POLITECNICO
MILANO 1863

Design and Implementation of Mobile Applications
plants

2020-2021

Authors:
Federico Ferri
Alessio Galluccio

Professors:
Prof. Luciano Baresi
Giovanni Quattrocchi

Contents

1 Introduction

This document is to describe the design and the implementation of the *plants* mobile application with emphasis on the user experience and the technology stack used.

This project is the evaluation work for the Design and Implementation of Mobile Applications class at Politecnico di Milano taught by Professor Luciano Baresi and Giovanni Quattrocchi as teaching assistant.

1.1 Idea

The idea came from a very simple need that arose during this 2020 pandemic as many people were forced to shelter at home and thus had much more time to dedicate to hobbies and personal activities.

During this idea generation step we both shared a common hobby which is maintaining a small home vegetable garden.

While sharing this hobby we both shared also the same problem related to such an hobby which is watering. Watering requires a small attention span but at a very constant interval because even a single miss could cause the plants to die.

We thus had the idea to create a specialized digital tool to manage watering but not of a single plant but of an entire garden because each plant requires different amount of water depending on many different factors.

1.2 Requirements

After coming up with the idea we started devising the functionalities of the app. The app main goal is to remind you exactly at the right time to water the plant(s) that need to be watered.

This has to be very precise because a wrong timing on when to water the plants can damage them or make the application useless by giving too many wrong indications to the user.

Moreover the application should keep the data and the service going regardless if a user changes phone and it must be simple to set up.

In the end we decided to divide the requirements into functional and non functional requirements, the first being related to the core functionalities and the latters being related to a good user experience.

1.2.1 Functional requirements

The functional requirements are all the elements that must be present in the app to make the core requirements possible.

- Sign-in using social accounts such as Google or Facebook to ensure a fast and seamless sign-up and login experience.
- View in a glance your plants and their current status.
- Adjust the status of a plant if the application is currently wrong about it.
- Inform the application about relevant events for a plant such as when it has been watered.
- Add new plants to the application.
- Inform with a push notification when its time to water a plant.
- Use as many information as possible from the user and from external services such as the weather service to predict best when a plant needs water.
- Delete a plant from the application.
- Modify a plant from the application.

1.2.2 Non functional requirements

The non functional requirements are as important the functional ones but related to the aspects of the application that are not core but still necessary to make the user experience enjoyable.

- The application should run both on Android and iOS with the same level of service and features.
- The notifications about important events should be generated and delivered in a timely manner even if the app is in deep sleep mode or frozen.
- The application can use the photo camera when needed.
- The application can use the positioning services when needed.
- The user can disable push notifications if deemed too invasive.

1.3 Assumptions

We assume to simplify the design and the development some characteristics about the environment and the device where *plants* runs on.

- The connection to the internet is always available, reliable and with a good speed.
- The third parties API (application programming interface) and services are always available.
- The device where *plants* runs on is free of any limitations that would impact the functioning of the application such as unavailable storage or too strict permission limitations.

2 Design

2.1 Framework

Our framework choice between the ones proposed in class (Flutter, React Native, Android, iOS, Progressive Web Apps) has been decided based mainly on cross platform compatibility.

The requirement already excludes pure Android and iOS development from the spectrum.

We decided to land on React Native because we were inspired by the huge ecosystem of packages offered for React Native and the immense popularity it has.

We decided to exclude Flutter because it is based on DART a language that we have never used before. Moreover also Progressive Web Apps were a very interesting candidate but we felt that using them was not in the theme of the course because they are primarily developed using web technologies and adapted to be used as mobile applications.

2.2 Architecture

We decided to use as many external services as possible to keep the application codebase lean and to reduce the development effort.

The system architecture consists in the **client**, the firebase services such as **Firebase authentication**, **Firestore** and **Firebase storage**. The **notification service** and the **weatherapi.com** service. Moreover we have some other services we rely upon such as the **Google authentication**, the **Facebook authentication** and the **Google cloud functions**.

2.2.1 Client

The client consists in the React Native application and is the core of the entire project. We decided to use the SDK 40 of React Native because it is a good and stable version of the SDK.

The biggest choice we made is to use Expo. Expo is a platform that allow to speed up React Native development at the expense of less freedom in terms of functionalities. We analyzed our needs and decided to proceed in using Expo due to the advantages in terms of using a real mobile device for development and being able to run it off real iOS and Android devices without having to deal with platform specific stores.

We also decided to use an approach based more on classes than function in our React Native codebase because we have often to deal with internal state of the components that compose the screen so the usage of classes is advised.

For cleanliness we decided to divide the codebase into logic blocks:

- **Assets**: these represent the various binary imports of the components such as images and fonts.
- **Components**: the components represent some elements in a screen, we decided to group them together to reuse them as much as possible into different sections of the application.
- **Config**: the configuration files used to store configuration settings such as API keys.
- **Navigaton**: similarly as the components here we can find the elements related to navigating the various app's screens.

2.2.2 Authentication

Making sure that a good authentication process is in place is not an easy task and it is often overlooked even by major mobile applications. We decided to use social sign-in to completely avoid the risks of rolling our own authentication service.

We use Firebase authentication to aggregate login data of our users and to manage authentication. As social login providers we use Google and Facebook and to enable this we registered a Facebook application to obtain a Facebook App Id and we also registered a Google application to obtain the ability to login using Google.

2.2.3 Database and storage

To store all of our data we user Firestore a NoSQL database offered by Firebase. This database is quite different compared to a traditional database. It's a database that offers a client SDK to interact with.

Traditionally databases only offered interfaces for server side applications to interact with it, with Firestore it's possible to have an application that saves its state without having a custom backend.

We designed our database with these collections, collections are the equivalent of table for traditional SQL databases:

- **users**: here in each document (the equivalent of a row in a SQL database) we can find some data about user preferences in receiving notifications and its Expo notification token. The token's function will be better explained in the specific section.

- **plantTypes**: each document represent a specific plant type supported by the application. They are stored server side to allow adding a new plant without having to deliver a new release. Each plant type has a few characteristics parameters used by the watering algorithm.
- **plants**: the most important collection stores each plant added by any users and its status and history. It's a very dense collection but this is advisable in a NoSQL environment to allow for better performances since joins are not advised.

To store large binary files generated by users such as images we use Firebase storage a bucket service to store large files and serve them to the internet.

2.2.4 weatherapi.com

To gain information about the weather in a particular location we use a third party service. We landed on this service because it offers a nice API and moreover it serves icon to represent the weather in a graphical way.

2.2.5 Notification service

Notifications to the clients are not generated on the client itself in background but are generated and sent remotely. This is because many devices tend to kill or send to deep sleep and application that is not used very often to save on RAM and CPU usage. Such a behaviour would make our service useless due to unreliability.

We then decided to develop a Google cloud function that runs periodically and checks the Firestore database for upcoming events. If such events are found notifications are sent to the clients using Expo notifications. This is fine for an unpublished application in a public application available on the stores it would be wiser to use a service such as Firebase notifications.

Moreover inside Firestore as mentioned before is stored the Expo token needed to send such notifications to the device and this token is retrieved and updated on Firestore every time the user opens up the client application.

3 Implementation