POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

The Flutter Framework: Analysis in a Mobile Enterprise Environment

Supervisors

Prof. Giovanni MALNATI

Ing. Fabio FERRERO

Candidate

Daniele PALUMBO

July 2021

Abstract

In the mobile field there are mainly two operating systems, Android from Google and iOS from Apple. To release applications on both it is necessary to develop separate codes in the respective languages of the two operating systems. Over the years, many companies created frameworks for cross-platform development of mobile applications that allow the same code to be used on multiple operating systems. Unfortunately, the advantages of having a single codebase are often balanced by the limitations that a framework introduces mainly regarding performance and integration with the mobile operating system. For this reason it is necessary to analyze a new framework when it is released to evaluate its effectiveness and in this thesis I analyze the Flutter framework.

Flutter is a framework released by Google in 2018 that experienced a rapid growth in popularity in recent years. Flutter proposes a declarative approach to UI programming and promises both a performance equivalent to the native execution, and graphics which are indistinguishable from native ones. Flutter is written in Dart and is a cross-platform framework based on a compiled approach. It uses a custom engine to render each element on the screen within a 2D Skia canvas. Flutter's entire rendering pipeline is implemented directly by the framework and is independent of the native components of the underlying operating system. It therefore differs from other cross-platform frameworks such as React Native or Ionic. Dart is a procedural language with functional aspects released in 2011. Furthermore, Dart is a language that supports both compiled and interpreted execution. This allows Flutter to have a fast and dynamic development environment, offering features such as the Hot reload. Subsequently, the code can be compiled into machine code and executed without the overhead of interpretation.

In this work I study Flutter focusing on the needs of an enterprise work environment, where it is necessary to rely on tested and effective systems. My work begins by analyzing what is the state of the art in cross-platform development and what approaches exist to create a single codebase. I also include in the analysis the other best known frameworks on the market. Next, I focus my thesis on the technical aspect of Flutter, analyzing the techniques for building the UI and the mechanisms for managing data and the internal state. In addition, I expand my work analyzing Dart and the whole development environment.

One of the main objectives of this thesis is to analyze the execution performance of Dart and the Flutter framework in different contexts. The results obtained show that Flutter performance values are two to three times lower than native results. The deterioration in performance remains constant with increasing complexity. However, in very simple cases the Flutter results are identical to native ones. On

the other hand, compared to React Native, Flutter achieves better results both in resource usage and average FPS (Frames Per Second), making applications written in Flutter effectively indistinguishable from native applications. Moreover, the documentation of Dart and Flutter is complete and well detailed and the developer community has grown rapidly thanks to the open-source approach. From my analysis I obtained positive results on the effectiveness of Flutter as a valid alternative to native development.

Acknowledgements

Scrivo questi ringraziamenti l'ultima sera prima di consegnare questa tesi. Mi sento che ho completato la prima parte del percorso che ho iniziato tanti anni fa. Questi 3 anni a Torino sono stati incredibili e per questo voglio ringraziare tutte le persone che mi sono state vicino. Sono davvero contento di aver preso la decisione di partire. Tante cose sono cambiate nella mia vita e ho avuto la libertà di fare le scelte che sentivo più giuste.

Voglio però con tutto il cuore dedicare questi ringraziamenti alla mia mamma Cristina. Quando sono partito per Torino mamma aveva già scoperto di avere il cancro. Nonostante questo, dal primo momento ha appoggiato la mia scelta di partire. Mi ha sempre incentivato a dare il meglio e non vedeva l'ora di vedermi laureato. Mi diceva che era orgogliosa di me per come ero cresciuto, riferendosi al me stesso del primo anno di ingegneria a Roma che l'università l'aveva mollata. Fino a quando hai potuto sei stata accanto a me e mi hai dato forza e voglia di fare. Sento di non aver disatteso le aspettative e ho fatto tesoro di tutto quello che mi hai detto negli anni mamma. Oggi concludo questo percorso di studi e vorrei tu fossi qui per vedermi e festeggiare con me.

Ringrazio il mio papà Biagio, Gino per gli amici, che insieme a mamma non hanno fatto mancare mai nulla a me e mio fratello. Ci avete dato la possibilità di studiare e di diventare gli adulti che vogliamo. Mi avete dato tanta fiducia quando ne ho avuto bisogno. Ti voglio bene papà.

Ringrazio mio fratello Davide che è la persona più spensierata che conosco, ti voglio bene perché sei sempre pronto a ridere e scherzare, o a parlare dei razzi per Marte, o per cantare Shinzo So Sasayego. Ti voglio bene fratellino e sono sicuro che arriverà anche per te il momento di festeggiare la laurea.

Ringrazio la mia Giulia che è stata con me nella parte più difficile di questo percorso. Sei entrata nella mia vita come una compagna di corso e ora sei la scoperta più bella che ho fatto qui a Torino. Devo ringraziarti sicuramente per avermi supportato nei momenti peggiori, quando inizio a lamentarmi di tutto e borbotto. Ma ti ringrazio anche per le possibilità che ci stiamo costruendo insieme e che mi danno fiducia per il futuro.

Voglio ringraziare i miei amici di Roma che anche vivendo lontano mi hanno

fatto sentire a casa ogni volta che stavo con loro, come se in realtà non fossi mai partito. E' anche grazie a voi che Ostia per me rimane un posto speciale.

Infine lascio un ringraziamento e un saluto a tutte le persone che hanno condiviso con me questi anni e che, ognuno a modo suo, hanno contribuito a rendere speciale la mia magistrale.

"Nel dubbio spegni e riaccendi"
Primo informatico della storia

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's highly disruptive and competitive mobile app development world, businesses would not risk missing their presence on both platform stores, Google Play Store and the Apple App Store. However, going for native apps usually means a budgeting problem, as native development involves multiple teams. This is why cross-platform app development has emerged as a valid choice for mobile app development: it aims to reduce costs without giving up the presence of an application both on Android and iOS. Cross-platform framework are also commonly used in mobile development, as they offer a wide range of advantages and simplifications during development. Cross-Platform development - also known as multi-platform software or platform-independent software - is based on the idea of writing the code once and running it everywhere.

Although cross-platform development exists since a long time in desktop environments, in this project we are interested mainly in mobile platforms. In particular, I take into consideration only two platforms, Android and iOS, which are by far the most used mobile OS in the world, since all other alternatives failed to reach a minimum market share. Indeed, in the development of a mobile app, the developers have to write the code for two different platforms in order to actually cover 99% of all active mobile devices worldwide.

There are significant differences among the various cross-platform frameworks for mobile development both for the architecture and internal functioning and for the programming languages and methodologies used. One of these cross-platform frameworks is Flutter[1]. Flutter is the solution proposed by Google in 2018 to develop cross-platform applications. It supports the mobile worlds, therefore Android and iOS, as well as the desktop and the web world. It is a very versatile and powerful framework which places itself in an advantageous position compared

[1] `https://flutter.dev/`

to other cross-platform solutions, as I show in this thesis.

Nevertheless, it is not a given that Flutter advantages could be exploited inside an enterprise context. In the enterprise environment, the choice of using a framework for cross-platform development, instead of proceeding with native development, is often seen as a disadvantageous choice. It is a common feeling to consider cross-platform as a less valid choice in the long term than the native one. Typically, this feeling is caused by the inherent limitations of using a single code to create an app that runs on multiple products, which very often have marked differences. However, Flutter is a new framework with precise and thoughtful design choices, thus it is interesting to analyze whether it may be worth using it. The purpose of the thesis is to evaluate the effectiveness of the use of Flutter in the enterprise environment. My evaluation starts studying the features offered by Flutter and then considering them in the context of an enterprise environment. The analysis also includes a discussion of the Flutter language Dart and the environment and workflow needed to work with Flutter. This work also includes a practical evaluation based on a coding experience in Flutter, which is done through the development of real use-case scenarios. These code implementations help create a more solid view of Flutter's true effectiveness.

In chapter 2 I expose the state-of-the-art of cross-platform technologies in the mobile field. There are different architectural types for cross-platform that need to be introduced to better understand Flutter, in order to talk about the various frameworks that exist on the market. Many of these have existed since a long time, so there exist numerous papers and articles on their performance and their versatility. One of the main parameters to evaluate the effectiveness of a system compared to other solutions is the usage trend of a framework. Hence, for each framework taken into consideration, I present its trend of adoption during years and the general opinion of other developers on this technology.

Chapter 3 is dedicated to detailed analysis of Flutter, which is a framework mainly focused on reducing the usage of native languages. For this reason, it offers multiple components for the development of the user interface and the user experience. A section of this chapter is dedicated to all these graphic components and to the construction of the necessary layouts. Also, the analysis of the internal navigation mechanisms of the application provided by Flutter is separated to better analyze its complexity. Then, I analyze how Flutter allows the integration of application logic, databases, text internationalization and http communications. Finally, in this chapter I present Flutter mechanisms for accessing the APIs offered by the underlying device.

Following the previously introduced chapter, in chapter 4 I contextualize the same analysis from the point of view of a company truly active in the field of mobile application development. In a company, Flutter can only be adopted once developers learn the Dart programming language, accepting also the Dart

programming style and its workflow. Furthermore, this chapter also analyzes Flutter performance and scalability in depth. The last part presents a particular Flutter feature that allows its integration into existing native applications, aiming at optimizing the development of cross-platform features without having to sacrifice what has already been developed. These feature is tested to understand if this solution is really achievable and which are its consequences.

In chapter 5 I describe and implement two different real case scenarios in Flutter. These implementations are compared to analogous interface natively developed by IrisCube Reply S.r.l.[2] in one of their mobile applications realized for a bank company. The first scenario is a simple example and aims to emphasize the versatility of Flutter graphics components. This example concerns the creation of the private area for a user. This user area includes an interface for showing personal data and another interface for editing them. The reference model comes from the user area of the mobile application of one of the most important Italian banks. The second case scenario is more complex and it aims to create an interface for personal financial management. This implementation uses an articulated layout which also includes animated graphs. Also in this case the starting model comes from the same application. In this last case, it is important to consider that in the native case a public library available on the network was used: this made necessary the comparison between the availability of third-party libraries for Flutter against native environments.

Finally, in order to offer a correct point of view on the effectiveness of Flutter, in the chapter 6 I show the results achieved by combining the considerations obtained from the analysis of the framework from chapters 3 and 4 and the results of its usage in the development of real use cases. In the results of this thesis I highlight the limitations and capabilities of Flutter and understand if it can be suitable for enterprise environments, such as companies with a large development team. I also evaluate Flutter performance both against native development, and against less recent frameworks. This helps to understand if it offers a better solution than the previous frameworks and if Flutter can really be a winning choice over native development. At the end, I draw my conclusions and I suggest some enhancements and future work for this project.

[2] `https://dart.dev/`

Chapter 2

State of the art in cross-platform frameworks

Cross-Platform development is a concept idea which includes a family of totally different approaches to exe cute a computer software on different platforms. Mobile cross-platform fever began soon after the release of the first SDK for iPhone and the born of Android OS. PhoneGap was one of the first framework made to speed up the mobile development and ease app deploying on many different platform (at his pitch, PhoneGap supported iOS, Android, Bada, Tizen, Ubuntu Touch, Windows phone).

Before introducing these approaches and related structures, it makes sense to objectively evaluate the limitations of native programming and understand what goals the cross-platform approach sets.

2.1 Goal and scenario

The main purpose of this thesis is to analyze the versatility of Flutter as the main platform in a business context. The goal is to evaluate the effectiveness of the cross-platform approach and understand if Flutter offers everything needed to create a complete and functioning mobile application. The analysis therefore includes a study of Flutter approach to mobile development, the performance of the framework and the programming language used, Dart. The goal is to define in a realistic way in which context Flutter is most suitable and what its main limits are. I compare Flutter both with other cross-platform frameworks and with purely native development.

I start from native development to identify which are its characteristics and which difficulties a cross-platform framework faces. An application developed following the native approach is developed separately for Android and iOS. The

development for the two operating systems uses different languages and programs: for Android the default language is Kotlin and the reference IDE is Android Studio, for iOS instead there are Swift and XCode. Google and Apple fully support these two environments, because in both cases they are in charge of developing and maintaining both the programming language and the reference IDE. This is indeed the main advantage of the native approach: Google and Apple guarantee developers the most stable language and IDE for development, a complete and reliable SDK for the mobile operating system and a large series of third-party libraries ready for the development. use. For native development, there are two disadvantages in particular: cost and consistency. To achieve the goal of distributing applications of a high quality level, it is necessary for a company to have talented and capable programmers available. The development team must be able both to create new features and to maintain applications over time - OS version updates, development SDK updates, and so on. Furthermore, even a design team is often needed to have a functional and aesthetically pleasing interface.

Android and iOS are conceptually similar - both were mobile operating systems born at the same time and with the same needs - but they differ in several key aspects. These platforms differ not only in terms of what native applications look like; they also differ in terms of the structure and flow. We need to keep these differences in mind to provide the best user experience through the native application design. Google and Apple over the years have developed their own philosophy on how apps should be developed. For Google this resulted in the creation of the Material Design[1], while for Apple the philosophy has become the Human Interface[2]. In practice, there are long lists of guidelines that Google and Apple encourage to use. The primary objective is to obtain a sort of uniformity in key operations for the applications on the store, such as internal navigation. For end users it is a great advantage, because even between very different applications, recurring patterns and behaviours can be found in common. In fact, continuing the example of the internal app navigation, guidelines by Apple and Google recommend to use platform-standard navigation controls whenever possible: page controls, tab bars, segmented controls, table views, collection views, and split views. Users are familiar with how these controls typically work on each platform, so if a developer uses the standard controls, its users will intuitively know how to get around the app.

Therefore the problem with the cost of the native development is what I have just explained: a company needs deep experience in both operating systems. This turns out into creating very distinct development teams that work according to the

[1] `https://material.io/design/`

[2] `https://developer.apple.com/design/human-interface-guidelines/`

platform specifications, so it is necessary to add to this separate teams for design and testing. Furthermore, we have to consider that often human cost is the largest expense in the cost balance of a company.

In addition it is difficult to build graphically similar applications on both platforms, because the graphic elements made available by the Android SDK are not always present on iOS SDK. It happens not to find the exact counterpart on one operating system that is used on another. And even if the necessary component is present in both OS, it is possible that graphically the two are profoundly different. I summarize the main and most obvious differences between Android and iOS:

1. **In-app navigation patterns**. There are few navigation options in the Material Design Guidelines. One well-known pattern is a combination of a navigation drawer and tabs. A navigation drawer is a menu that slides in from the left (usually) by pressing the hamburger menu icon. Tabs are located right below the application bar and enable content organization at a high level, allowing the user to switch between views, data sets, and functional aspects of an app. The other commonly used option is the bottom navigation component. This bar makes it easy to explore and switch between top-level views in a single tap on the bottom part of the screen. Material Design Guidelines recommend not to use bottom navigation and tabs at the same time because it may cause confusion when navigating. In the Apple Human Interface Guidelines there is no standard navigation control that's similar to the drawer navigation menu. Instead, Apple's guidelines recommend putting global navigation in the bottom navigation bar - similar to the Material design counterpart. Usually, this bar contains no more than five destinations. That is because Apple believes that primary navigation elements should be in the foreground and that the hamburger menu should be used only to store functions that are not daily tasks performed by the user.

   In any Android device there is a universal navigation bar. Using the back button in the navigation bar is an easy way to go back to the previous screen or step, and it works in almost all Android apps. On the other hand, in the Apple design approach there is no global navigation bar. Internal application screens should have a native navigation bar with a back button in the top left corner. Apple also includes a left-to-right swiping gesture in applications to go to the previous screen. This gesture works in almost all apps - in Android usually this gesture is used to move left in a tab view introduced before;

2. **Native UI components**. All common components for building the interface are present as native components in both Android and iOS. However, the graphics of these components are often profoundly different and follow the graphics of the operating system decided a priori by Google or Apple. Components such as buttons, checkboxes, sliders and switches are different in shape,

design and colors - for example, in Android an active switch has the main color of the app, while in iOS it is always green by default. The icons, although similar, differ in the thickness of the strokes and in the filling. The date picker and the time picker are modals in both OS - screen above the current screen - but they differ profoundly in structure and design. The differences are omnipresent and not always just graphic: for example, the material design makes great use of the fab button as an always accessible element for the creation of new contents; iOS lacks a counterpart. Some examples of those differences can be seen in Figure 2.1.

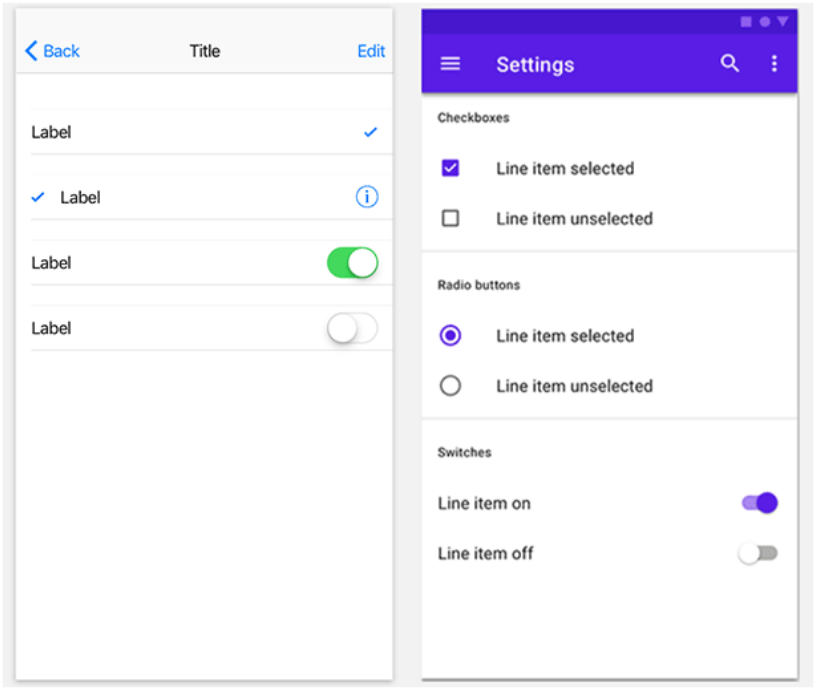

Figure 2.1: The figure shows some noticeable graphical differences between the most common components of the graphical interface in iOS and Android.

3. **Android bottom sheets and action sheets vs activity views in iOS**. There are two types of bottom sheets in Android: modal bottom sheets and persistent bottom sheets. Modal bottom sheets have two types of content: modal bottom sheets with different actions and an app list that appears after the user taps the Share icon. We can find the same types of content in native iOS action sheets and activity views. But these components look different than Android bottom sheets.

4. **Micro-interactions and animations**. Interactions help users to orient themselves in the app by showing how elements are related to one another. Familiar, smooth, and unobtrusive transitions from one screen to another keep users engaged. Motion indicates how to perform actions and offers helpful suggestions. Both the Material Design and Human Interface have guidelines about micro-interactions - or micro-animations - and they are quite similar, but with some differences. iOS users are used to smooth transitions, fluid changes in device orientation, and physics-based scrolling. iOS users can feel disoriented when movements do not make sense or appear to defy the laws of physics. Material Design, on the other hand, suggest that in a transition several interface elements are converted. Each element is classified as outgoing, incoming, or permanent. The category to which the item belongs affects how it is converted. Usually, interface elements in Material Designs during transitions can lift up and expand, or go down and contract during movements.

5. **Other minor differences**. There are different guidelines for touch targets - 44px for iOS and 48dp/48px for Android. Material design suggests also to align all elements to an 8dp square baseline grid. Android and iOS use differents fonts - Roboto (Noto as default) for Android, San Francisco in iOS and both suggest a different default font size and several ways to emphatize it - for example large title component in iOS screen.

It is clear that there are many differences between Android and iOS. Each platform has its unique interactions. Good design is design that respects users habits in each operating system. It is really important to keep in mind the differences between platforms when designing a mobile application for both iOS and Android, so that the final result will meet the expectations of users. If a company wants each element in native applications to look the same across platforms, additional development effort is required to develop custom view implementations - as iOS-like controls on Android or viceversa.

However, consistency is not always an important goal for a company and pursuing consistency across different platforms is not always an unsolvable problem. There are examples that clearly show how consistency can be obtained also through native development. In particular, the Gmail app, developed by Google - Figure 2.2, and the Instagram app - Figure 2.3, developed by Facebook, are interesting. The first follows the rules of Material design, while the second follows the Human interface. In both cases, differences between Android and iOS versions are minimal.

The cross-platform approach aims to mitigate the flaws of native development. Costs can be reduced by specializing the development team towards a single technology, and consistency can be easily achieved by using unique graphics components for both platforms. Furthermore, a single development pipeline also has the effect of speeding up development and decreasing time to market. However, if the benefits of

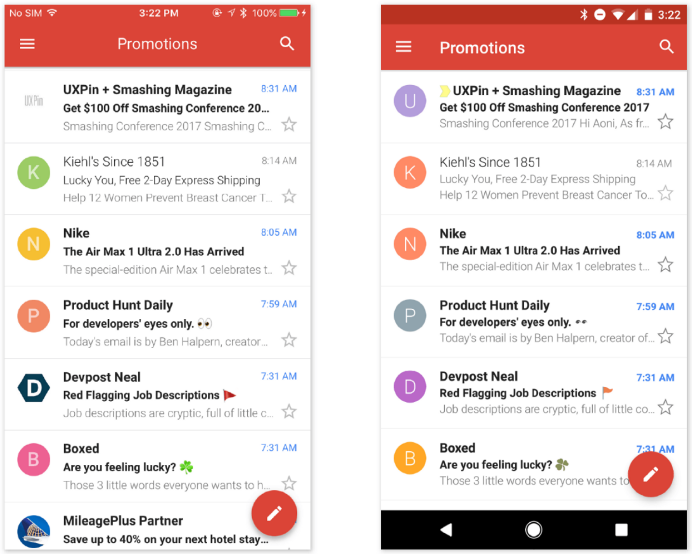

Figure 2.2: The figure shows the same version of Google's Gmail application on iOS - left - and on Android - right. The graphical differences between the two apps are minimal with a trend towards Material Design on both.

cross-platform were really that obvious and overwhelming ones, native development would be less used than cross-platform frameworks. The reality instead shows the opposite, with native development beating the cross-platform approach in all usage statistics.

The goal of developing a single codebase for different OSs is not trivial and this has held back cross-platform approach adoption and diffusion. In particular, the task of developing a single codebase can be initially divided into three aspects: user interface development, hardware feature implementation and maintainability over time.

These three aspects are by far the most important because they represent the common part in the development of any application. In particular, the first point can be further divided into:

- In-app navigation;
- layout definition;
- UI construction;
- animations and transitions.

Hardware feature implementation instead concerns exclusively the framework itself and the APIs it offers. Similarly, maintainability involves the framework

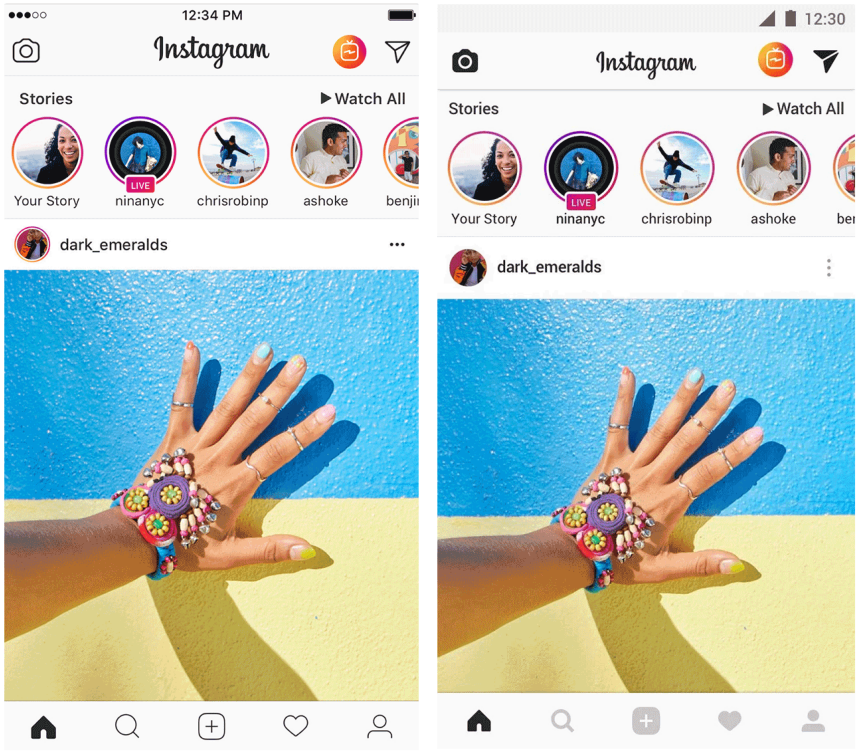

Figure 2.3: The figure shows the same version of Facebook's Instagram application on iOS - left - and on Android - right. The graphical differences between the two apps are minimal with a trend towards Apple Human Interface on both.

ability to stay stable and updated over time. The objective of the thesis becomes to understand how Flutter manages the key aspects of the development of an application and if it is able to do better than previous competitors. The analysis begins in the next section with the introduction of the various approaches to cross-platform development and the main frameworks that derive from them. After that, the work will be focused mainly on Flutter itself and related to native development.

