TREBALL FINAL DE GRAU

TÍTOL DEL TFG: Boletify: a machine learning-based application to identify mushrooms

TITULACIÓ: Grau en Enginyeria Telemàtica

AUTOR: Eric Blanca Gómez

DIRECTOR: Antoni Oller Arcas

DATA: 29 d'octubre del 2021

Títol: Boletify: a machine learning-based application to identify mushrooms

Autor: Eric Blanca Gómez

Director: Antoni Oller Arcas

Data: 29 d’octubre del 2021

Resumen

Cada vez son más frecuentes las soluciones basadas en machine learning. Aunque ya han pasado varias décadas desde su creación la tecnología ha ganado popularidad y adopción estos últimos años gracias a herramientas como Jupyter o Anaconda, así como también la investigación sobre redes neuronales, deep learning e inteligencia artificial. Ejemplos de esto podrían ser la conducción automática, los clasificadores o el análisis de datos.

Esta aplicación permite clasificar setas mediante reconocimiento de imagen a través de un modelo clasificador implementado con Tensorflow Lite, el cual está especialmente pensado para trabajar en móviles y “edge devices”. Además, debido a que un tipo de aplicación como esta es común usarla en lugares donde no hay conexión a internet nos hemos asegurado de que mantenga todas las funcionalidades en modo offline. Esto último lo hemos conseguido moviendo el modelo clasificador dentro de la aplicación nativa en lugar de obtener la respuesta del servidor, y creando una copia de la información de las setas en el almacenamiento local del móvil.

El frontend de la aplicación ha sido desarrollado con Flutter, un framework relativamente en auge que nos permite crear aplicaciones para Android, iOS, web y escritorio usando prácticamente el mismo código. Aunque por la particularidades de nuestra herramienta solo consideramos las plataformas móviles.

El resultado de este proyecto es la publicación de una aplicación Android en versión alfa capaz de identificar diferentes tipos de setas mediante un modelo clasificatorio basado en machine learning. El proceso de publicación e integración se ha automatizado, de forma que los cambios realizados en el repositorio de Github generan una nueva versión la cual es subida a la Play Store directamente.

Títol: Boletify: a machine learning-based application to identify mushrooms

Author: Eric Blanca Gómez

Director: Antoni Oller Arcas

Date: 29 d'octubre del 2021

Overview

Machine learning-based solutions have become very popular in recent years. Although it has been decades since its first use, the technology has gained popularity and adoption thanks to tools such as Jupyter or Anaconda, as well as the increase in the research on neural networks, deep learning and artificial intelligence. Examples of this could be automatic driving, classifiers or data analysis.

The presented applications allow classifying mushrooms by image recognition through a classifier model implemented with Tensorflow Lite, which is specially designed to work on mobile and edge devices. Furthermore, as the classification would usually be used in places where there is no internet connection we have developed an offline mode that keeps all functionalities in those cases. We have been able to achieve it by moving the classifier model inside the native applications instead of getting the response from the server, and also by creating a copy of the mushroom information in the mobile's local storage.

The frontend of the application has been developed with Flutter, a relatively booming framework that allows us to create applications for Android, iOS, web, and desktop using almost the same code.

The outcome is an Android app in Alpha version that is available from the Play Store only for internal testers, it is able to identify several types of mushrooms and to work offline. The publishing and integration process has been automated, so changes made in the Github repository generate a new version which is uploaded directly to the Play Store.

ÍNDEX

INTRODUCTION

Even though machine learning has been a reality in research labs for decades, it is now getting enormous attention for real-world applications. Nowadays topics such as machine learning, deep learning, and artificial intelligence are reaching a wide public and we can find them almost everywhere. The opportunities are big and for a lot of companies, it is the right time to take advantage of the benefits, sophistication, and power of ML-based tools.

The main goal of this project is to use a classifier model built using machine learning in order to identify different types of mushrooms through image processing. Furthermore, I want to speed up the time to market as much as possible using powerful development technologies, and to be able to automate the integration and delivery process so the product is as similar as possible to what we would find in a real-world project.

Chapter 1 will present some context about the motivations to build this tool, which is the state of the art in the market and which are the goals to achieve with this project.

Chapter 2 will go through the design process: branding, user interaction, and user experience.

Chapter 3 will present the project architecture, as well as some considerations about the functionalities, requirements and data model.

Chapter 4 will expose the considerations to choose a tech stack for the frontend, backend and classifier; as well as the working environment.

Chapter 5 will be related to a few development details: how tha classifier was trained, how we started the development and some technical implementations.

Chapter 6 will focus on the app delivery to the Play Store: how to set up the app for release and the CI/CD pipeline, what we want to test, how we automate the process.

Chapter 7 will present a time estimation of each phase of the work and how we managed it.

Chapter 8 will finally expose the final conclusions of this project: whether the goals have been achieved or not, which are the learnings and which could be the future improvements in the short and long term.

CHAPTER 1. CONTEXT

This chapter will contain a brief explanation of the personal and social motivation to build this tool, the current state of the market for similar projects, the goals that I want to achieve building it, and which requirements should the product meet.

1.1. Motivation

I am from a small town located in the Catalan Pyrenees, it is quite a touristic place and there are several activities that people can do depending on the season.

During summer people usually come to visit our Romanic-style churches and the beautiful National Park, while in winter people can enjoy skiing in our big snow-covered mountains. You can also enjoy trekking and walking, as well as different activities such as fishing or mushroom picking. I really enjoy this last one, and I know that a lot of people come here during spring and autumn to enjoy it as well.

The biodiversity in our mountains is really huge and it is sometimes hard to distinguish between the different kinds of mushrooms, so I started to figure out how to build a tool able to identify them so people could pick only the mushrooms that they know while learning their features; which would result in a more environment-friendly activity that is more conscious about the goods and dangers of picking them.

I was inspired by a tool that I usually use to learn about plants which is called PlantNet[1], and that can be used to identify them using just an image. I was really curious about the application and how it worked, as I knew thanks to some subjects that I have coursed during my degree that it was probably using some kind of machine-learning model to make that classification.

This last fact, together with my passion for developing frontend products and software, triggered my curiosity to start the initial research about how I could validate my idea with the least effort, while building a free tool focused on *mycology*[2] that could be useful for me and other people that enjoy the same activity.

The results of my research quickly started to result in adding up several isolated pieces that would lead to a product able to perform that task. However, there were still a lot of things to consider, learn, and define.

[1] You can check the official product webpage here: https://identify.plantnet.org

[2] *Mycology* is the branch of biology focused on the study of fungi (properties, toxicity, food, etc.).

1.2. State of the art in mushroom classifier apps

Searching for similar apps that either provide the same functionality or are focused on mushrooms was part of the initial research. From a personal point of view, any of them were targets to learn, improve and compare. The results showed that even though there are some tools that aim to provide the same functionality, none of them is completely aligned with the scope of this project. Here you have the most interesting ones:

ShroomID [1]: Free with paid features. It is able to identify mushrooms through images, probably using a machine learning classifier model as well, it also provides a heatmap based on mushroom localization which makes it a powerful app. As an extra point, it has a nice modern UI and a huge mushroom database. The main drawback is that It is focused on the USA and its local varieties, so it is not appropriate to be used in our case.

Shroomify [2]: Free with paid features. It is able to identify mushrooms by letting the user choose manually its color, cap shape, body shape, etc. The main drawbacks are that the classification system is not automated, the app is focused on the UK varieties and it does not really have a beautiful, enjoyable UI/UX.

Picture Mushroom [3]: Completely free. It allows identification using images and lets you chat with mycology experts to confirm the classifications. It has a professional UI, but users usually complain about bugs and errors, as well as low confidence in its results.

eBolets Catalunya [4]: Fully paid application. It is not able to identify mushrooms, but shows the location and density zones for different mushrooms in Catalonia, which is really important for mushroom pickers. However, it has the worst UI/UX.

Others: There are a few more apps focused on finding or identifying mushrooms; however, they are in some way or another worse than the previous ones. This is usually related to a bad trade-off price/functionality, not automating the identification process, or having a poor UI-UX.

As you can see there are already several applications focused on finding or identifying mushrooms, where the strongest competitor would be ShroomID as it also uses a machine learning classifier and it is modern, performant, and offers an additional feature based on heatmaps. However, mycology is something really focused on the ecosystem and the zone where they are found. Mushrooms from the USA and Europe are quite different, and this could lead to mistakes or misunderstandings. As a proof of concept, I have tried to identify some mushrooms using it, and did not show the expected result, but similar varieties or completely different ones. That is why our application, which is specifically focused on the Catalan Pyrenees varieties, is better for local users.

1.3. Goals

The main goal of the project is to release to the official market a free tool to help identify the different types of mushrooms that we have in the Catalan Pyrenees, it should be available for the biggest number of users and be released within the scope of the project.

I only consider releasing the application to the Google Play Store as a first approach due to publishing fees, setting as the main goal the automation of the code integration and application delivery.

Here is the summary of the specific goals that the project should achieve:

- Identify mushrooms through camera or gallery pictures.
- Automate the classification using a machine learning classifier model.
- Automate the Play Store release process for the Android app.
- Support any app functionality in offline mode.
- Support filtered and named search to display mushroom information.

CHAPTER 2. DESIGN PROCESS

During this chapter we are going to present everything related to the application visual design; including the branding process, the color palette, the user interface flow, and also the custom icons that have been designed during the creative phase.

2.1. Branding

Two main items were considered during the branding process of the mobile application: the application logo and the application icon. The logo will contain the name and identify the brand, while the icon will be used to identify the application inside the device.

Regarding the name, the final result is a portmanteau[3] word using a mixture of: 1) *bolet* which stands for mushroom in Catalan, but also *Boletus Edulis* which is the scientific name of one of the most valued mushrooms in Catalonia; and 2) *identify* word, which is the main feature of the tool. The result of the mixture is *BOLETIFY*, which is a catchy, easy-to-remember name that also represents the complete app functionality.

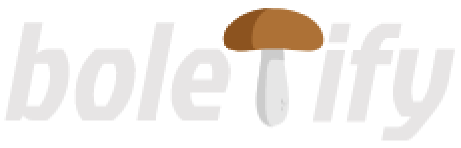


Fig. 2.1 Boletify application logo

As the T letter remembers the shape of a mushroom it was decided to convert it to a meaningful image for the logo (see Fig. 2.1), which also evokes the scope of the application: mushrooms.

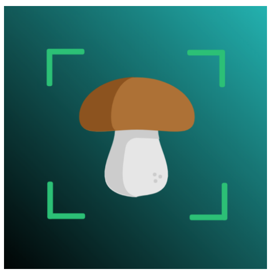


Fig. 2.2 Boletify application icon

The same image was used to design the application icon together with a camera focus shape, which reinforces the idea of a tool to identify mushrooms while adding an extra visual item (see Fig. 2.2).

[3] A portmanteau word is a blend of words in which parts of multiple words are combined into a new one.

2.2. Color Palette

The color palette aims to represent the look and feel of Nature using the colors that we could find in the Pyrenees.

For that reason, the chosen colors contain green and blue tones (see Fig. 2.3), even though other colors will be used such as white for the text, or black, brown and orange (FFAB40) in some places like the navigation bar.

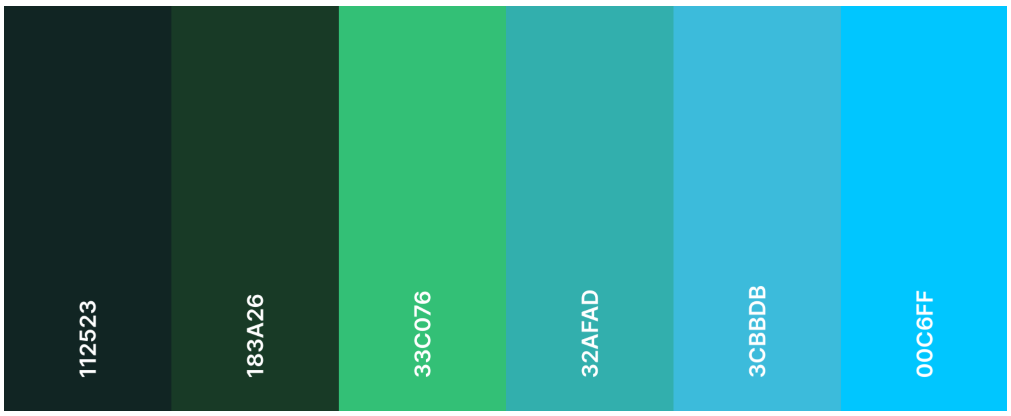


Fig. 2.3 Boletify color palette

2.3. Custom Icons

Mushrooms will be classified using tags, so several custom icons have been designed in order to make the filtering feature more friendly and visual.

