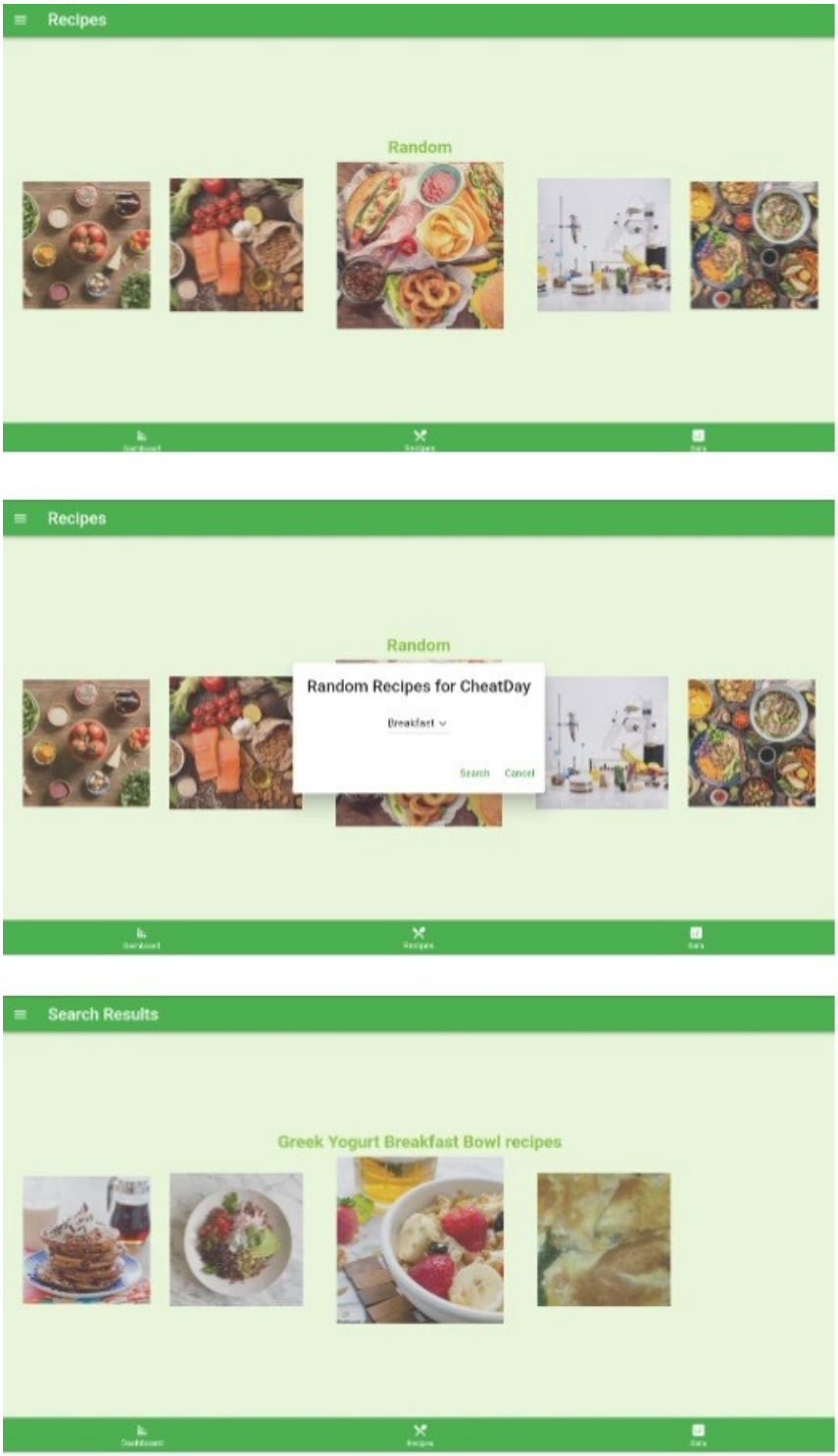

Figure 7 The scroll menus on a tablet device

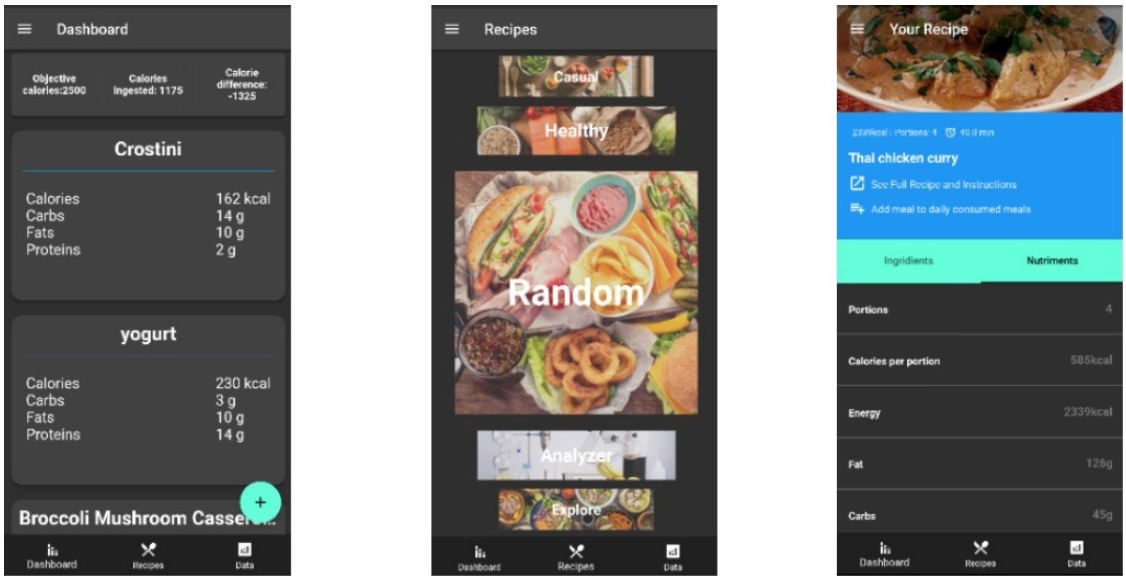

Figure 8 Screenshots of the application with dark theme on

3 Requirements

The following section contains a list of the requirements that the application should satisfy. A brief explanation for each requirement is provided.

- **R1**: The system allows the login/registration of a user. This can be done as described in section (1.2.1)
- **R2**: The system allows user to manually insert a meal, giving as input the name and the nutritional values.
- **R3**: The system allows the user to insert a meal by scanning his barcode scan.
- **R4:** The system allows the user to choose between a light or dark theme.
- **R5:** The system allows the user to insert a caloric target.
- **R6:** The system shall provide recipes to the user depending on mealtime, country of origin, main ingredients and maximum number of calories.
- **R7:** The system must calculate the nutrients of a meal given the main ingredients as input.
- **R8:** The system must allow the user to add a recipe to the list of meals in his dashboard.