2.2 Cross-platform approaches

Talking about cross-platform approaches, the starting point is always what can be achieved through native development. Native apps are developed with the tools and programming languages provided by the producer or the vendor of a certain mobile platform. A native app runs only on mobiles with the target platform and usually can be installed from the store. In the following sections instead, I present

several solutions adopted by cross-platform frameworks.

2.2.1 Hybrid approach through WebViews

The first solution is an hybrid approach that exploits WebViews. A WebView is an embeddable browser that an application can use to display web content. This embedded browser hides the address bar, bookmarks and others typical controls. In a WebView, the operating system natively supports the execution of Javascript code and is able to render elements on the screen formatted with CSS and HTML using webview rendering engine.

The difference within a purely webapp is the implementation of an abstraction layer that exposes the device native APIs. This becomes an interop layer that connects the Javascript APIs with the platform specific APIs. An hybrid cross-platform framework implements this layer and expose javascript APIs which exploit native capabilities. Projects based on this type of framework can both execute JavaScript inside the WebView and call native system features when needed. This means that the application capabilities aren't limited by the traditional browser security sandbox. There is support to WebViews since iPhoneOS 2 - iOS before it was called that - and from the first public version of Android.

The flexibility of the web views was the basis for the birth of several cross-platform frameworks. The use of web languages has allowed many developers to quickly start developing for mobile devices. Furthermore, in an hybrid app, every responsive interface written for a browser can be easily reused with minor modifications inside a mobile context. The hybrid frameworks mainly had to provide support to the device API through Javascript and tools for building the UI.

This type of approach is hybrid because the applications developed in this way are neither native applications nor pure web applications. Applications developed with this approach are not native because they are rendered entirely in a WebView without using the native UI framework. They are not even webapps because they are packaged and distributed as native applications through the stores and have access to the operating system APIs.

In the hybrid approach, performance is a problem. In particular, the loss of performance is due to the execution of the whole application within a webview. Furthermore, an additional overhead is the bridge needed to implement native functionality. The loss of performance is estimated to be of 40% with respect to native execution[1]. It is not trivial to obtain a native look and fell through web languages and this is another common problem to overcome in the hybrid approach; however, this problem can be certainly mitigated by the flexibility of the cross-platform framework in adapting to the underlying physical and software system.

2.2.2 Interpreted approach

The interpreted approach differs from the hybrid approach mainly because it does not use device browser engine through a WebView. An app built with this approach has an internally self-contained runtime component. There are many names for the interpreted approach that depends on technology and implementation of a certain framework: common names are interpreted approach, runtime approach, web-native approach and javascript-to-native approach - for Javascript implementations which are by far the most popular. The framework vendor develops a runtime layer composed by one runtime for each targeted platform. The runtime layer hence offers a common set of APIs which the developers can use to develop the business logic. Native funcionalities are accessed through a different bridging system than in webviews. Instead of a Cordova-controlled WebView component, in the interpreted approach frameworks use proprietary plugin-based bridging systems. These plugin allow for invocation of function interfaces implemented in native code. Thanks to the runtime, performance is relatevely close to native: it renders code components directly to the native APIs using the Javascript virtual machine both on iOS and Android. Javascript markup language is interpreted to platform-specific interface components with on-device language interpreters. Apple includes the JavaScriptCore engine - a Javascript VM - with the operating system, as well as bindings for Objective C/Swift. On Android instead there is not an embedded Javascript VM, but it can be added in the app APK through one of the available libraries which slightly increases the Android app size.

Each framework has to implement its own runtime and each runtime has its specifics, implementation and underlying plugin architecture. So one drawback of this approach is the resulted fragmented space containing numerous frameworks and tools. Consequently, plugins developed for React Native for example will not work in another framework without modifications. In a long term project changing framework - for example from React Native to NativeScript - means rewriting both the user interfaces and business logic and all the plugins and other custom native bridging components.

2.2.3 Web app approach

Another possible way to pursue cross-platform applications is using a web app. A mobile web app is essentially a web application developed with web technologies: HTML, CSS and Javascript. With respect to standard web app, the difference is in the extra optimization required to properly fit the layout to the typical display format of a mobile device. In particular, mobile screens are usually developed vertically and have a very high pixel density since the resolution is similar to a computer display but in a quarter of the space. Naturally a web app cannot be installed on the device, but is used via the device's browser.

Over the past few years we have seen increasing standardization of browsers - for example Chromium by Google is the base for many browsers - and increasing support for accessing device functionalities. This made it possible for fully web applications to both behave and appear as native applications installed via the platform store. For example, a web app has now the possibilities to access features such as localization and internal storage not needing a bridging system but only using the features offered by the browser. Standardization has made it necessary to compensate for the raw look and feel of a pure website to make the interface to look more like a native application. This reason led Google in 2015 to coin the term "Progressive Web App (PWA)" to describe web apps that take advantage of the new features offered by modern browsers. PWAs improve traditional web apps with so-called service workers (to allow for running code in a background thread), a web app manifest (to provide metadata), off-line capabilities, and an installation-like user experience. Google also provides several interface creation tools that are based on the material theme. These tools allow to enhance the graphic aspect of a PWA as well as provide it with more advanced behaviours than a normal website.

A PWA however remains a web app. It needs the browser to work and depends on the functionalities that the browser provides. If a certain functionality is not available through the browser or on a specific platform, a PWA remains limited with no way around the problem. Also, the average user looks for an application available in the platform's official store, and getting a web app through the browser is counterintuitive. Finally, the PWA concept is young and in some ways still immature. For these reasons, the web app approach is often considered more of an additional tool to increase a company's presence in the mobile market rather than a true cross-platform approach.

2.2.4 Compiled approach

The cross-platform problem has been existing since there were several operating systems. An effective solution to reuse the same engine on different platforms has been found in the field of video games. This is called the compiled approach.

In particular, in the case of video games, performance cannot in any case be one of the disadvantages because generally a video game is greedy of computational resources. For these performance reasons it is not possible to add intermediate execution layers or use interpreted languages. The solution is to bypass what the native system offers as APIs and rely solely on low-level implementations that rely primarily on OS ABIs. This allows a specialized component to be even more performing than the native one in its specific context of use. The compiled approach aims to build cross-platform applications by implementing all the functionalities an application needs to function, bypassing what the underlying OS offers at high-level.

The complexity of working at a low level of abstraction for each of the target

platforms has the consequence that the compiled approach is typically focused on a few key aspects - for example the implementation of the business logic, leaving out the graphic part to be built with classic tools. Flutter in this context is the most recent and flexible solution because it provides a complete application development tool, from business logic and user interface development, to the integration of all native physical features. Flutter has as its primary objective to mitigate all the typical limitations of cross-platform approaches.

The main difference between Flutter - or in general a framework based on the compiled approach - and the interpreted approach is that Flutter does not render any native components offered by the OS. Instead, internally all Flutter renderings are done on a the Skia Graphics Engine. This works through painting on a 2D Skia Canvas. The Flutter rendering engine is able to re-create the look and feel of native user interfaces. An application created with Flutter must be compiled in bytecode before being executed and this would greatly extend the development time in favour of a faster execution time. However, Flutter introduces a great novelty because it integrates a Dart VM. This virtual machine allows advanced features in the development phase such as the hot reload. The final application is always compiled AOT - ahead of time compilation - which eliminates the need for any intermediate layer of interpretation and guarantees optimal performances comparable to native ones.

The main disadvantage of using a compiled cross-platform framework is that any code changes must be published through the official stores. The hybrid and interpreted approaches offer the ability to push code changes using tools such as Microsoft CodePush. However, even native apps suffer from this flaw, and both Apple and Google offer solutions to quickly deploy hotfixes by streamlining the approval process.

2.3 Cross-platform frameworks

In this section, I quickly introduce the main four cross-platform development frameworks on the market. I start by including a graph that analyzes the trend of interest of each framework based on the number of global searches on the Google search engine in the last five years. The chart is in Figure 2.4.

The most popular cross-platform frameworks among developers are the following ones:

1. **Ionic.** This is an open-source SDK for hybrid mobile app development - based on the Hybrid approach. The original version was released in 2013 by Drifty and was built on top of AngularJS and Apache Cordova. However, the latest release was re-built as a set of Web Components, allowing the user to choose any user interface framework, as Angular, React or Vue. It also allows the use

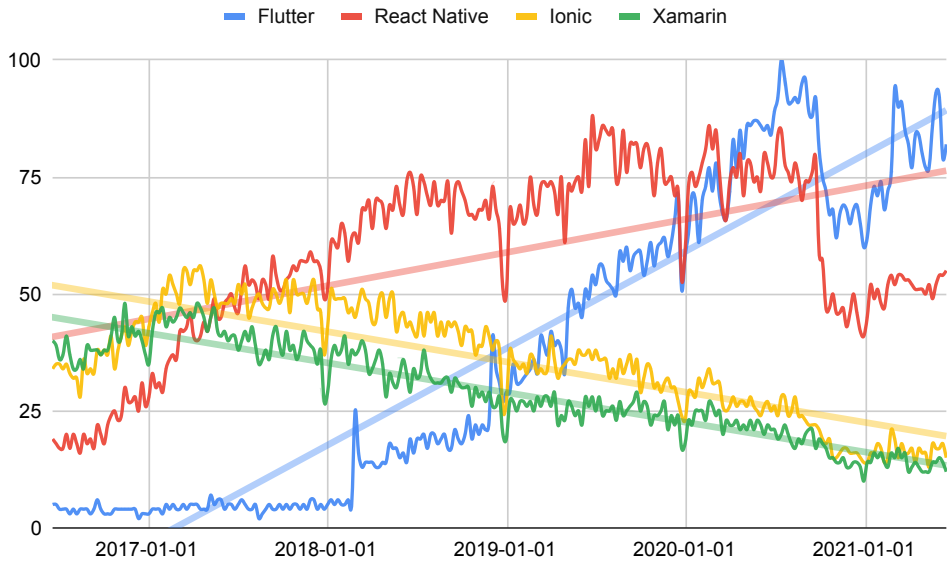

Figure 2.4: The graph represents the trend for last 5 years of world research regarding the four main frameworks for the development of cross-platform applications.

of Ionic components with no user interface framework at all.

Ionic uses Cordova plugins to gain access to host operating systems features such as Camera or GPS. Ionic supports Android, iOS, Windows UWP, web and Desktop - with Electron. Ionic offers several mobile components, together with typography and interactive paradigms. The usage of Web Components allows Ionic to provide custom components and methods to interact with them. The tabs component creates a tabbed interface, which supports native-style navigation and design and includes the history state management. Besides the SDK, features such as code deploys and automated builds are provided by Ionic with other services that developers can use and also a its own IDE known as Ionic Studio.

Ionic supports Android from version 4.4, while for iOS is supported from version 10. Ionic 2 also supports the Universal Windows Platform for building Windows 10 apps.

As can be seen in Figure 2.4, Ionic's popularity has been waning in recent years. I have tried to contextualize this decline based on the opinions of the developers. There are three most criticized aspects that have stopped the spread of Ionic [2]:

(a) **Too many releases**. The developers criticized the Drifty choice of releasing too many versions of Ionic and announcing the release of Ionic 4 when Ionic 3 was still in its infancy. As a result, many developers felt that Drifty developers intended to reinvent Ionic from scratch without careful planning.

(b) **Lack of documentation**. Ionic's documentation has always been deemed poor and incomplete. The situation became unmanageable for developers due to the many versions of Ionic supported at the same time which made the documentation fragmented.

(c) **Performance issue**. The Cordova bridge necessary for the operation of Ionic has much lower performance than the native code. In case of heavy operations the limit of the bridge becomes even more evident.

2. **Xamarin**. This framework was originally created by the developers who made Mono and MonoTouch. These software became the foundation for Xamarin.iOS and Xamarin.Android in 2011. This makes Xamarin the oldest cross-platform development framework out there. In 2013, version 2.0 of the SDK was released with the integration with MS Visual Studio. In 2016, Microsoft acquired Xamarin and made the framework open-source. Today, Xamarin SDK is a part of Microsoft's .NET platform and is fully integrated with Visual Studio IDE. It supports the C# programming language and XAML in Xamarin.Forms to build UI. However, heavy graphics and complex animations require tons of UI customization for each platform. Xamarin can be used with Visual Studio only. It does limit a company in IDE choice, every programmer have to learn how to use Visual Studio.

   Xamarin makes it easy to recycle business logic and thus create a common skeleton for cross-platform applications. When Microsoft bought Xamarin, they intended to push hard for Windows app adoption, so it made sense to leave UI development to native code. Unfortunately, today this choice does not allow to share most of the necessary code.

   As can be seen in Figure 2.4, Xamarin's popularity has been waning in recent years as for Ionic framework. The developers have expressed a lot of criticism in recent years and which can be summarized in these two points [3]:

   (a) **Confusion about Xamarin**. There are several frameworks for Xamarin. Previously there were Xamarin.iOS and Xamarin.Android, now wrapped under Xamarin.Native. Then Xamarin.Forms was released in 2020 which promoted an innovative approach to reusing UI code, but which was incompatible with Xamarin.Native and instead relied on Xamarin.Essential. At the end of 2020, a new framework for cross-platform development by

Microsoft was announced called MAUI - Multiform App User Interface - which will be released at the end of 2021. The result of these choices has been total confusion for the developers.

(b) **Difficulty and complessity**. Many developers comment on Xamarin describing it as unnecessarily difficult and cumbersome. Debugging is complicated and often crashes, logging is incomplete and heavy, multiple IDEs are needed to have a complete tool - Visual Studio for compiling, Android Studio for UI development and JetBrains Rider for coding. The Jan Rabe article that I have included in the bibliography details the many technical difficulties encountered when using Xamarin [4].

3. **React Native.**

   This is an open-source cross-platform framework created by Facebook to develop mobile applications. React Native was born thanks to the experience gained with the development of ReactJS. React Native supports application development for Android, iOS, MacOS, Windows, Web, UWP, Android TV and TvOS and it combines the language and skills of ReactJS with the capabilities of the native.

   React Native works virtually identical to the React framework, except for how React Native manipulates the DOM which is not via the Virtual DOM. React Native DOM runs in a background process, which interprets the JavaScript code directly on the end-device. Communication with the native platform are via serialized data over an asynchronous bridge. This bridge is th foundation of the React Native features and capabilities.

   Native components are wrapped by React components in a way in which they interact with native APIs using the declarative UI paradigm and JavaScript. This allows Javascript developers to develop native app and can let existing native teams work much faster. React Native styling has a similar syntax to CSS, but it does not use HTML or CSS. Instead, JavaScript messages are used to manipulate native views. Furthermore, React Native allows developers to write native code inside React Native projects, which makes them even more flexible.

   As can be seen in Figure 2.4, the popularity of React Native has been growing in recent years. ReactJS is one of the most popular web development frameworks and this has greatly influenced the popularity of React Native. Many developers like React Native and Javascript. Additionally, Javascript is establishing itself as a new industry standard globally, and React Native has access to the world's largest community and millions of libraries. However, the growth of React Native has not been as great as Facebook hoped. Compared to native development, React Native has lower usage rates and there

are criticisms regarding its heaviness, long technical times to build code and general lower performance than native.

On an industrial level, React Native is Flutter's biggest competitor. Right now, it is the only large cross-platform framework that is healthy and with great growth opportunities. For this reason I will quote React Native many times in this thesis as an example or comparison.

4. **Flutter**

   Flutter is an open-source UI cross-platform framework developed by Google released in 2018. Flutter allows to develop applications for Android, iOS, Web, Windows, MacOS and Linux from a single codebase.

   Flutter main focus during release presentation was to provide a way of being able to render consistently at 120 frames per second. In 2021, version 2.0 was released, which was the first version to add official support for web-based and desktop applications for Windows, macOS, and Linux and an improved Add-to-App feature.

   During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile. For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web. The framework is open source, with a permissive BSD license, and has a thriving ecosystem of third-party packages that supplement the core library functionality.

   Flutter's success was immediate as also demonstrated in Figure 2.4. The developer community has welcomed flutter for its potential. In the following chapters of this thesis I will try to analyze all the main aspects of Flutter and objectively evaluate its capabilities.

Besides the four frameworks I've shown above, there are many other frameworks for cross-platform development. If a framework is not on this list it is for one of this two reasons: either the framework is not specifically designed for creating cross-platform applications, or the framework is old or not very widespread. Furthermore, I think that not including further frameworks as comparisons does not affect the quality of Flutter's analysis. This is because I will be comparing Flutter primarily to native development and React Native - currently the industry standard for cross-platform mobile applications.

Chapter 3

Flutter framework analysis

3.1 Flutter architectural overview

This section aims to provide an high-level overview of the architecture of Flutter, including the core principles and concepts that form its design.

Flutter is designed as a layered and extensible system. Layered because is a system structured in a way that different services of the framework are split into various layers of components, where each component has a specific well-defined task to perform. This design promotes modularity. Instead, Flutter is extensible because every high-level feature exists as a series of independent libraries - known as packages - where each of them depends on the underlying core called Flutter Engine. For this reason the Flutter framework is relatively small. Developers choose what to add and this helps in the long term to keep the app dimension small. Flutter is composed by three main layers:

1. **Flutter Engine.** It exposes the primitives necessary to support all Flutter applications and it is mostly written in C++. Whenever a new frame needs to be painted, the engine is responsible for rasterizing - converting an image from a vector graphics format to a bitmap image - the scene composed by the widgets tree. It provides the low-level implementation of Flutter's core API, including graphics (through Skia), text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain. The engine is exposed to the Flutter framework through dart:ui, which wraps the underlying C++ code in Dart classes. This library exposes the lowest-level primitives, such as classes for driving input, graphics, and text rendering subsystems.

2. **The Flutter Framework**. High-level developers interact with Flutter

through the Flutter framework, which provides a modern, reactive framework written in the Dart language. It includes a rich set of platform, layout, and foundational libraries, composed of a series of layers. Working from the bottom to the top, we have:

- **Basic foundational classes**, and building block services such as animation, painting, and gestures that offer commonly used abstractions over the underlying foundation.
- **The rendering layer**, which provides an abstraction for dealing with layout. It is used within the construction of a tree of renderable objects which can be dynamically changed and auto-updated reflecting changes in the layout.
- **The widgets layer**, which renders object in the rendering layer has a corresponding class in the widgets layer. In addition, the widgets layer allows developers to define reusable custom classes. This is the layer at which the reactive programming model is introduced - reactive programming is a development model built around asynchronous data streams and propagation of changes.
- **The Material and Cupertino libraries**, which implements the Material and iOs design - and behaviours - through a comprehensive set of pre-made components based on primitives from the widgets layer.

3. **Platform specific Embedder**. There is one platform embedder for each supported platform and it provides an entrypoint for the core engine functions; The embedder works in coordination with the underlying operating system to use rendering surfaces and access user data input, as well as managing the message event loop. The embedder is written in a language that is appropriate for the platform: Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Due to the embedder layer, Flutter code can be both integrated into an existing application and distrubuted as a stand-alone app.

Flutter's architecture is summarized graphically in Figure 3.1. Here it is possible to see the three main layers that make up the internal architecture and their main responsibilities.

Considering what I explained in the chapter 2, Flutter is a cross-platform framework based on the compiled approach, so an application to work need to include all the layers of the framework and this adds a fixed overhead in size. In particular - according to the description in the official documentation - a minimal hello-world app bundled and compressed as a release APK has approximately 4.3 MB of size for ARM, and 4.6 MB for ARM 64 [5]. This is mainly due to the core

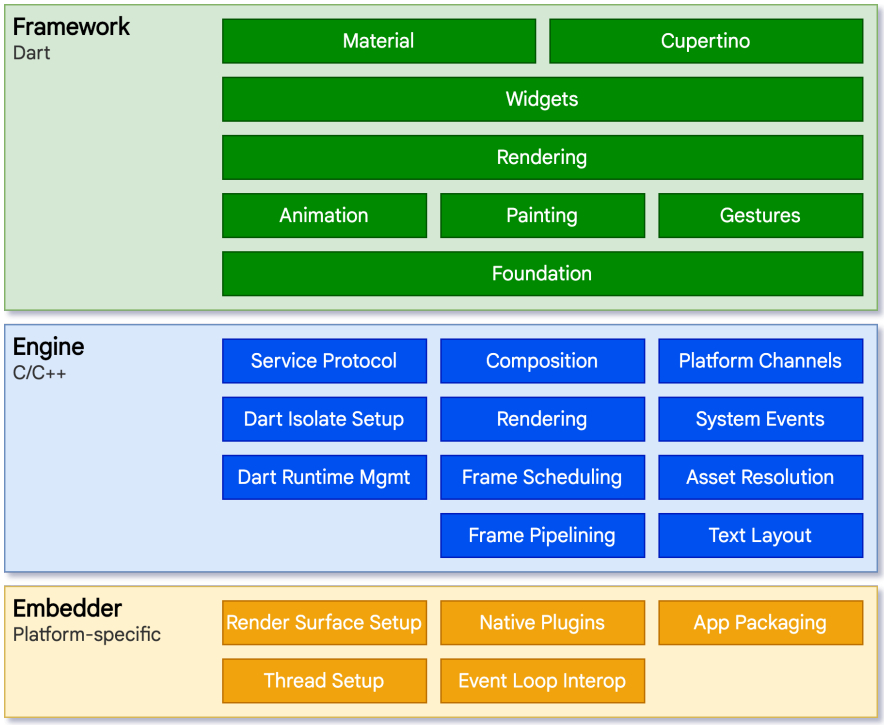

Figure 3.1: Graphic representation of the internal architecture of the Flutter framework.

engine - which is approximately 3.2MB (compressed) - and the framework code - approximately 840KB (compressed). For this reason the platform plugins like the camera or the webview are available as packages. In the end, are packages also platform-agnostic features like http or the animations packages.

3.2 User interface and UI development

3.2.1 Reactive and declarative approach

From the outside, Flutter is a reactive and declarative UI framework. As said before, "reactive" means that the internal architecture of Flutter is built around asynchronous data streams. These stream are necessary for flexibility in managing the life cycle and behaviour of the UI components and for the state propagation. State propagation is necessary to keep the declarative UI independent from its internal state. In other words, Flutter is in charge of updating the interface at runtime as a consequence of a state change. From a developer perspective, the main goal is to provide a mapping for the application state - intended as data

concerning the application behaviour - to the interface state - data representing how the UI is currently displayed[6].

This model is a fairly recent approach in the world of UI development and is inspired by work that came from Facebook for their own React and React Native frameworks. This work includes serious rethinking on many traditional design principles with the aim of keeping simple the management of interfaces that usually grow very quickly in terms of graphical and behavioural complexity.

In most traditional UI frameworks the imperative approach reigns: the user interface initial state is described once and then separately updated via code at runtime, in response to events. Parameters in the constructor of a component are used to describe the initial state and then the field relating to the change of state is explicit modified. For example, a widget A can be initialized with `new` keyword or retrieved with `find_by_id()` method. Afterwards, the field color of A can be changed explicit invoking the field reassignment: `instance_widget_A.set_color("new color code")`. This system is obviously also used for the construction of widget children subtree, creating a problem on how move parameters down the hierarchy. Another challenge of this approach is the state management that could become very tricky: as the application grows in complexity, developers need to keep track and be aware of how and where the state changes and which are the consequences for the entire UI. Let us consider a simple application with a color box or a hue slider used to choose the color of a text: in addition to those, consider also some radio buttons to choose whether the text is italic or bold and another one for graphic effects such as shading. There are many places where the state can be changed and, as the user interacts with the UI, changes must be reflected in every other related place. Minor distractions can create intricate ripple effects to unrelated pieces of code that are difficult to predict and maintain over the long term. One solution to this problem is to assign different responsibilities to different conceptual components, as for example in MVC approach: user triggers data changes to the model via the controller, and then the model pushes the new state to the view via the controller. Although this is a consolidated and functional approach in a client-server context, in UI workflow it is not feasible, since in MVC creating and updating UI elements are two separate steps that can easily get out of sync: as consequence unpredictable UI behaviours in a real time use can happen.

The solution adopted by Flutter, along with other reactive frameworks, uses an alternative approach based on explicitly decoupling the user interface from its underlying state. Flutter APIs, similar in style to React APIs, allow developers to only describe how to create the UI. The framework then uses this information - widget configuration - to both create and update the user interface as needed. Flutter widgets, again similar to React components, are represented by immutable classes: these components promote a pure functional approach and they are used to

configure a tree of objects. In a hierarchical way, some widgets are used to manage a specialized tree of objects for layout, which becomes the container where manage another tree of objects dedicated to composition - for example a color property of a text or aspect ratio of a video source. Basically, Flutter is built around mechanisms to efficiently walk complex trees, convert trees of objects into lower-level objects, and propagate changes across these trees. In this context, a widget practically is only a lightweight "blueprint". The widget, to declare its user interface, needs to overrides its the build() method, which is a function used to convert its own state in UI code - this code is later used to actually draw the user interface on the Skia canvas. To make a change in the UI, a widget triggers the build method on itself - through the setState() for StatefulWidgets - and it constructs a new widget subtree where the UI is now affected by the last change.