The tags have been split into two main categories: edibility and season. Each one of these tags is represented through an icon (see Fig. 2.4 and Fig. 2.5), edibility ones are mutually exclusive while season icons will be complementary.

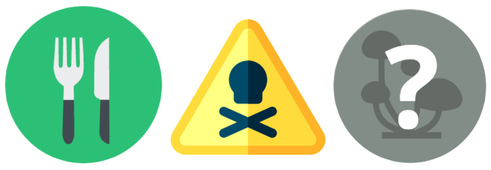


Fig. 2.4 Edibility icons: edible, toxic, and unknown

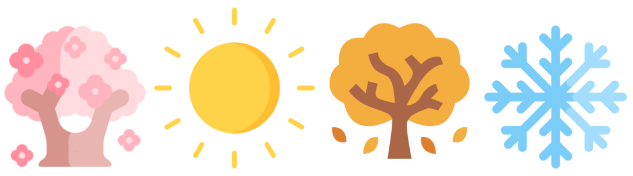


Fig. 2.5 Season icons: spring, summer, autumn, and winter

2.4. User Interface Flow

The user interface flow has been simplified in order to keep the application as intuitive as possible to highlight its main feature: the identification of mushrooms. The main concern regarding the user experience is that a user has to be able to open the app and classify a mushroom as easily and fast as possible.

However, a user has to be able to check which are the mushrooms available by season and which of them are edible or not, so a filtered search and a named search have been added as well to the main flow.

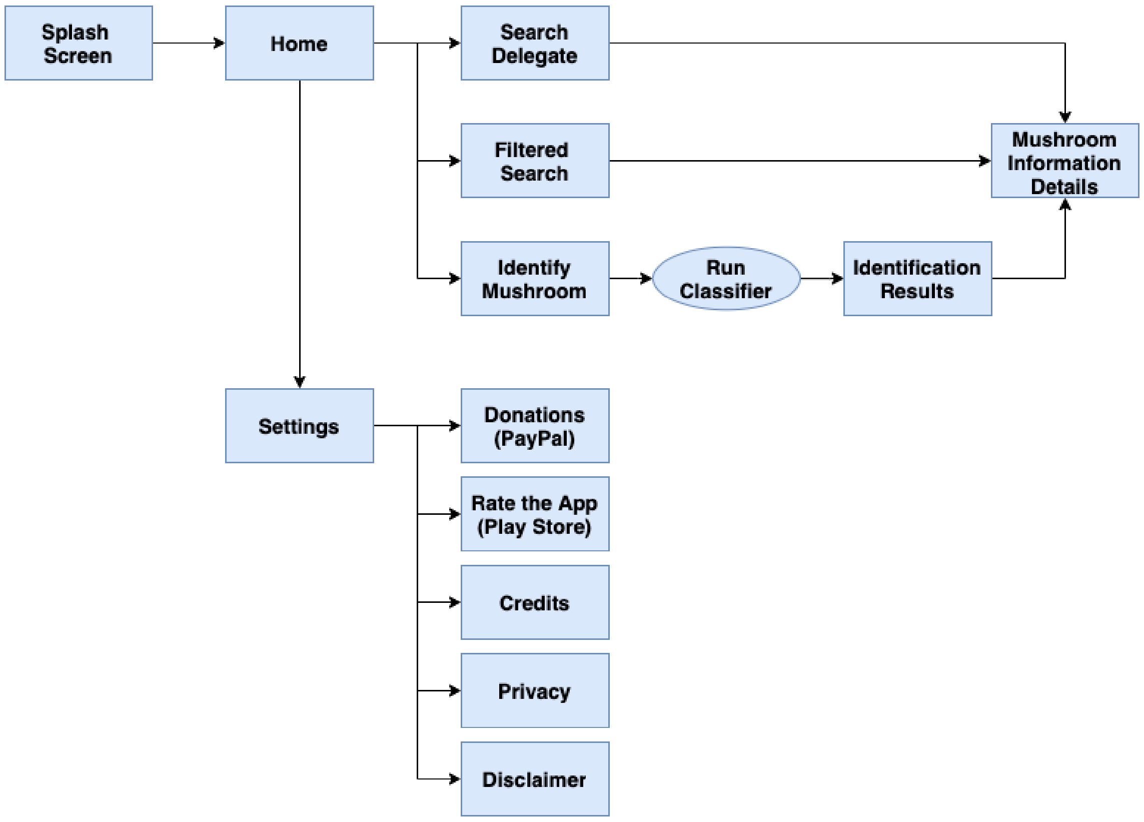


Fig. 2.6 User interface flowchart and screens

In order to highlight the main functionalities only two main screens have been defined: Home and Settings, so the user is able to access the main features from the home page and no additional navigation is required. Due to previously stated, the following items will be added to the Home screen:

- A floating button to choose an identification method and get the classification result.
- A filtered search list to find mushrooms based on tags.
- A search delegate to find a mushroom by its name.

CHAPTER 3. APPLICATION ARCHITECTURE

The following chapter presents several considerations related to the app architecture. Specifically, the base architecture schema, how the offline mode should work, and which will be the main data model entities.

3.1. Base schema

The classifying tool will take the shape of an Android or iOS native application. Even though building a web application would take little effort, that approach has been dismissed, as it is not coherent with the idea of classifying a mushroom *in situ*.

This application has to be able to interact with a Backend using an API in order to get a list of mushrooms information that will be displayed in our user interface. The only hard requirement for this backend is to have a database where we can save and get that information. In addition, another nice-to-have would be an analytics system to track how the product is performing. Other functionalities such as Authentication, Push Notifications, or any other backend services have been dismissed for this project.

Last but not least, the application also has to be able to run a machine learning model locally using camera or gallery images, as well as to save and recover fallback information from its own local storage.

You can see the base schema of the project in the image below (Fig. 3.1).

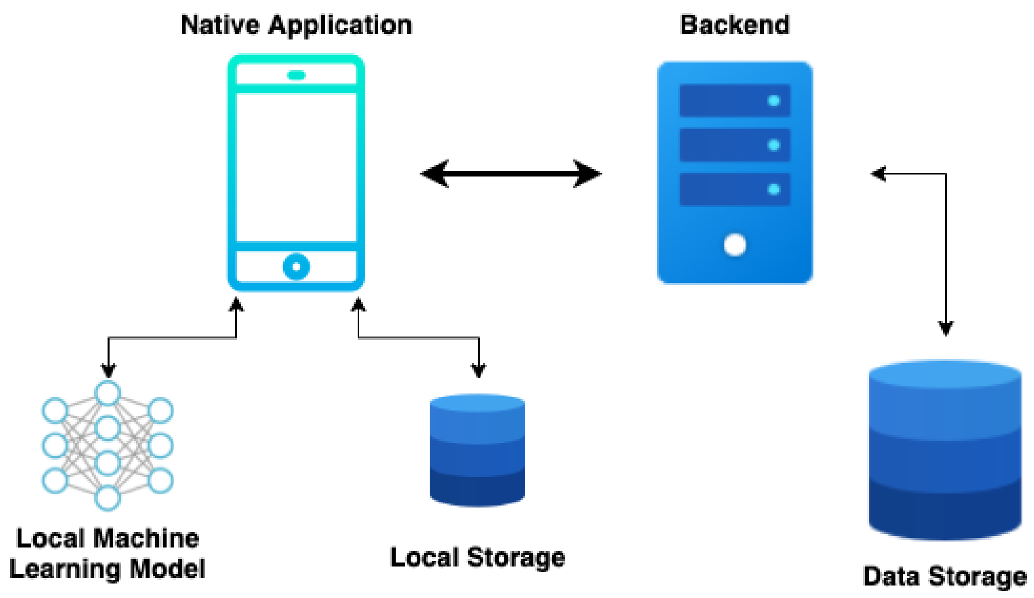


Fig. 3.1 Base schema for our application

3.2. Offline mode

The offline mode will be a key feature of the application: it has to be able to keep all functionalities without an internet connection, as a user could probably use the application in remote places where there is no mobile connection coverage.

Two main action points have been defined in order to support the offline mode:

- Place the classifier model inside the native app instead of getting the result from the server using an API request.

- Keep a fallback copy of the mushroom information that will return the server in the device local storage, so we can recover that data later.

The last action point requires some additional definition regarding how and when to save or access that data.
When the user has a stable internet connection we will fetch the mushrooms data from our Backend, then we will save that information to the device's local storage, and finally we will redirect the user to the Home screen. When there is no internet connection, we will try to get that data from the local storage, and then redirect the user to the Home screen as usual.

If for any reason we do not find a fallback copy of the data we will have to redirect to an error screen and let the user know that we could not fetch the information. The classifier will work perfectly fine, but we will not be able to show any mushroom detail. For instance, this edge case could happen when a user downloads the application but does not open it while having a connection, so we won't be able to fetch and save the information the first time.

You can see in Figure 3.2 the flowchart of our application startup.

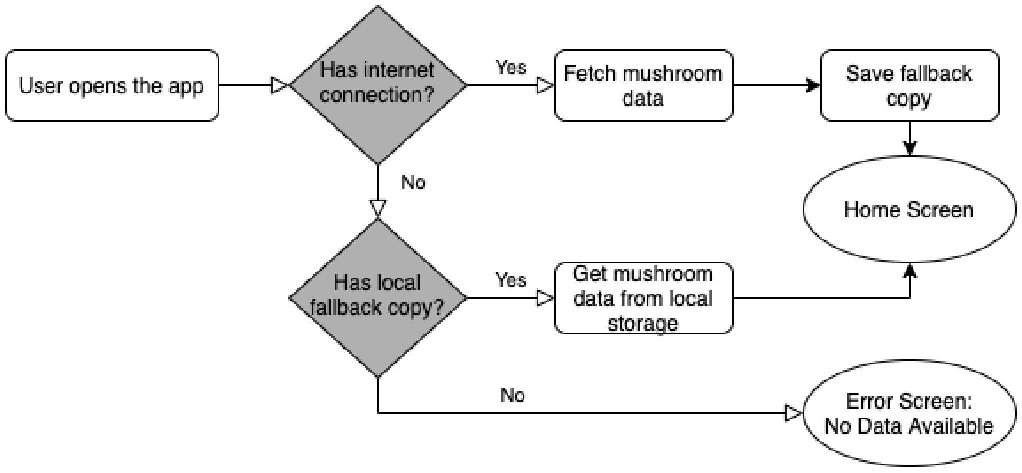


Fig. 3.2 App startup with offline mode flowchart

3.3. Data model entities

The application does not need complex entities or relationships, so only two main entities have been designed: ClassifierOutput and Mushroom.

The ClassifierOutput is used to map the classifier output to a defined object model that we can use. This model will have a confidence percentile, a label that will identify a Mushroom and the image sent to it. To build the object we will use a factory method ClassifierOutput.fromTflite, which will return a new ClassifierOutput (see Fig 3.3).

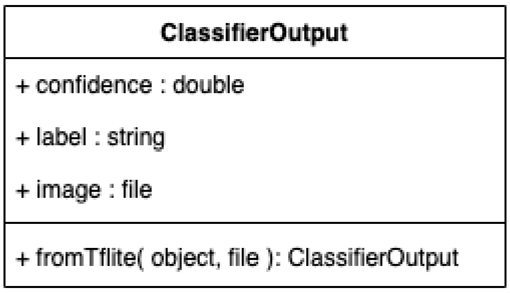


Fig. 3.3 ClassifierOutput entity

The Mushroom model (see Fig 3.4) will have an *id*, a *name*, and several parameters that will be used to display any information regarding that kind of mushroom. It is worth mentioning the *tags* property that will be used to filter the mushrooms by class. This property will contain an array of TagInfo, another entity that is used to link a Tag with an image, description and label. Finally, the Tag model will be an enumeration of the different qualities: spring, summer, autumn, winter, edible, toxic, unknown.

Regarding the mushroom methods, most of them will be used to manage the offline mode (*fromJson, toJson*) or to get the information from Firebase (*fromFirebase*).