- **R9:** The system must provide the user a link containing the step for a given recipe.
- **R10:** The system must provide data cards and charts to track the nutrient intake in a daily, weekly and monthly manner.

4. Implementation & Architecture

MVC is an architectural pattern that divides an application into three main logical components: the model, the view, and the controller. Each of these components is designed to handle specific aspects of an application's development.

In this project we adopted the MVC+S Pattern which provides us with a much more readable, controllable, and organized architecture. Nothing special other than organizing your files. The mindset is straightforward, but the benefits are numerous. In contrast to MVC, we separate the Service layer. Services are concerned only with making external API calls, and parsing/returning any data they receive back. Services never touch the Model directly. Instead, Controllers will make Service calls, and update the Model with the results, if they choose to.

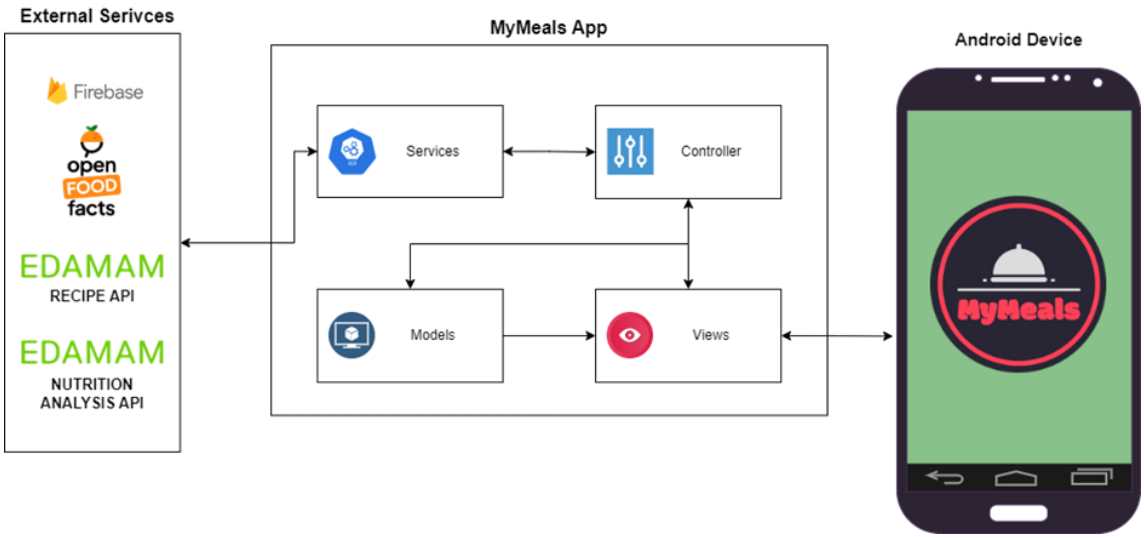

Figure 9 The application has Model View Controller + Services architecture.