Listing 3.1: Simple widget declaration in Flutter. Available property for a widget are defined in its constructor. More on that later.

```
return WidgetA(
    color: red,
    propertyX: valueX,
    child: WidgetB(...)
);
```

The build() method is designed to have a fast execution and it should be free of side effects, all heavy computational work should be done asynchronously and stored as part of the state - more on state management later. This allow Flutter to call it whenever needed, potentially once for each frame rendered - similarly to how React handles the DOM and its updates. In conclusion, updating is done by telling the framework to replace a widget in the tree with a new one updated. Consequently, new and old widgets are compared by the framework to efficiently updates the user interface. Flutter automated comparison process is quite effective, and this enables high-performance and promotes interactivity. This approach creates some memory side complications caused by numerous fast objects instantiation and deletion, but this is handled by how the framework language manages these complications. Fortunately, Dart is particularly well suited for this task.

As explained, in Flutter widgets are mandatory to pursue a declarative paradigm approach: in a nutshell, declarative means that we simply declare how an element should be rendered given the current state without never actually touching it. To make it more clear in practice, it's good not to think of how to accomplish a certain result - intended as flow of statements that change a program's state - but instead how a component should render itself in it's new state [7].

The application structure is the most important consideration to embrace a declarative approach and Flutter declarative UI is entirely built around widgets as a unit of composition. As mentioned, a widget is a lightweight "blueprint", a basic building block heavily customizable, that represents an immutable declaration of

part of the user interface.

Composition is obtained through a hierarchy structure: from a simple text widget all the way up to the root widget, each of those nests inside its parent and receive both context and constraints from the parent. The root widget is usually the container that hosts the Flutter app, which typically is a MaterialApp or a CupertinoApp. Both are convenience widgets that wraps a number of widgets - design and behaviour - commonly required in Google Material Design or Apple Human Interface. The following example is useful to summarize what I have said up to this point:

Listing 3.2: Simple Material app in Flutter. This is the entire code needed to make a hello world application for Android and iOs. As already suggested by the previous explanation, all classes instantiated in the following example are widgets.

```
1  // Flutter Material library
2  import 'package:flutter/material.dart';
3
4  // Dart default entrypoint. Can be override in configuration.
5  void main() => runApp(HelloWorldApp());
6
7  // I extend Stateless widget because this part of the UI does not
       depend on anything other than the configuration information in the
        object itself.
8  class HelloWorldApp extends StatelessWidget {
9
10   // As mentioned, my widget overrides the default build method.
       Build returns always a Widget object. New keyword can be omitted
       as it is superfluous.
11   @override
12   Widget build(BuildContext context) {
13     return MaterialApp(
14       home: Scaffold(
15         appBar: AppBar(title: Text('Hello World Example')),
16         body: Center(
17           child: Text('Hello World'),
18         )
19       ),
20     );
21   }
22 }
```

The result of the previous code is represented in Figure 3.2. The resulting screen is really simple, but it is interesting to note that with very few lines of code it is possible to launch complex applications.

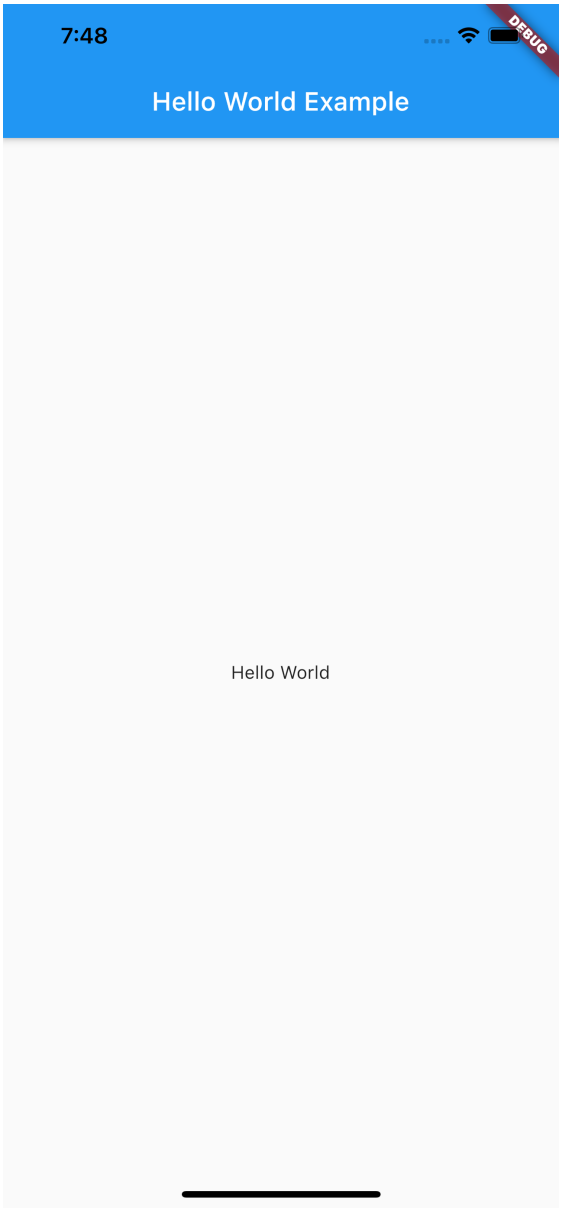

Figure 3.2: The Hello-World application running on iPhone. To create this application, the code is in the hello-world snippet.

3.2.2 Building complex layouts

To understand how Flutter layout mechanism works, we need to understand how Flutter building components are made. "Composition over Inheritance" is a key aspect in Flutter architecture. Also known as composite reuse principle in OOP programming, composition over inheritance promotes the idea that classes should

achieve polymorphic behaviour and code reusability by composition of other classes that implement the desired functionalities. This idea competes with the inheritance approach that promotes polymorphic behaviour through extension of a parent class. In practice, in Flutter widgets are composed of many other single-purpose widgets. These are usually very small widgets that combine to produce powerful effects. The class hierarchy is deliberately shallow and focus on maximize the possible combinations of simple and reusable components. It's no surprise that Flutter made the choice to follow composition as an architectural approach, because that's what React recommends too. Everything in React is a react-component, and each of them follows a strong component based model which promotes composition over inheritance. This is a growing trend in all frameworks built for UI development, including very recent solutions such as SwiftUI[1] and Jetpack Compose[2].

In Flutter this results in core features also being abstract and implemented as separate components. In contrasts with more traditional APIs, where common features like padding or alignment are built into the inherited core of every layout component, in Flutter they are available as stand-alone widgets. In Flutter to center a widget one wraps it in a Center widget, rather than using a polymorphic align property.

Since the widget may be a very simple and focused component, it is possible to find widgets without a visual representation of their own: layout widgets - as padding, alignment, rows, columns, and grids - have the sole purpose to manage their own layout responsibility in another widget. But the real potential of the compositional approach and the flexibility of widgets is evident in some utility widgets Flutter has. For example, the Container widget is made up of different simpler widgets, each of them responsible for a specific feature as layout, painting, positioning or sizing. More specifically, these widgets are ConstrainedBox - allows to impose additional constraints on its child, DecoratedBox - paints a Decoration object either before or after its child paints, and Transform - applies a transformation just prior to painting its child without accounting in space occupation. Prior to this, a Container is such a versatile and powerful component useful in practically any situation as a basic building block. It is used from a clean construction of the layout up to heavily customized and animated components. Compositional approach is not only useful to developers who made Flutter, but can be actively used also by developers working with the framework. Flutter allows a developer to drill down into the source code for any widget and examine it. Continuing with the container example, in its source code is possible to see all the widgets I described before, along with other simpler ones such as padding and alignment. In

[1] `https://developer.apple.com/xcode/swiftui/`

[2] `https://developer.android.com/jetpack/compose`

this way a developer can compose a custom container from core and basic widgets in novel ways, instead of subclassing the default Container Widget to introduce a customized effect. Anyway, the most important thing for a widget is its visual representation inside the build method, because it represents the widget part of the user interface in a concrete way. Flutter recursively asks each widget to build itself. This process continues in depth until a fully described tree made of concrete renderable objects is obtained. Finally, objects of this tree are stitched together into a renderable object tree. Efficiently lay out a hierarchy of widgets, defining their size and position, before they are rendered on the screen is one of the Flutter main tasks.

In order to explain the process of defining a layout in more detail, it is useful to first investigate how Flutter handles the rendering process: during the build phase, each node in the render tree is a RenderObject, which is an extremely general abstract model for layout and painting. RenderObjects have slots for memorizing a parent and parentData - where the parent RenderObject can store child-specific data. However, a RenderObject does not define a child model nor a coordinate system or a specific layout protocol. Knowing the minimum necessary of the child object and not binding to any reference system, make RenderObject a flexible component which has sufficient abstraction to handle all variety of use cases. Basically, a RenderObject contains primitives that are easy to specialize: for example, RenderParagraph is in charge of rendering text, RenderImage renders images and RenderTransform manages transformation before painting. Most of the widgets in Flutter are rendered by an object that inherits from the RenderBox subclass of RenderObject. This uses Cartesian coordinates as layout system and represents a RenderObject of fixed size in a 2D Cartesian space and provides the basis for a box constraint model: for each widget to be rendered, a minimum and maximum width and height are established. The RenderView is the root of all RenderObjects ant it represents the total output of the render tree. When a new frame has to be rendered for any reason - the underlying platform demands when to render - a call is made to RenderView compositeFrame() method. Consequentely, a SceneBuilder is built to trigger an update of the scene. RenderView waits for the scene to be completed and then passes the composited scene to the Window.render() method in dart:ui. Finally, the composited scene arrives to the GPU that controls how to render it. This is in a nutshell how Flutter render mechanism works. Hence to actually perform layout, Flutter has to traverse in depth-first the render tree, passing down from parent to its children information about size constraints. Flutter layout can't really be understood without knowing the main rule that defines how widgets decide how much space they can use:

"Constraints go down. Sizes go up. Parent sets position."

In more detail:

- A widget receives its own constraints from the parent. Constraints are a set of 4 doubles: a minimum and maximum width, and a minimum and maximum height. By default, minimum values are 0 and maximum values depends on platform screen.

- This widget has to say to its parent the size he wants to be within the constraints. To discover it, the widget passes to its list of children their constraints. Many widgets have just one child, but others - as Column, Row or ListView widgets - have more children. Each children can receive different constraints based on parent widget internally behaviour. One by one, children are asked what size they want to be.

- Then, using children size, the widget positions them one by one - horizontally in the x axis and vertically in the y axis.

- Finally, the widget computes its own final size and tells it to its parent.

As I said, a widget deciding its size must respect the constraints given by its parent and each widget passes up its own size. This process is implemented as single walk down in the widget tree. At the end, the result is that each object has a defined size and is ready to be painted with the paint() method. This Flutter layout engine based on box constraints approach is very effective and it is able to layout the widgets tree in $O(n)$ time. However, this process also has some limitations:

- Depending on constraints means that a widget can't have any size it wants. This situation could results in a typical error that the space needed for a widget is out of bounds. Hence, the widget overflows.

- A widget may have no limits on a specific axis. For example a ListView widget does not have a limit on the main axis because it's implemented as a ScrollView and because it wants to be tall as all the space needed vertically. So a ListView will have the total internal height value equal to the sum of all children heights. If one of its children wants to have an infinite size on an axis without constrains, this will produce an infinite dimension error on that axis.

- The parent always decides children position on the screen, so a widget does not decide - and cannot know - its own position. This usually does not happen in a UI built with an imperative approach.

- Is not possible to precisely define position and size of a widget without taking into consideration all the widgets tree. A parent size and position depends on its parent size and position, and so on up to the root.

- If a parent does not have enough information in its constraints on how to align a child and this child wants to have a different size, the parent might ignore child size. To avoid this, is necessary to be specific in alignment declarations.

To better understand the Flutter layout mechanism, it is useful to analyze a real case. Consider the following code:

```
1  Row(
2    children: [
3      Expanded(
4        child: Center(
5          child: Container(
6            color: Colors.red,
7            child: Text(
8              'This is a very long and big text, which will not fit in
     one line.',
9              style: TextStyle(fontSize: 20),
10           ),
11         ),
12       ),
13     ),
14     Container(color: Colors.green, child: Text('Short text!')),
15   ],
16 )
```

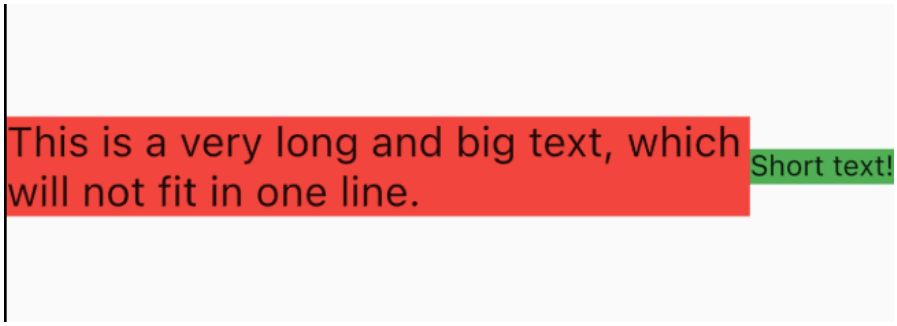

Figure 3.3: Graphical representation of the Expandend widget that guarantees the maximum allocation of the available space to its child based on the space occupied by the parent widgets.

This code produces as result the screen in Figure 3.3. I analyze in detail the interactions between the various widgets:

- The first Text widget - line 7 - wants to be long as all the space necessary to the text to fit in one line. A Row widget does not impose any restriction on

width, so by default the first text widget overflows the physical screen on the right.

- The Container - line 5 - does not define any additional constraints on its text widget child. In this example is useful only to visually show the text widget dimension by setting the background color to red. Same regarding the center widget - line 4.
- Instead, the Expanded widget - line 3 - tries to get all the available space on the row main axis. Therefore, it does not impose any limit on its child width and asks to its parent Row to have an infinite size.
- When one of Row children is wrapped in an Expanded widget, the Row does not let this child define its own width anymore. Instead, the Row widget computes Expanded child width according to width of others children. In this example, the second Text widget - line 14 - just needs to occupy the space dedicated to the short text, so the Row widget forces the Expanded child widget to have a width equal to the remaining space. In other words, the original child's width becomes irrelevant, and is ignored.

I said that a parent widget might ignore child decided size, but it is also common to have a parent that dictate precisely the size of a child widget by setting maximum and minimum constraints to the same value. Let's think to the topmost render object which constrains its child to be the size of the underlying physical screen. A more flexible situation is when a parent dictates to its child only one dimension - width for example - leaving to its child to decide the desired height. For a simple example, see the FlowText widget, which is built with respect to a given width but it can be tall as needed by the text to not overflow.

Finally, also a child can decides the best use of the space the parent made available. In this case, a child object decides how to render its content basing on parent constraints. For example, in a LayoutBuilder[3] widget the builder child uses the passed-down constraints to determine which widget to return. Check the following code:

```
1 Widget build(BuildContext context) {
2   return LayoutBuilder(
3     builder: (context, constraints) {
4       // As example, consider the first condition for a vertical
    layout and the second condition for the same layout in portrait
    mode
```

[3] https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html

```
5         if (constraints.maxWidth < 600) {
6           return OneColumnLayout();
7         } else {
8           return TwoColumnLayout();
9         }
10      },
11    );
12  }
```

Flutter provides developers with a rich catalog of widgets, which are classified in visual, structural, platform, and interactive widgets. There are about two hundred widgets available and they are deeply different from one another: each one is built around a specific responsibility and it is easy to find always the most suitable widget for the current problem. Flutter documentation is incredibly comprehensive and exhaustive: each widget has its own web page where it is possible to find an introduction to the widget, documentation about constructor, parameters and options, examples of use and often also a summary video. In addition, Flutter provides numerous tutorials and usage examples that immediately help developers to learn the most common and used widgets.

In conclusion, the layout mechanism offered by Flutter is powerful, flexible and incredibly fast. Developers are required to fully understand the main rule that defines the entire layout process: "Constraints go down. Sizes go up. Parent sets position". With this rule in mind is possible to build actually complex interfaces with minimum effort using what Flutter offers as building components. To avoid errors or unpredictable behaviours in the various nested layers of the UI, it is always important to recognize which widgets define constraints, which ones adapt to available space, which ones require infinite space and which ones use only the space strictly needed.

3.2.3 Assets management

In a Flutter app is obviously possible to include also assets among the code. Assets are also commonly known as resources, especially for people coming from Android background. An asset is simply a file - or in general a content - that is bundled and deployed with a Flutter app. This file is accessible at runtime and can be used in many ways. Common types of assets are static data, as JSON files, configuration files, icons, and images. All common image extensions are supported - JPEG, WebP, GIF, etc.. - but Flutter lacks by default in supporting vector graphics formats. This lack is due to the rasterization cost of the SVG format if is used without understanding it. One SVG image can completely prevent applications from running smoothly at 60 frames per second if blindly used. Anyway, Flutter can

support SVG format by importing the flutter_svg[4] dependency in the pubspec.yaml file.

This file is also used by Flutter to identify assets required by the app. Every app needs a place where defining some metadata, including its dependencies and imported assets. All of this metadata goes in the pubspec file, named simply pubspec.yaml. This file is located at the root of a Flutter project and it is written with the YAML syntax.

The assets subsection of the pubspec.yaml specifies files that should be included with the app. Each asset is identified by an explicit path relatively to the root, where the pubscpec file is located. The order in which the assets are declared does not matter. The directory name used to save the assets does not matter too, it can be the default assets folder as well as a user defined folder. During the build process, Flutter places assets into a special archive called the asset bundle.

The app can read from the bundle at runtime through an AssetBundle object. This provides two main methods that allows a developer to load a string/text asset - loadString() method - or an image/binary asset - load(). AssetBundle requires in both methods a logical key to take out of the asset from the memory. The logical key map is the path to the asset specified in the pubspec.yaml file at build time.

For what concerning string/text asset, it is possible to load one of them directly using the rootBundle global singleton. Each Flutter app has a rootBundle object and it contains all the resources that were packaged with the application when it was built. However, it's recommended to obtain the AssetBundle from the rootBundle for the current context in which the widget is being built - using DefaultAssetBundle.of(context). This adds a layer of indirection in which one of the ancestors of the current widget can provide a specific AssetBundle at runtime, rather than directly use the default rootBundle built-in the app. However, if a specific Assetbundle is not defined at runtime, the DefaultAssetBundle.of(context) returns the rootBundle; obviously, outside of a Widget context, rootBundle is used to load such assets directly.

Instead, AssetImage class is in charge to load an image. It is used inside a widget build() method. The AssetImage class fetches an image from the asset bundle provided by the context. If this asset bundle derives from Flutter AssetBundle, it inherits resolution awareness feature when loading an image. This means that given a main asset and a set of variants, AssetImage chooses the most appropriate asset for the current context. It understands how to map a logical asset request into the physical asset, based on the device pixel ratio and size given in the configuration. In order for this mapping to work, assets must be arranged in a particular directory structure which reflect the scale factor.

[4] `https://pub.dev/packages/flutter_svg`

Flutter assets can be accessed also in platform specific code. On Android the assets are available using the AssetManager API provided by "android.content.res". The logical key needed for retrieving the asset is obtained from lookupKeyForAsset on PluginRegistry.Registrar; this process available when developing a plugin for Flutter in native code, and allows Java or Kotlin code to rely on Flutter assets configuration.

The process is similar for the iOS counterpart. Instead of the AssetManager, in iOS Flutter assets are available using the mainBundle. The process to retrieve the key is the same, and this allows to use Flutter assets configuration with Objective-C/Swift code.

In other occasions is not possible to rely on Flutter for managing assets, but it is necessary to work with assets in the platform projects directly. Two common uses are the icon app and the launch screen.

Firstly, updating a Flutter app icon works the same way as updating icons in native Android or iOS environment. For Android the new icon goes in the Android folder inside the Flutter project root directory, under .../android/app/src/main/res. Developers have to replace default placeholder images named ic_launcher.png with the desired assets and after they have to update the AndroidManifest.xml file.

For what concerning iOS, the process is similar but it is done inside iOS folder from the Flutter root directory - .../ios/Runner. In AppIcon.appiconset directory there are always several placeholder images to be replaced with the desired new icon and names as dictated by the Apple Human Interface Guidelines.

Secondly, also for drawing transitional launch screen Flutter uses native platform mechanisms. The launch screen starts on app initialization while the Flutter framework loads. The launch screen persists until Flutter renders the first frame of the application. The Launch screen, also called splash screen in Android, usually is a simple image, but it could also include animated content or special transitions. It is absolutely necessary to consult the native documentation of the platform because Flutter does not include any of these features.

3.3 Flutter navigation and routing

Navigation and routing are one of the core aspects of a mobile application. They allow the user to move among different pages that the application contains for displaying different types of information. In a nutshell, navigation means defining the workflow of an application as the act of moving between a collection of screen to complete tasks, while routing is the way to handle the navigation.

In Flutter screens and pages are just widgets and are known as routes. This approach is different from Android and iOS: in the first one, a route is similar to an Activity, whereas in the second one, a route is equivalent to a ViewController. Until

the end of 2020, Flutter provided a basic widget for routing, called Navigator. The Navigator manages a stack of Route objects and provides two methods implemented with an imperative approach: those API are Navigator.push and Navigator.pop methods. This two method were used to navigate between two routes: push() navigates to one route from another route, while pop() navigates back to the previous route.

As were implemented, those imperative APIs allows targeted modifications to the routing stack, without offering flexibility: developers asked extensions to the imperative API to obtain full control over the Navigator stack. Furthermore, the imperative approach on which Navigator is based feels outdated and not very Flutter style. In a generic widget, to change its children is enough just rebuild the widget with a new set of children, but this concept is not valid for the Navigator: to change the set of routes with a new set, it is not enough just to rebuild the navigator, but must be used a mix of imperative APIs to achieve the goal.

The other known limitation for developers with the old routing system is nested Navigators. These are commonly used in tabbed interfaces where each tab comes with its own Navigator, which is nested with respect to the root Navigator. Nested Navigators are not necessary to keep track of the route as a whole, but they are needed to know the single tab in which the user is. Using the default Navigator meant that developers had to manually implement nested routing to avoid unexpected behaviour for the back button. If a user hits the back button to go back from the current tab to the previous one, the button will not get him back. Instead, it will pop the route from the Navigator, bringing the user to the previous view in the stack instead of the expected tab.

For these reasons, at the end of 2020, the Flutter team released a big update for the internal navigation and routing system, called Navigator 2.0. Now Flutter provides developers with a complex and fully customizable system for implementing and managing in-app navigation. New Navigator APIs allows to set the navigator's history stack in a declarative way, by providing a list of Pages objects.

Before explaining the various components in detail, it is useful to show the innovations introduced with the new Navigator and what the several components are in charge of:

- **Page**, is an immutable object which acts as a blueprint describing a route that will be placed into the Navigator stack.

- **Router**, is a widget which wraps a Navigator. It acts as a dispatcher to open and close pages of the app. The Router is able to configure the Navigator current list of Pages relying on the current app state. In addition to this, the Router is also able to change the Navigator configuration listening to events from the underlying operating system.

- **RouterInformationParser**, is in charge of parsing a RouteInformation object in a user-defined type. When the state change, it emits a new route and the RouteInformationParser converts its abstract data type T in the type defined by the developer.
- **RouterDelegate**, is the hearth of the Router. It defines the behaviour of how the Router listen to changes in the app state and how to react to these changes.
- **BackButtonDispatcher**, it reports to a Router when the user taps the back button. This happens only on platforms that support back buttons, such Android.

In Figure 3.4 is showed in detail how all navigation components interact with each other, with the operating system and with the status of the application.

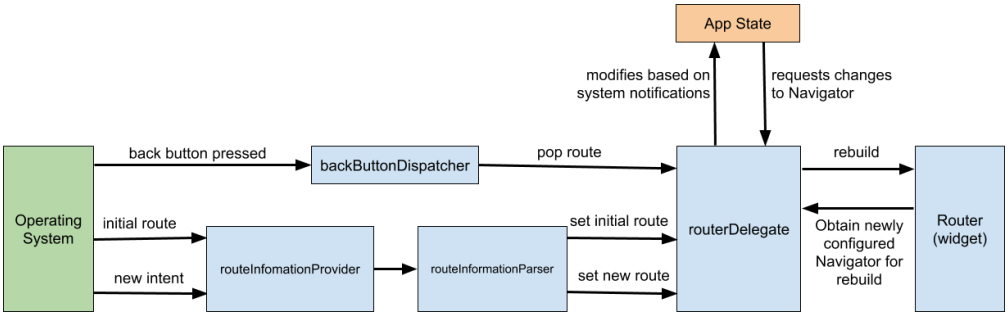

Figure 3.4: Flow model within Flutter Navigator version 2 [8]

Main task of the Navigator is to transform a Page object in its corresponding Route. The relationship between Pages and Routes is similar to the one between Widgets and Elements. As a Widget describes the configuration for an actual Element, a Page describes an actual Route living on the Navigator stack. Due to new declarative approach, when the Navigator receives an updated list of Pages, the Navigator is able to update its internal list. The new list is compared to the old one and replaced. As a consequence, the route order in the history stack of the Navigator is also updated to match the order of the new pages list.

