Politecnico di Milano
Dipartimento di Energia,
Informazione e Bioingegneria
Computer Science and Engineering

Design and Implementation of Mobile Applications

Realiteye - Design Document

Professor:
Prof. Baresi Luciano

Candidates:
Nicolò Albergoni - 10562083
Andrea Falanti - 10568850

Academic Year 2020/2021, Milano - 05/02/2021

Contents

Introduction

1.1 Scope

Realiteye is a new type of e-commerce application that aims to enhance the standard functionalities offered by a common e-commerce service with the help of the augmented reality technologies. The core feature of the Realiteye's mobile application is to provide the users the ability to watch, in an augmented reality environment, the products to which they are interested. The user simply taps the AR button on a product to enter an augmented reality scene in which the 3D model of the object is displayed.

The e-commerce is mainly focused on products in which viewing 3D models in augmented reality can be useful for the purpose of purchasing decisions. For example it could be useful for furniture related products, where it's important for buyers to check the real dimensions of a product and how they look in the current environment. When the model visualizer screen is open, users can use the phone camera to frame the space in which they want to place the object, the app will then render the object's 3D model, in augmented-reality, onto that plane; this provides the user the ability to see how the furniture will look in their apartments. Another meaningful example could be the one related to mechanical parts. Users that are interested in a mechanical piece can have a 3D-look of the object in augmented reality before buying it. Through the app they can display and control (rotate, zoom, etc..) the model of that object on a plane.

Although Realiteye is intended for the commerce of products that have an associated 3D model to be displayed, the app also allows the visualization of products that don't have a 3D-Model yet. In general the app provides the usual actions related to an online shopping service such as: searching a product, placing an order, checking your cart and visualizing product details.

1.2 Why Flutter?

Since we wanted to emphasize Realiteye's augmented-reality feature, we decided to develop the e-commerce app for mobile. Augmented reality requires a camera to point an area of interest and should also allow users to move in the environment in order to check the products from different perspectives. Therefore, portable devices are the most suited for this task and the only one that could give the final users a pleasant experience in using this new technology.

Since the only target devices are mobile phones, we decided to adopt a technology that could provide us the ability to develop a single code-base but that could run on a wide range of devices, even using different operating systems. We chose Flutter as development technology mainly for its cross-platform development features, allowing us to develop a single app both for Android and iOS. Moreover, with Flutter, we are able to develop smooth and powerful UIs without the typical loss in performance given by standard cross-platform framework, as Flutter release builds are native.

Architectural Design

2.1 High Level Overview

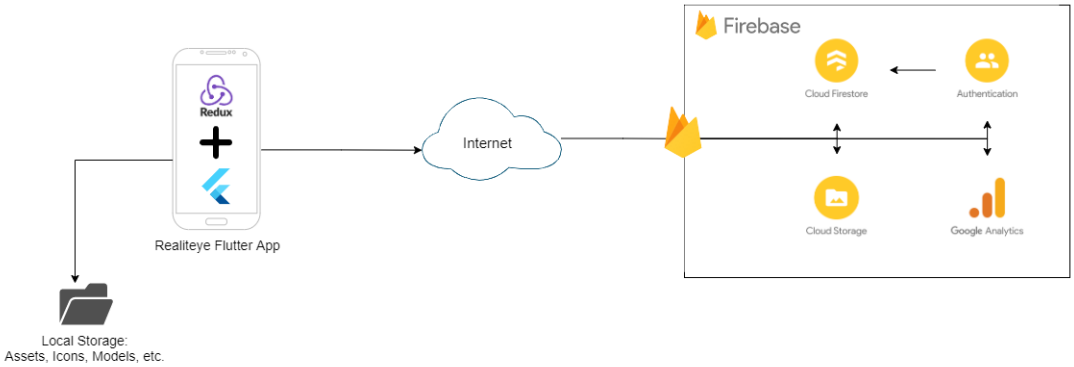

Figure 2.1: High level overview of the Realiteye system architecture

Realiteye is a cloud-based mobile system that heavily relies on the BaaS (Backend-as-a-Service) functionalities offered by Google Firebase. Conceptually Realiteye's architecture is divided into two macro-layers.

Frontend

The Realiteye flutter application, that will be distributed on the main mobile stores and will be installed by the users on their devices to interact with the system. The mobile application embeds the presentation and the business logic layers of the system. The presentation part is responsible for showing on screen the data, fetched from the system's database. It also provides the UI elements to interact with the data and perform the necessary operations required by the system. The presentation layer is managed by the Flutter's UI widget system, that allows to build application screens as a hierarchy of widgets to display. Also the business logic is embedded in the widget hierarchy: functions that

perform operations on the local application context or on the system database are associated to interactable widgets, following common UI standards. Moreover, we make use of Redux for Flutter, that helps us centralize certain parts of the state and logic of the application in order to have a consistent behaviour.

Backend

The backend side of the application is entirely handled by different Firebase products, externalizing completely the backend logic. This choice has been taken to focus the most part of developing effort on the frontend, that has a strong relevance for the success of a mobile application. In particular, we make use of Cloud Firestore and Cloud Storage to handle all of our data infrastructure (user data, product data, assets, etc...), and rely on the Firebase authentication system in order to provide a simple and secure authentication infrastructure for the users.