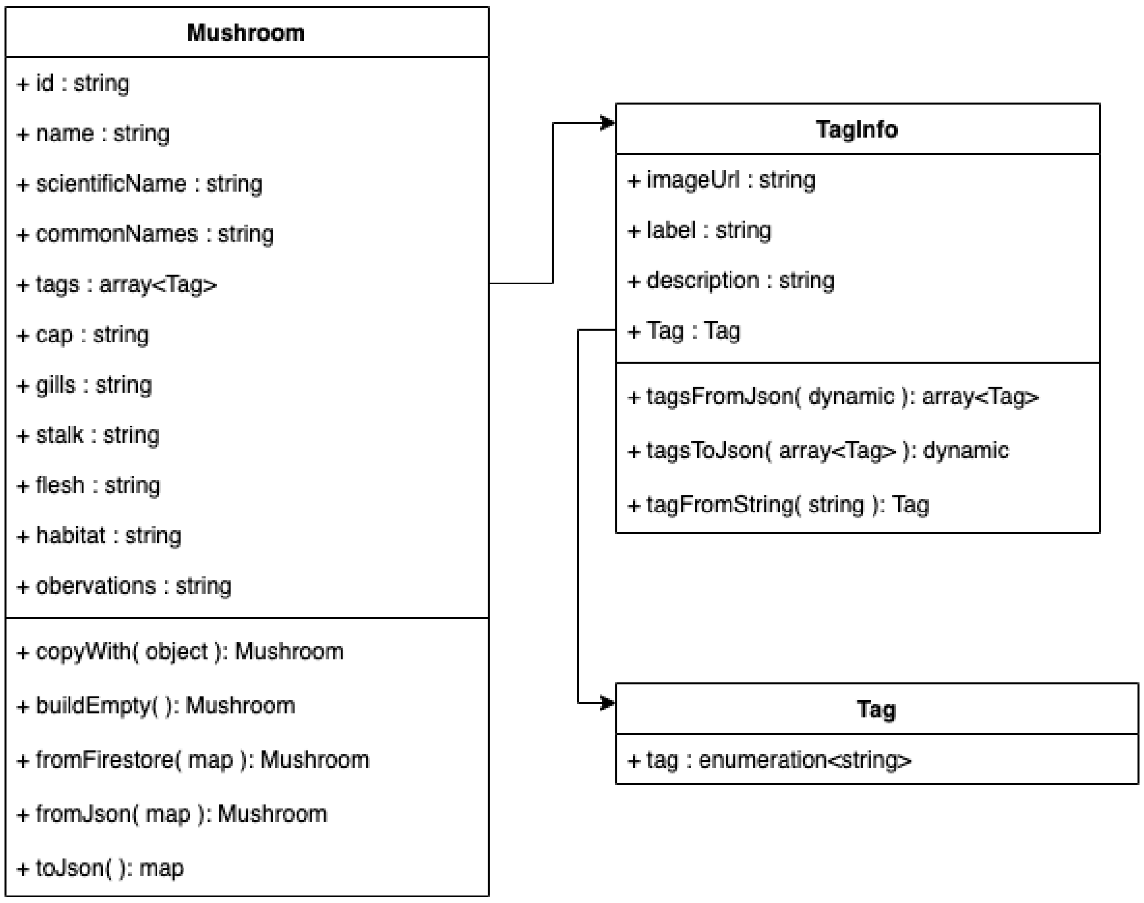


Fig. 3.4 Mushroom-related entities

CHAPTER 4. TECH STACK AND WORKING ENVIRONMENT

This chapter aims to present and choose a technical stack for our applications and which will be the environment used for the development phase. First of all, I am going to consider the technologies used for the frontend, backend and machine learning engine. Finally, I will present the hardware and software that will be used.

4.1. Choosing the Frontend

Deciding how to implement the frontend will be a key choice to make the application performant while being able to be agile in the development. Thus, it is important to know which is the current state of the art for mobile apps development and then to perform some research to make the right choice. This section will present those previous considerations, a comparison summary and the final decision.

4.1.1. State of the Art in Mobile Development

There are two main mobile operating systems in the current market: Android and iOS. As we would like to make our tool available to as many users as possible we will aim for both platforms. In order to support both platforms there are two options:

- Develop a native application for each of them.
- Use a cross-platform framework to build a single non-native application.

One one hand, the first thing to consider is that native applications are much more performant than any cross-platform approach. On the other hand, going native means having two completely different codebases, each one developed with a different language and with very little things reusable between them. So basically there is a trade-off between performance and workload. Our tool does not have expensive performance requirements and developing two different applications in such a short time is completely out of scope for our project. In addition, there are several examples of successful products that are built using a cross-platform approach [5] [6] [7].

Regarding the cross-platform frameworks that could be used for the task, the most popular ones are: React Native, Ionic, Xamarine and Flutter.

The main difference between them is that the React Native, Xamarine and Flutter render through the native engine, while Ionic is rendered using a browser engine. The Ionic approach is the less performant, as it consists in embedding websites on a mobile platform through a WebView and styling them to look native, which does not benefit from using the native engine.

That is the main reason why React Native has easily overcome hybrid app solutions such as Ionic in the last years. Regarding Xamarin, it works consistently and it is performant but has a hard learning curve, and even though it is free for startups and individuals, companies have to pay to use it.

In order to know which framework is currently more popular we have checked the amount of questions that developers are asking on Stack Overflow using their Trends tool[4].

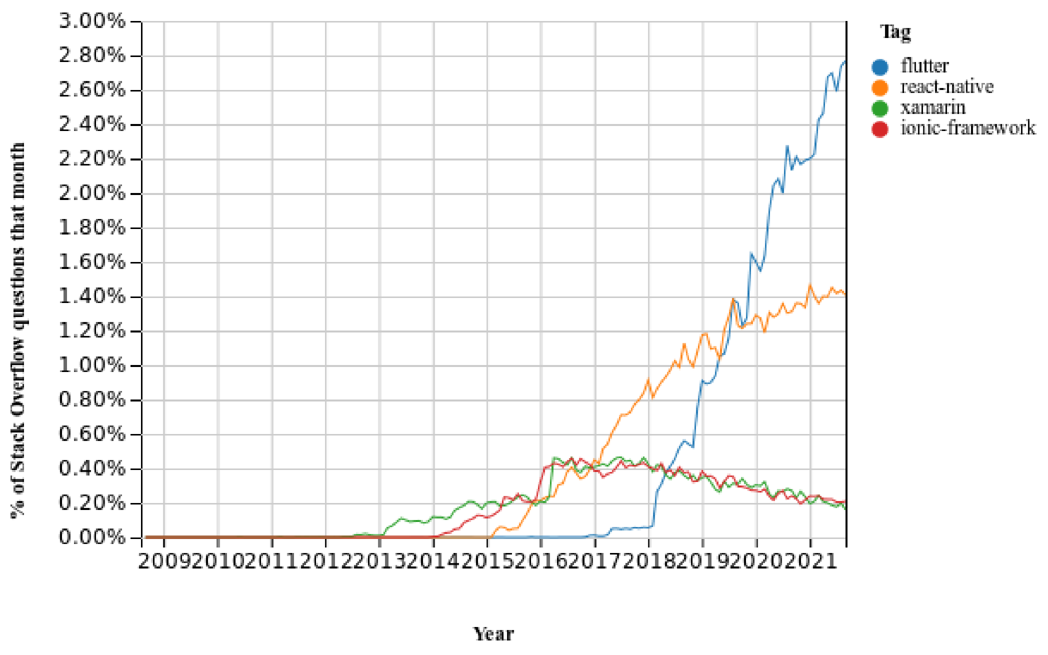


Fig. 4.1 Percentage of questions per month for each framework in StackOverflow

As you can see in the Figure 4.1 Xamarin and Ionic are slowly losing the interest of the developers, not only because there are less questions about them if we compare it with Flutter and React Native, but also because there are half the questions compared to their best period during 2016-2018. Regarding Flutter and React native, we can see that there are double the questions about the first one, which reinforces the idea that people love Flutter even though React Native was the preferred option before 2019.

We decided to use a framework that is “alive” and could be a game-changer in the short term, for that reason we will choose either Flutter or React Native. However, in order not to be biased by the hype, it is important to deeply understand the pros and cons of each one, as well as to confirm a statement that can be recursively seen: Flutter performance is huge compared to React Native.

[4] Stack Overflow Trends lets us query the percentage of questions per month for a certain topic. For more information, check https://insights.stackoverflow.com/trends .

4.1.2. Flutter vs React Native

Flutter is gradually increasing its market share and becoming a close competitor to React Native, which has been the most popular open-source and cross-platform mobile app framework for a long time, as it allowed developers to easily jump from the web world to the mobile world targeting multiple mobile platforms just using Javascript.

However, Flutter popped up with a brand new language: Dart; that quickly started to be widely used by the community due to its impressive performance results and its soft learning curve[5].

Both frameworks are based on a reactive paradigm where an application “reacts” based on actions that lead to state changes, which is an approach that was successfully used first by ReactJS and which is also used in React Native to manage views and actions.

Even though both frameworks implement a reactive approach, the rendering engine implementations are completely different.

React Native apps have a virtual tree of native widgets (just like the VDOM of ReactJS), but as we are using Javascript we need a “bridge” to compare and upload the changes, as well as for any asynchronous communications between the JS realm and the Native realm. The “bridge” translates Javascript to Native using the RN engine and a different compiler for each specific platform (Android, iOS, etc), which has a negative effect on performance in favor of an easier development experience.

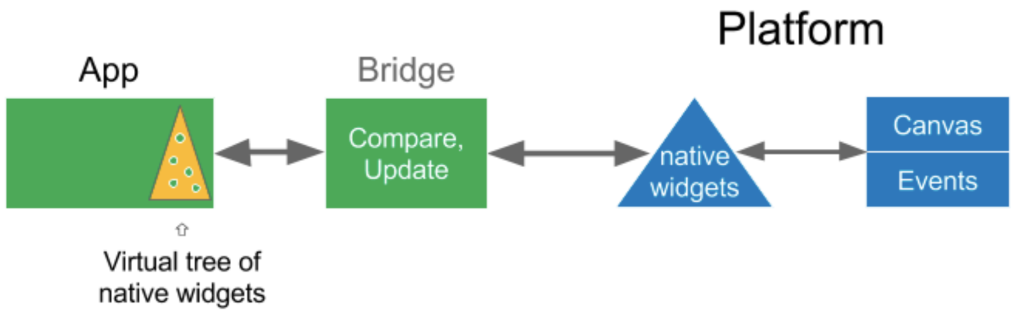


Fig. 4.2 Rendering approach in React Native

Flutter takes this idea a step further (see Fig. 4.3) as it does not use OEM[6] widgets like React Native, but defines their own ready-to-use, customizable widgets that look and feel native. For that reason the bridge is no longer needed

[5] For more information about this topic I recommend the lectures *What is revolutionary about Flutter* and *Why Flutter uses Dar*t, which you can find on Hackernoon webpage.

[6] Original Equipment Manufacturers: Native widgets for iOS and Android applications.

and thus performance is incredibly boosted, which leads to performance benefits such as running consistently at 60fps. The main drawback is that we have to download the whole Flutter engine to work with widgets; which boosts the startup time but also makes the app size bigger (see [8]).

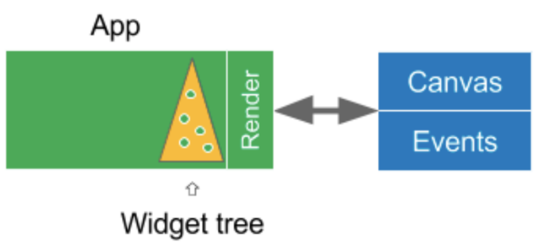


Fig.4.3 Rendering approach in Flutter

Regarding the performance of each of them, there are a bunch of articles about the topic, but just to get a bit of the whole picture we are going to specially check out one of them called *Flutter vs Native vs React Native: Examining Performance* [ref] where they measure the raw performance (CPU ordinary calculations) using two algorithms to calculate pi numbers.

There are a couple of interesting results from the lectures. On one hand we can see that React Native is almost 15 times slower than Java, while Flutter speed is 1.2 times slower. On the other side we can see that Flutter is even faster than Swift for iOS, being 1.5 times slower than Objective-C. The second table also gives us significant results, but most importantly, reinforces the tendency where Flutter outperforms RN and also achieves in some cases almost native speeds.

4.1.3. Summary: Cross-platform frameworks

In Table 3.1 you can see a summary comparison of the hybrid and cross-platform frameworks that have been considered to build the Frontend application. The following items have been considered to make the comparison:

- **Learning Curve:** Low, Medium or High. This is both subjective and objective criteria; Ionic and React Native are usually easy to learn, as there is good documentation and the curve becomes even softer if you have used Javascript, ReactJS or Angular previously. In contrast, working with Flutter means learning Dart programming language, even though the framework is designed to be easily used.

- **Time to Market:** This criteria has a direct correlation with the learning curve. However, this also takes into account the development speed once we know the framework: how easy it is to build screens, if there is documentation and community support, etc.

- **Performance:** Flutter and Xamarin performance is very close to native due to the approach they take, so we have considered that performance is high. In addition, it has been already exposed that Flutter outperforms React Native, while Ionic has the lowest performance because the approach implies embedding a webview.

- **Hot-Reload:** This feature allows code changes to be updated to the emulator during development without losing the app state, drastically speeding up the development life cycle. It has become a must for any framework and all of them are currently implementing this feature in one way or another, so I am not going to consider it to make the final decision.