A Page object can always be converted in the corresponding Route in the history stack, but not every Route corresponds to a Page. This strange situation is due to the previous imperative Navigator approach which also allowed the existence of pageless routes. To minimize breakdown in existing apps, the pageless mechanism will continue to work. However, from the implementation point of view, there have been some changes to generalize the Navigator logic.

A transition delegate is used to decide how a Routes should enter or exit the screen when its corresponding Page is added or removed from the Navigator list.

Basically, the transition delegate has to make two decisions. The first one is to decide the behaviour of a transition between two routes: the new route can animate in/out or just appear/disappear. Secondly, the transition delegate decides how to order Routes which are added and removed at the same location in the history stack. Developers can both configure the transition delegate behaviour of the Navigator or provide a different transition delegate. It is also possible to choose a specific transition delegate for each update of the Navigator list of Pages. This is useful to implement different transition styles; if no custom delegate is provided, the default one will be used and it will implement transitions in a Material push-like effect.

The Navigator also support a new onPopPage() callback in addition to the Pages list. The callback is usually invoked by the Navigator in response to a Navigator.pop() call. This asks to the receiver of the callback if the given Route - corresponding to a Page - should be popped or not. If the receiver agrees, the Navigator changes the state and triggers the update of the Navigator with a new list of Pages without the popped one. Otherwise, in case of error or false response, the popped Page is not removed from the list and it will be treated as a new Page to show changes in the state according to the error.

Despite the effectiveness of the new routing introduced by Flutter in solving the criticalities of the previous approach, the acceptance by developers has not been great so far. Principal criticisms regard the API complexity and the numerous components needed to implement in a custom context. Developers expected the presence of pre-made solutions for common scenarios offered by an extra abstraction layer which is absent by now. In fact, even in the main article about Navigator 2.0, the author refers to the possibility of using these new Navigator APIs to build an higher-level API package. This is certainly an aspect than could be further explored in the future [9][8].

3.4 Data and backend

In this section, I introduce more technical aspects of the process of building an application. So far, the focus was primarily on Flutter graphics capabilities and the goal was to emphasize the flexibility of Flutter's internal architecture. In this and the next section instead, I analyze what tools Flutter provides for managing information within the application and how to use the functionality of the underlying platform.

3.4.1 State management

In order to simplify state management, Flutter has adopted a clear distinction into two major super classes from which all widgets derive: **stateful** and **stateless** widgets.

Most widgets do not need a mutable state, because they do not have any properties that change over time. Icon, IconButton or Text are examples of stateless widgets. To implement this immutable condition, a widget subclasses the StatelessWidget[5].

A Stateless widget is used when the part of the user interface described in its build method do not depend on dynamic factors: anything the widget needs is the configuration in the widget itself and the BuildContext in which the it is inflated. For this reason, the build method of a stateless widget can be called only once for each widget instance when the app is running. This call happens when the widget is responsible for drawing itself. If the framework needs to redraw a Stateless widget, then it will need to create a new instance of this.

On the other hand, a Stateful widget is dynamic. If a widget can change in response to a user interaction - for example changing its appearance - it needs to subclass the Stateful superclass. Checkbox, Radio, Slider or TextField are examples of Stateful widgets[6].

Implementation is slightly different for stateful widgets with respect to stateless ones. It requires creating two classes, one subclass of StatefulWidget, which defines the widget, and another one subclass of State, which contains the widget state and defines the widget build() method. This separation is in place because a Widget is always an immutable object and Flutter wants to not treat stateless and stateful widgets in a separate way. For this reason, a widget state is stored in a State object, well distinct from the widget appearance.

The state consists of values that can change. Whenever this happens, the State object calls its setState() method, which tells the framework to redraw the widget.

So, let's analyze the code example below. Consider this simple widget with a counter and a button in its view. The counter increments whenever the user taps the button, and then the new value is showed. In this example, the counter is the state for that widget, while other components are just stateless widgets - ElevatedButton, Text and Padding widgets. After the value changed, the widget needs to be rebuilt to show the updated value in the UI. However, the build method is the one of the object extending the State class.

```
1 // This class is the configuration for the state and it holds values
     provided by the parent
2 class MyCounter extends StatefulWidget {
3   @override
4   _CounterState createState() => _CounterState();
```

[5] https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html

[6] https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html

```
}

class _MyCounterState extends State<MyCounter> {
  // local variable counter
  int _counter = 0;

  void _increment() {
    // This method calls setState which tells Flutter that something has changed in this State. As consequence, Flutter rerun the build method below.
    // If _counter is changed without calling setState() nothing will change UI side.
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Padding(
          padding: EdgeInsets.all(16),
          child: Text('Counter value: $_counter')
        ),
        ElevatedButton(
          onPressed: _increment,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

As I said before, having the state separate from the widget permits to treat both stateless and stateful widgets exactly in the same way, without concerning about losing state. Is not necessary to hold an object to preserve its state, because the parent can create a new child instance at any time, without losing the persistent state. Flutter is in charge of finding and reusing existing state when appropriate.

Going deeper into the aspect of state management, it is evident that stateful widgets are suitable for managing a specific type of state, usually called "Ephemeral state". A useful definition of state in an application context is "all data the app needs to build the UI at any given time". Following this sentence, the app state can be separated in two conceptual types; the first one is the ephemeral state and the second one is the shared state.

As I explained in this section, the ephemeral state usually represents the state

in the context of a Widget. This contains the local data actually needed to the widget to build its interface. The counter example perfectly describes this type of local data, where the counter value is useful only to the widget containing it. Other example could be the current page index in a PageView or the current selected tab in a BottomNavigationBar. This state is generally simpler than the shared one, it do not require to use state management techniques, as serialization, and it do not change in complex ways during app execution. For local state, all you need is a StatefulWidget.

On the other hand, the shared state represent the application state, which consists in all the information to share across many parts of the application. Furthermore, shared state also includes data developers want to keep between user sessions. Example of shared state are the user preferences or the login info, but also data about items in a shopping cart and read/unread state of notifications.

For this type of state there is not a single rule on how to manage it. The right implementation of share state management depends on the complexity and nature of the application, the team experience and the current development phase in which the app is. Flutter does not pose any constraint on the approach the developers prefer. They can freely use State and setState() of Stateful Widgets to manage all of the application state. The Flutter team takes this approach in many simple app. It is also possible that, as the application grows in features, a local state might need to be moved to app state and vice-versa.

For that reason, the diagram in Figure 3.5 can help in simple way to summarize which is the right choice between ephemeral and shared state for a given piece of data.

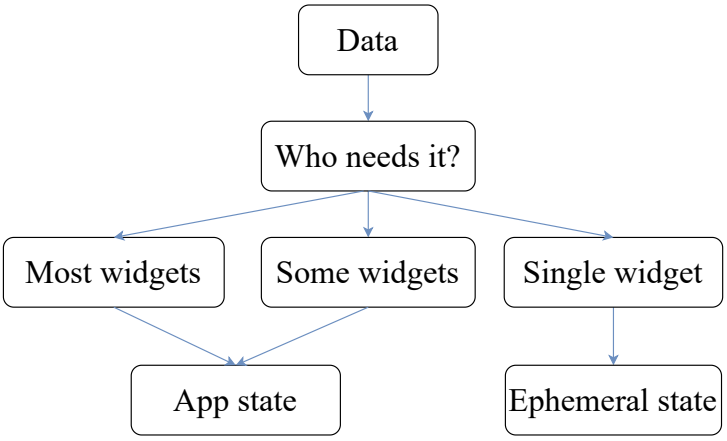

Figure 3.5: Graphical summary of essential questions to answer when deciding how to manage state within a Flutter application[5]

For managing the shared state - or application state - Flutter suggests to use the

Provider package[7]. If there are no strong reasons for using an alternative approach - as Redux - then the Provider package is probably the right choice, because it is easy to understand and use.

In Flutter, thanks to its declarative approach, it makes sense to manage the state above the widgets that access and use it. The main reason for this, for how Flutter is structured, it is hard to imperatively change a widget from outside, for example by calling a method on it. The Provider package is based on three main components:

- **ChangeNotifier**. This is a simple class provided by the Flutter SDK. It in charge of providing change notifications to its listeners. Simplifying, a ChangeNotifier is a form of Observable - from publisher-subscriber pattern. If an object is a ChangeNotifier, other objects can subscribe to its changes.

  In the Provider approach, the ChangeNotifier role is a way to encapsulate the application state. The main method offered by the ChangeNotifier is the notifyListeners() method: this has to be called any time the underlying model changes and these changes need to appear in the UI. ChangeNotifier does not depend on any higher-level classes in Flutter so it is easy to implement and test.

  To actually use it, a class has to extends it. If for example we consider a shopping cart app, the cart model is the one that will extend the ChangeNotifier. The notifyListeners() method will be used when a new item is added to the cart.

- **ChangeNotifierProvider**. This widget provides an instance of a ChangeNotifier to each descendant that needs it; it is part of the Provider package. The ChangeNotifierProvider has to be put above the widgets that use it. It can be declared in each suitable widget higher in the tree. If any widget in the tree needs to access the ChangeNotifier, the provider can be placed in the main method, as you can see in the code from the following example.

```
1 void main() {
2   runApp(
3     // this object provides the definition of a builder that
      creates a new instance of MyModel. ChangeNotifierProvider is a
       smart component and it will not rebuild MyModel unless
      necessary. Furthermore, it automatically calls dispose() on
      the model when the instance is no longer necesary.
4
```

[7] `https://pub.dev/packages/provider`

```
5       ChangeNotifierProvider(
6         create: (context) => MyModel(),
7         child: MyApp(),
8       ),
9     );
10  }
```

- **Consumer**. This is a Widget from the Provider library that offers methods to interact with models provided by the ChangeNotifierProvider. The exposed instance of such a model can be used to interact with the model state. Consumer needs a type - Consumer<MyModel> - because without it the Provider package, which is based on types, does not know what the developer wants. The only other required parameter along with the type for a Consumer widget is the builder. This is a function that is called whenever the notifyListeners() method of the ChangeNotifier is called.

  Compared to a classic Flutter builder, the Consumer builder requires three parameter: the context, which is also present in every build method, an instance of ChangeNotifier, provided from the above, and a child, which is in place for optimization. It is a best practice to put the Consumer widgets as deep in the tree as possible, to avoid to rebuild large portions of the UI for minor changes in the application state.

There is a last component, called Provider.of, that is useful to access data in the shared model, but these data are not needed to show the UI. For example, consider again the cart example and now add a "clear cart" button or a "add item to cart" button. These buttons do not need to display data from the provided model, but they only have to trigger a change in the model. It is just possible to use a Consumer, but that would result in a waste of resources.

By combining all these components together, any data in the application state can be easily managed. Furthermore, as I explain in the next section, this management mechanism is also necessary to manage live data coming from the internal storage or the network.

3.4.2 Data persistence

It is a common need for developers to save simple values, or user preferences, to simplify the operation of the application and improve the user experience. For example, these values can be strings, booleans, integers or small lists. This task can be easily achieved without complication if developers save these data inside the local database of the underlying platform.

Android and iOS provides different solution to perma-save small set of data. Android uses the SharedPreferences class: this represents an XML file/files managed

by the app with a key-value structure and this data could also be shared among different applications. Instead, iOS provides the UserDefaults component to achieve the same objective, with the only difference that this is a simple synchronous database with cache features.

By default, Flutter does not have a built-in mechanism to access this platform specific classes, but there is available an official package made by Flutter team which resolves this problem, called shared_preferences [8]. This is a simple library which allows developers to use SharedPreferences and UserDefaults without worrying about the underlying platform. The package shared_preferences provides methods to asynchronously read and write data using Flutter Future mechanism. These methods are different for different type of data. Methods to write are set[type], so there are setString, setInt, setBool, setDouble and also setStringList and all of them return a boolean value to confirm the writing operation. Instead, for reading the methods are similar but they are composed with get plus the type to read. These methods are available only through an instance of shared_preferences library. Hence, to implement a simple string memorization, the code in following example is enough.

```
1
2  Future<bool> saveStringToDevice(text) async{
3    SharedPreferences sp = await SharedPreferences.getInstance();
4    return sp.setString('string_key', text);
5  }
6
7  Future<String> getStringFromDevice() async{
8    SharedPreferences sp = await SharedPreferences.getInstance();
9    return sp.getString('string_key');
10 }
```

It is important to remember to not overuse shared_preferences database, because it is a system built on top of saving mechanism offered by the platform and there is no guarantee that the data will be stored in the long term. Also, in both iOS and Android, application data is completely loaded into memory at startup, making it mandatory to keep the database size down. Last thing, shared_preferences is not good to store sensitive data, like passwords or token.

To store sensitive data locally, the best solution is to persist them data in a secure storage. iOS provides developers with secure Keychain, which stores small bits of user data in an encrypted database. Keychain items are encrypted using two different AES-256-GCM keys. Apple Keychain is built mainly for string and allows to store passwords, other secrets that the user cares about - as credit

[8] `https://pub.dev/packages/shared_preferences`

card information - and cryptographic keys and certificates the app needs to work with. On the other hand, Android has a similar system called KeyStore, which from the outside works similar to the Apple counterpart, but supports different encryption algorithms. Furthermore, Android KeyStore create a system wide credential mechanism.

As for shared_preferences, Flutter does not have built-in support for secure storage, but this support can be achieved using a third party library called flutter_keychain[9]. This library provides developers with a singleton to write and read asynchronously secure data. It uses secure Keychain in iOS and Keystore in Android - in particular, for Android the library uses the AES encryption. The following code helps to understand how to use it.

```
1
2  Future<bool> saveSensitiveValue(string) async{
3    result = await FlutterKeychain.put(key: "string_key", value: string
       );
4    return result;
5  }
6
7  Future<String> getSensitiveValue() async{
8      result = await FlutterKeychain.get(key: "string_key");
9      return result;
10 }
```

The previous solutions are good to store simple data in an easy way, but they are not a real database. To actually persist data available to many users in necessary to link the mobile application to a real database. Firestore is a common used solution to achieve both data persistence and real-time data updates. Firestore is a NoSQL database built by Google on top of Google Firebase database. A piece of data linked to its remote version in the database is called a LiveData in Android. It can auto update its value when the remote value changes.

Flutter does not exactly have LiveData, but the same result can be achieved through an official package, called cloud_firestore [10], and through a stream widget called StreamBuilder[11]. This is widget is based on a sequence of snapshots representing changes in its state. The build() method of the StreamBuilder is called at the discretion of the Flutter pipeline. This will receive a timing-dependent sequence of snapshots that represent the interaction with the stream.

[9] `https://pub.dev/packages/flutter_keychain`

[10] `https://pub.dev/packages/cloud_firestore`

[11] `https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html`

Configuring Firestore is not trivial and require to write platform-specific configuration code inside Flutter folder for both Android and iOS. Altogether, by combining Firestore and StreamBuilder, it is possible to use "LiveData" within Flutter. Any change made by a user be reflected to all other users in the app.

In conclusion, in the context of persistence, Flutter has no obvious limit. It is possible for developers to obtain the required behaviours by Flutter while maintaining a cross-platform approach. The only platform-specific code concerns the configuration phase.

3.4.3 Internationalization

In a business context, it is essential to provide customers with localized applications in the users' language. In this case, it is therefore necessary to rely on the internationalization of the application. With this, it is possible to substitute the application simple static strings with a key-value mechanism to retrieve saved translations from file, based on the language chosen by the user.

Flutter provides both a built-in mechanism to internazionalize the application and a good documentation on how to set up it for each locale supported by the mobile app.

Flutter has a localization package, called flutter_localizations, for translating its own components. This package has to be specified in the pubspec.yaml file as a dependency. Developers must add also every file used to store the translations. For example, if developers decide to use json files, they will create a file for each locale with .json extension and they will import them under the assets entry in the pubspec.yaml.

Flutter two main widget for building applications both support localization; the first one is the MaterialApp widget and the second one is the CupertinoApp widget. In particular, Flutter localization uses delegates for initialization localization files, so developers has to implement a class in charge of loading the translated strings from file to the memory during execution. The follwing code snippet shows how to add localization to a MaterialApp.

```
1  class LocalizedApp extends StatelessWidget {
2    @override
3    Widget build(BuildContext context) {
4      return MaterialApp(
5        title: "Simple localized app",
6        localizationsDelegates: [
7          GlobalMaterialLocalizations.delegate,
8          GlobalWidgetsLocalizations.delegate,
9          GlobalCupertinoLocalizations.delegate
10       ],
```

```
11        supportedLocales: [
12          Locale('en', 'US'), // english + country code
13          Locale('it', 'IT'), // italian + country code
14        ],
15        home: MyApp(),
16      );
17    }}
```

For the sake of brevity, I do not show in this thesis all the steps necessary for the implementation of localization. My main interest in this case is to show that Flutter has no shortcomings from this point of view. In practice, Flutter offers similar mechanisms to other cross-platform frameworks, such as ReactNative, and native solutions. Furthermore, third party libraries, as easy_localization[12], allow developers to minimize the amount of code they have to write and offer pre-made complete solutions.

3.5 Platform integration

Flutter includes as add-on packages to access all the fundamental APIs to interact with the functionalities of the underlying operating system. These add-on packages are created both by the official team and other developers. The package url_launcher[13] published by flutter.dev team implements many of these native features at a high level and it allows developers to call them with just a few lines of code.

In particular, I show in this section the most important features made available by the url_launcher package. The main component of this library is the launch method: this method takes as main parameter a string argument containing an URL. This URl is not just a web url, but it can be formatted using a number of different URL schemes, where each scheme represents a specific feature of the underlying OS. In general, the availability of a URL scheme depends on the platform where the app is installed. For example, since Flutter also supports the creation of desktop applications, if the phone module is not available on a desktop platform, then the APIs for the phone will not work and their invocation returns an error. As a general rule, an URL scheme is only supported if there there are apps installed on the device that can support them. Another example is the iOS simulator available through Xcode, which does not have a default email or phone apps installed, so some launch scheme will not work on this simulator. Usually, common schemes are supported by both iOS and Android with just a few specific schemes for one

[12] `https://pub.dev/packages/easy_localization`

[13] `https://pub.dev/packages/url_launcher`

platform or the other.

To call the telephone screen, the scheme consists of the keyword tel followed by the telephone number to be called. Similarly, it is also possible to send an SMS with the sms scheme followed by the telephone number. Since the scheme is a URL, it is also possible to define parameters. In the case of sms for example, one can specify the default text for the sms to be sent as a parameter. The following code snippet shows the implementation of these two features.