4.2 Implementation

Below a list is provided containing all the components present in the MVC + S.

4.2.1 Model:

The model holds the state of the application and provides an API to access/filter/manipulate that data. Its concern is data encapsulation and management. It contains logic to structure, validate or compare different pieces of data that we call Domain Logic.

In our application the models consist of the user model that is initiated once the user is logged in and the recipe model where the selected recipe data is handled to be then displayed in the view.

4.2.2 View:

Views are all the Widgets and Screens within the Flutter Application. They are the core components on app built of flutter.
Our application has 11 screens and 9 widgets.

4.2.3 Controller:

The controller layer is represented by various Commands which contain the Application Logic of the app. Commands are used to complete any significant action within the app.

- **Authentication**: By relying on Firebase, this controller handles user signing and sign-out and holding the logged-in status.
- **Data controller**: Sets and gets user data from Firebase Firestore and formats before sending it to the view or model and data plotting.
- **Meal Services**: implements barcode scanning and mediates between the view and 3rd party meal services.

4.2.4 Services:

Services fetch data from 3rd party services and return it to the app.

- Firebase
- Open Food Facts: get product nutrition data based on barcode.
- Edamam Recipe API, it includes functions for:
  - Ingredients & Calories services
  - Random meal Services
  - World Cuisine Services
- Edamam Nutrition Analysis API, it returns nutritional facts given a recipe and his ingredients in grams.

The view does not directly interact with data since it must go through the controller and the same applies for external services.

The data solution used is Firebase's Cloud Firestore which is NoSQL (Documents and Collections) database. We chose Firestore because of its flexibility and availability along with its excellent integration with flutter.

Authentication (Sign-in, Google Sign-in, password recovery) is handled by Firebase Auth SDKs, we chose Firebase Auth because of its reliability and security since passwords are hashed and never stored as is and uses auth tokens to verify the authenticity of each request, it also handles automatically token release, refresh, and expiration.

4.3 Dependencies

- **Http ^0.13.5**:
  - Sends http GET and POST request to 3rd party services.
- **Flutter_barcode_scanner ^2.0.0**:
  - Scans barcode using the device's camera.
- Firebase modules:
  - **firebase_core: ^2.4.0**
  - **firebase_auth: ^4.2.0**
  - **google_sign_in: ^5.4.2**
  - **cloud_firestore: ^4.2.0**
- **provider ^6.0.1:**
  - A wrapper around InheritedWidget to make them easier to use and more reusable.
- **font_awesome_flutter: ^10.3.0:** Font and Font Icons.
- **syncfusion_flutter_charts ^20.4.38:** Chart plotting.
- **shared_preferences ^2.0.15:**
  - Saves settings locally.
- **settings_ui ^2.0.2:**
  - Settings page widgets
- **vertical_card_pager ^1.5.0:**
  - Cards carousel
- **url_launcher ^6.1.7:**
  - Open URLs in WebView
- **horizontal_card_pager ^1.1.1:**
  - Cards carousel
- **pull_to_refresh ^2.0.0:**
  - Refresh state (or any action) after pull gesture

Testing dependencies:

- **Mockito ^5.3.2:**
  - Provides navigator mocks and watcher.
- **fake_cloud_firestore ^2.1.0:**
  - Mocks Firestore.
- **firebase_auth_mocks ^0.10.3:**
  - Mocks Firebase auth.
- **google_sign_in_mocks ^0.2.1:**
  - Mocks Firebase Google sign in.
- **faker ^2.1.0:**
  - used to generate fake emails for tests.

4.4 Project Policies:

- **Modular Development:**
  A software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

- **Separation of Concern**
  Each function has only one specific duty

- **Code Smells Prevention**
  - Avoid duplicate and dead code.
  - Avoid long methods.
  - Validate Merges and Pull requests.

- **Strict Project Structure**
  - Each dart file should be in model/screens/services/widgets folder.
  - Each widget file has only one widget

- **Variable Naming Convention**
  - Classes, enums, typedefs, and extensions name should in UpperCamelCase.
  - Libraries, packages, directories, and source files name should be in snake_case(lowercase_with_underscores).
  - Variables, constants, parameters, and named parameters should be in lowerCamelCase.

- **Git**
  - Features are split over branches which are merged once a set milestone has been reached. The pull requests and branch merging require the two members to be present to solve any conflict.
- **Null Safety**
  Null safety means that a variable cannot have a null or void value. This feature improves user satisfaction by reducing errors and app crashes. Null safety ensures that all runtime null-dereference problems are shown at compile-time.

5 Testing

5.1 Performed tests.

To ensure a better stability of the app and its functionalities, different tests have been performed: Unit tests, Widget tests and Integration tests.