The Figure 2.1 depicts the overall architecture of the app, which is quite simple, but helps to visualize the logical separation between the backend services and the app itself. Moreover, this separation allows the system to preserve the safety of data as they are all protected by firebase safety rules, and by adopting such cloud-based infrastructure we can distribute an app that is scalable, in the sense that there is no performance limitation given by the backend.

2.2 External Services

In the following paragraph we describe the main external services that the app uses.

2.2.1 Cloud Firestore [1]

Cloud Firestore is one of the Firebase main solutions for storing data, it is a flexible and scalable NoSQL cloud database. Cloud Firestore is designed specifically for mobile development, as it provides features to store and sync data across devices in a real-time fashion. By adding real-time listeners, data is always consistent with respect to database updates; whenever an update occurs, a notification is sent to those listeners, thus, preventing the retrieval of the whole

database. It also offers offline support for mobile, so it's possible to build responsive apps even in absence of internet connection.

Cloud Firestore's NoSQL database is based on the document-collection data model. Documents are the fundamental data objects, they contain sets of values mapped to fields; they support different types of data for their fields (string, number, array, object, reference, etc...). Documents are stored in collections that can be used to organize data. It is also possible to create sub-collections inside documents, in order to create a more complex structure. Firestore exploits this document-collection pattern in order to provide the ability to perform optimized queries. You can retrieve data at the document level without the need to retrieve the whole collection (or any subcollections). Moreover, because Firestore is designed to scale, data is replicated in multiple regions in order to provide consistency, and whenever you perform queries their performances are scaled to the size of the results, not the size of the dataset.

2.2.2 Firebase Authentication [2]

Firebase Authentication provides the backend services needed for the users authentication procedures, together with an easy to use SDK for the developers. Authentication can handle both classic email-password based authentication and external providers methods, like Google, Facebook, etc. Authentication also comes with the out-of-the-box services of: SMS confirmation, registration/confirmation email and password reset mechanisms; providing all the most common user authentication-related services.

The only drawback is that you can't directly save some user-specific data during the registration phase. However this can easily be solved by coupling Authentication with Cloud Firestore, hence saving those data to the database (with a reference to the user ID) whenever the user registers.

2.2.3 Cloud Storage [3]

Cloud storage is a hosted storage repository, that can be used to store almost every kind of digital file like: images, videos, 3D models, music, etc. With the use of this service, a system can benefit from a secure storage for assets that are also available for download for external usages. In our app this it's particularly useful for storing the 3D models used in the AR visualization of products; whenever the app requests a model, it is downloaded from Cloud storage to the phone.

Also images should be stored here, but for the demo they are directly retrieved from specialized websites.

2.3 Screens Architecture

In this paragraph we describe what are the main views of the app and their behaviour. Before diving into each screen description, here we present a flowchart-like diagram of the UI navigation flow.