```
1
2  phone_launch() async {
3      if (Platform.isAndroid) {
4        const scheme = "tel:+39333123123";
5        await launch(scheme);
6      } else if (Platform.isIOS) {
7        const scheme = "tel:0039-333123123";
8        await launch(scheme);
9      }
10 }
11
12 sms_launch() async {
13     if (Platform.isAndroid) {
14       const scheme = "sms:+39333123123?body=Hello%20World";
15       await launch(scheme);
16     } else if (Platform.isIOS) {
17       const scheme = "sms:0039-333123123&body=Hello%World";
18       await launch(scheme);
19     }
20   }
```

It may be seen in the above code example how the launch method is implemented as an asynchronous call to the underlying platform. Furthermore, via code it is possible to know on which operating system the app is installed and then implement different behaviours if needed.

To send an email via the default app of the device the code to implement is the same as for tel and sms, but the scheme is mailto. Anyway, creating and sending an email is a more complex action than a call or a text message. This is because, in addition to the email address, in the mailto scheme it is possible to specify the subject and body of the message, even in html format. For the sake of brevity, I'm not showing the related code snippet. The official documentation recommends using the URI object constructor for more code clarity.

Finally, the url_laucher package allows developers to open web pages with the http or https scheme either by opening the default browser or with an in-app webview. Javascript can be enabled if needed. All these options are parameters supported by the launch method and do not require special attention. The only

useful thing about implementing a url scheme is knowing the default behaviour of the operating system. For example, Android opens a url in the browser, while iOS uses a webview by default.

Of all the native features, two in particular are not included by default in Flutter and are not available through official packages. In particular, url_launcher does not allow access to the device geolocationing and contact list features. However, both of these aspects are available through third-party developer libraries.

To access the geolocation services there are mainly two libraries: the first is geolocator[14] and the second is location[15]. The two libraries are very similar and both offer high-level functionality. It is important to underline that it is always necessary to implement the right controls to obtain access permissions to geolocation. By default, on both Android and iOS, access is restricted to user permissions. Without going into the specifics of the implementation via code, it is useful to know that in both libraries it is possible to specify the accuracy of the geolocation - this affects energy consumption - and to implement a callback when the user's location changes - very useful for showing changes to the UI in real time.

Instead, to use the device contact list devs can use the contacts_service[16] library. This allows to easily access the functions related to the contact list: view the list of contacts - with or without the relative contact image - insert, delete or modify a contact and also open the default phone book of the device if necessary.

The last of the absolutely necessary native features in a real app are the access to both media files from the photo gallery and from the camera. Both can be easily achieved by adding a library created by the Flutter team called image_picker[17]. The code to be written to actually implement these features in an application is very little. In particular, it is sufficient to define what the input source will be and then store the return path to the source. As mentioned above for geolocation, it is essential to correctly manage user permissions for accessing the device memory. I am not showing the code necessary for the implementation here, because in chapter 5 I will deal more specifically with this aspect.

In conclusion to this section it is fair to mention that Flutter offers a flexible system that allows developers to call platform-specific APIs for both Android and iOS. This system is available for all native languages: Kotlin or Java code on Android, Swift or Objective-C code on iOS. The Flutter solution built-in to support platform API does not rely on code translation from Dart to native language, but rather on a flexible message passing style. This style relies over platform channel

[14] `https://pub.dev/packages/geolocator`

[15] `https://pub.dev/packages/location`

[16] `https://pub.dev/packages/contacts_service`

[17] `https://pub.dev/packages/image_picker`

concept: the Flutter portion of the app constantly sends messages to its host, the iOS or Android portion of the app, over this platform channel.

The host listens on this channel and receives any message from the Flutter app. The message has to be strictly defined in a codec because the platform channel support only binary communications. The host platform, after a new message on the channel, can call any number of platform-specific APIs using the native programming language. When the computation is done, the host sends a response back to the Flutter portion of the app. This system works in both directions and in practice does not limit Flutter's flexibility towards the underlying platform.

With platform channel, Flutter can therefore interact with native platform APIs, but actually not directly. This approach is different from Xamarin and NativeScript cross-platform framework, where developers can call native APIs directly using the same language of the framework. That is because of language bindings. For common and simple cases this overhead due to platform channel is irrelevant. However, in more custom scenarios, developers have to write native Swift or Java/Kotlin code to interact with native APIs, using different IDEs, and communicate via platform channels. It is important to understand that developers enter in a low level context and that they will need to write not trivial native code.

Anyway this flexibility is a long-term guarantee that Flutter's cross-platform approach is not limiting, although the actual implementation could be very tricky. This article written by Mikkel Ravn addresses in detail how to use the Flutter platform channels [10].

It may seem like a disadvantage to have always to import an external library to implement native functionalities, and the question arises why Flutter does not provide these capabilities out-of-the-box. Despite this, as I said previously, these libraries offer high-level methods that allow devs to obtain complex functionalities with a few lines of code. It is also common in the native context to import libraries that wrap the functionalities of the underlying platform to simplify implementation and speed up development. For this reason, the Flutter team's choice to keep the framework lean is a winning choice. Developers can clearly decide what to import and what not, relieving the framework of all those features that are certainly useful but not always necessary. In conclusion, the purpose of this section is to demonstrate that Flutter has no evident limits on the use of native features.

3.5.1 Supported platforms

In this short section I show the Android and iOS versions that Flutter supports. The following information refers to the latest version of Flutter available which is 2.2, released in May 2021. The definition of supported platform is clearly written on

the dedicated page of the Flutter documentation [18]. To define a platform supported it is necessary that the Flutter team at Google tests in continuous integration every new commit of the framework. In addition to this, the team also has to run post-commit tests before moving from the master channel to the dev channel.

Flutter supports Android from Android API version 19, which is Android Kitkat 4.4 released on 3 September 2013, while iOS is supported from version 9, released on 16 September 2015. Flutter also supports numerous other desktop and web platforms but they are not the subject of this thesis.

In addition to fully supported platforms, there are best effort platforms, which are supported by the community testing. In general these are old platforms which are no longer widespread, but that Flutter team thinks they can support in the future with ad-hoc testing.

In the end, there are unsupported platforms, which includes very old version of Android and iOS. On these platforms Flutter may work, but neither the development team nor the community directly test or support it.

According to the data collected by Google on the distribution of Android versions, 99.3% of the Android devices in circulation have an Android version greater or equal to the 4.4 one. This means that Flutter supports all of these devices. For iOS, on the other hand, according to Apple's data, devices with a version lower than 9 are just 0.2% of all active iOS devices. This means that practically 100% of Apple devices out there are supported by Flutter. Despite this, it is possible that not all libraries work correctly on older operating systems, even if supported, and that it is sometimes necessary to write specific code for these platforms. For example, permission management in Android has changed dramatically in the latest versions of the operating system, so the code to acquire permission must be different on older platforms.

3.5.2 Flutter wearable support

The Flutter team focused heavily on developing a flexible and versatile cross-platform framework that could be used on different classes of devices and operating systems: Flutter can in fact be used on desktop systems - Windows, MacOS and Linux - mobile systems - Android and iOS - and on browsers too. Despite this, there is only a weak support for wearables, especially supported by the community.

The wear environment is split between stand-alone apps created specifically to be installed on wearables and companion apps which are an extension of a phone app. This second option has actually been used less and less in recent years, at least in the Android context. This because Google has pushed a lot in the direction

[18] `https://flutter.dev/docs/development/tools/sdk/release-notes/supported-platforms`

of stand-alone applications for wearOS. Since the release 2.0 of WearOS, Google wanted devs to think of the watch as a separate device instead of just an extension of the phone.

Instead in the context of Apple, which has its WatchOS operating system for wearables, the situation is more balanced between the two possibilities. In general, the choice between one solution or the other always falls on the question of whether or not the application for the Apple Watch will need to communicate with the phone. If so, Apple recommends creating a companion app, otherwise a stand-alone app.

Unfortunately, Flutter does not offer much in either cases. In particular, with some not trivial changes to the native Android code, it is possible to create a stand-alone application for WearOS using a third-party library called wear [19]. This library offers several possibilities, including support for both rectangular and round screen clocks. However, it is fair to point out that this library wear is not yet finished and not very popular. In the article written by Souvik Biswas[11], he shows a workable solution to create a WearOS application without too much difficulty. From my point of view, the solution is quite cumbersome and not actually ready for a business context.

The situation is even worse for Apple wearable. In fact, Flutter does not officially support WatchOS in any way. In case a developer wants to create a WatchOS extension there is a solution elegantly shown in this article written by Rafael Ferraz [12]. From my point of view, as a Flutter analyzer, it is useful to point out how constantly it is necessary to use the swift code and to use the platform channels that I introduced in the previous section. As in the case of WearOS, I therefore consider this solution not ready for a business context.

It is clear that the Flutter team has not yet invested enough to support the world of wearables. Furthermore, at the time of writing this thesis, there does not seem to be any intention from the Flutter team to work on wearable support anytime soon. This is actually a lack of Flutter, although personally I do not consider it particularly problematic. Indeed, applications for wearables are not often a priority business and many companies have not yet invested to create their application for wearables. This is also because in general users use the smartwatch mainly to access notifications, to quickly manage calls and to use the features related to fitness.

In conclusion, I add in this section that the limitations for wearables are present and accentuated also for other devices that are part of the mobile ecosystem. With Flutter it is not possible to create applications neither for televisions - AndroidTV and TvOS - nor for cars - Android Auto and CarPlay. Again, the situation will not

[19] `https://pub.dev/packages/wear`

change anytime soon.

Chapter 4

Flutter for Enterprise mobile app development

In this chapter I try to answer the question of whether Flutter is suitable for an enterprise environment. I begin by examining the Dart programming language and then move to seeing which tools the Dart environment offers. After that, the focus of the chapter is on Flutter performance. In these section I explain the methodology that I used to evaluate Flutter and which works have helped me in this task. Finally, I show the Flutter Add-to-app functionality, which is designed to simplify the migration of native applications to the Flutter framework.

4.1 Development environment and tools

This section is dedicated to Dart and its development environment. Beginning I analyze the functioning of Dart and the main features it offers, then I deepen the analysis by including the development context in which Dart operates.

4.1.1 Dart language

In this section I explore Dart, the programming language behind Flutter.

Dart is a programming language written by Google developers in 2011. Now it has reached the second stable version and it constantly in development. Dart is a general-purpose programming language - approved as a standard by Ecma (ECMA-408). It is an object-oriented language, based on classes with a C-style syntax. As other high-level modern languages, Dart has a garbage collector to free memory and it supports abstract classes, interfaces, mixins, reified generics and type inference.

Dart goal is to be a client-optimized language created especially to quickly develop apps on any platform. So it is a productive programming language for multi-platform development, which comes with a flexible execution runtime platform.

Languages are defined by their technical envelope — the choices made during development to shape a language. Dart is designed to prioritize both client side development and development time. Client-optimized means that Dart emphasizes and simplifies the development of user interfaces, at the expense of other aspects. For example, in Dart there is no such thing as RDBMS - Relational Database Management System.

A key aspect of Dart to simplify UI development is to use the UI as code. When creating UI, usually developers switch between the UI markup language and the backend language in which they are writing the app and logic. This usually leads to an overhead for developers that decreases productivity. To reduce context switching and focus development, Dart uses the same syntax to define both purely logical components - functions, classes, variables - and the components of the graphical interface. For this reason, Dart has some specific features, as, for example, that const and new keywords are optional, which helps cleaning up the definition of UI widgets without using a single keyword.

The Dart language is strongly statically typed. At compilation time, it uses static type checking to always check that variable values match their static type. This mechanism is also known as sound typing, which means that the program can never get into a state where the evaluated value of an expression do not match the expression static type. However, in Dart type annotations are optional due to type inference. Although the strongly typing system, Dart allows developers to use dynamic type combined with runtime checks in development to speed up the experimentation phase.

From version 2, released in 2019, Dart provides null safety, which means that a value can not be null unless it is specified it can be. This works in the same way of Kotlin null safety system and also the symbol “?” - is the same.

The Dart code is compiled by the DVM - Dart Virtual Machine. The Dart VM works very similar to the JVM but it mainly works in two modes. The first is the AOT mode - Ahead of Time compilation, in which the Dart code is compiled directly into executable code for the underlying execution environment. In the case of an Android and iOS mobile device, compilation will produce native executable code for the ARM platform, in the context of a browser it will produce Javascript code that can be executed by the browser, while for a desktop system it will produce native desktop code for the X86 platform. In the AOT mode Dart VM does not support dynamic features as loading, parsing and compilation of Dart source code in real time. It can only load and execute precompiled machine code. The Dart VM is still needed also for compiled code because it is in charge to provide the

runtime context which includes the garbage collector and various native methods for dart as libraries to function, runtime type information and dynamic method lookup. The AOT mode is used to package and distribute the completed app.

During the development phase, the dart code is interpreted by the DVM using the JIT mode - Just In Time compilation. The DVM is able to compile code to low level very fast and it can dynamically load Dart source, parsing it and compiling it to native machine code on the fly to execute it. In JIT mode, Dart VM offers advanced features, as hot-reload which simplify and speed up development and debugging.

The hot-reload allows developers to immediately run the latest changes made to the source code without having to restart the application or lose the state of the application. Dart VM reloads all the libraries with updated code, updates classes with the new versions of fields and functions and the Flutter framework automatically rebuild, repaint and re-layout the widget tree.

Considering the background common to many developers, Dart has a short learning curve. The typical background of a mobile developer includes languages such as Java, C # and C ++ or similar languages commonly used in software development. This background is enough to start programming immediately in Dart because it requires concepts already present in many other languages. In particular, to program in Dart developers need to have knowledge about variables, built-in types, flow control structures - for loops, while loops, if-else, switch - functions - lexical Scope and Lexical closures - classes and concepts about OOP programming, async programming and Dart keywords.

For a Javascript programmer starting to develop with React Native is probably more intuitive due to the possibility of really reusing all his knowledge. Despite this, Dart does not introduce complicated novelties; indeed, the common impression using Dart is that one already seems to know Dart. This is due to the ability of the language to re-propose concepts already present in many other languages in a simple and intuitive key. The Flutter framework, on the other hand, requires the programmer to master its mechanisms especially for what concerns the functioning of the UI components. In chapter 3 I analyzed Flutter's mechanism for managing 2D space and it is evident how Flutter's approach is different from other solutions such as React Native. But for Dart itself these complications do not exist: to program efficiently in Dart a developer just needs to study the complete and exhaustive documentation which is well explained and full of examples, including interactive ones. Personally, since I had to learn Dart to work on this thesis, I can subjectively confirm the simplicity of Dart. My programming background also includes very high-level easy-to-write languages like Ruby, or modern functional languages like Elixir, and Dart did not represent a stumbling block for me, rather it made learning Flutter the main focus of my work.

In conclusion, for a company investing in Dart is a low risk operation. The

language is robust and open source, the concepts behind Dart are common to many other languages so they easily become reusable knowledge. The ability to compile and run Dart code on virtually any platform - desktop, browser and mobile - can potentially help a company to keep development costs down by reducing the number of people specializing in different programming languages. On the other hand, from a programmer's point of view Dart has great support for all the tools needed to work, it is a language with an intuitive syntax, easy to learn and that promotes productivity. These aspects are sufficient to promote Dart as a good business investment, like other more known languages in the enterprise context.

4.1.2 Editors for Dart and Flutter

In this section I expose the various editors and IDEs that officially support the Dart language. First of all, to use Dart, you need to locally install the Dart runtime available for all desktop operating systems - Windows, MacOS and Linux on your machine. The installation procedure is absolutely simple and does not require any particular configuration. The only restriction is on the versions of the supported operating systems. As Dart is a fairly recent language, it only supports some of the versions of the various desktop OS. In particular, only the Windows 10 version is supported for Windows, while MacOS Mojave 10.14 or higher is required for MacOS. For Linux, however, there is no reference to a specific version, but the official documentation specifies that Dart is fully supported only on stable versions of Ubuntu or Debian. However, the necessary packages are available online for other versions of Linux. Additionally, Dart includes support for running on ARM architecture only on Linux, which should soon be extended to new ARM-based Macs in the second half of 2021.

Installing Dart as a standalone package is really necessary only if you want to use Dart without installing Flutter as well. This is because the Flutter SDK from version 1.21 - released in 2019 - already includes the Dart runtime inside.

Dart gives totally freedom to developers to decide which editor or IDE they prefer to use. The reference IDE for developing on all platforms is Android Studio[1], co-developed by Google and JetBrains. To develop in Flutter, developers need to install the Flutter plugin from the IDE marketplace. The procedure is very simple and with a single click the plugin provides Android Studio with all the features needed to develop applications with Flutter. The main features offered by the IDE are a flexible Gradle-based build mechanism, generation of multiple versioned APKs and template support for Google Services and various device types.

Obviously, to emulate and build an application for iOS it is mandatory to

[1] `https://developer.android.com/studio`

use Android Studio on a Mac computer. Android Studio natively interfaces with Apple's Xcode and no further configuration is required.

As JetBrains is actively involved in the development of Android Studio and the Flutter plugin, this plugin can also be used on other IDEs distributed by Jetbrains. It is therefore possible to develop Flutter applications for example using IntelliJ, which in fact is the second recommended IDE on the official Flutter page.

Similarly, for Visual Studio Code is enough simply to add the officially available Flutter extension to start developing Flutter applications. A Mac computer is always needed to build and emulate iOS. Visual Studio Code is for many developers the main IDE to work, thanks to the quality of the extensions and the general lightness and speed of the program when compared to Android Studio, well known for its heaviness of execution. Are good news to be able to use it without having to configure anything manually.

Finally, among the programs recommended by the official Flutter team, there is also an online editor called DartPad[2]. It is used to develop code on the fly via browser or to share code snippets online. The Flutter documentation itself makes extensive use of interactive examples built on this editor. Being online, no configuration is required and one just need to access the web page to start writing Dart code. Since late 2019, DartPad also supports creating Flutter applications directly via the browser. Obviously DartPad does not completely replace all the features of an IDE like Android Studio, but it is certainly a particularly fast and simple add-on tool that can be handy in many situations.

The official Flutter page also reports the existence of plugins developed by the Dart community for the Emacs and Vim editors and for the Eclipse IDE. It is also possible to use Xcode as an IDE even if you need at least Flutter version 1.15 and Xcode 11.4 version or higher to have an automatic integration of the framework. If not, some manual procedures are needed, well explained in the official documentation[3].

For this thesis I developed all the practical examples of chapter 5 through Android Studio and I was positively impressed by the completeness of the IDE. My previous experience with Android Studio was based on development with Kotlin and I had the initial perception that I would have a penalized experience of using another language. In practice, however, I found that all the main features offered by Flutter are well implemented and easy to use.

In conclusion, it is evident that every developer can find the most suitable program for its work. Also, as all major and well-known IDEs are well supported, each company can decide which editor or IDE is most convenient for its business

[2] `https://dartpad.dev`

[3] `https://flutter.dev/docs/development/ios-project-migration`

without compatibility issues.

4.1.3 Test suite

The success of any mobile application depends primarily on its quality. Customers prefer and advertise via word-of-mouth apps which provide a consistent user experience, with a premium UI and free of bugs.

Quality assurance has the important role to address application's defects before it reaches production. Usually, in every company there is some form of QA as part of the development lifecycle. This is true also when there is not a dedicated QA team but the same developer team is in charge to test the app quality.

The time spent in QA increases every new feature added on-top of the application. Bugs can easily go unnoticed and slip into the user's hand. Automation testing really helps in automating some of the work that the quality assurance team would do manually.

Automated testing is a much more common practice for web applications or desktop programs than for mobile applications. On one side there is certainly the fact that mobile development is relatively young compared to other platforms and not all the steps of classic software development have already reached the qualification of industry standards in the mobile context. On the other hand, classical testing techniques applied to a desktop program or web application are not suitable for exhaustively testing a mobile application.

This mainly because mobile users are on the move rather than in front of a desktop with a fixed location - also portable computer are usually considered stationary during usage. Furthermore, web applications are built for bigger screens where it is less important to make the most of all the space, whereas mobile applications are optimized for smaller screens both to easily access all the various features and to have a graphical interface that is not oppressive but on the contrary clean and pleasant to the eye. The team responsible of QA should consider those and other aspects moving from web testing to mobile testing.

Dart provides testing libraries focus on three kinds of testing which are unit, component and end-to-end - or integration - testing. This are only three type of testing among many kinds that developers are familiar with.

These three test typologies are structured as follows:

- Unit tests is in charge to verify the smallest piece of testable code source. Usually unit testing checks code like a function, a method, or class. Generally unit test suits are the most common and used among all other kinds of tests: to feel safe, developers usually try to achieve very high coverage of the source code by testing all the individual components. This does not guarantee that the system will perform as a whole, but that each of its components will perform as expected;

- Component tests verify that a component behaves as expected. This usually consists of a class or multiple classes. Component testing is a black box testing and involves testing of each object or parts of the software separately. Usually this type of test often requires the usage of mock objects to mimic events, API response, user actions and instantiate child components.

- End-to-end or integration testing verify the flow and the behaviour of two or more components by combining them. An integration test generally runs in a real context, which could be on a real device or an OS emulator for a mobile app, or on a browser for a web app, and consists mainly of two pieces: the source code - or app - itself, and the test code that puts the app through its paces. In addition to testing the correctness of the execution, an integration test often measures also performance and app behaviours on different devices and OS.

Dart introduces two libraries to implement these 3 types of tests. The first book is test[4], which provides a standard way of writing tests. This package can be used to write groups of test using the test annotation. It offers features common to many other language testing library as the @TestOn annotation which is used to restrict tests to run on a specific environment or the @Tag annotation to group tests. A tag also allows to create a custom configuration for some tests. Dart allows to write asynchronous tests in the same way developers would write synchronous tests and this helps testing commonly used Dart asynchronous features. Test configuration in Dart is done through yaml file, creating a dart_test.yaml file. There could be more than one configuration file and only the nearest one is used to configure the current test.

The second most important test library of Dart is the mockito[5] package. This provides a simple way to create and configure mock objects, to use them in fixed scenarios. Mocks are usually used to verify that the system under test interacts with the mock object in expected ways.

Flutter introduces its features by building them on top of what Dart offers and this is also visible in testing. Flutter adds two major libraries to simplify cross-platform application testing. The first one is the flutter_test[6] package which adds several utility methods to test that a widget's behaviour is as expected. It allows to easily invoke the construction of a widget with the necessary parameters and check that the result contains text, images or even whole widgets. For example, take the case of a developer who wants to test a product listing page. The developer

[4] `https://pub.dev/packages/test`

[5] `https://pub.dev/packages/mockito`

[6] `https://api.flutter.dev/flutter/flutter_test/flutter_test-library.html`

mock 3 products and can test with this library that the resulting list contains 3 item product list instances.

The second main package is the flutter_driver[7], which provides complex APIs and methods to test and check Flutter applications that run on real devices or emulators. This library runs the application in a separate process from testing and allows to accurately simulate complex behaviours that an average user would do with the application. In particular, flutter_driver allows developers to run complex flows along the application and simulate both the short or long taps and the user's swipes.

By combining these various libraries it is possible to thoroughly test an app written with Flutter. However, considering that a Flutter application can potentially run on very different systems with equally different input mechanisms, test management is still immature in my opinion. Compared to other cross-platform frameworks however, Flutter has the advantage of relying little on the underlying operating system because it implements everything by itself and this reduces problems of incompatibility and complexity of the test suite. In conclusion, in a business context, testing Flutter code is not trivial, but testing a single code base can be a great incentive to test more and with more quality. The difficulties could arise especially if a TDD - Test Driven Development - policy is applied.

4.1.4 Code Analysis and Linting in Flutter and Dart

At the enterprise level, it is often essential to introduce mechanisms to stimulate developers to uniform the code style. A large company usually has numerous programmers working in parallel both on the same project and on projects that are unrelated to each other. In both cases, it is in the company's interest to promote a cohesive and consistent code style between the various pieces that make up a project and the various projects that are part of the company's assets. The benefits of a uniform style help both developers during programming and the company in the long run, as it will be easier to introduce new staff to already written code and it will be easier to maintain that code. Especially if the developers who created a certain code are no longer present in the company or allocated to another project.

Furthermore, writing uniform code allows the company to boost the quality of the code since it helps to obtain a consistent code base from which to draw conclusions on performance and efficiency.

Over time, many high-level languages have introduced automatic systems to highlight portions of code that violate pre-established rules. Generally, these practice is known as linting. Dart, like many other languages, has libraries that

[7] `https://api.flutter.dev/flutter/flutter_driver/flutter_driver-library.html`

deal with defining a set of rules that the code must respect and highlight the invalid portions of code to be corrected.

Linting primarily promotes two ways to improve code quality and efficiency. The first is through the use of a common syntax. Many high-level languages allow a developer to achieve the same result with multiple code implementations. For example in Dart it is possible to declare a string with single quotes but also with double quotes. The result is the same in both cases but it can be confusing to have some strings with double quotes and some not. This suggests to an inexperienced person that there is a difference between the two cases, when in reality the two approaches are equal. The case of quotes is however simple: the advantage of uniforming the syntax is more evident when applied to the whole set of analogous diversities that can create ambiguity and differences. For example, a developer might have a habit of initializing variables to null, or using camel-case or snake-case indiscriminately, or always declaring the type of the variable even when not needed.

All these cases fall under the Style Rules heading. Defining at project level which style rules you want to have allows the linter to advise developers on how to change the code. It also allows to automate the correction of these errors with a single command since usually the style rules are only aesthetic and not functional. In the long run, the most important achievement for developers is being able to read and change non-their code without running into style differences that increase the chance of errors or misunderstood.

The second approach of linters is to improve code quality through advanced static analysis. These are more complex techniques that include checks on compliance with programming patterns, analysis of complexity metrics, support for multiple safety and security-focused coding standards.

Dart's main linting library is called lint [8] and was created by an independent developer. Although this library has more than 200 rules that can be defined in the configuration file, these rules are all about style rules. There are also other linters but overall they are inferior in quality and capacity to the lint library.

In conclusion, compared to other languages, Dart linters are less mature and less ready for a business context. Other languages, such as Ruby or Javascript, do better by providing deeper and more intelligent static code analysis tools.

4.2 Flutter performances

In this section I aim to evaluate the execution performance of the Flutter framework in relation to the native execution. Considering the assumptions on which Flutter framework is based and having already extensively discussed its architecture in

[8] `https://pub.dev/packages/lint`

chapter 3, I start from the hypothesis that the actual performances of Flutter are analogous to the native execution. The Flutter architectural model does not present obvious weaknesses on the performance side, but rather the whole framework has been designed and developed to maximize efficiency and avoid slowdowns common to other cross-platform frameworks.

In some ways Flutter is much more like Unity than other cross-platform frameworks like React Native. It does not rely on the high-level APIs that the underlying operating system exposes to the outside, but works at a lower level, implementing everything necessary from the ground up. Flutter implements its execution model avoiding costly interactions from a performance point of view. The similarities with Unity are bidirectional: some developers have long used Unity, which is a game engine, to develop non-game applications with excellent results. In fact, there is a library for Unity that simulates Flutter's widget model to build UI elements[9].

Using Unity instead of a framework specifically designed for mobile applications of course has its drawbacks. It's hard to find ready-made solutions online and help from other developers on channels like StackOverflow. There are dozens of utility libraries for Flutter that reduce development time which do not exist for Unity. Furthermore, as a non-gaming project in Unity grows, it is difficult to find other staff already trained on the subject. Unity is a game engine and is great for developing mobile games, not developing mobile applications. However, the fact that some developers have used it confirms that so far the cross-platform performance has not been up to the native.

However, the advantage of using Unity is to rely on a really fast framework on any platform because it is built specifically to maintain high performance even in stressful contexts using a single codebase. Flutter has the same goal and that is why I am assuming that Flutter's performance are good enough to match native ones.

To evaluate performance, it is first necessary to identify what the analysis consists of. There are different types of performance and each type relates to a specific area of interest. In particular, these are sufficient to provide a satisfactory picture of Flutter's overall performance:

- Rendering speed - animation smoothness, frames per second while UI is changed or some UI effects that take place in time;
- Computation speed - the speed of mathematical calculations and memory manipulations for business logic.

To confirm my initial hypothesis on Flutter I will analyze the behaviour of the framework with respect to the native execution in these 2 contexts. However, it

[9] `https://github.com/UnityTech/UIWidgets`

is not enough to evaluate Flutter performance on a small project or a mockup application. This is because a real application is profoundly different from simple and specific cases. Results of such analysis will not be sufficient to draw satisfactory conclusions for an enterprise context.

I therefore decided in this thesis to combine data from different sources to outline a general picture of Flutter's performance. The sources I have used are articles and academic works available online that over the years have compared Flutter against both native execution and other frameworks such as React Native. The goal is therefore twofold: to evaluate both whether the execution speed of Flutter is comparable to the native one and if the Flutter framework, compared to other cross-platform frameworks, has actually an advantage and how great it is.

4.2.1 Rendering speed

Several articles and papers have collected data about the execution of Flutter applications to evaluate rendering speed.

The first work is the article written by Jozef Petro, co-founder and CEO of Subdolabs[13]. Petro's article compares the development of a simple list of 300 elements. Each element is a card made up of an image and a related text. The performance analysis consists first of making this list in parallel between Flutter and React Native and then collecting data on the fps during execution. The data were collected with the performance profiling systems present for both languages as development tools. The analysis procedure consists of a manual routine performed on the emulator which consists of i) starting the app and waiting for the first image, ii) start scrolling as fast as possible up to the end of the list and, iii) change direction of scrolling three times in total.

This test does not involve changes to the UI components, but only the scrolling of a long list which is implemented on both frameworks as a RecyclerView. This means that the elements are loaded lazily as the list scrolls. The test was repeated several times with the emulator restarted and the results are summarized in Figure 4.1.

From the results of the Figure 4.1 it is possible to see the average FPS of Flutter is higher than that of React Native[13]. The Flutter framework from the third second onwards exceeds 30 fps on average and all the time maintains high FPS very close to the ideal 60. React Native on the other hand takes an extra second to stabilize above 30 FPS. Furthermore, at 10, 15 and 20 seconds, the drop in frames due to the change in direction of scrolling is much more evident in the React Native graph. From these results I therefore deduce that, without updates to the UI element, the allocation and deallocation of objects in Flutter is really fast and the results are excellent.

The second work deepens the previous article and is made by the inVerita company[14]. This article analyzes two cases: the first is a list of a thousand items

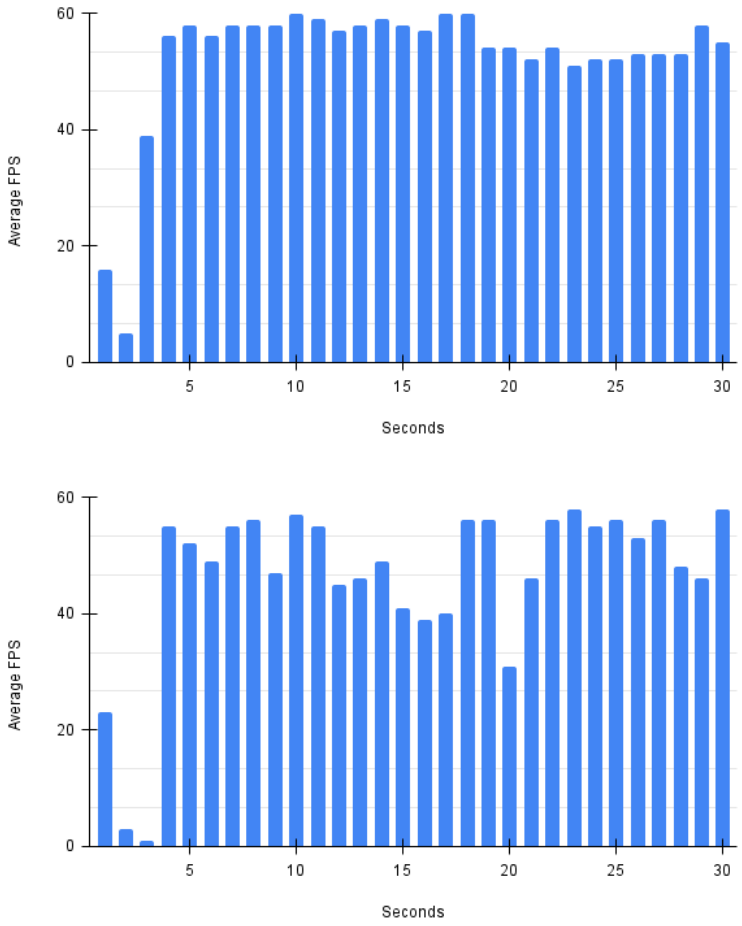

Figure 4.1: the graph shows the FPS values of the same app in Flutter and React Native.

and the second is a screen full of complex animations.

Each element of the first list is made up of a card with inside a static image and an animated image in rotation. Unlike the previous article, here the test is performed with an automatic script and consists in programmatically scrolling the list from the up top to the bottom. The performance comparison includes Flutter, React Native, and the native Android and iOS implementation. The test results represent the average FPS of the test combined with the average CPU usage and the maximum memory occupied by the application. The performance test was performed several times and the results are an average of the various runs.

The results of the first test for Android are summarized in the table Table 4.1, while for iOS in the table Table 4.2.

	Average FPS	CPU%	Max Memory MB
Native (Kotlin)	60	2.4	58
Flutter	60	5.4	114
React Native	58	11.7	139

Table 4.1: Performance for the execution of the same Android application made of a list of a thousand elements in Kotlin, Flutter and React Native.

	Average FPS	CPU%	GPU%	Max Memory MB
Native (Swift)	60	12.72	21.24	154
Flutter	59	33.3	10.75	159
React Native	60	113.13	19.56	220

Table 4.2: Performance for the execution of the same iOS application made of a list of a thousand elements in Swift, Flutter and React Native.

From the Table 4.1 it can be seen that the average FPS is excellent for all three platforms and remains around 60 FPS. However, native execution is more efficient. The Flutter application uses twice the CPU on average and takes up twice as much memory as the native application. The performance of React Native is worse and shows further consumption of hardware resources. For iOS, the situation is slightly different[14]. From the Table 4.2 it is visible that even in this case the FPS average is great for all three applications, but in this test Flutter is closer to native execution. The GPU consumption column is necessary as Swift renders the UI using the GPU. In fact, Swift and Flutter's CPU and GPU usage rates are similar but opposite. However, memory consumption is similar. React native also in this case shows again a greater consumption of resources. The CPU and memory consumption data are the most interesting of the test and are detailed in the graph of Figure 4.2 and in graph of Figure 4.3.

Before drawing conclusions, I also analyze the second test case. This test consists of a screen with two hundred images arranged in a grid. The performance test applies graphic effects to the images at the same time: the two hundred images are treated in groups of three images. In each group the first image is faded, the second scaled and the last rotated. The execution platform is similar to the previous test and also in this case the results are the average of several executions.

The results of the second test for Android are summarized in Table 4.3, while for iOS are in Table 4.4

The results of Table 4.3 clearly show the advantages of native execution in the Android context. The application in Kotlin is the only one that manages to maintain an excellent FPS average, while both the Flutter and React Native applications struggle to reach an acceptable FPS value, remaining below an average

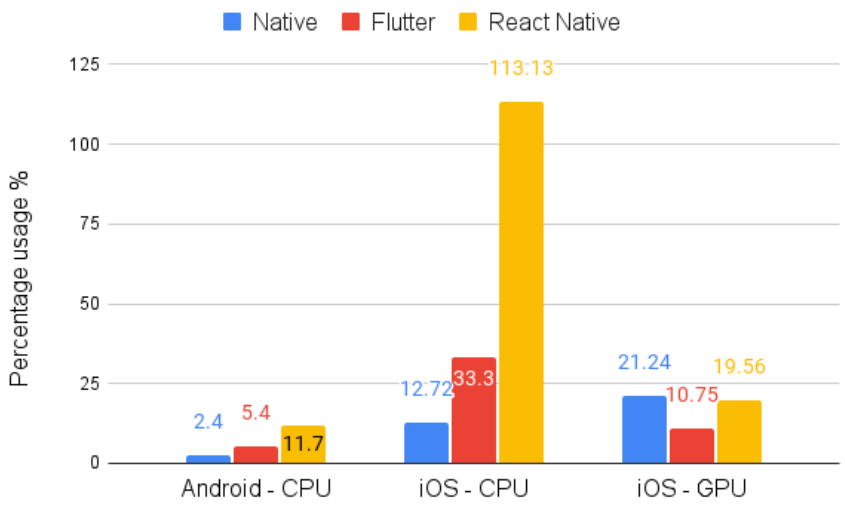

Figure 4.2: CPU consumption comparison between the same application made by a list of thousand elements written in Native, Flutter and React Native

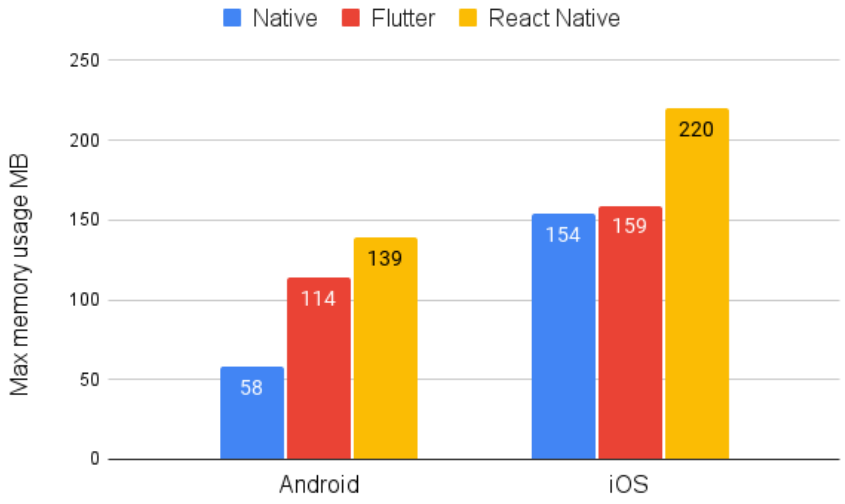

Figure 4.3: Max memory usage comparison between the same application made by a list of thousand elements written in Native, Flutter and React Native

of 20 FPS. Is also visible in this test that Flutter uses on average about double the CPU and double the memory than a native application in Kotlin, without however achieving an acceptable result. The situation is even worse for React Native[14].

In iOS the situation is different and more balanced. As can be seen from

	Average FPS	CPU%	Max Memory MB
Native (Kotlin)	58	6.53	80
Flutter	19	10.28	168
React Native	18	12.5	424

Table 4.3: Performance for the execution of the same Android application made of a grid of 200 animated images in Kotlin, Flutter and React Native.

	Average FPS	CPU%	GPU%	Max Memory MB
Native (Swift)	59	61	48.28	158
Flutter	59	69	81.91	191
React Native	59	118.6	19.8	220

Table 4.4: Performance for the execution of the same iOS application made of a grid of 200 animated images in Swift, Flutter and React Native.

Table 4.4, the three applications - native, Flutter and React Native - manage all to reach 60 FPS. However, the consumption of hardware resources remains in favor of the native solution. Swift's advantage over Flutter is less pronounced than Kotlin's advantage, and especially on the memory side there is a substantial draw. As with the previous test, Flutter's performance for the CPU has to be compared to the GPU performance for Swift due to the different execution model.

It is important to underline that the performance of React Native appears to be excessively negative compared to the native execution. The code of the React Native applications has been implemented using only javascript code which is not suitable for excessively expensive computational tasks. More on this topic in the subsection 4.2.2. React Native allows you to easily integrate native code or use natively implemented libraries for specific tasks. In this thesis, however, the interest is not to analyze the performances obtainable by optimizing React Native. The compiled approach chosen by Flutter easily allows to improve the performances obtainable without using native code, external libraries or complex optimizations compared to the React Native approach.

The data of this test are further analyzed in the following graphs: in Figure 4.4 for FPS average comparison, in Figure 4.5 for CPU usage comparison and in Figure 4.6 for memory comparison.

Another interesting analysis is that made by Matilda Olsson in her thesis "A Comparison of Performance and Looks Between Flutter and Native Applications"[15]. To collect the data, Olsson created two very simple applications for Android and iOS using native programming and the Flutter framework. The applications include a twenty-item list and a bottom navigation bar. The test is performed manually and consists in scrolling the list of 20 elements and navigating from the list screen

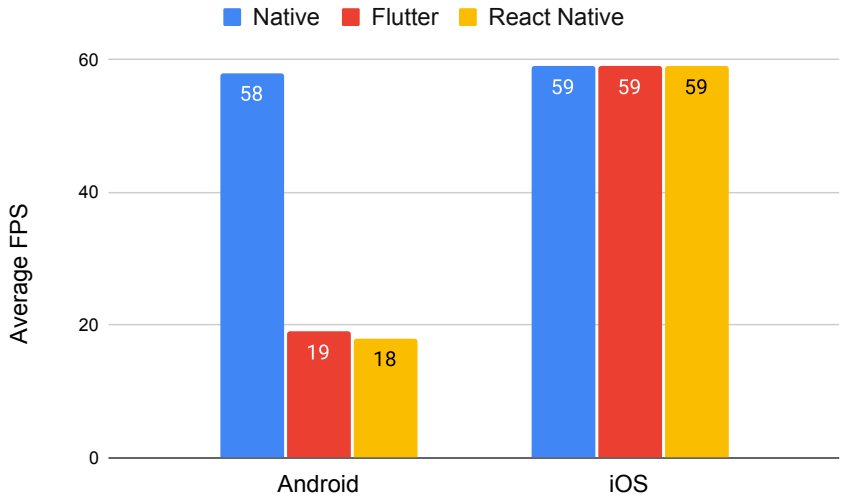

Figure 4.4: Average FPS comparison between the same Android and iOS application made by a grid of 200 animated images written in Native, Flutter and React Native

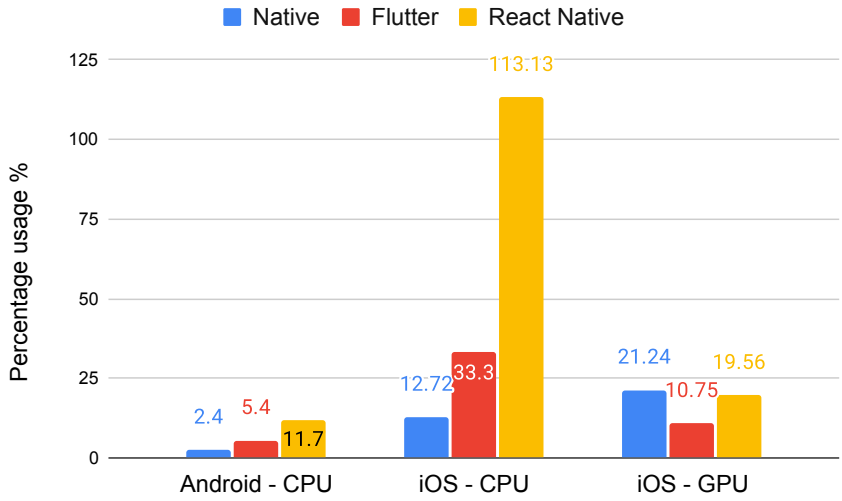

Figure 4.5: CPU consumption comparison between the same Android and iOS application made by a grid of 200 animated images written in Native, Flutter and React Native

to the home and vice versa.

The results of this work are shown in Table 4.5 and further shown in the

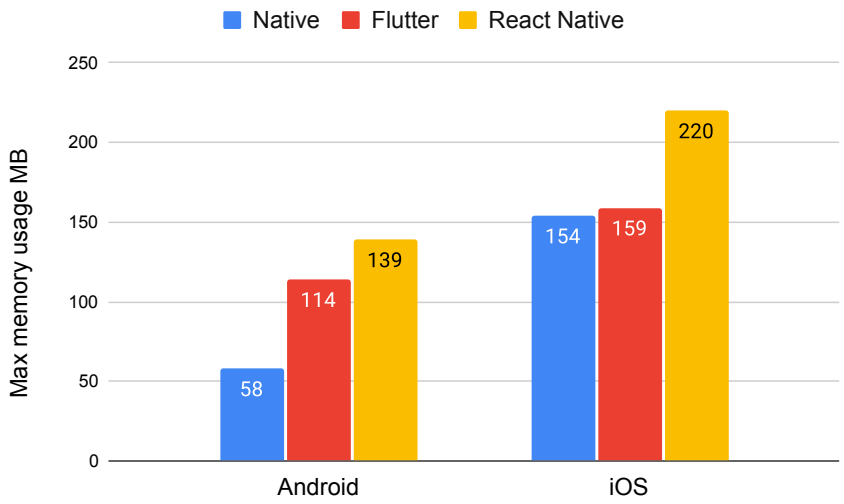

Figure 4.6: Max memory usage comparison between the same Android and iOS application made by a grid of 200 animated images written in Native, Flutter and React Native

Figure 4.7.

	CPU max %	CPU min %	CPU average %
Native - Kotlin	34.6	1	11.7
Flutter - Android	32.3	1	13.2
Native - Swift	30.9	4.8	11
Flutter - iOS	33.9	6.2	11.8

Table 4.5: Table with CPU usage comparison obtained with several iterations on Matilda Olsson's test application made of a simple list page, a Home page and a navigation bar.

The application developed by Olsson is very basic. There are no long lists, changing widgets or complex animations. I integrated her work because the data clearly shows that in this simple context Flutter's performance is absolutely analogous to native performance on both platforms[15]. This is in contrast to the results obtained from previous articles and shows that Flutter's scalability is not yet at the level of native scalability.

Going deeper into Flutter's scalability discourse, the next article I include in this thesis analyzes this aspect. Andrea Bizzotto has made an article in which he analyzes the creation of a stopwatch app in Flutter[16]. In particular, the application consists of a text, which represents the current time elapsed from the

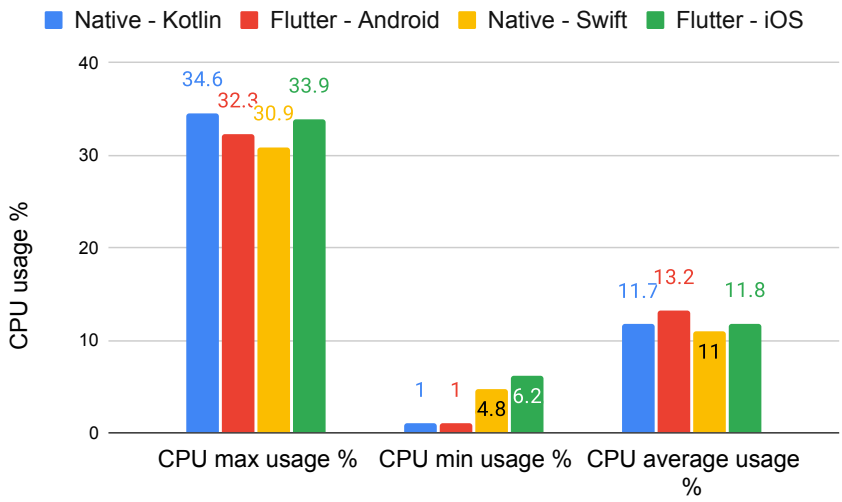

Figure 4.7: Graph with CPU usage comparison obtained with several iterations on Matilda Olsson's test application made of a simple list page, a Home page and a navigation bar

start of the count, and two buttons: one to start the stopwatch and one to restart the count.

The problem with scalability I expressed concern the application state management which Bizzotto analyses. Flutter's declarative approach dictates that the text might change in response to the change of the internal state of the application, which in this case represents the time passed so far. The stopwatch text is made up of minutes, seconds and tenths of a second. The latter are managed as hundredths of a second via software to improve accuracy. This implies that the internal state of the application changes every 30ms and the interface is refreshed at each change.

Proper state management in this case is crucial. Every 30ms, the widget in charge of state management will be redesigned, along with all its children. In the first version of the application, Bizzotto enclosed the state management in the outermost widget within the Scaffold widget. In the second iteration he moved the management of the state to a custom widget that contains within it the text of the stopwatch. Finally, in the third version, the application state has been further divided between a new custom widget for the exclusive management of hundredths of a second and the previous custom widget, now only responsible for managing seconds and minutes.

In Figure 4.8 the evolution of state management is shown.

In his article, Bizzotto analyzed the CPU and memory consumption of the stopwatch application written in Flutter in each of the 3 steps, and then compared

the data with the Apple stopwatch application inside iOS. The results are shown in the graphs of Figure 4.9.

The results of Bizzotto analysis show that without a thoughtful management of the state Flutter efficiency suffers. The first version of the Flutter application consumes up to four times more CPU and memory than the native iOS application. The situation improves by optimizing the number of widgets that need to be redrawn at each state change. In the intermediate version the application, Flutter application consumes about three times the resources of the native application, while in the most optimized case it consumes only twice as much. The difference in consumed resources is also evident in the number of allocated threads. The Apple stopwatch application allocates a single thread while the Flutter application allocates 3 threads in total: one for the event loop, one for I/O operations and another one for UI refresh[16].

From the Bizzotto tests it emerges that the declarative approach consumes more resources in the presence of frequent and continuous refreshes of the UI. The comparison with the stopwatch application is deliberately disadvantageous for Flutter, because the imperative approach allows to change the text of the stopwatch with a simple assignment. The results also highlights the importance of code quality in Flutter state management. It is important to note which UI components must be updated at each frame and limit the interface update to as few widgets as possible. This is a disadvantage of Flutter declarative approach and needs attention from developers.

4.2.2 Computation speed

In this section I quote the article published by two members of the development team of the company inVerita. The two employees are Ihor Demedyuk and Nazar Tsybulskyi who carried out an interesting experiment in March 2020[17]. Their work consists in a comparison between Flutter, React Native and the native execution for two mathematical algorithms very expensive in terms of hardware resources.

The first is Borwein's algorithm used to calculate $1/\pi$. This algorithm is known to be cpu-intensive, meaning that the main limit to the execution time of the algorithm is the maximum computational power that the CPU can offer.

The second instead is the Gauss–Legendre algorithm. This is used to calculate the decimal digits of π. While this algorithm is CPU intensive to compute, it is also known to be a memory-intensive algorithm. In fact, the availability and speed of the memory limit the execution time of the algorithm.

It is essential to point out that these operations do not make much sense when performed on the hardware of a phone. This part of the analysis is very specific and distant from a real use case. The goal is to stress the hardware components and observe any Flutter limitations from the results.

The experiment consists of performing these mathematical operations repeatedly and then calculating an average of the results. In particular, the Borwein algorithm has been executed a few hundred times, while the Gauss–Legendre algorithm has calculated π one hundred times with an accuracy of 10 million decimal digits. The hardware platform is deliberately simple for both operating systems: for Android a Xiaomi Redmi Note 5 with Android 9 was used, while for iOS an iPhone 6s with iOS 13 was used. Finally, in the experiment the algorithms were implemented for Android in Java, Kotlin and Dart, while in iOS in Objective-C, Swift and Dart.

The results of the experiment are described for Android in the graphs of Figure 4.10 - Borwein algorithm - and Figure 4.11 - Gauss–Legendre algorithm. While for iOS the results are in the graphs of Figure 4.12 - Borwein algorithm - and Figure 4.13 - Gauss–Legendre algorithm.

The InVerita data results show similar Flutter performance to the results of previous articles[17]. Flutter on both platforms achieved similar outcome to native but overall lower performance. On Android, Borwein Flutter's algorithm test takes twice as long as it does native Java and Kotlin execution. In the test with the Gauss-Legendre algorithm the differences are negligible and the time taken by Flutter is similar to native languages.

The iOS results show a more marked difference in testing with Borwein's algorithm[17]. In this case, Flutter's execution time is five times greater than Swift and even ten times greater than the same code in Objective-C. On the other hand, in the Gauss-Legendre test the time taken by Flutter is slightly worse than the time obtained with the code in Objective-C, but at the same time better than the time performed by Swift.

The results of React Native are not important for this thesis but I have included them in the graphs anyway to offer a further reference. I also specify that no developer would implement these mathematical algorithms in Javascript, and that the included results of React Native have a single purpose: to actually show the computational weight of the Javascript bridge that has to serialize and deserialize communications with the underlying platform.

From this last test and the previous tests, the results are consistent in evaluating the performance of Flutter below the native one, except in specific contexts. On average, the execution of Flutter applications is between two and three times slower than native execution. However, there are no obvious drawbacks to Flutter's architecture and the Dart programming language. The AOT compilation to platform specific code really shows the advantages of running the code natively without any interpretation steps. In conclusion, Flutter's overall performance is not the same as its native performance, as previously assumed. Conversely, the data shows that Flutter results are superior in any context when compared to React Native.

Flutter is a young technology and it has surely room for improvement. We must

also considered that both Android and iOS native programming have been around for more time and they have been incredibly optimized over the years to better cooperate with the hardware. The Flutter framework starts well and can only improve from this point on.

4.3 Development time

In this section, I analyze data from multiple sources that provide feedback on application development times in Flutter. This section does not include my programming experience which is reported in the section 6.2.

The first source is Corey Sprague and Larry McKenzie's article "eBay Motors: Accelerating With Flutter" published in September 2020 for the Ebay company[18].

The article describes their development experience in their internal Ebay team in charge of creating a mobile app for enthusiast car trading. The development team was given freedom in choosing the technology. When the works began, in 2018, Flutter 1.0 had been released few months before and the team's choice fell precisely on this technology[18].

The team experience developing with Flutter results to be very positive. They write that using Flutter allowed them to solve interesting problems in less than the estimated time. They describe that the development experience in Flutter was much more enjoyable than the native programming and it was possible to quickly create a very consistent user experience between iOS and Android[18]. In addition, they also speak positively of the application testing done through automated tests. To their surprise, Flutter's testing experience was among the best ever and they achieved very high code coverage near to 100%[18].

From the team's work, Sprague and McKenzie have come up with two important metrics. The first is the amount of shared code they have managed to obtain; the second is a survey done by developers about their feeling on Flutter development time.

The first metric summarizes a year of development in Flutter by highlighting the amount of code shared between Android and iOS. The results are shown in the Table 4.6 and further shown in the graph of the Figure 4.14.

Dart Code (∼220 000 lines)	Android - Kotlin	iOS - Swift	Other
98.30%	0.35%	0.25%	1%

Table 4.6: The table shows the distribution of programming languages in the Ebay Motors team project. The code in Dart is shared between Android and iOS applications.

The results of the work of the Ebay team show an incredible value of code shared between the two platforms. Overall, the Dart code makes up 98.3% of the total code. The native code represents just 0.6% of the total code and has been used only for platform specific configuration and for icon and splash screen definition[18].

The second metric collected by the Ebay team concerns a survey carried out among developers regarding the programming speed in Flutter. The results are shown in the Figure 4.15 graph.

The survey results show that the vast majority of developers think that developing in Flutter has increased their productivity. The 70% of developers say that development in Flutter is more than twice as fast as native programming. The other interesting fact is that no developer in this sample has expressed the opinion that developing in Flutter is slower.

In addition to the article by the Ebay team, I decided to integrate the data of two videos published on Youtube respectively by the user Gabriel Basilio Brito[19] and by the Smart Code App channel[20]. These two videos are conceptually similar but were made 3 years apart from each other. The first was released in December 2018 and the second in January 2021. The videos show some time metrics in common actions in software development by comparing the time needed by Flutter and React Native to perform these action. The results of the Basilio Brito video are shown in Table 4.7, while the results of the Smart Code App video are in Table 4.8

	Flutter	React Native
Create new Project	12 sec	1 min 40 sec
First Build and run - Android	23 sec	39.4 sec
Build relaese APK	41 sec	1 min 30 sec
APK size	4 MB	7 MB

Table 4.7: Performance comparison for the same development tasks between a Flutter app and a React native app made by Gabriel Basilio Brito[19]

The data from Table 4.7 and Table 4.8 show that on average Flutter takes less time to perform common software development tasks. On average, to create a new application, React Native needs ten times less it takes for Flutter. On the other hand, in the case of the first app build, in Android Flutter takes about half the time, while in iOS it takes about five times less. The time required in Flutter for hot reload is also two to four times less than that required in React Native. Finally, it is interesting to note that Flutter's debug build weight is higher than its React Native counterpart, but the final APK weight is lower for Flutter[19][20].

In conclusion, the data in this section show Flutter efficiency in optimizing development times. This is definitely an advantage for developers who can be more productive using Flutter than other technologies.

	Flutter	React Native
Create new Project	10 sec	1 min 52 seconds
First Build and run - Android	14 sec	21 sec
First Build and run - iOS	32 sec	2 min 46 sec
Hot Reload	<1 sec	0.5 to 2 sec
Build size DEBUG - Android	39.2 MB	37.1 MB
Build size DEBUG - iOS	97 MB	66 MB
PC CPU usage	17% - Dart	55% - Node

Table 4.8: Performance comparison for the same development tasks between a Flutter app and a React native app performed by the channel Smart Code App[20]

4.4 Migration to Flutter

Develop a new mobile apps using the Flutter framwerok is a good and modern choice for a project that starts from scratch. However for those enterprises that already run native apps an immediate migration to Flutter is not trivial. The problem is both monetary - how much would the migration costs - and productivity - how long does it take to train developers. Flutter team have developed a solution called Add-to-app to simplify the migration to Flutter without having to immediately give up all native code. The Add-to-app goal is both to simplify the migration to Flutter for companies that have already decided to do so and to entice new companies to try Flutter in parallel with native development.

Flutter has introduced from version 1.12 the possibility to add the Flutter framework to existing native applications. In the beginning the system was limited and did not allow adding any Flutter library and it was possible to have only one Flutter screen. In June 2021, the Flutter framework reached stable version 2.2 and many strides were made with Flutter's Add-to-app functionality.

Add-to-app now allows you to add the Flutter framework to a native application that will be able to invoke the Flutter main via native code. All Dart code will work similarly to a stand-alone Flutter app, and the same goes for communicating with the underlying operating system. Furthermore, it is possible to integrate third-party libraries.

The main limitation of this solution is that Flutter must include the Dart runtime, the Skia canvas and all the necessary components in order to work. This means that a native app that adds Flutter will basically have two separate systems inside it, both of which are required. Adding Flutter therefore increases the size of a native app.

On the other hand, the advantages are obvious. A company may decide from a certain point onwards to start developing new screens using only Flutter. The

other parts of the application will continue to work as before, while the Flutter code can retrieve the data from the native backend and thus focus solely on UI development. If the migration process goes well, a company can then decide to go so far as to completely eliminate native code and build a stand-alone Flutter app. Otherwise, if Flutter's approach is not suitable for business purposes, you can delete the Flutter library and related screens added to the native application.

The migration period can easily be more or less long depending on the knowledge and size of the development team and the complexity of the native application. As I showed in chapter 3, developing in Flutter generally requires fewer and simpler lines of code. During the migration, the development team can get in touch with the features and mechanisms of Flutter without immediately having to develop all the knowledge necessary to build an application from scratch.

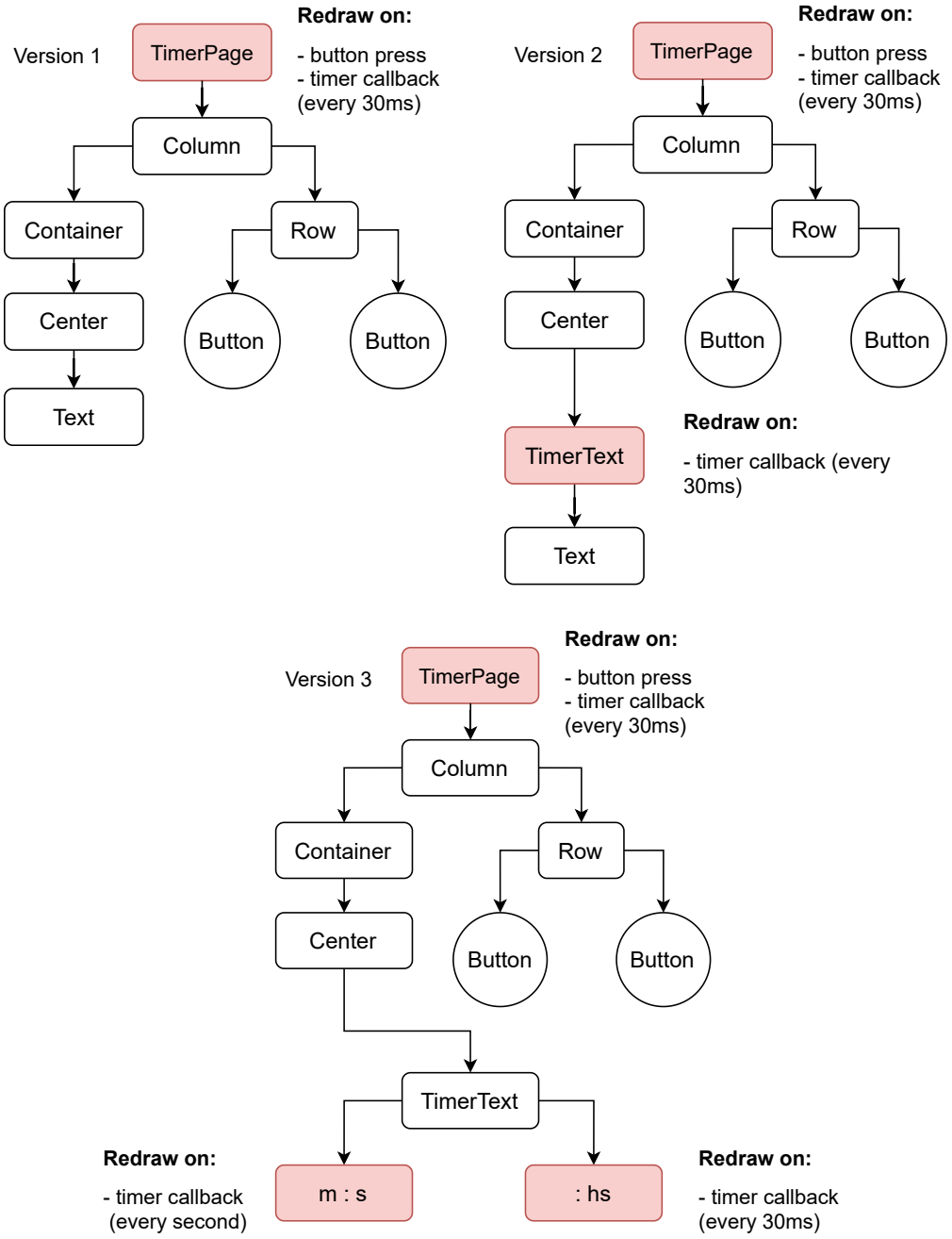

Figure 4.8: Widget tree evolution of the stopwatch application through 3 steps. The red boxes indicate stateful widgets, the white ones mean stateless widgets.

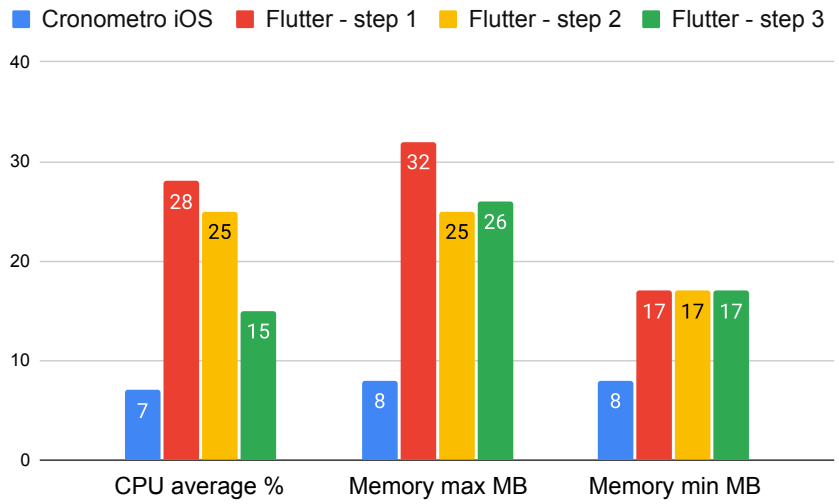

Figure 4.9: Performance comparison between the iOS stopwatch and the Bizzotto Flutter stopwatch application with each of the evolution steps of the widget tree[16]

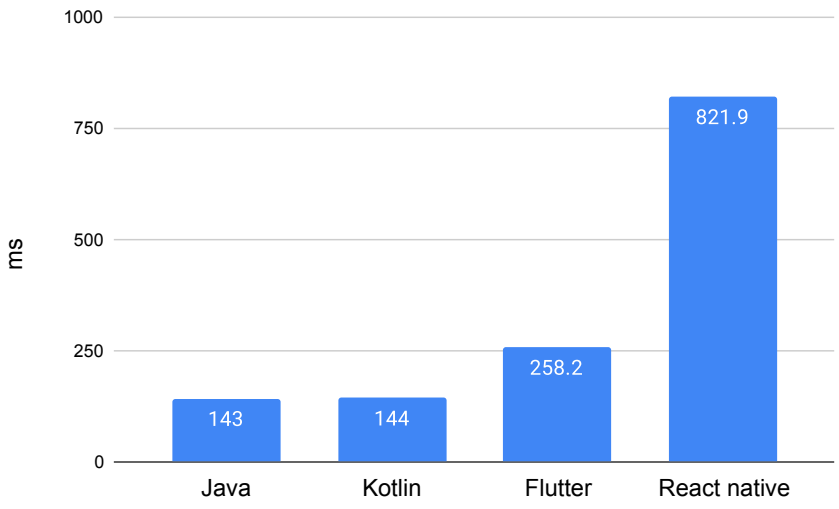

Figure 4.10: Performance for the execution of the Borwein algorithm in Android between Java, Kotlin, Flutter and React Native implementations.

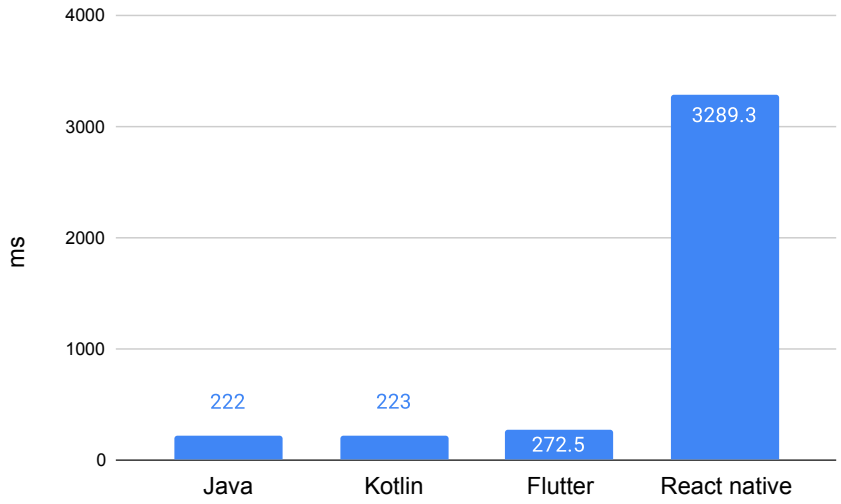

Figure 4.11: Performance for the execution of the Gauss-Legendre algorithm in Android between Java, Kotlin, Flutter and React Native implementations.

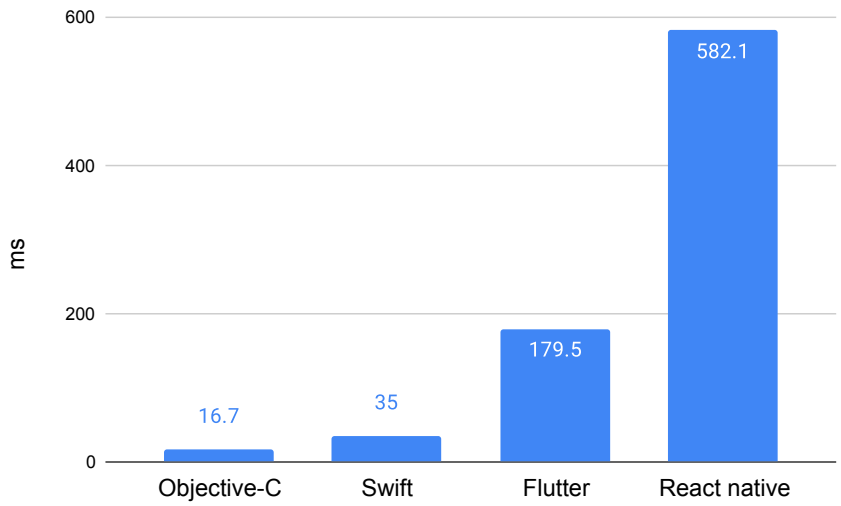

Figure 4.12: Performance for the execution of the Borwein algorithm in iOS between Objective-C, Swift, Flutter and React Native implementations.

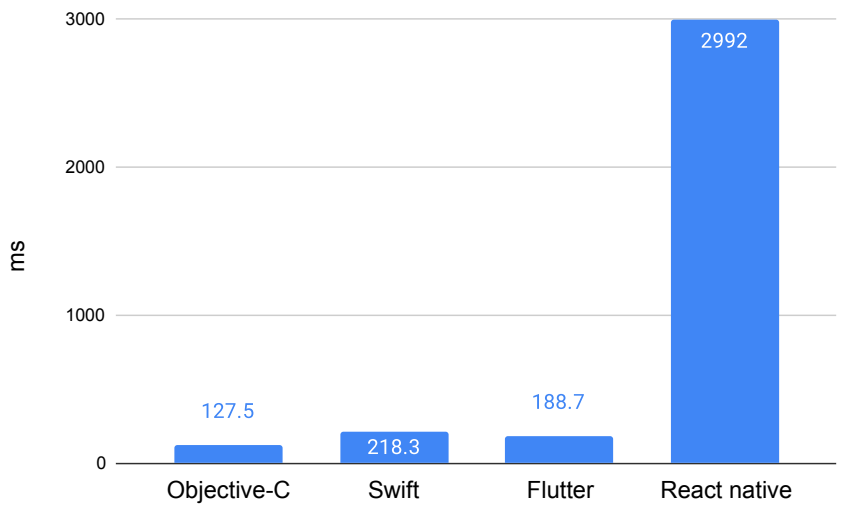

Figure 4.13: Performance for the execution of the Gauss-Legendre algorithm in iOS between Objective-C, Swift, Flutter and React Native implementations.

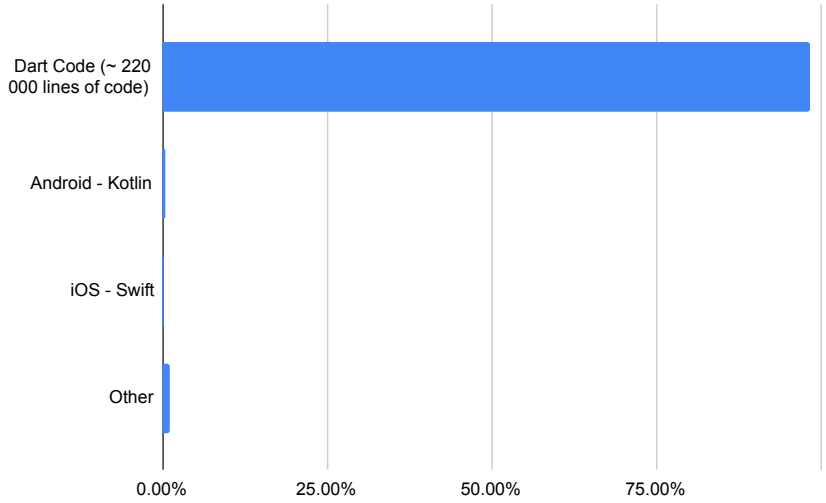

Figure 4.14: The graph shows the distribution of programming languages in the Ebay Motors team project. The code in Dart is shared between Android and iOS applications.

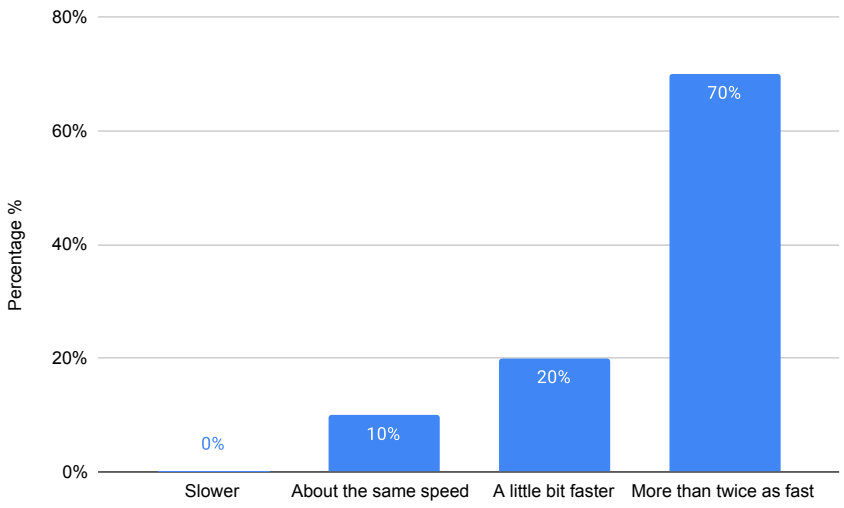

Figure 4.15: The graph shows the results of the Ebay Motors internal team survey on how fast programming was in Flutter compared to Native programming.

Chapter 5

Implementation of a real case scenario

In this chapter I analyze two real use cases to practice some of the key aspects of Flutter that I showed in chapter 3. The first case concerns the construction of a section of an application in charge to show the user's personal information. The second case instead consists of a screen for personal financial management. In both cases the focus is on programming, but with different objectives. In the first case - section 5.1 - the goal is to build a custom layout using the widgets provided by Flutter, while in the second use case - section 5.2 - the goal is to generate some graphs to show financial data using native Flutter or a library available for Flutter.

5.1 Simple case: user personal information section

In this section, I describe the process of programming an interface in Flutter. The IrisCube Reply company has provided me with mockups of their application for a bank company concerning the profile section of a user. These mockups were made by the product and design team and contain all the features that these teams have considered important to include in a profile section.

I started from these mockups to desing my app and I wanted as closely as possible to create the same interface in Flutter as the one in mockups.

To build the layout I started by building the skeleton for internal navigation. I used the `bottom_navigation_bar` field of the `scaffold` widget to instantiate a widget `BottomNavigationBar`. This widget works in conjunction with the `body` field of the Scaffold widget. Body field receives a list of pages and the buttons in the bottom navigation bar set the page to display. The buttons in the bar below

are all `BottomNavigationBar` widgets. The order and style of the buttons depends on the customization you apply and I used it to recreate the bottom navigation bar of the mockup. Finally, the `onTap` field wants a function, which in my case only takes care of updating the index of the widget list of the body field so that another page is shown.

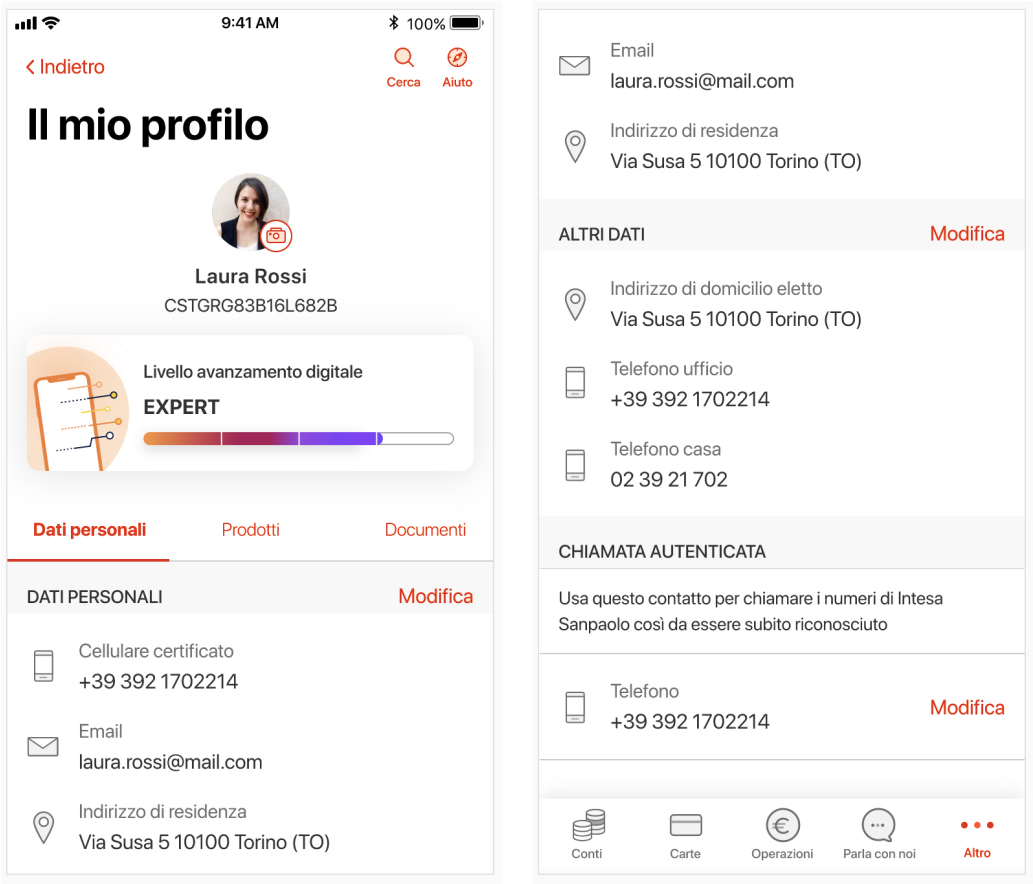

Figure 5.1: The figure represents the mockup of the user page created by the product team. This screen contains all the personal information that can be used by the user. The central card represents the experience that the user has achieved using the application.

The profile page consists of an header and a tab section below. The header contains the user's profile photo together with the name and tax code. Immediately below there are info about user experience with the digital products. The tab section it is divided into three pages that include personal data, activated products and user documents.

The header is built with a Column Widget which has as `children` three widgets.

The first is the round profile photo with the photo change button which is built with a Stack widget. This widget allows to graphically overlay multiple widgets. In my case there is a CircleAvatar widget and a RawButton widget. The first is a native Flutter widget that allows in a very simple way to have a round profile photo. On the other hand, I had to create by hand the button to equal the mockup because the pre-made buttons were not customizable enough. Anyway, it was still easy to realize the button because the documentation is clear and exhaustive. The code for these two widgets is shown in Listing 5.1. At the company level, it is advisable to define a custom class that allows developers to use the customized button in several parts of the code without redefining it.

Listing 5.1: Header section - circle avatar and modal button to trigger the avatar photo change.