Table 4.1 Summary table of different frameworks

Framework	**Learning Curve**	**Time to Market**	**Performance**	**Hot-Reload**
Ionic	Low	Fast	Low	Yes
React Native	Low	Fast	Medium	Yes
Flutter	Medium	Fast	High	Yes
Xamarin	High	Medium	High	Yes

4.2. Choosing the Backend

The application does not need complex requirements for the Backend, so it has been considered from the very beginning to use a Backend as a Service (BaaS) in order to avoid as many problems as possible. Another important reason to leave apart traditional Backend approaches is to speed up the whole development phase, as using Backend as a Service it is possible to set up a server in a few minutes. We have considered the following providers:

- **Firebase [9]:** It is Google's BaaS and also the preferred choice if you are going to use other Google products. Even though all of them explicitly state that you can integrate the BaaS with Flutter, there were more documentation in this case and it is a relief to know that Google is behind the project. Some of the services that it provides are: database, realtime database, storage, push notifications, crashlytics, analytics, authentication, machine learning functions, cloud functions, etc. Which makes the provider a powerful choice for starting and escalating our solution.

- **Supabase [10]:** Its slogan is "The open source Firebase alternative", which speaks by itself. Currently supports Database, Storage and Authentication services. It uses Postgres relational database, unlike Firestore Database which is NoSQL. Even though it is open source and cheap the main drawback is the amount of services that it provides; for instance, we need another solution to handle analytics, push notifications, and other features that we may want for our application.

- **AWS Amplify [11]:** It is the Amazon Web Services BaaS solution. There are a lot of services such as: database, storage, analytics, machine learning, CMS, push notifications, cloud functions, CI/CD and much more.

It has been decided to choose Firebase from the different BaaS providers due to several reasons.

On one hand, Supabase is a great open source solution but it could not be enough if we want to implement additional features in the near future such as push notifications.

On the other hand, AWS Amplify provides even more services than Firebase and would be like using a sledgehammer to crack a nut. It could lead to complex, over engineered solutions for the simple functionalities and approach that we want to implement.

Another important fact is that both Flutter and Firebase are Google products, which may ease the integration between them while not having issues due to a lack of documentation. Regarding which Firebase service to use, there are no real-time communication requirements between devices, so it has been decided to use the Firestore Database instead of the Realtime Database.

4.3. Choosing the Machine Learning model builder

The main goal of using a machine learning model is to provide active support to users in the classification task, hence identifying each mushroom without having to manually compare them. Nowadays, it is very common to use Machine Learning based classifiers, as the technology is mature and accessible; which has led to it being widely adopted by a lot of companies and researchers.

Building a machine learning classifier model would require some time that could slow down the project development. As it is not required to build a custom, high-confidence classifier, it has been decided to use a Google powered tool: Teachable Machine.

Teachable Machine is a web-based tool able to build classifiers based on images, sounds, and poses without writing any code. It lets us export these models in different formats and use them almost everywhere. Under the hood it uses a type of supervised machine learning algorithm, which means that it just needs to be fed with a single dataset for each class, and then Teachable Machine does the rest of the work: training, validation and test sets, classifier model, etc.

4.4. Working Environment

The application will be developed using Android Studio in a MacBook Pro. I will be using the Flutter SDK v2.2.3 for the development phase, using the ADB emulator with a 29 API level device and a Samsung Galaxy S9 physical device to test the application. I will also use Xcode to emulate an iOS device during the research phase to prove and test how it works, even though I do not aim to release it to the App Store due to the high publishing fees.

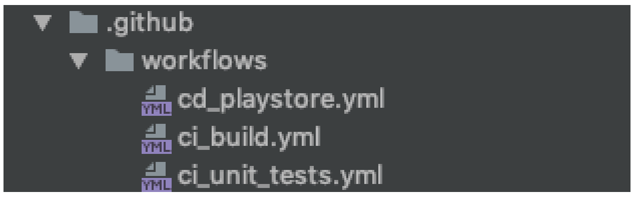
Regarding the Backend and the Machine Learning classifier, two online tools will be respectively used: the Firebase Console and the Teachable Machine tool. Finally, I will use a Git version control system to keep track of the changes and to host the codebase, as well as Github Actions to manage secrets and to set up the continuous integration and delivery pipeline.

Here it is the detailed list of the software and tools that will be used:

- MacBook Pro Mojave v.10.14.6
- Git version control system
- Flutter SDK v2.2.3 (using null-safety)
- Android Studio v3.5.3
- ADB emulator and Samsung Galaxy S9
- Firebase Console
- Teachable Machine online classifier tool
- Github Actions

4.4.1. Github Actions setup

In order to set up the Github Actions we just need to add a .github folder in our root project. Inside this folder we will place a yml file for each of the actions we want to launch, you can see an example in Figure 4.3.



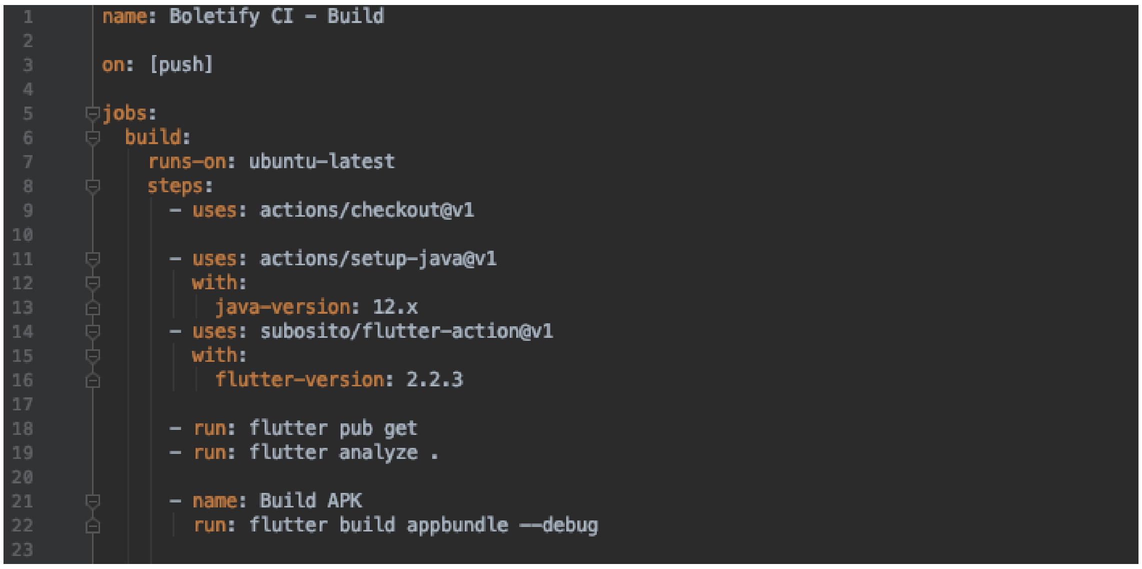
```
▼ .github
    ▼ workflows
        cd_playstore.yml
        ci_build.yml
        ci_unit_tests.yml
```

Fig. 4.3 Actions setup inside the project

Below you can see how the yml file looks like (see Fig. 4.4), some of the important fields are:

- **name:** Name that will be displayed in the Github Action hook.
- **on:** When the action should be triggered, we can do it on push, pull request, specific branches, etc.
- **jobs:** The action that will be triggered.

There are some specific steps that we can get and reuse from the Github Actions Marketplace [12], in our case the most important action will be subosito/flutter-action@v1 [13], which will help us install the Flutter SDK in the specified environment.



```yaml
name: Boletify CI - Build

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1

      - uses: actions/setup-java@v1
        with:
          java-version: 12.x
      - uses: subosito/flutter-action@v1
        with:
          flutter-version: 2.2.3

      - run: flutter pub get
      - run: flutter analyze .

      - name: Build APK
        run: flutter build appbundle --debug

```

Fig. 4.4 Actions setup inside the project

CHAPTER 5. DEVELOPMENT DETAILS

This chapter is focused on several development details that are worth mentioning, each section reflects a part of the architecture. Firstly, it is explained how the classifier was trained. Secondly, which was the approach followed to organize the app and some detail about the main developments. Finally, the backend structure is exposed.

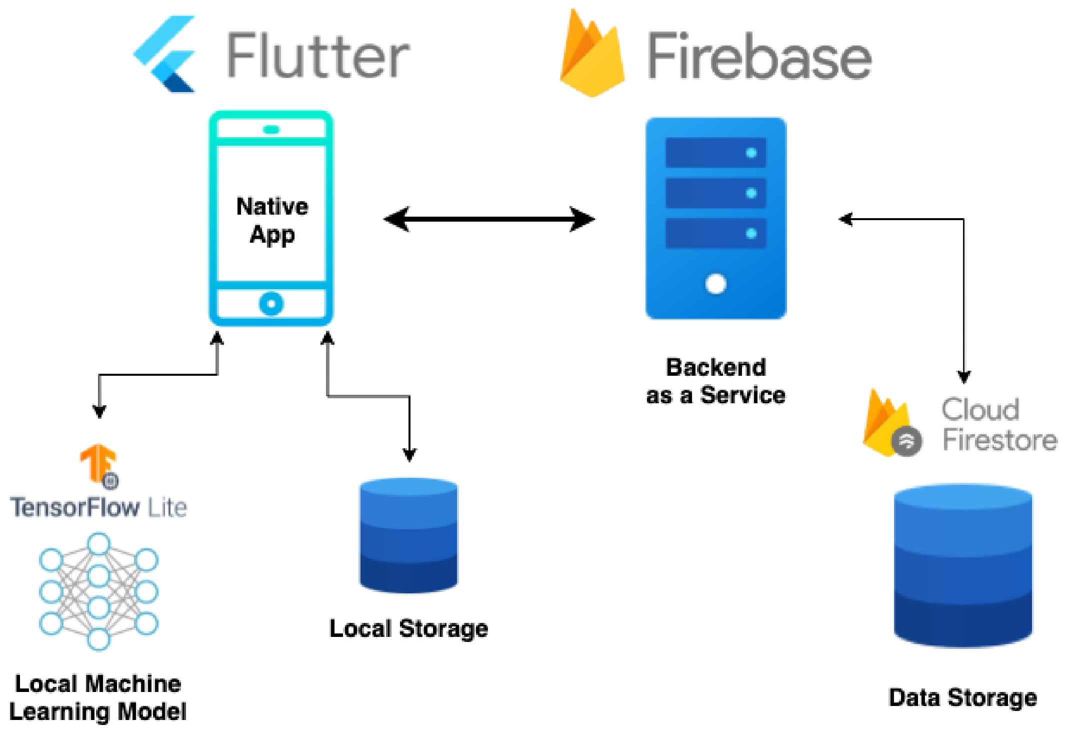


Fig. 5.1 Application architecture with used technologies

5.1. Teachable Machine: Training the classifier

As the classifier has to identify mushrooms, the first step is to create a different class for each of them in the classifier. For each class we will have to create a dataset of images to feed the training model. For example, the class “Cep: Boletus Edulis” contains around 50 images of one or multiple instances that have been collected and updated in the corresponding class of Teachable Machine; the same procedure has to be done for any kind of mushroom that we want to be classified (see Fig. 5.2).

It is also a good practice to create a class “Not a mushroom”, which will be useful to discriminate pictures that do not contain any mushroom, so from a statistical point of view it is more possible that an image that is not a mushroom falls to this category instead of mistakenly being classified as a mushroom because of the shape, color, or other attributes. Once all the classes have been created we can start training the model.