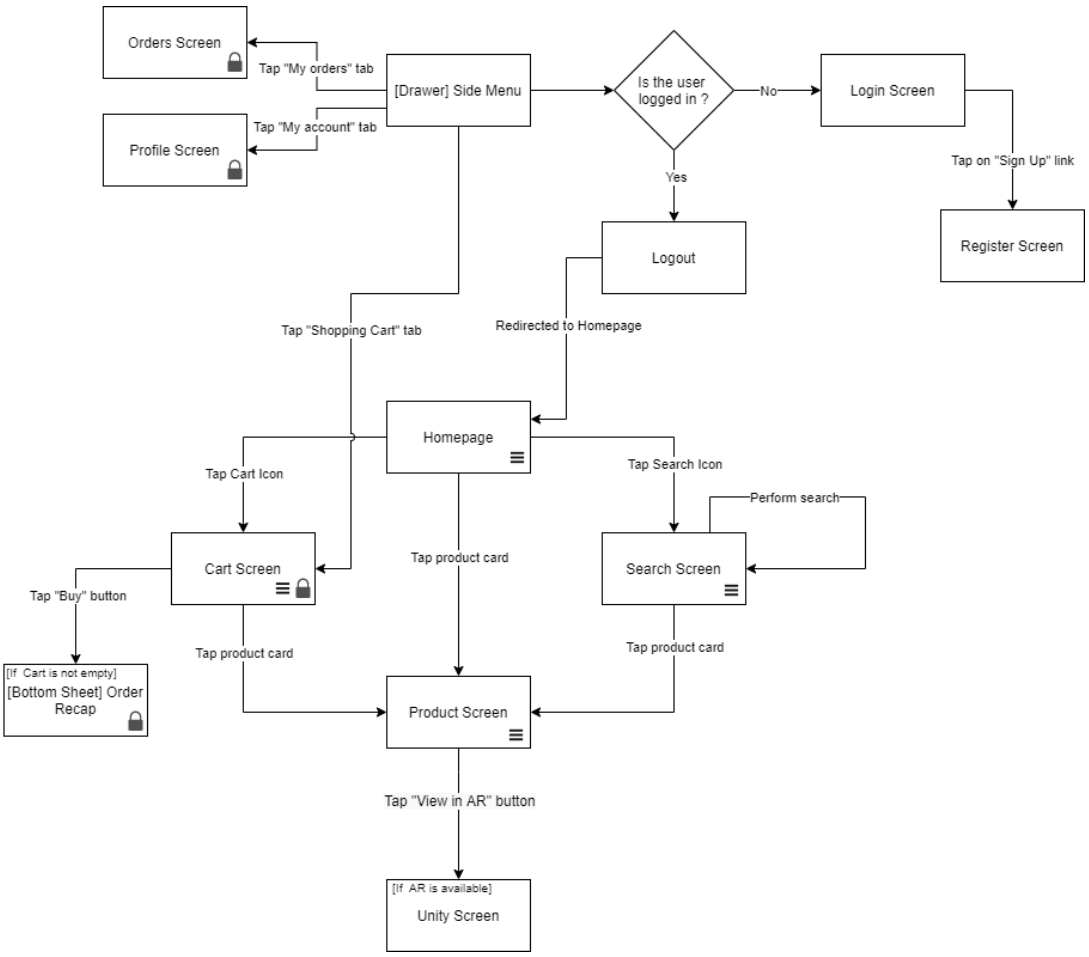

Figure 2.2: User navigation flow diagram of the Realiteye mobile application

In order to be able to fully understand the Figure 2.2, here we list some useful clarifications about the diagram:

- Although every arrow points only forward, it is implicit that if the user clicks on the back button (either from the appbar icon or from the system button) the app returns to the previous page, as it happens in all common mobile applications;

- The lock icon inside some boxes means that that screen is visible only if the user is authenticated. Hence, if the user is not logged in, some screens are simply not reachable;

- Those boxes that contain a menu icon intend that from that screen it is possible to open, on top of the current page, the drawer (which is a lateral menu that appears on the left of the screen), by simply performing a swipe to the right from the left edge of the screen. In the diagram, the drawer is depicted with the "[Drawer] Side menu" box and from this menu it is possible to reach other pages as shown in the diagram. Moreover, since from the drawer it's always possible to go back to the page from which it was opened (simply by tapping out of it), in the diagram, in order to avoid filling the figure with arrows, it is represented as a sort of standalone component.

2.3.1 Views description

Homepage

The homepage is the starting point of the application. It displays a list of products that are tagged as "Popular" or as "Hot Deals". The products' important information are visualized in cards that contain: name, thumbnail, discounted price, discount amount and rating. Those cards are clickable, and if so they redirect to the related Product Screen. From this page, it is possible to reach the Search Screen and the Cart Screen (if the user is authenticated) by tapping on the relative icons.

Drawer - Side Menu

As mentioned before, the drawer is a lateral menu that can be viewed from some screens (indicated in Figure 2.2 with a menu icon) by swiping to the right from the left edge of the screen. It is not really a screen by itself, as it appears in overlay with respect to the screen it was opened from, but it's worth mentioning for some of its unique functionalities.

If the user is logged, it shows the current user account together with its avatar, and by tapping on the respective tabs, it's possible to reach: the Profile Screen, the Orders Screen and the Cart Screen (if not already on that screen). Moreover, the user can logout from the app by clicking on the relative logout tab; the user

will then be redirected to the homepage. On the contrary, if the user isn't logged in, the logout tab will be substituted by a login tab that redirects the user to the Login Screen.

Regardless of whether the user is logged in or not, from the drawer it's always possible to switch the language of the app (between English or Italian), and the app theme (between dark and light theme) by clicking on the respective icons.

Product Screen

This page contains all the product related information. It consists of: a carousel of images that shows some pictures of the product, the name, the price together with the discount applied, the categories to which the product is associated, the rating and the description. If the selected product has an associated 3D model, the "View in AR" button is enabled, otherwise it's disabled. If the user is currently logged in, in the appbar it is also visible an add/remove cart item icon, allowing the user to add or remove that product from its cart.

Cart Screen

This page shows the products that are in the user cart and it is not visible if the user isn't authenticated. From the cart, it is possible to change the quantity of the products and, if needed, the user can remove the item from the cart by doing a right-swipe. The items in the cart are consistent across devices; initially they are fetched from the database and stored in the Redux store at the start of the app. Then, if the user updates its cart, those changes are reflected only in the local Redux store and, when the app goes to background or it's terminated, the current cart is pushed to the database.

If the user has at least one product in the cart, then the "Buy" button is visible; if pressed, the Bottom Sheet widget appears. This widget is a recap of the order: it lists the products in the cart, the total price and provides the user the ability to select the delivery address and the payment method they want to use (those last two information are retrieved from the user profile). Whenever the user clicks on the confirmation button, an order is created and will then be visible through the Orders Screen.

Orders Screen

The Orders Screen allows an authenticated user to see, in different tabs, the currently "In progress" or "Completed" orders they have. The user can view the order id, the date in which the order was issued and the expected delivery date.

Profile Screen

This screen contains the user's personal information. In detail it shows: the user avatar, the email, the birthday date, the list of saved delivery addresses and the list of payment methods.

Login/Registration Screen

The user that wishes to login can do so by pressing the login tab on the drawer. The user will then be redirected to the Login Screen, in which it will be asked for the email and the password. If one wishes, it is also possible to login with Google of Facebook credentials.