```
// CircleAvatar widget wants only the background image - AssetImage widget within the resource path - and the image radius
CircleAvatar(
    radius: 50,
    backgroundImage: AssetImage(_image.path),
)
// The raw button widget must be configured in both graphical aspect and behaviour. It is totally graphically customizable but many fields have to be defined.
child: RawMaterialButton(
    onPressed: () async {
        await _dialogCall(context);
    },
    elevation: 2.0,
    fillColor: Colors.white,
    child: Icon(
        Icons.camera_alt,
        size: 20.0,
        color: Colors.green,
    ),
    padding: EdgeInsets.all(8),
    shape: CircleBorder(side: BorderSide(width: 2, color: Colors.green)),
    materialTapTargetSize: MaterialTapTargetSize.shrinkWrap,
    constraints: BoxConstraints(minWidth: 0)
)
```

In the header there is also a custom card that represents the user's digital experience. The product team inserted it to entice the customer to explore the products and security features of the application. The biggest technical difficulty of this widget is only the experience bar. I made it with a list of transparent but edged containers on a bar made with a `FractionallySizedBox` widget. The length and colors of the bar depend on the user experience. Colors are selected from four

possible palettes defined by the product. The FractionallySizedBox code is visible in Listing 5.2.

Listing 5.2: This FractionallySizedBox width depends on user experience normalized between 0 and 1. The linear gradient receives a colors array.