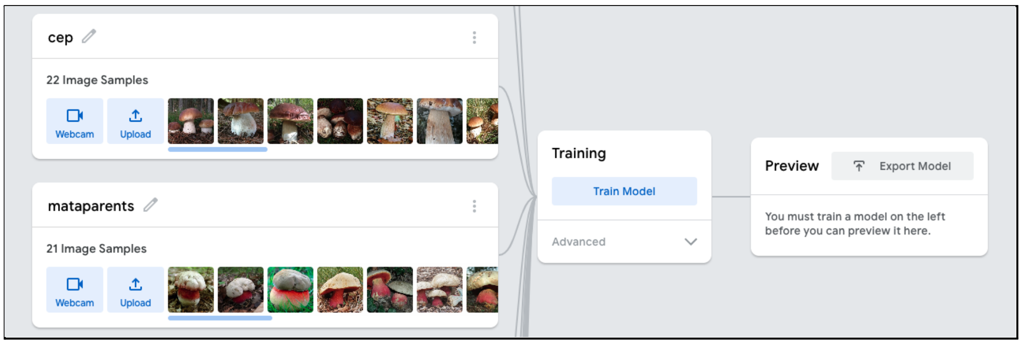
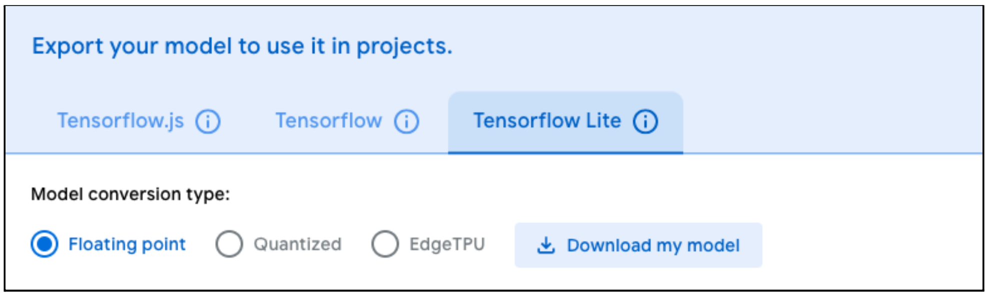


Fig. 5.2 Teachable Machine visual classifier builder

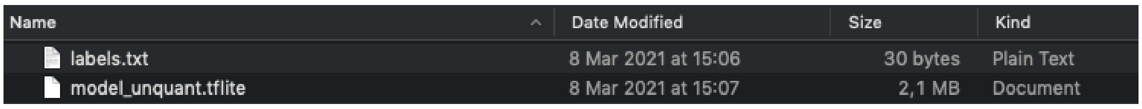
If at some point it is required to fine tune the results, we can tweak the training configuration from the "Advanced" dropdown tab. Once the classifier is completely trained the Tensorflow model can be exported to be used natively or in a browser, mobile, or edgetpu device.
In order to use the model for mobile devices it has to be exported with the Tensorflow Lite option (see Fig. 5.3); in this particular case, the model will be converted to a Floating Point type, as we do not want it to run on any edgetpu device and it is important that the identification is as precise as possible no matter the time it takes to classify the mushroom [16].



Model conversion type: [✓ Floating point] Quantized EdgeTPU

Fig. 5.3 Correct way of exporting the model to be used in mobile apps

The downloaded model will contain two files: 1) labels.txt, which will be a file with as many lines as different classes our model has, where each line will have the format "classId className"; and 2) model_unquant.tflite, which has the classifying logic itself in a valid TensorFlow Lite format.



Name	Date Modified	Size	Kind
labels.txt	8 Mar 2021 at 15:06	30 bytes	Plain Text
model_unquant.tflite	8 Mar 2021 at 15:07	2,1 MB	Document

Fig. 5.4 Files that will be used for the app classifier

5.2. Flutter: App Frontend

5.2.1. Folder structure

The approach followed for the Flutter folder structure aims to improve the app maintainability organizing files by separation of concerns. In any typical Flutter application most of the developments take place in the */lib* file, which you can see in Figure 5.5.

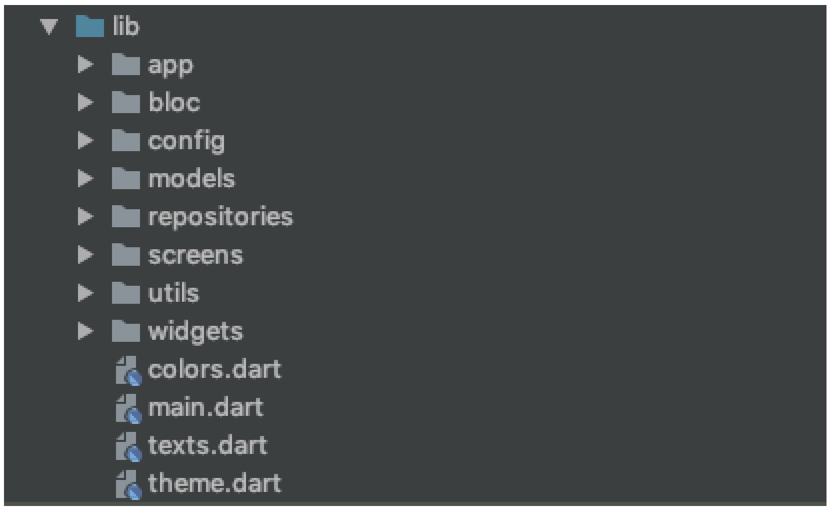

```
▼ lib
    ▶ app
    ▶ bloc
    ▶ config
    ▶ models
    ▶ repositories
    ▶ screens
    ▶ utils
    ▶ widgets
      colors.dart
      main.dart
      texts.dart
      theme.dart
```

Fig. 5.5 /lib folder structure of Boletify

First of all, it was decided to create a single file to define the colors, texts, and theme of the application; so we can easily change them without going file by file. The *main* file is the starting point of the application, and it is directly related to the */app* folder, where the Boletify layout is defined once the application is loaded.

API calls will be added to the */repositories* folder. It is intended to add an extra layer to be used as an adapter instead of making the request from the widget or the BLoC component directly, so changing the Backend or the requests will require to modify just a single file. In addition, it will not only get the query snapshots from Firebase and return a model we can use but also manage the offline mode feature to get a local copy of the mushrooms using the fileManager.

Business logic will have a special folder */bloc* where there will be a subfolder for each one of the BLoC's with the events, states and implementations of each of them. However, if it does not make sense to add the logic to a BLoC and it is reusable it will be added to the */utils* folder, where we will find functionalities that are used in several places.

Data entities will have their model defined inside the */models* folder. The folder will contain a file for each model with the class properties and methods that have been defined previously plus any other model needed to manage the widget properties.

Everything regarding UI will be added to */screens* or */widgets* folders. The first one will be used to compose the specific views using widgets and will have a subfolder for each of the screens. That subfolder will contain widgets that are only used on that screen and the screen itself. Settings, Home, Details, Search, etc. are examples of our screens. In contrast, the second folder will only contain generic-purpose widgets that are used in several places, such as FadeInAnimation, ShowUpAnimation, CustomIcon, BlackWhiteFilter, etc.

Finally, the */config* folder will contain hardcoded configurations that we do not want to move to our database but we want to easily access to change the app behaviour. For example, for the *summer* Tag is associated with an icon, a label, a description, etc. Each Settings Item is associated with an icon, a title, a screen or redirect url, etc.

5.2.2. Core setup

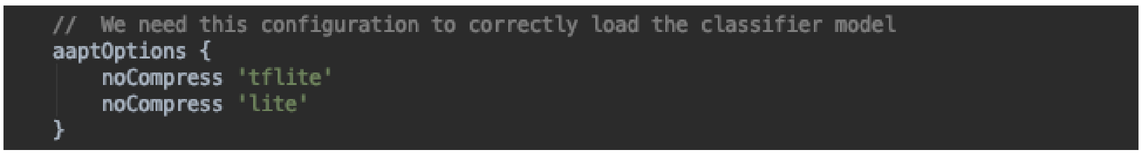
The starting point of the application was to add the required packages to initialize Firebase [14][15] for Android and iOS applications to be able to retrieve the mushrooms data. Then, the following packages were also added in order to implement the classifier functionality:

- **tflite:** A plugin for accessing Tensorflow Lite API for both iOS and Android. We will define the labels and the .tflite model, as well as the image to classify and some extra configuration such as the number of results we want to receive, the minimum confidence threshold to get a result, etc. [17].

- **image_picker:** A plugin for selecting images from both Android and iOS image library, or taking a picture using the camera. We will define an image source that can be either camera or gallery, and we will get a file that will be used to feed the classifier [18].

The next step after installing the packages is to add to the assets folder the files generated by Teachable Machine in the previous step (*labels.txt* and *model_unquant.tflite*).

To make them work properly, the following configuration will be needed as well (see Fig. 5.6).



```
// We need this configuration to correctly load the classifier model
aaptOptions {
    noCompress 'tflite'
    noCompress 'lite'
}
```

Fig. 5.6 Additional configuration to work with a TF-lite model in Android

In order to test the functionality it has been used a Stateful Widget that performs all the work: Initialize and deinitialize the model, pick the image, and handle the views.

In section 5.2.3 you can see how this widget was optimized by moving the logic to a BLoC and using a Stateless Widget instead, gaining performance and maintainability by optimizing renders and decoupling the business logic.

To finish the initial setup, a simple Github workflow was added in order to make a build and launch the unit tests anytime a Pull Request is opened, so we can be sure nothing breaks when new changes are uploaded.

5.2.3. Using the classifier in BLoC

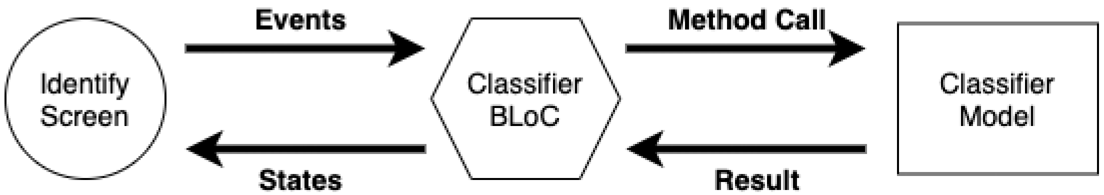


Fig. 5.7 Classifier using BLoC implementation

As you can see in Figure 5.7 the Identify Screen will emit different events to the Classifier BLoC and receive different states from it. Each event will be the action we want to perform and the state will define which UI and information we want to display. On the other side, the BLoC will also interact with our classifier model using the *tflite* package to run the model on an image and get an output from it. The classifier BLoC will allow only two events:

- **Initialize:** To set up the classifier.
- **Classify:** Which accepts an image source argument (Camera or Gallery) that will be used to get a file using the *image_picker* package and call the *runModelOnImage* method from the *tflite* package.

Each one of the yielded states will trigger a UI change reactively in the main Identify Screen container widget. The classifier BLoC will yield the following states:

- **Initial:** When the classifier is setting up.
- **Loading:** For any asynchronous action that requires a loader.
- **Error:** When there is an uncontrolled error that requires giving feedback to the user.

- **Methods:** When the classifier is ready to accept a classification method to get the image from it.

- **Result:** When we get any output result from the classifier, no matter if the classification fails or succeeds.

In the tables below (see Table 5.1 and Table 5.2) you can find the two events and the respectives actions, states and screens that will be triggered, yielded or shown in the application.

Table 5.1 Actions, states and screen for the Initialize event

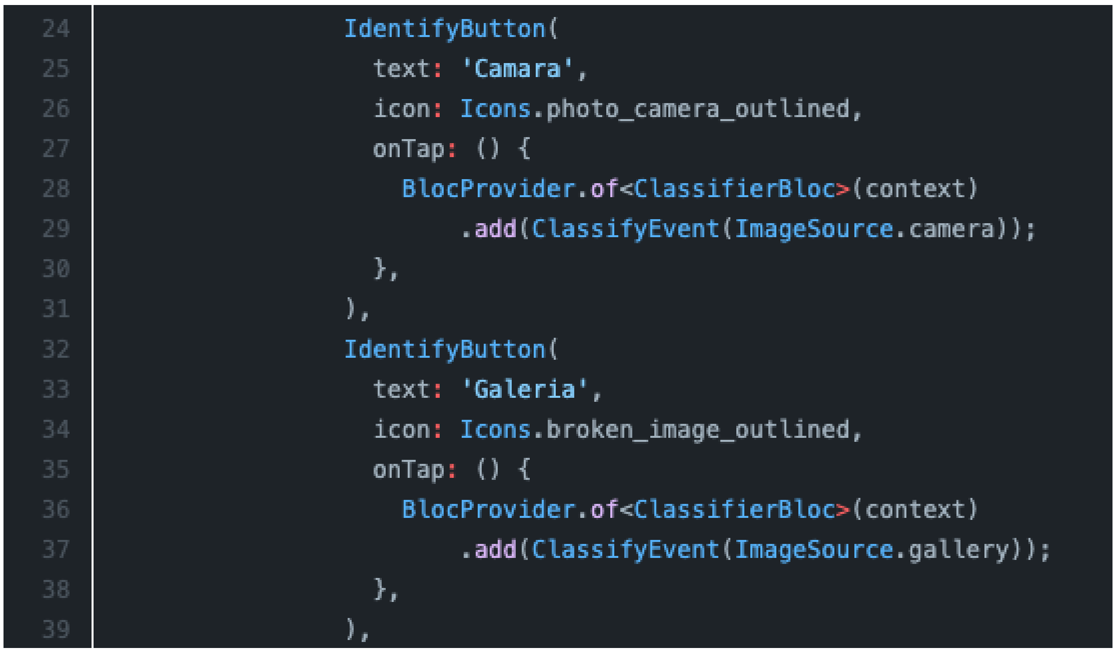
Initialize Event		
Actions	**States**	**Screens**
Initialize *tflite* with the model and labels	Initial	Loading
Set as ready after initialization	Methods	Identification Methods
Handle error if any	Error	Error

Table 5.2 Actions, states and screen for the Classify event

Classify Event (imageSource)		
Actions	**States**	**Screens**
Use image_picker to get a file and run the classifier to get the result	Loading	Loading
Yield a state with the classifier output	Result	Identification Result
Handle error if any	Error	Error

Initializing the classifier is an asynchronous action, for that reason it has been defined an Initial state that will set up the TensorFlow Lite model and show a CircularProgressIndicator widget in the meanwhile.