If the user doesn't have an account, he can go to the Registration Screen by tapping on the "Sign Up" link. The Registration Screen must be filled by the user with the following information: email, first name, last name, password. However, it is also possible to register through the aforementioned methods, that is to say, Google or Facebook. In that case, the user just has to login through the desired method and all the required data (the same as the classic registration) is gathered directly from the relative authentication provider.

Search Screen

This screen is used to make search queries on products. The user can simply search for a specific item by entering its name in the search box. If one wishes to perform a more detailed search, some additional filters can be specified. Such filters are: view only the products that have a 3D Model, set a price range and select a set of categories. Finally, the user is able to order the result by price (cheapest/most expensive first). The products that satisfy all the filters are then displayed in a scrollable list inside the screen. Each product's most important information is displayed in a card, like the ones in the homepage. Each cart is clickable and, if so, it will redirect the user to the relative Product Screen. The app also saves the search history of the user so that it can quick-search that same

items later on.

Unity Screen

Whenever a user taps on the "View in AR" button in a product page, the app checks if the relative 3D model had already been downloaded; if not the bundle is downloaded from Cloud Storage and saved to the local device storage. The location of the bundle in local storage is then passed to the Unity Screen. This screen has the job of opening the Unity Scene inside the app using the `flutter_unity_widget` package. Once the scene is loaded, the assetBundle path is passed to the Unity Scene, then the model is loaded from the assetBundle and it is displayed in the augmented reality environment. The widget also offers a way to dynamically change the main color of the texture of the model.

2.4 User Interface Mockups

This section shows some mockup version of the main screens of the mobile application. Note that these mockups are only a design prototype of the main screens used as a baseline for the real UI layout; hence they do not represent the final appearance of the application. All the mockups were designed using Figma web application.