```
FractionallySizedBox(
    alignment: Alignment.centerLeft,
    widthFactor: _normalizedUserExperience,
    child: Container(
        height: 20,
        decoration: BoxDecoration(
            borderRadius: BorderRadius.circular(5),
            gradient: LinearGradient(
                begin: Alignment.centerLeft,
                end: Alignment.centerRight,
                colors: _colorArray
            ),
        ),
    ),
)
```

The tab section is built with native Flutter components, but has been deeply customized by me. In Flutter it is not trivial to insert a TabView inside other widgets using the `DefaultTabController`. To do it, I need to decompose the TabView into the `TabBar` and the related views. The whole page is therefore structured as a Column widget composed of the ProfileHeader, the TabBar and the views of the TabBar. The TabBar requires a custom TabController which, however, can be easily created from the basic TabController. The most important thing is to define the method to update the body of the TabBar. This method updates the index after the user click or swipe on the interface. The views of the TabBar are very simple and are all defined as a list of items, where each item is composed of a flat card with icon, name of the field and the respective value. Each section can be modified with the appropriate button.

To complete the layout and have all the necessary space inside each view of the external BottomNavigationBar, the layout of each page is contained entirely within a `CustomScrollView`. This widget supports the `Sliver` field which requires a `SliverAppBar` and a `SliverList`. The SliverList wants as input a list of widgets as children. This system allows me to simultaneously manage the behaviour of the screen scroll and the behaviour of the AppBar as I prefer. I automatically determine the height of the list by delegating the decision to the children of the list.

Overall, the most difficult aspect in creating the layout was automatically defining the height of the screen. Being made up of columns and lists, there is in principle no widget that sets a maximum height. The initial solution was only one:

manually set the overall height. But I wanted to deduce the height automatically from the number of components on the screen. To solve this problem it was enough to use the properties of the Container widget, which is built by default to have the minimum size equal to the space needed by its child. Each of the nested elements of the TabBar was then enclosed in a Container and in this way the List widget is able to deduce the necessary space from its children, to uniquely determine the overall height and finally to communicate it to the external CustomScrollView. Using a CustomScrollView allowed me to keep the bottom navigation bar always visible by scrolling within the view. Instead the AppBar is fixed at the top of the internal view and disappears by scrolling. The code of the layout can be consulted in Listing 5.3. The result of this work is visible in Figure 5.2. As you can see from this image, the result is very close to the mockup. Although I had no previous Flutter experience, it was easy to manipulate the framework's native widgets. The UI as code concept is well expressed. It is intuitive to program the UI thanks to the expressiveness of the native widgets. These widgets allow even with minimal knowledge to build advanced interfaces.

Listing 5.3: The code in the figure represents the skeleton of the profile screen within one of the views of the BottomNavigationBar.