5.1.1 Unit test

A unit test tests a single function, method, or class. The goal of a unit test is to verify the correctness of a unit of logic under a variety of conditions. External dependencies of the unit under test are generally mocked out. Unit tests generally don't read from or write to disk, render to screen, or receive user actions from outside the process running the test.

5.1.2 Widget tests

A widget test (in other UI frameworks referred to as component test) tests a single widget. The goal of a widget test is to verify that the widget's UI looks and interacts as expected. Testing a widget involves multiple classes and requires a test environment that provides the appropriate widget lifecycle context.

5.1.3 Integration tests

An integration test tests a complete app or a large part of an app. The goal of an integration test is to verify that all the widgets and services being tested work together as expected. Furthermore, you can use integration tests to verify your app's performance. Generally, an integration test runs on a real device or an OS emulator, such as iOS Simulator or Android Emulator. The app under test is typically isolated from the test driver code to avoid skewing the results.

5.1.4 User/Acceptance testing:

The application was distributed to alpha testers to check the functionalities of the application and to evaluate the user experience of the app from someone other than the developers.

In order to capture how the user interacts with the application, live test sessions were set up, where we(developers) would oversee the usage of the app by testers to check any issues with not only the implementation but also the user experience.

5.2 Code Coverage:

Code coverage is a measurement of the proportion of code that is executed while automated tests are being run. Code coverage helps in determining the quality of code and investigating how thoroughly a software product has been verified.

The goal set for the code coverage of the project is at least 75% and for individual functions to files to be at least 70%.

5.3 Summary of testing:

File	% Branch	% Lines
lib/		
main.dart	100.00	90.00
lib/model/		
recipe.dart	100.00	100.00
user_model.dart	100.00	100.00
lib/screens/		
dashboard_screen.dart	100.00	83.93
data_screen.dart	100.00	81.72
home_screen.dart	100.00	93.10
login_screen.dart	100.00	73.91
recipe_details_screen.dart	100.00	76.22
recipes_screen.dart	100.00	61.60
recipesearch_results.dart	100.00	82.65
register_screen.dart	100.00	85.11
resetpass_screen.dart	100.00	85.71
setttings_screen.dart	100.00	73.33
tabs_screen.dart	100.00	91.67
lib/services/		
auth.dart	100.00	91.67
data_services.dart	100.00	96.77
meal_services.dart	100.00	98.75
lib/widget/		
bottom_navbar.dart	100.00	55.26
calories_chart.dart	100.00	72.73
daily_counter.dart	100.00	79.41
dashboard_card.dart	100.00	100.00
data_tile.dart	100.00	100.00
ingridient_tile.dart	100.00	100.00
insertData.dart	100.00	71.43
insertproduct_popup.dart	100.00	70.33
main_drawer.dart	100.00	51.61
All files with unit testing	100.00	77.12

Note: Bottom_navbar.dart and main_drawer.dart have low coverage rate (52%) as they included already tested parts or code note reachable via unit tests.

Overall Coverage Achieved: **77.1%**

5.4 Testing Details:

97-unit tests have been written for testing the application. These unit tests cover the business logic present in the services folder which consists of 3 files: Auth, data services, and meal services. The tests verify the behavior of the methods and check that errors are handled. For data related functions, mocks have been used to emulate the behavior of these modules such as Firebase auth and Cloud Firestore.

Widgets tests check that the widgets and screens render correctly with no exception. The tests also verify that the functions within the widgets interact with the states in the expected manner. Navigator mocks were used along with database mocks.

Models have been tested too to verify that data is constructed as expected.

A set of end-to-end integration tests were implemented in order to test the interaction of the app across the entire application. The integration tests capture the path that a user would take, which is:

Registration -> Meal Addition -> Recipe Lookup -> See Recipe Details -> Consult Data and Trends -> Update UI settings -> Set nutrition parameters in settings.

6 Future implementations

In the future development of the app there are different improvements and new features that could be implemented. Below is provided a list containing a brief description of each possible new addition.

- Implementing the tracking of more nutrients:
  In the current version of the app only calories, carbohydrates fats and proteins are registered. It could be useful to track other nutrients such as vitamins and fibers.

- Implementations of a filter for allergies and personal preferences:
  In a future version it will be possible to filter the results of the recipes based on allergies or personal preferences (gluten free, lactose free, vegan, kosher…).

- Improvement of the recipe database:
  The introduction of an algorithm filtering the results would improve the quality of the recommendations.

- Implementing suggestions and insights to the user about his diet:
  A new functionality would be added where the user gives as input the objective of his diet and the app returns suggestions based on the user eaten meals.

- Improve IOS support:
  The codebase is compatible with IOS, but at the moment we didn't test the app on an IOS devices, for this reason there is no iOS compatibility yet.