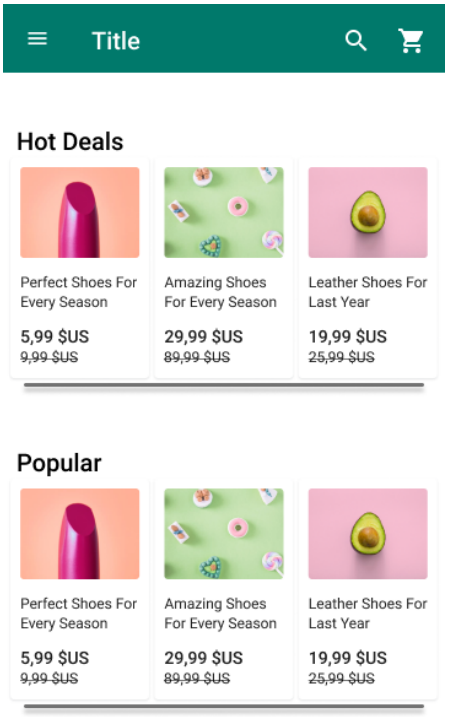

Figure 2.3: Homepage

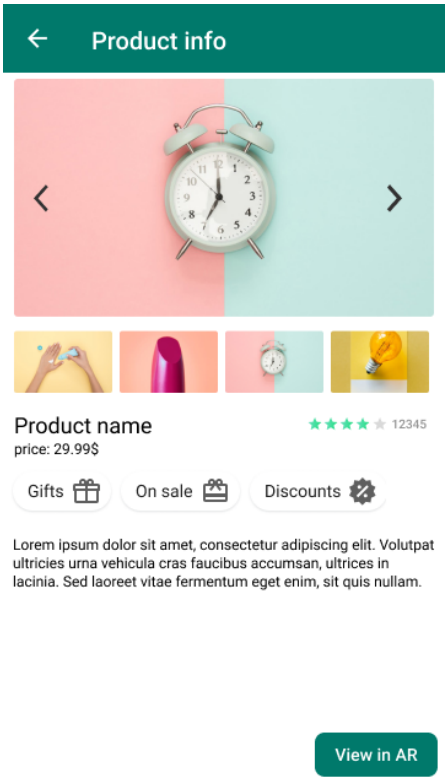

Figure 2.4: Product Screen

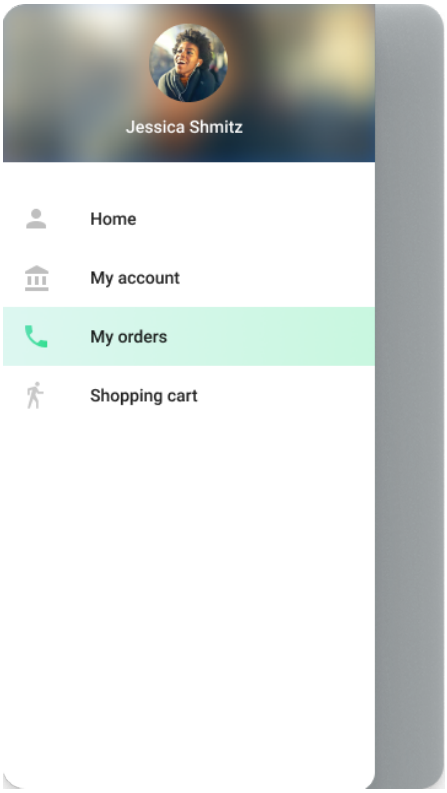

Figure 2.5: Drawer - Side Menu

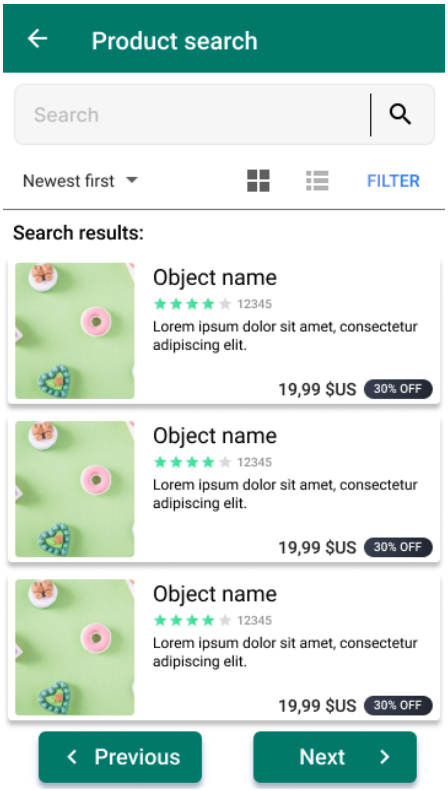

Figure 2.6: Product Search

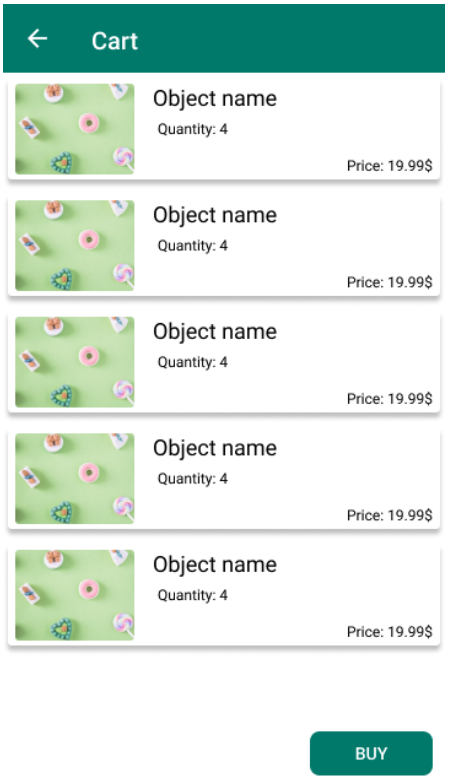

Figure 2.7: Cart Screen

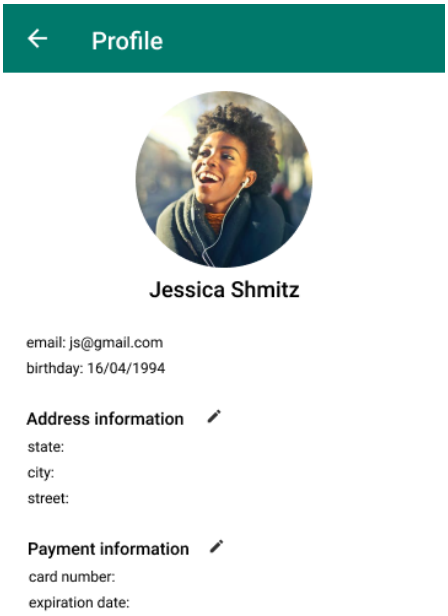

Figure 2.8: Profile Screen

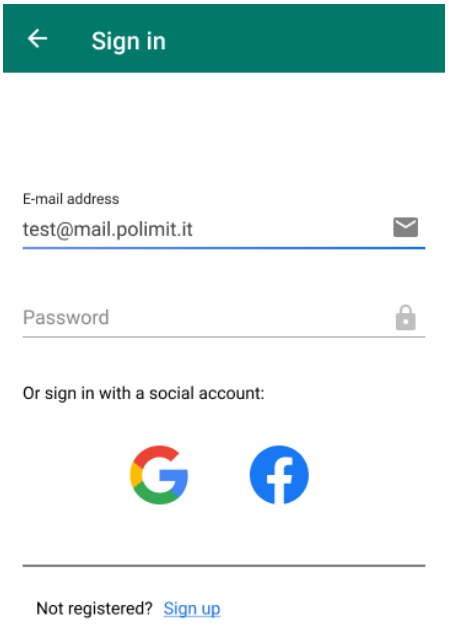

Figure 2.9: Login Screen

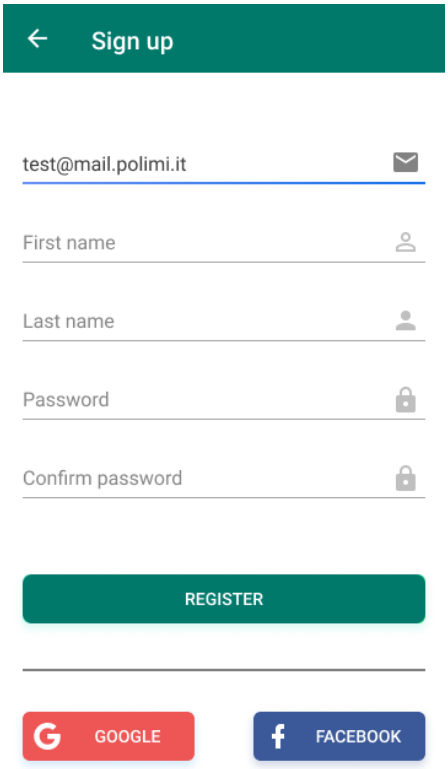

Figure 2.10: Registration Screen

2.5 Database Architecture

2.5.1 High Level Overview

As mentioned before we make use of Cloud Firestore, which is a cloud hosted NoSQL database. For our system, it's important to store data about: users, products, users' cart, orders made by the users