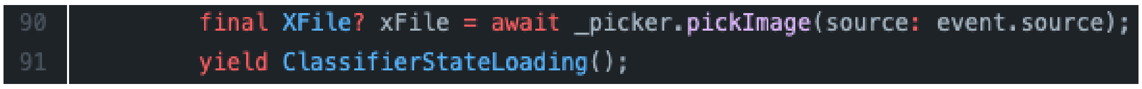
Once the classifier is loaded we will show the Identification Methods screen to let the user choose between taking a picture with the camera or uploading it from the gallery. For this reason we will show two custom buttons that will trigger the ClassifyEvent with the chosen image source on the onTap event (see Fig. 5.8).



```
                IdentifyButton(
                  text: 'Camara',
                  icon: Icons.photo_camera_outlined,
                  onTap: () {
                    BlocProvider.of<ClassifierBloc>(context)
                        .add(ClassifyEvent(ImageSource.camera));
                  },
                ),
                IdentifyButton(
                  text: 'Galeria',
                  icon: Icons.broken_image_outlined,
                  onTap: () {
                    BlocProvider.of<ClassifierBloc>(context)
                        .add(ClassifyEvent(ImageSource.gallery));
                  },
                ),
```

Fig. 5.8 Identification methods screen buttons

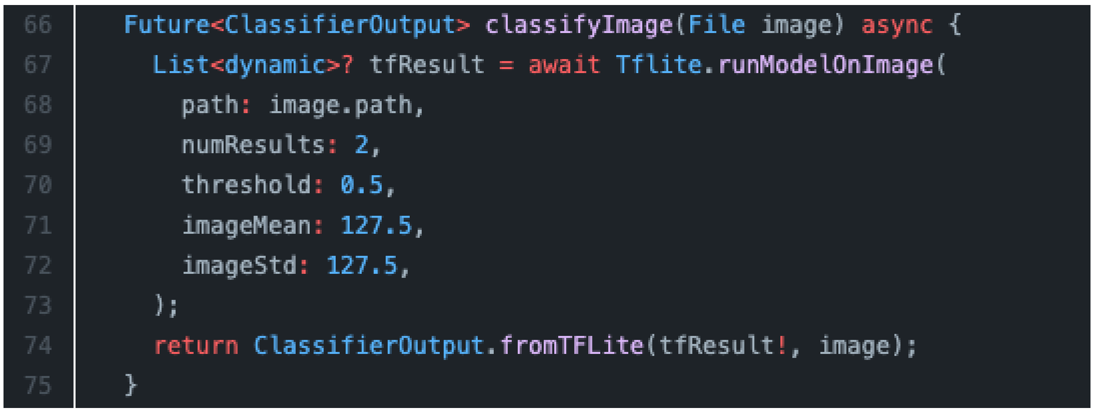
After adding the ClassifyEvent with an Image Source we will use the *image_picker* package to handle the gallery or camera access. This will be an asynchronous action that will not be resolved until the user cancels the action and chooses a file that we can use.



```
        final XFile? xFile = await _picker.pickImage(source: event.source);
        yield ClassifierStateLoading();
```

Fig. 5.9 Using image_picker package from the Classifier BLoC

Lastly, the image will be sent through the classifier model using the *tflite* package, which will return a mushroom from it.



```
  Future<ClassifierOutput> classifyImage(File image) async {
    List<dynamic>? tfResult = await Tflite.runModelOnImage(
      path: image.path,
      numResults: 2,
      threshold: 0.5,
      imageMean: 127.5,
      imageStd: 127.5,
    );
    return ClassifierOutput.fromTFLite(tfResult!, image);
  }
```

Fig. 5.10 Using Tensorflow Lite from the Classifier BLoC

5.2.4. Building the offline mode feature

The main functionality of the offline mode is to allow the app working without connection by saving a fallback copy of the API data information, so we are still able to connect classification results with mushroom details. You can find the schema workflow in the section **3.2 Offline mode**.

In order to handle saving and recovery of the fallback information a utils file called *file_manager.dart* has been created. I have created a utils file called *file_manager.dart*. First of all, we have to add a package called *path_provider* that we will use to access and get the file from the Android and iOS file system. Then we will create a class called FileManager that will have three methods:

- **initializeFileManager:** Will initialize the file system and get file path in order to read or write on it.

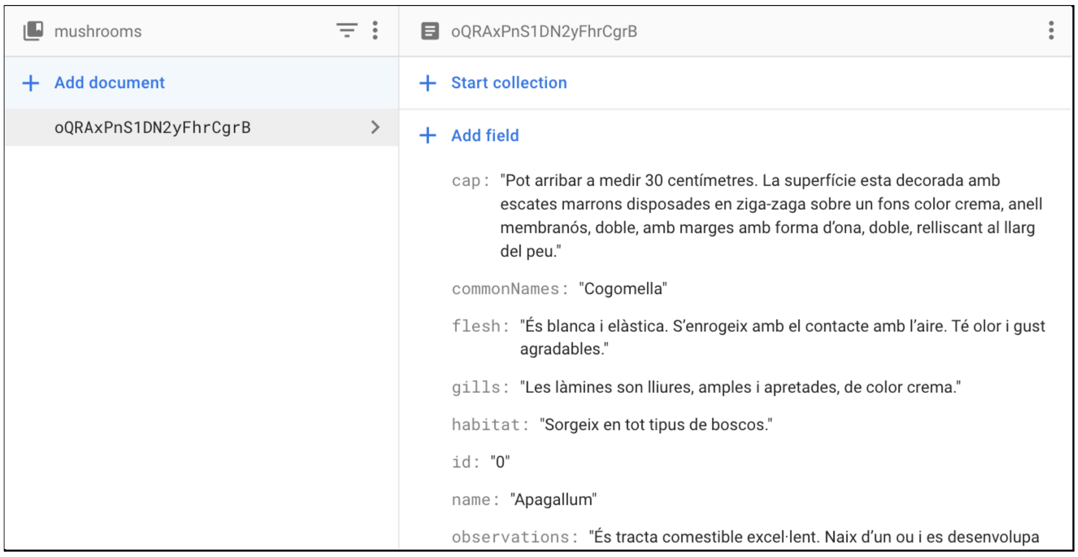
- **setMushroomsFallbackList:** Will receive an array of Mushrooms, convert them to JSON and create or update the fallback file.

- **getMushroomFallbackList:** Will get the JSON information from the fallback file, convert it to a list of Mushrooms and return it.

5.3. Firebase: Backend as a Service

As has been stated before, setting up a database using Backend as a Service takes minutes. We just have to create a new project in Firebase and set up a Firestore database. Then we just need to configure the frontend to be able to access our information, which basically consists in adding our project package name and adding the Google services file to our Flutter project.

Next, we will create a new collection called “mushrooms” and add a new document to it with the information of each of the mushrooms we want to classify (see Fig. 5.11).



```
cap: "Pot arribar a medir 30 centímetres. La superfície esta decorada amb
      escates marrons disposades en ziga-zaga sobre un fons color crema, anell
      membranós, doble, amb marges amb forma d'ona, doble, relliscant al llarg
      del peu."
commonNames: "Cogomella"
flesh: "És blanca i elàstica. S'enrogeix amb el contacte amb l'aire. Té olor i gust
        agradables."
gills: "Les làmines son lliures, amples i apretades, de color crema."
habitat: "Sorgeix en tot tipus de boscos."
id: "0"
name: "Apagallum"
observations: "És tracta comestible excel·lent. Naix d'un ou i es desenvolupa
```

Fig. 5.11 Example of a mushroom document in the Firestore "mushrooms" collection

CHAPTER 6. DELIVERY AND RELEASE

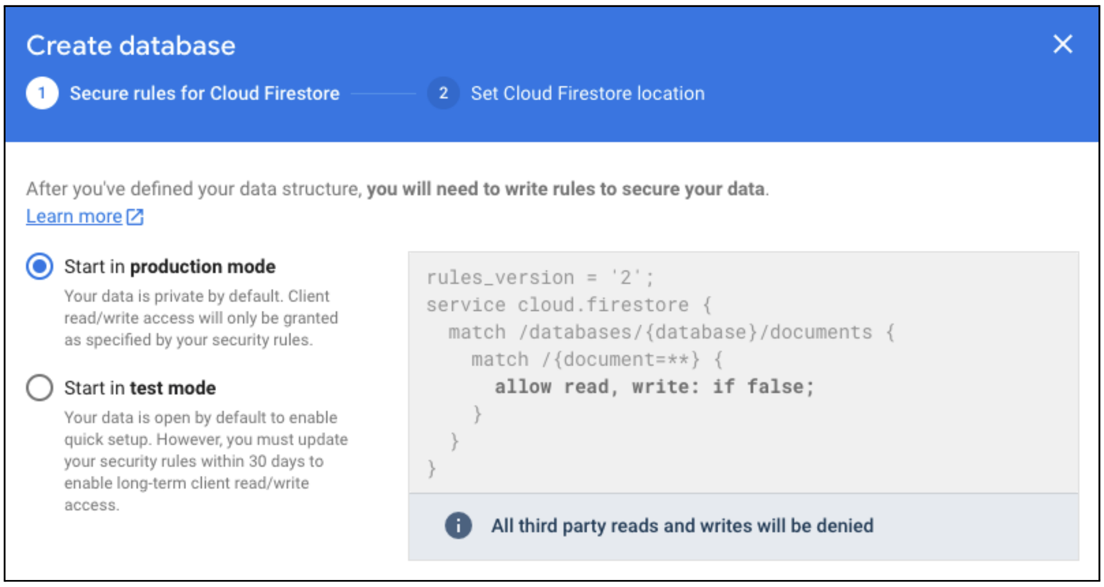
In this chapter we are going to go through the main steps needed to release the application to the official channels. I will explain how to set up the backend, how to sign the application, and which steps we have to follow in order to use Github Actions for continuous delivery.

Even though Flutter allows us to generate the application for both iOS and Android, we are going to release it exclusively to Google Play. The main reason is the publisher license price, which is about 25€ per year for Google Play, whereas it costs about 100€ per year to publish to the App Store. We think that is the right moment for now, as we do not really know if people will use or not our application.

Due to the previous consideration, we are only going to show the Android configuration and release process.

6.1. Setting up a production-ready Backend

First of all, a new project has been created in Firebase in order to be production-ready (see **Fig. 6.1**). This is important to isolate production data from development data, so we can use a development project to test future improvements.


Create database

1 Secure rules for Cloud Firestore — 2 Set Cloud Firestore location

After you've defined your data structure, **you will need to write rules to secure your data**.
Learn more

● Start in **production mode**
Your data is private by default. Client read/write access will only be granted as specified by your security rules.

○ Start in **test mode**
Your data is open by default to enable quick setup. However, you must update your security rules within 30 days to enable long-term client read/write access.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

All third party reads and writes will be denied

Fig. 6.1 Creating a production database for Boletify

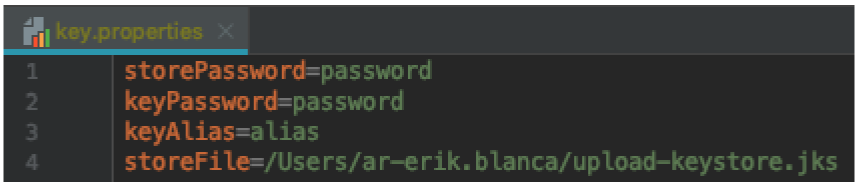
Starting the project in the production model does not allow access to anyone by default, so the rules have been changed as well in order to give read access to our frontend application. Besides this step, the only requirement is to configure the iOS and Android apps to use this new production database by downloading and adding the Google services JSON file to the code.

6.2. Setting up a production-ready Application

Android applications must be digitally signed [19] with a certificate before being used in the real world. Currently, we are only able to build and install an unsigned application, which we cannot use for a release.

The most common way of signing an app is to configure several fields in a *key.properties* file that will be located in the root of our Android project. If we use Flutter, the path will be */name_of_project/android*.

This file will contain a *storePassword*, a *keyPassword*, a *keyAlias*, and a path to our *storeFile*.

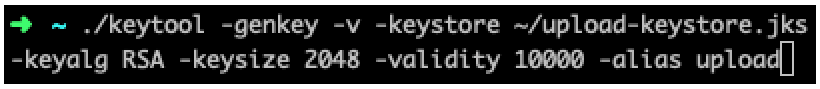


```
storePassword=password
keyPassword=password
keyAlias=alias
storeFile=/Users/ar-erik.blanca/upload-keystore.jks
```

Fig. 6.2 Example about how it looks a *key.properties* file

In our case, the same password has been used for the *storePassword* and the *keyPassword*; while any string can be used to fill the *keyAlias*, for example upload or release. Regarding the storeFile, it has to contain the path where our signing key is being stored, which will be created using a tool from the Java JDK called keytool.