and their reviews about the products. The data about these topics are stored in documents, which are aggregated in collections, according to the standard NoSQL paradigm. The following diagram gives a high level detail of the collection design of the database.

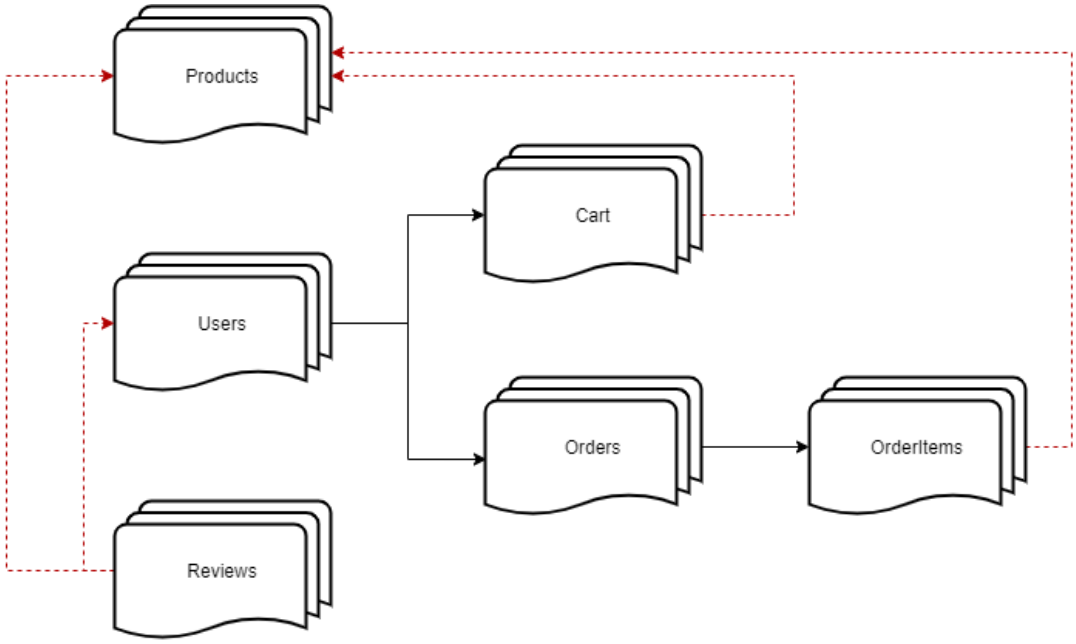

Figure 2.11: High level representation of the database structure

The Figure 2.11 shows that data is organized into three top-level collections: Products, Users and Reviews. Documents under the Users collection contain two subcollections, Cart and Orders. Furthermore documents of the Orders subcollection contain the OrderItems subcollection.

The red dashed arrows, depicted in the diagram, indicate that documents of the starting collection contain references to documents of the pointed collection.