The key is created using the command below, notice that if you have not set a global path for the JDK you will need to move to its *location /bin* to run it.



```
➜  ~ ./keytool -genkey -v -keystore ~/upload-keystore.jks
-keyalg RSA -keysize 2048 -validity 10000 -alias upload
```

Fig. 6.3 Command used to generate the Java KeyStore file

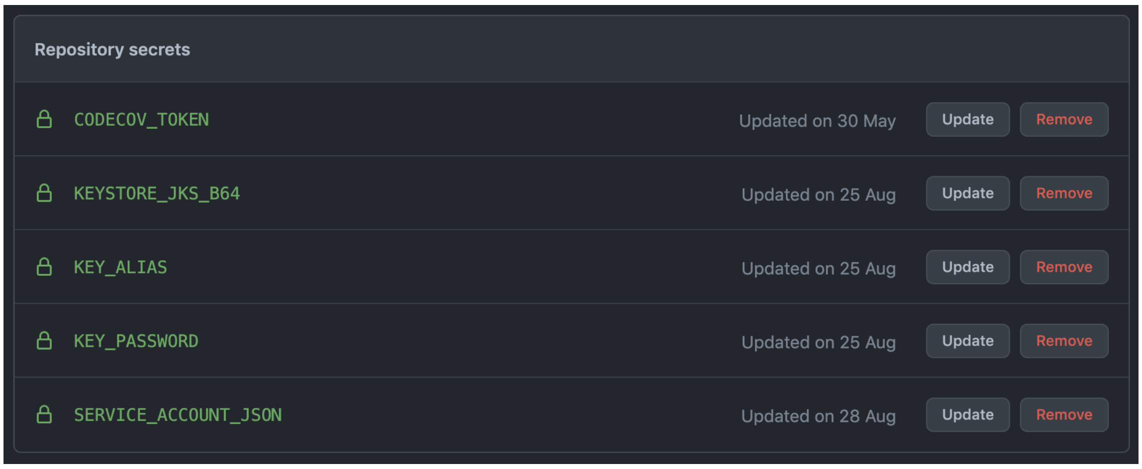
Now that all the fields are filled we can use the key file to sign the build locally and - with some changes in our *android/app/build.gradle* file - update it manually to the Play Store. But we want to build a Deployment pipeline to automate that process using Github Actions, and we cannot simply update the keys to Github, as it contains sensitive information that other people could use to sign and unsign the application bundle. To solve this, we will proceed to:

1) Update the key information to Github Secrets so we can use it later in our Github Action;

2) Generate and fill the *key.properties* file dynamically from those secrets in our Github Action.

6.3. Setting up the Continuous Delivery pipeline

First of all, we have to update our key properties as secrets to Github Actions so we can use them later as environment variables in our build process. Uploading them is as easy as going to the Settings/Secrets section of our project Github repository, you can see the result in the image below.



Repository secrets

CODECOV_TOKEN	Updated on 30 May	Update	Remove
KEYSTORE_JKS_B64	Updated on 25 Aug	Update	Remove
KEY_ALIAS	Updated on 25 Aug	Update	Remove
KEY_PASSWORD	Updated on 25 Aug	Update	Remove
SERVICE_ACCOUNT_JSON	Updated on 28 Aug	Update	Remove

Fig. 6.4 Github secrets from Boletify repository

As you can see in Figure 6.4 there are 5 secrets, but not all of them are used for the Android deployment system.

- **CODECOV_TOKEN:** It is used in our Continuous Integration pipeline to be able to get access to the Codecov tool, so test coverage can be uploaded and tracked.

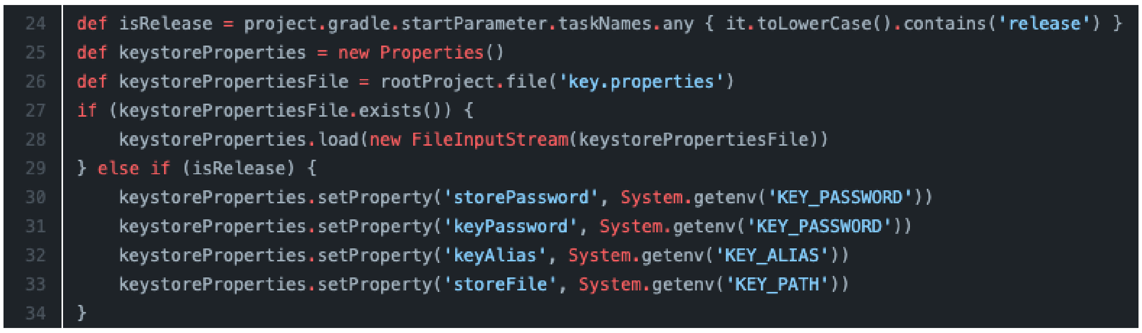
- **KEYSTORE_JKS_B64:** It is the content inside the keystore_file.jks. As you can see it has been converted to base 64 format, as Java KeyStore format is a binary format. This means that it will have to be decoded again at runtime in the release signing process.

- **KEY_ALIAS:** The *keyAlias* field from the previous step. As said before, any alias such as “release” or “update” can be used.

- **KEY_PASSWORD:** The password that has been set up in the previous steps for the *storePassword* and *keyPassword* fields. Using a different password for each of them will require an additional secret.

- **SERVICE_ACCOUNT_JSON:** It is a unique identifier that is used to get access to the Google Play Developer API in order to upload changes to the Android application on the Google Play Console.

The next step is to make some changes in our *app/build.gradle* file so we can dynamically generate the *key.properties* file from the environment variables that we will set up from the Github Action.



```
24 def isRelease = project.gradle.startParameter.taskNames.any { it.toLowerCase().contains('release') }
25 def keystoreProperties = new Properties()
26 def keystorePropertiesFile = rootProject.file('key.properties')
27 if (keystorePropertiesFile.exists()) {
28     keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
29 } else if (isRelease) {
30     keystoreProperties.setProperty('storePassword', System.getenv('KEY_PASSWORD'))
31     keystoreProperties.setProperty('keyPassword', System.getenv('KEY_PASSWORD'))
32     keystoreProperties.setProperty('keyAlias', System.getenv('KEY_ALIAS'))
33     keystoreProperties.setProperty('storeFile', System.getenv('KEY_PATH'))
34 }
```

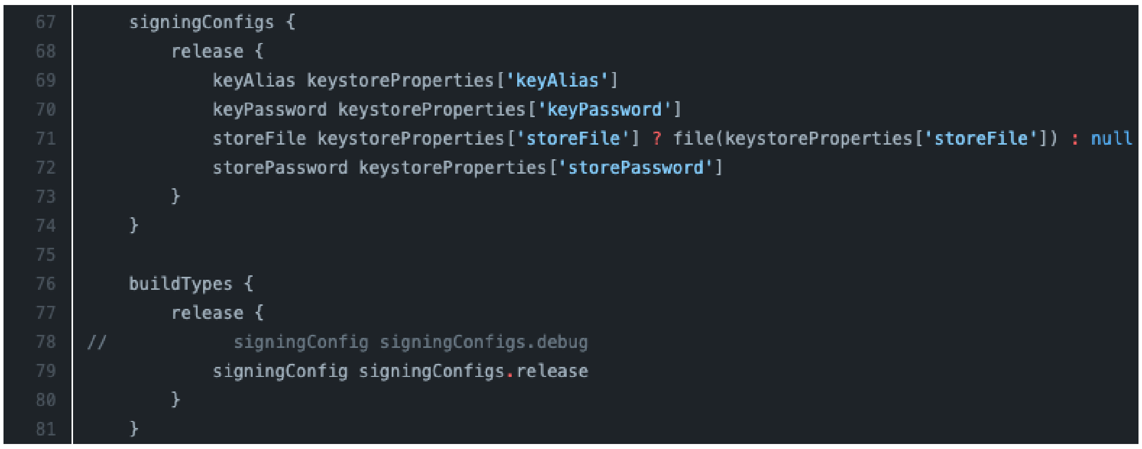
Fig. 6.5 Code used to get or generate the *key.properties* file

In line 24 it is defined as a local variable *isRelease* to know whether the build is flagged as a release or debug one, as we only want to get the *key.properties* from the environment variables if it is a release build.

In line 27 it is checked if there is an existing *key.properties* file, if it is found that means that we are launching the build locally, so we will use the local key values.

In lines 29-34 we check if it is a release build, and if that is the case we fill the values from the environment variables provided by the Github Action. As you can see in line 33, we are using a *KEY_PATH* environment variable which will be the path to our JDK key once it is decoded from the Github Action.

As a last step, we also have to define the release signing config in the same file, as well as bounding the release build type to that concrete signing config.



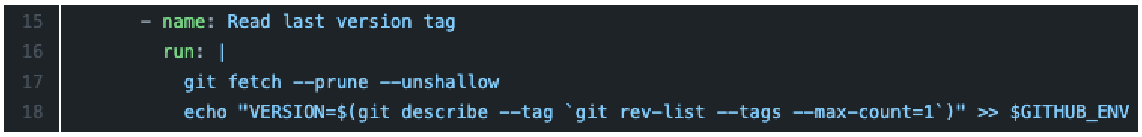
```
    signingConfigs {
        release {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile keystoreProperties['storeFile'] ? file(keystoreProperties['storeFile']) : null
            storePassword keystoreProperties['storePassword']
        }
    }

    buildTypes {
        release {
//              signingConfig signingConfigs.debug
            signingConfig signingConfigs.release
        }
    }
```

Fig. 6.6 Boletify release signing config for Android

6.4. Continuous Delivery Github Action

Once we have set up our secrets and configured the application to be production-ready it is time to implement the continuous delivery Github Action. This action should be able to get the latest version tag from the Boletify repository, increment the patch, build the new version, upload it to the Play Store, and push the new tag to our repository. As well as the other actions, this one will be a .yml file inside the *.github/workflows* folder, that will be located in our Flutter project root path.

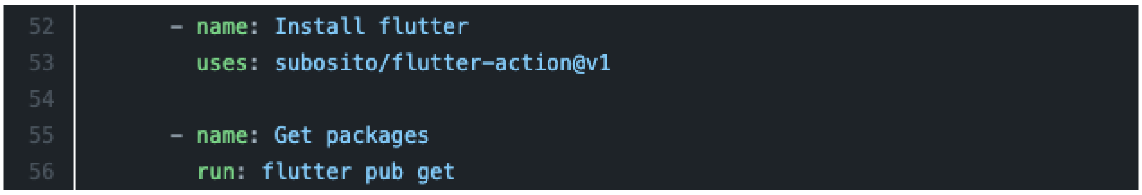
First of all, we will fetch the tags from our Github repository and to keep the latest one (see Fig. 6.7). We will use it to increment the patch number in order to generate the next build version and the next patch version.



```
    - name: Read last version tag
      run: |
        git fetch --prune --unshallow
        echo "VERSION=$(git describe --tag `git rev-list --tags --max-count=1`)" >> $GITHUB_ENV
```

Fig. 6.7 Steps to install Flutter and project packages

Once we have defined the next build version we will proceed to install Flutter using the *subosito/flutter-action*, and then we will install the project packages as you can see on the image below.



```
    - name: Install flutter
      uses: subosito/flutter-action@v1

    - name: Get packages
      run: flutter pub get
```

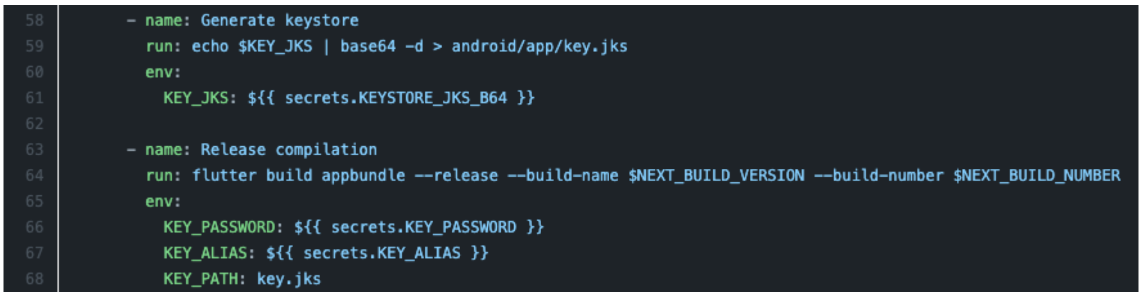
Fig. 6.8 Steps to install Flutter and project packages

Then we will start the signed build process using the secrets that we have added to Github Secrets. First of all, we need to create the key.jks file from our key secret that we have encoded in Base64 format, so we have to decode the secret and write it inside the key.jks file to match the *key.properties* location (see Fig. 6.9, line 61).

The next step after decoding the key is to build the appbundle[7] with the release flag, and then use the fields that have been defined using the previous release tag in order to set the next build name and next build version flags (see Fig. 6.8, line 64).

[7] An *Android App Bundle* is a publishing format that includes all your app's compiled code and resources. Google Play uses your app bundle to generate and serve optimized APKs for each device configuration, so only the code and resources that are needed for a specific device are downloaded to run your app.

Notice that the rest of secrets have been mapped to match the same environment variables defined in the *app/build.gradle*, so they can be used to fill the *key.properties file*.

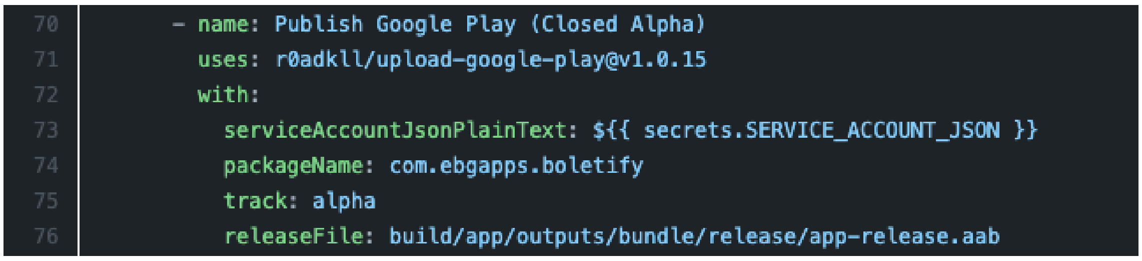


```yaml
    - name: Generate keystore
      run: echo $KEY_JKS | base64 -d > android/app/key.jks
      env:
        KEY_JKS: ${{ secrets.KEYSTORE_JKS_B64 }}

    - name: Release compilation
      run: flutter build appbundle --release --build-name $NEXT_BUILD_VERSION --build-number $NEXT_BUILD_NUMBER
      env:
        KEY_PASSWORD: ${{ secrets.KEY_PASSWORD }}
        KEY_ALIAS: ${{ secrets.KEY_ALIAS }}
        KEY_PATH: key.jks
```

Fig. 6.9 Steps build the production appbundle using secrets

The last step is to update the appbundle using the *r0adkll/upload-google-play* action in order to manage the Google Play API interactions.

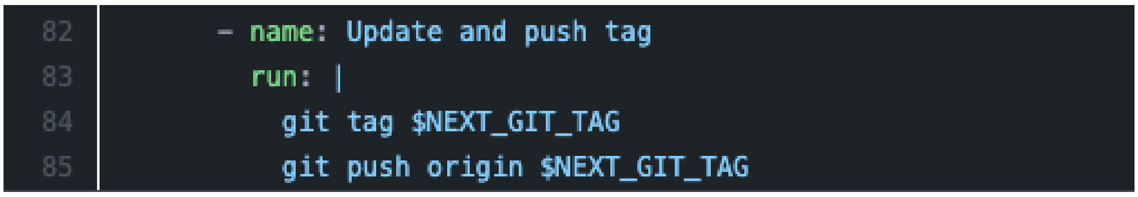
This action will require to set the correct *packageName* of the Android application, and then use the SERVICE_ACCOUNT_JSON secret to grant the access via API to our Google Play project. It is also important to add the correct .aab path to the action, as well as to define the track where the build should be uploaded (see Fig. 6.10). In this case only the Alpha track is used to make the release available for internal testers, but the track could be defined as Production or Beta to upload the release to those tracks.



```yaml
    - name: Publish Google Play (Closed Alpha)
      uses: r0adkll/upload-google-play@v1.0.15
      with:
        serviceAccountJsonPlainText: ${{ secrets.SERVICE_ACCOUNT_JSON }}
        packageName: com.ebgapps.boletify
        track: alpha
        releaseFile: build/app/outputs/bundle/release/app-release.aab
```

Fig. 6.10 Release step to publish build into Google Play

Once the appbundle has been correctly uploaded the last step is to upload and push the tag to Github, so the next release is able to get that tag and increment it in order to start the delivery process again next time (see Fig. 6.11).



```yaml
    - name: Update and push tag
      run: |
        git tag $NEXT_GIT_TAG
        git push origin $NEXT_GIT_TAG
```

Fig. 6.11 Update and push new tag to Github

CHAPTER 7. WORK PLANNING

In this chapter we are going to explain the workload and how it has been distributed during the whole project. Here you can find a brief explanation of each of the main tasks and the time estimation for each of them (see Table 7.1).

Project definition: Time dedicated to define the idea and the project's scope.

Research: Time spent in any kind of research; which includes research about native and cross-platform frameworks, data storage and machine-learning, and setup of the development ecosystem and tools.

Design: Time spent defining and improving the user interface, the icons used on the application, and the multiple mockups.

Development: This phase includes the coding of the application frontend and the different features, as well as the later performance optimization.

Release: This phase includes the release of the application to the Play Store, as well as the process automation using Github Actions.

Documentation: Time spent writing down the project report.

Table 7.1 Time estimation distribution for each task

Task	**Time Estimation** (days)	**Time Estimation** (%)
Project Definition	10	9.4%
Research	20	18.8%
Design	14	13.2%
Development	30	28.3%
Release	14	13.2%
Documentation	20	18.8%
TOTAL	106	100%

It is worth mentioning that a big amount of time has been invested in the research process in order to decide the best architecture and how to implement the ML engine. Another important task to consider is the release, which usually takes less time. One of the goals of the project was to automate the release process, so a lot of time was spent in the delivery pipeline as well.

CHAPTER 8. CONCLUSIONS

This last chapter will go through the final conclusions of the project: if the initial goals have been achieved, what I have learned personally, and which future improvements could be done in the short and long term.

8.1. Goals achievement

At the beginning of the project we have defined a set of goals that have been achieved. The result is a native Android application with an automated integration and delivery process that uses a server as a service and it is able to classify mushrooms through a machine learning model that also works perfectly fine when a user is offline.

Training a machine learning model and using it on the client side to identify mushrooms using images was the main objective of this project. Time spent during research has been useful to find a tool that allows us to get rid of most of the complexity of these techniques so the workload is reduced to only building the dataset. Building the classifier without Teachable Machine would be a project itself.

Similarly, I have used a backend as a service approach that has proven very useful to speed up development compared to a traditional approach, which would be not only slower to build, but also would have less scalability and more complexity for adding features such as push notifications or analytics.

Another one of the main goals was to automate the release process though a continuous integration and continuous delivery pipeline. The result is a system that integrates and publishes our changes from the master branch of our version control system to an alpha track on the Play Store that is only used for internal testers until I decide to release the public version.

8.2. Personal conclusions

During this project I have worked in all the phases of the software development life cycle: from planning to releasing a product. Furthermore, we have also automated the integration and delivery of the application, which is exactly how things work in real-world companies. That makes me really proud and I have learned a lot.

In fact, most of the tools that I have used were unknown to me. Flutter, Firebase and Github Actions are tools that I have never used before, and with which I feel now very confident and ready to use again in future MVPs or side projects.

The whole process has been very agile. Firebase works like a charm, there is a bunch of documentation, and you are able to set up a new project in minutes. In any case, I still have curiosity to use a relational database in a BaaS, so probably I will give Supabase a try at some point. Regarding Flutter, I think it is a very powerful framework and I am totally convinced it will take over React Native in the following years, as the development speed and performance outperform its competitors. By the way, automating the hard work using Github Actions allows me to reuse it in other projects, which will make the development even more agile.

The work will be available in a public repository that you can find on my Github [21], so other developers will be able to fork it to improve the project or learn from it.

8.3. Future Improvements

One one hand, there are several things that I want to improve in the short term before the public release. For example it would be really nice to have a small tutorial that will be shown only the first time in order to teach the user how to correctly take the picture to maximize the classifier confidence. Another nice-to-have would be to make the user explicitly accept our responsibility disclaim, as you know that some mushrooms can be toxic and that could help to make the user more conscious about that fact and also to relieve ourselves from any misuse. Finally, I have recently found out that Firestore supports a functionality that allows developers to configure a cache to work offline, so we could use it to replace our offline feature while getting rid of some of the code.

On the other hand, I consider long term improvements for the classifier that will consist basically in growing our mushroom database and fetching the classifier with more images, so we can improve our confidence. At some point I even considered building a custom classifier to improve its precision. This will require some changes in the UI as well, as mushrooms can be identified by different details: cap, gills, stalk, etc. so the results would improve a lot if instead of using a single image we could use several images of the same sample.

As a last step, if the app is used by a big number of real users I will consider releasing it for iOS.

BIBLIOGRAPHY

[1] **ShroomID PlayStore.** ShroomID application details (online). [Last access: October 10th 2021]. URL: https://play.google.com/store/apps/details?id=com.shroomid

[2] **Shroomify Play Store.** Shroomify application details (online). [Last access: October 10th 2021]. URL: https://play.google.com/store/apps/details?id=com.mushroom.shroomify

[3] **Picture Mushroom Play Store.** Picture Mushroom application details (online). [Last access: October 10th 2021]. URL: https://play.google.com/store/apps/details?id=com.glority.picturemushroom

[4] **eBolets Catalunya 2021 Play Store.** eBolets Catalunya 2021 application details (online). [Last access: October 10th 2021]. URL: https://play.google.com/store/apps/details?id=com.unusualapps.eboletscatalunya2021

[5] **Official Discord Blog.** How Discord achieves native iOS performance with React Native (online). [Last access: October 10th 2021]. URL: https://blog.discord.com/how-discord-achieves-native-ios-performance-with-react-native-390c84dcd502

[6] **Google Developers Youtube channel.** Alibaba used Flutter to build 50+ million Xianyu app (online). [Last access: October 10th 2021]. URL: https://www.youtube.com/watch?v=jtYk3gWRSw0

[7] **Official Flutter webpage.** Apps take flight with Flutter (online). [Last access: October 10th 2021]. URL: https://flutter.dev/showcase

[8] **Zazo Millán, Cristian. UPV.** ”Migración de aplicaciones Android hacia Flutter, un framework para desarrollo de apps multiplataforma” (online). [Last access: October 16th 2021]. URL: https://riunet.upv.es/bitstream/handle/10251/128486/Zazo%20-%20Migraci%C3%B3n%20de%20aplicaciones%20Android%20hacia%20Flutter,%20un%20framework%20para%20desarrollo%20de%20apps%20mult....pdf?sequence=1

[9] **Official Firebase webpage.** Firebase products (online). [Last access: October 11th 2021]. URL: https://firebase.google.com/products-build

[10] **Official Supabase webpage.** Supabase homepage (online). [Last access: October 11th 2021]. URL:
https://supabase.io/

[11] **Official Amazon Web Services webpage.** AWS amplify (online). [Last access: October 11th 2021]. URL:
https://aws.amazon.com/amplify/

[12] **Github Marketplace webpage.** Github Actions (online). [Last access: October 12th 2021]. URL:
https://github.com/marketplace?type=actions

[13] **Github Marketplace webpage.** Github Actions - Flutter Action (online). [Last access: October 12th 2021]. URL:
https://github.com/marketplace/actions/flutter-action

[14] **FlutterFire official webpage.** FlutterFire overview (online). [Last access: October 11th 2021]. URL:
https://firebase.flutter.dev/docs/overview/

[15] **Firebase official webpage.** Add Firebase to your Flutter app (online). [Last access: October 11th 2021]. URL:
https://firebase.google.com/docs/flutter/setup?platform=android

[16] **Tensorflow official webpage.** Model optimization for Tensorflow Lite (online). [Last access: October 10th 2021]. URL:
https://www.tensorflow.org/lite/performance/model_optimization

[17] **Official package repository for Dart and Flutter apps.** Tflite Flutter package (online). [Last access: October 10th 2021]. URL:
https://pub.dev/packages/tflite

[18] **Official package repository for Dart and Flutter apps.** Image picker Flutter package (online). [Last access: October 10th 2021]. URL:
https://pub.dev/packages/image_picker

[19] **Official Android Developers webpage.** Sign your app with Android Studio guide (online). [Last access: October 10th 2021]. URL:
https://developer.android.com/studio/publish/app-signing

[20] **Google Developers webpage.** Getting started with Google Play Developer API (online). [Last access: October 16th 2021]. URL:
https://developers.google.com/android-publisher/getting_started#creating_a_new_project

[21] **Github webpage.** Boletify repository (online). [Last access: October 18th 2021]. URL: https://github.com/erikbg7/Boletify