2.5.2 Structure Overview

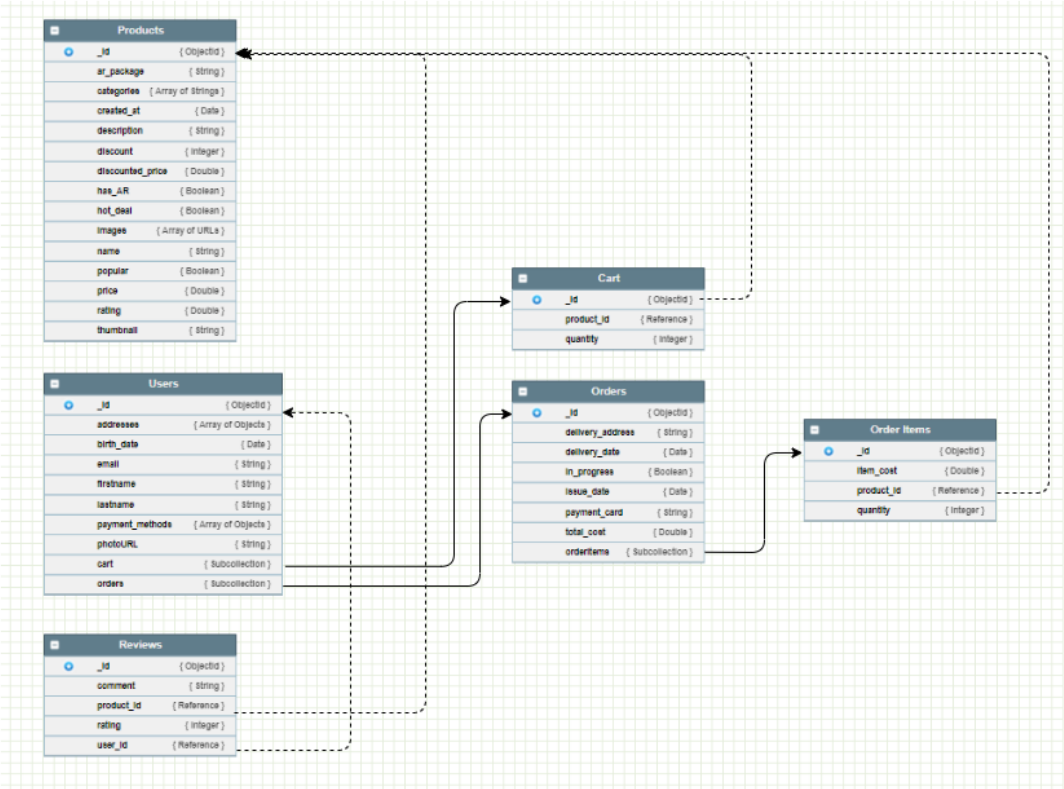

Figure 2.12: More detailed view of the database collections structure

The Figure 2.12 gives a more detailed overview of which data is stored in each collection's document. Every collection is represented as a table and all documents of that collection follow the data scheme provided by the table. In the data scheme, for each field, it's specified its name and the type of data contained in it.

Design Choices

3.1 Redux [4][5]

Redux is a predictable state manager born initially for javascript web applications, but that has been ported for multiple technologies and frameworks, Flutter included. Redux works through simple principles: the state of the application is centralized in a single unique store, that can be accessed for read operations by all application components. To change the state of the application, any component can issue an action, customizable through the action constructor. Actions are handled by the defined reducers to provide a new state. States are immutable, therefore a new state is created from scratch each time a change is performed, avoiding the possibility of side-effects. When a new state is created, all components that depend on the Redux state are updated with the new changes, providing therefore a centralized logic to manage important aspects of the application state, in particular if they need to be persistent across screens. In our application, we decided to use Redux as a state manager to store information about:

- Logged user;
- Cart items;
- Current app theme (light or dark);
- Search history.

This data is used in multiple components of the application and must be kept and updated in a consistent way during all the application lifecycle. For this reason, we found Redux a good choice for persisting state variables through screen navigation and keep consistency between current state variables value and the widgets appearance.

3.2 Firebase BaaS

Firebase with its flexible and scalable services, provides a out-of-the-box environment for creating a fully functional backend. Firebase also lay out a well documented API to interact with their services, defined for many languages and frameworks (Flutter included). Main aspect and benefits of the services used in the system have been described in the external service section.

As mentioned in the architectural design part, the main component of the system is the client mobile application, on which the most part of the development effort has been spent. In order to fast prototype the demo and have a functional and polished application for the given deadline, we decided that Firebase services, even if less customizable than other options, were well suited for supporting the backend bone of the system.

3.3 NoSQL

Our initial idea was to implement the system database with a SQL one, because we have much more experience with the SQL technology due to other university courses. Firebase Cloud Firestore is instead a NoSQL database, but we decided to use it because it could be easily integrated together with other Firebase services, resorting completely to Firebase for the backend part of the system. NoSQL presents as advantages a flexible data structure, that has been useful for faster prototyping, as we had not to modify the database directly when we made changes to the data structure of the collections. A disadvantage of NoSQL is that complex queries can be slow. Moreover, Firestore has limitations regarding complex queries, not every kind of query can be executed yet; also that may require additional composite indexes to work.

3.4 Unity / Augmented Reality (AR)

One of the main goals of the application is to provide a way to have a meaningful 3D representation of a product, that users could view to help them during their shopping decisions. The augmented reality technology is well suited to provide this kind of innovation, because potential buyers can see a 3D model of the product directly in their homes or working places, view it from different

angles and possibly interact with the models through some scripts to provide even more information and features. Before starting to think about the whole project structure, we have checked the possibility of using the AR technology in a Flutter application and we found that it was possible with the help of a package that integrated a Unity project as a Flutter widget. Unity is a well known game engine used by professionals and amateurs, but in recent years has evolved to work also very well with AR and VR technologies, making it a perfect candidate for our system.

Implementation

4.1 Unity Project

A Unity project is embedded into the project codebase, under the `unity` folder. A Unity project can be composed of multiple scenes, but in our application only one scene is exported. To provide AR functionalities, AR foundation is enabled in Unity package manager, so that its GameObjects and scripts can be used in the scene. The scene contains scripts that allow to find planes for the AR environment and to place a 3D model on a found plane, covering the product visualization functionality. The 3D model is not directly linked in the scene, instead it is loaded at runtime from an assetBundle, that is downloaded by the Flutter application logic, from remote storage, and passed as a local URI. In this way, using a single scene, it's possible to visualize and interact with any possible 3D model, it's only needed to convert it in an assetBundle with an apposite script provided into the Unity project itself. An additional script allows to change the color of the model's texture, given a string encoding. The Unity project is exported following the `flutter_unity_widget`[6] package guidelines.

After exporting the project, the package allows not only to visualize the Unity scene in a widget, but also to call functions and exchange messages from Flutter to Unity and vice versa. To demonstrate this behaviour, in the Unity screen of the Flutter application it's possible to click on color boxes to activate the color change script mentioned before, applying the color on the model.

It's important to note that, with advanced knowledge of the Unity engine and AR technologies, much more sophisticated functionalities can be developed in the same way. However, here for the sake of simplicity, we limited the scripting and interactions to a simple showcase.

An important consideration is that, because AR foundation uses ARcore and ARkit to leverage AR compatibility on Android and iOS platforms, their API level requisites must be satisfied to correctly build the Unity project. In

particular, for Android, a minimum API level of 24 (Android Nougat 7.0) is required.

4.2 Flutter Project

The code of the Flutter application can be found under the `lib` folder. It is organized in various subfolders:

- `generated` folder simply contains keys used by the localization package, generated from the JSON files that contain the translations (one for each language). These keys are useful to avoid hardcoding strings into the widgets, making the translation less error prone and easier to maintain;
- `models` folder contains simple classes, composed of only variables (sometimes also very simple functions), that gives a fixed structure to commonly used data;
- `redux` folder contains the classes that define the Redux state structure, the actions that can be performed on the state and the reducers that specify how these actions are handled, to generate the new state;
- `ui` folder contains widgets and screens. Screens are complex nested widgets that define a screen; each of these screens is associated to a route used by the navigator, defined in the main file. Widgets instead are simpler and smaller widgets, parametrized and stylized to make them reusable in multiple screens with low effort;
- `utils` folder contains general utility functions and the database operations, used by all the various screens of the application to read data from Firestore and perform operations on it;
- `view_models` folder, like `models`, contains simple classes too. These classes are used in some screens to extract the subpart of the Redux state required for displaying them and defining the Redux actions that could be performed in the screen;
- the `main` file, that is the entry point of the application. The application have a complex initialization due to the services and features that supports, composed of these steps:

1. The downloader is initialized;
2. The initial Redux state is initialized;
3. Screen orientation is fixed to portrait;
4. Localization package is initialized and then run the actual application widget;
5. Firebase is initialized, waiting it to complete before building the actual application widget;
6. Redux store provider, a gesture detector, used to improve focusable widget interactions, and the application life cycle watcher are instantiated, wrapping the whole application;
7. The material app is defined, along with the themes and navigation routes.

4.3 Important Flutter Packages

- `flutter_unity_widget`: enable the integration of a compiled Unity project into a Flutter application, providing a widget to visualize Unity scenes that can be integrated in the Flutter widgets tree;
- `flutter_downloader`: provide methods to easily download external files, given their URLs, like the Unity assetBundles stored in Firebase Storage;
- `redux, flutter_redux, redux_logging, redux_thunk`: package ensemble required to implement Redux into a Flutter application;
- `firebase_core, firebase_auth, firebase_storage, cloud_firestore`: package group required to initialize Firebase and use the services required by our application;
- `easy_localization`: enable localization in the entire application in an easy and scalable way. JSON files with the translation must be implemented and imported in the application's assets.

4.4 Seed Application

To properly showcase the application, Cloud Firestore database is seeded with dummy data produced by a simple NodeJS application (found under the

`Seed_App` folder in the project codebase). The seed application generates the data through the Faker npm package, complimented with URLs from sites like LoremPicsum (https://picsum.photos/images) to provide images URLs used in product photo galleries and thumbnails. After the data is generated, it is passed through the Firestore API to populate the database. Part of the dummy data produced is also saved in local JSON files, that can be used for testing database queries when using apposite mocks (see next point).

4.5 Testing

We have developed some unit testing regarding the database operations performed by the Flutter application, that are frequent and very important for the application functionalities. Testing environment is set up with a Firestore database mock, using the official `cloud_firestore_mocks` Dart package. Before all tests, the JSON files, containing dummy data generated from the seed application, are read and the data is stored into maps. Before each test, the data contained in the maps is used to populate a mock instance, so that a fresh instance with a fixed determined state is provided to all tests to perform (this avoids possible side-effects of previous tests).

We also tried to perform widget testing, but unfortunately because of the complex initialization of the application caused by Firebase, Redux and EasyLocalization packages, it is very hard to perform this type of tests, because each widget and screen is dependent on them and so they should be mocked and initialized before each widget instantiation. Extended manual testing has been made to ensure the quality and stability of the application, to cover the inconvenience of not having an automatic test suite for testing the UI.

Testing has been performed on multiple emulated devices, having different aspect ratios and API levels, and our personal devices.

Bibliography

[1] Google Developers. *Cloud Firestore Documentation*. 2020. URL: https://firebase.google.com/docs/firestore.

[2] Google Developers. *Firebase Authentication Documentation*. 2020. URL: https://firebase.google.com/docs/auth.

[3] Google Developers. *Storage Authentication Documentation*. 2020. URL: https://firebase.google.com/docs/storage.

[4] Dan Abramov and the Redux documentation authors. *Redux*. 2020. URL: https://redux.js.org/.

[5] Brian Egan. *flutter_redux package*. 2020. URL: https://pub.dev/packages/flutter_redux.

[6] Rex Raphael. *flutter_unity_widget package*. 2020. URL: https://github.com/juicycleff/flutter-unity-view-widget.