```
class ProfileContainer extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: CustomScrollView(
        slivers: <Widget>[
          SliverAppBar(
            // behaviour and UI for the AppBar
            flexibleSpace: Row(...),
          ),
          // List with a single child which decides the size for the whole screen
          SliverList(
            delegate: SliverChildBuilderDelegate((_, i) {
              return Profile();
            },
            childCount: 1
            ),
          ),
        ],
      ),
    );
  }
}
```

Finally, I have included the state management for the user. I modeled the

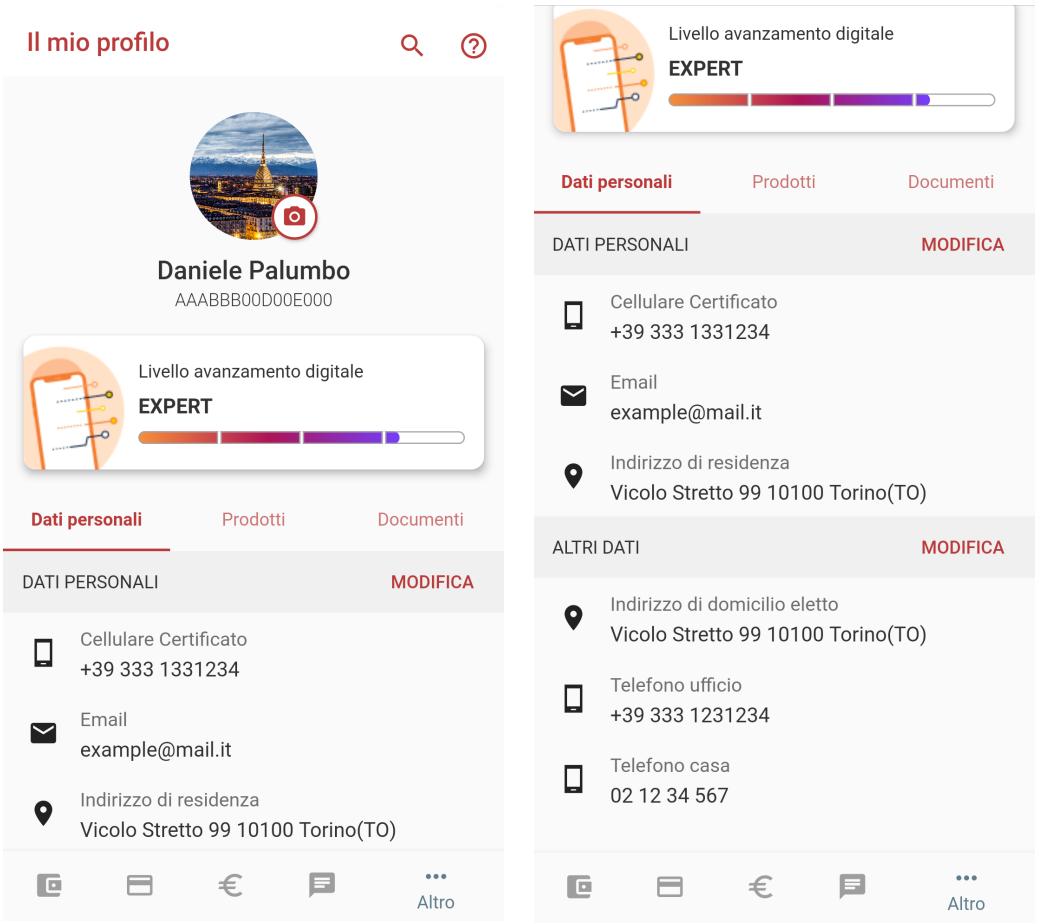

Figure 5.2: The figure represents the interface created by me following the design of Figure 5.1. All the features present in the mockup have been integrated, including internal navigation.

code to work with a `user` model which I have defined and which contains all the necessary fields. The user model is visible in Listing 5.4. The user is shared with the Flutter Provider pattern. I have defined a class `UserProvider` which keeps a User variable as an internal state and extends the ChangeNotifier Class. The UserProvider class and its methods are shown in Listing 5.5. The provider can notify any widget that subscribes to the state of the object. If there is a change in the state of the user object, this change will be propagated to all dependent views. A widget to retrieve the need status of the `Provider` class and takes the status of the user with the code shown in Listing 5.6. This code retrieves a top widget in the widget tree that has a Provider available. In my simple case I used the root widget to define the Provider. The code is shown in Listing 5.7.

Listing 5.4: User model definition.

```
class User {
  String name;
  String surname;
  String cf;
  String assetPath;

  List personalData;
  List products;
  List documents;

  User({this.name, this.surname, this.cf, this.assetPath,
      this.personalData, this.products, this.documents});
}
```

Listing 5.5: UserProvider class which includes the getUser() and setUser() methods.

```
class UserProvider extends ChangeNotifier {
  var _user = User();

  User getUser() {
    return _user;
  }

  void setUser(User newUser) {
    _user = newUser;
    notifyListeners();
  }

}
```

Listing 5.6: Provider method to retrieve from a parent widget an instance of the shared state.

```
var user = Provider.of<UserProvider>(context, listen: false).getUser
    ();
}
```

Listing 5.7: MultiProvider widget is in charge to initialize the Provider. It's child subtree will have access to this Provider.

```
return MultiProvider(
      providers: [ChangeNotifierProvider.value(value: UserProvider())
    ],
      child: MaterialApp(
        title: 'Profile Demo',
        theme: ...,
```

```
6             home: HomeContainer(),
7           ),
8       )
```

5.2 Advanced case: personal financial management section

In this section I analyze charts and graphs in Flutter. Many real world apps uses graphs and charts in large number of use cases. Showing raw data can be really frustrating and complicated for both developers and users. The use of graphs and charts in applications helps everyone and it can give good insights of the data. In the case of the application of IrisCube Reply, the graphs are used to provide the user with an effective way to keep her financial management under control.

In the personal finance management section there are two graphs in particular. The first is a donut chart showing expenses and income on a monthly base and it is visible in Figure 5.3. The second is a sliding graph that shows the trend of the user account balance and it is visible in Figure 5.5. The donut chart have some differences in how the labels are showed between the horizontal and vertical version. In Figure 5.4 the horizontal version is shown.

Figure 5.3: The figure shows the donut chart realized by the IrisCube Reply development team. This is the vertical version of this chart. In this version, the labels no not include the amount related to the represented category.

Flutter is a UI framework of modern era and it provides a many ways to work with charts and graphs elegantly. It is possible to create some very great user experiences using Flutter graphs with native performance. Among those several

Figure 5.4: This is the horizontal version of the chart presented in Figure 5.3. In this version, the labels include the amount related to the represented category.

Figure 5.5: The figure shows the interface of the account balance management. Here is possible to see the graphic representation of the nack account. The graph is scrollable and the user can select the date interval.

options that Flutter provides, I focused my work on the Flutter native library and a third parties library.

To make these two graphs I explored the possibilities offered by native Flutter and some libraries. Flutter's main chart library is `charts\_flutter`[1]. This library has not yet reached version 1.0 and does not have null_safety support - required only from version 2.7 of the Flutter framework. This library offers bar, time, pie and line charts. Furthermore, it allows to customize the axes, the labels and the graphic aspects. However the library is still a work in progress. I found it difficult to create

[1] `https://pub.dev/packages/charts_flutter`

even just the pie chart respecting the specifications of the mockup. The library is not versatile enough and is only good for very simple and not very technical graphs. So I tried a different library. Among the Flutter packages I found the third party pie_chart library. This library offers a simple and very versatile way to customize pie or donut charts. The result is exceptional and allows me to create the first of the two graphics of the application.

The results of my programming show that it is really easy to make a donut chart. The chart can be dynamically animated by updating the data source. An animation time determines the speed of change. Since it is possible to customize the legend, the labels, the colors, the general dimensions and the animation time, this library makes it possible to fully respect the graphic defined by the product team. I have included the chart structure code in Listing 5.8, while the final chart result is available in Figure 5.6. In the Figure 5.7, on the other hand, the version of the horizontal chart is visible. To have different widgets based on the orientation of the phone, it simply enough to encapsulate the screen in a `OrientationBuilder` widget. In Listing 5.9 I have included a very simple code snippet that uses the OrientationBuilder input parameter `orientation` to determine which widget to use.

Listing 5.8: Code structure for a donut chart to represent expenses for a user account

```
Scaffold(
      appBar: ...
      ),
      body:
        ...
        child: PieChart(
            dataMap: expensesData,
            animationDuration: Duration(milliseconds: 800),
            chartRadius: MediaQuery.of(context).size.width / 3,
            centerText: TotalExpensesWidget(expensesData),
            legendOptions: LegendOptions(...),
            colorList: ColorList(expensesData),
            chartValuesOptions: ChartValuesOptions(...),
            ),
        ...
)
```

.

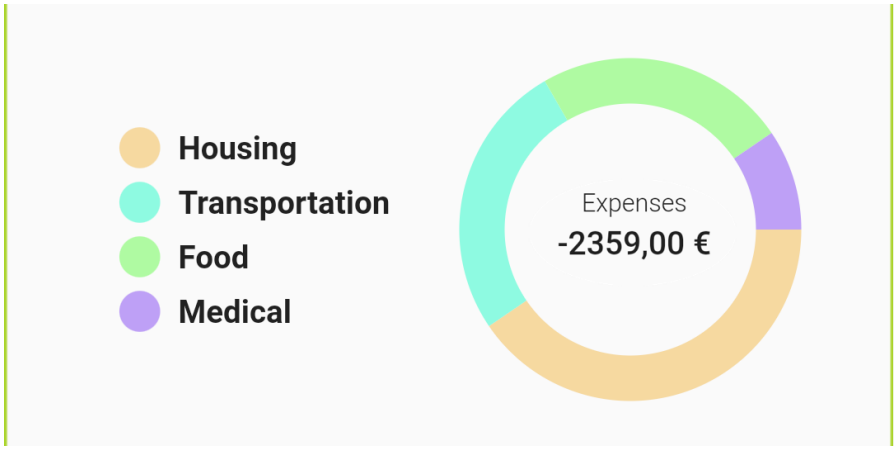

Figure 5.6: The figure shows the graphical result of creating a donut chart.

Listing 5.9: Code structure for an OrientationBuilder widget inside the pie chart screen

```
class PieChartScreen extends StatelessWidget {
  const PieChartScreen({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return OrientationBuilder(
      builder: (context, orientation) {
        return orientation == Orientation.portrait
            ? PieChartVertical()
            : PieChartHorizontal();
      },
    );
  }
}
```

Subsequently, I made the chart for personal finance management. This chart is technically an interactive way to show the user account data. In the case of the native application, this graph was created specifically by the development team. In my analysis I was interested in understanding if it was possible to make it with Flutter pre-made components or with some third parties library, in order to avoid to program myself from scratch. I have explored several alternatives, and I found a package in particular which have returned positive results. I considered the library `syncfusion_flutter_charts`[2] created by SyncFusion[3]. This company has

[2] `https://pub.dev/packages/syncfusion_flutter_charts`

[3] `https://www.syncfusion.com/`

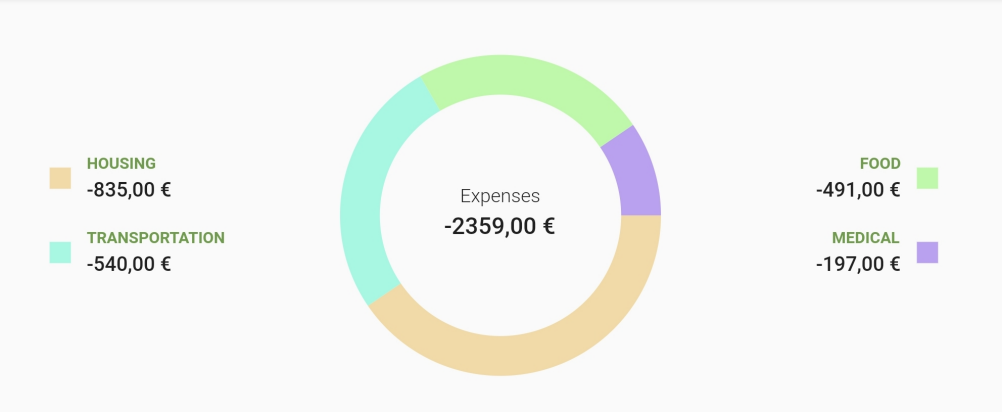

Figure 5.7: The figure shows the graphical result of creating a donut chart within label data in the horizontal view.

released its own charting library that is incredibly comprehensive. This library allows to create all types of graphs available through the native Flutter library and also adds pyramids and radial gauge graphs. The library also provides other components but these are not related to the discussion of graphs.

Using this library I tried to make a solution similar to the mockup. The graph in Figure 5.5 is a scrollable list. On the x axis there are the days. On the vertical axis there is the value range for the bank account. The max value of the range is defined by the greatest element of the data array. The chart is an area chart; the color of the area is a white to blue gradient. The color of the top line is the same as the area but with 100% opacity. The day selector is fixed in the center of the page and the intersection of the selector with the chart area value is a circle. I started by defining an array of data that I pass as a status to the graph's container widget. I defined it as a simple map with the key on the day and the value of the current account as the value.

For the realization I started from the library code for the creation of a `AreaChart` widget. Inside this widget I have inserted a scrolling graph. The x-axis takes the keys of the data list that I made with the data of each day from January 15th to March 15th. In Listing 5.10 I show the structure of the Cartesian graph. It can be seen how the structure is strongly generic. In my case, the `_getDateTimeSeries()` field returns a `AreaSeries` widget, which will have the task of drawing the graph on the screen. Each type of series has a different behaviour. The library documentation is complete and this helped me understand what to do to define the graph and the behaviour of the center pointer. This component is a `Trackball` widget in the syncfusion library.

The code for the trackball structure can be seen in Listing 5.11. I got the scrolling functionality by encapsulating the time list in a `InfiniteScrolling`

graph. In my case, this graph is responsible for showing only part of the list. The data loading is asynchronous, so it would be possible to integrate this graph directly with the http API to make the data request. While waiting for data, a circular loading bar is shown in the graph. In my simple case, however, the data is all local and there is no waiting.

Listing 5.10: Code for the definition of a Cartesian graph. This structure is common to all types of Cartesian graphs in the library. By customizing the various fields of the graph it is possible to obtain any combination of behaviours provided by the library.

```
SfCartesianChart _buildDateTimeAxisChart() {
   return SfCartesianChart(
       plotAreaBorderWidth: 0,
       title: ChartTitle(
           text: 'Conto ${_accountNumber}'),
       primaryXAxis:
           DateTimeAxis(majorGridLines: const MajorGridLines(width:
  0)),
       primaryYAxis: NumericAxis(
         minimum: 0,
         maximum: _maxAccountValue + 20000,
         interval: 20000,
         axisLine: const AxisLine(width: 0),
         majorTickLines: const MajorTickLines(size: 0),

       ),
       series: _getDateTimeSeries(),
       trackballbehaviour: _myTrackballbehaviour);
 }
```

Listing 5.11: Construction of the Trackballbehaviour object which is responsible for creating and managing the vertical bar. The position is fixed in the middle; indeed the label is visible only after a user touch and persists for two seconds.

```
_trackballbehaviour = Trackballbehaviour(
     lineColor: Color.fromRGBO(118, 135, 167, 0.41568627450980394),
     enable: true,
     markerSettings: const TrackballMarkerSettings(
       markerVisibility: TrackballVisibilityMode.visible,
       height: 10,
       width: 10,
       borderWidth: 10,
       borderColor: Color.fromRGBO(24, 108, 255, 1.0),
       color: Colors.white
     ),
     hideDelay: 2 * 1000,
     position: Position.center,
```

```
14          tooltipAlignment: ChartAlignment.center,
15          tooltipDisplayMode: TrackballDisplayMode.floatAllPoints,
16          tooltipSettings: InteractiveTooltip(format: 'point.y'),
17          shouldAlwaysShow: true,
18        );
```

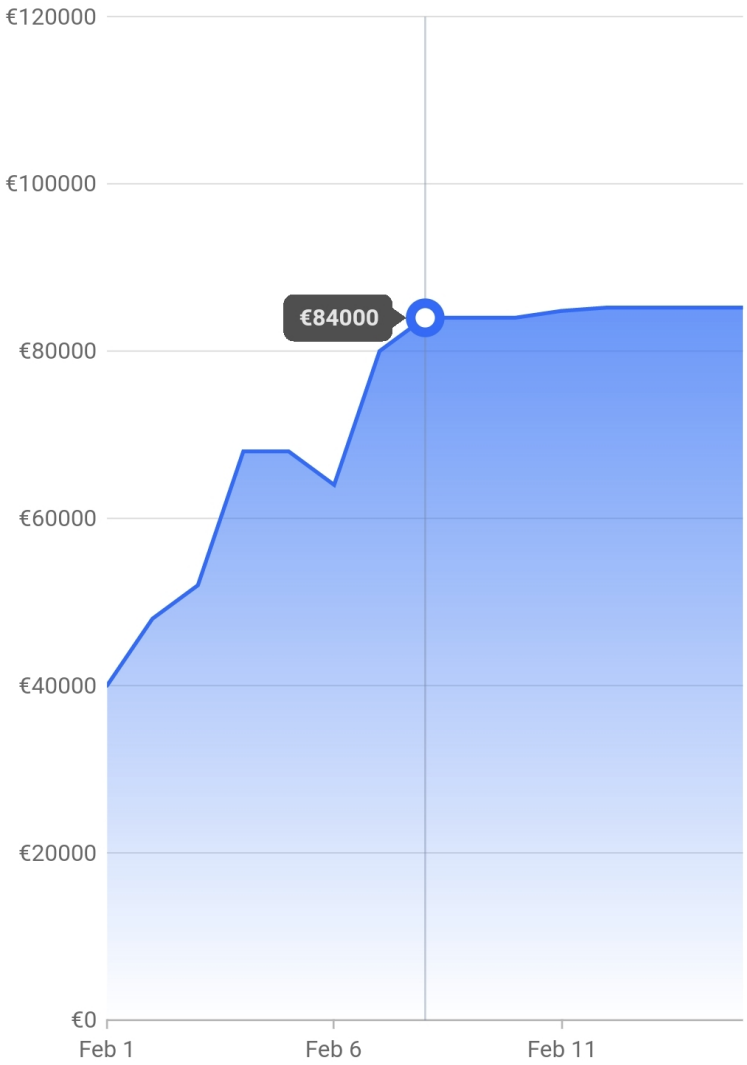

Figure 5.8: The figure shows the graphical result of creating a donut chart within label data in the horizontal view.

The result of this programming is shown in Figure 5.8. Overall, the result obtained for the creation of the mockup - Figure 5.5 - is a satisfactory outcome.

The libraries I choose allowed me to obtain a result comparable to the mockup. The main problem I had was when I tried to customize the label for the x axis. I was unable to get the x-axis labels represented as a group-by week. As a solution I have decided to show all single days on the x axis. However, the important result is that I managed to get a good result using libraries available for Flutter. I did not have to write much low-level code because the features I need were already present in the library. My situation is not an exception, because the Flutter community has produced numerous libraries covering various needs. The Flutter community has grown very rapidly from 2018 to 2021. In just 3 years, Flutter's reach has matched React Native in: GitHub stars, reddit channels, StackOverflow questions.

In conclusion, chapter 5 helped me to better understand how Flutter works. I am positively impressed by Dart choice to promote UI as code. This choice allows me to obtain a complete layout with a few lines of code. Widgets are deeply customizable in really every aspect. The feeling of working with Flutter widgets is that you always have a suitable component available to use in every situation.

Chapter 6

Results

In this chapter I show the results obtained from the Flutter analysis carried out in the previous chapters. The results shown here summarize the partial conclusions reached by talking about the various topics of this thesis.

6.1 Flutter advantages and disadvantages

In this section I summarize the highlights of Flutter's analysis and try to draw my own conclusions on the analysis performed. I started in chapter 2 explaining the cross-platform approach used to build the Flutter architecture. The Flutter team decided to use a compiled approach to develop the framework, as I explained in the subsection 2.2.4. This approach gave the team total control to deliver an effective, flexible and powerful solution. The effects of this process are evident by analyzing the possibilities offered by Flutter to developers.

The official SDK contains over two hundred Widgets that can be customized in appearance and behaviour. The efficient widget tree building mechanism allows developers to quickly learn how to combine widgets together. This results in more focus on the graphical aspect of UI construction rather than programming. Furthermore, the built-in Flutter widgets permit to easily obtain a graphic style that respects the guidelines of Google's Material Design or Apple's Human Interface. It is also possible to create the same application with both graphic styles at the same time based on the platform where the app is running. The flexibility and versatility of Flutter do not come at the expense of graphic consistency and Flutter is confirmed as the best solution so far among the cross-platform frameworks on the market.

In section 3.2 I have illustrated the widgets available to build UI elements and the declarative approach to define the interface. The set of widgets available is vast both to help developers and to respect the layout construction mechanism. As I

expressed in subsection 3.2.2, in Flutter even individual properties, such as padding, have their own widget that manages them. On the one hand, this means that the construction of the widget tree is very expressive and intuitive. On the other hand, developers need to know a large number of widgets to build complex layouts. In fact, building the interface using Flutter widgets takes practice and experience. The difficulty also includes mastering Flutter main layout principle: Constraints go down. Sizes go up. Parent sets position. Flutter layout construction is different from both native programming and web programming. Those who decide to use Flutter must know they need to rebuild their development experience. However, the development experience of several programmers is very positive. The expressiveness of the widgets layout helps keep ideas clear and focus on development.

Instead, as I analyzed in section 3.3, Flutter's internal navigation management is still immature. The navigator 2.0 released at the end of 2020 introduces numerous innovations to solve the problems of the first imperative Navigator dating back to 2018. However, the new declarative mechanism lacks in simplicity and proposes excessively complex and low-level solutions. For the future, Flutter needs to review internal navigation mechanism and the interactions between routing components. In this moment, indeed, developers will encounter a stumbling block due to the difficulty of learning how routing works. Native solutions are often cleaner and more expressive, and the same can be said of other frameworks like React Native. In Flutter there is no shortage of the necessary components to program any necessary internal navigation, but there is a lack of pre-made solutions to help developers.

After analyzing the UI-oriented features of Flutter, I analyzed state management in subsection 3.4.1. In these section, I introduced Flutter's state management mechanisms and I discussed about the choose to include the state management directly into widgets for UI. As I have already shown, in Flutter there is a type of stateful widget that allows to manage the application state directly in the widgets that deal with the UI. In addition, there are specific widgets that permit to keep track of a shared model, notify changes in status and react to these changes. For the management of the Flutter state there are essentially two currents of thought: the first that approves the current system and the second that instead criticizes it. In particular, these criticisms concern the consequences of managing the state at the same level as the widgets that define the UI. It is criticized that it is easy to lose sight of the graphical part of a widget due to the extra syntax required to manage the state. Also, a complex app state needs to be implemented across multiple widgets, which does not help in dividing responsibilities.

I partly recognize these issues, but from my analysis I have reached different conclusions: the Flutter team has made a very clear decision on how to structure the management of the internal state of an application. Flutter provides a few and well-defined widgets where everyone has a specific task in managing the state. Also, combining the state management with the part of the UI that will show it

graphically helps in my opinion to keep the code cohesive. In most cases, you should never exit the same file for all the code concerning a part of the UI. I therefore consider the Flutter solution to be flexible and suitable both in simple cases and in more complex cases with local and remote information. Flutter state management requires design by developers to be implemented correctly, but it pays off in the long run by simplicity.

In chapter 4 I analyzed the Flutter framework in a business context to answer the main question of this thesis: whether or not Flutter is suitable for a company active in the field of mobile applications development. The conclusions show lights and shadows, I go in order to summarize them.

In the subsection 4.1.1 section, I analyzed Dart, the programming language behind Flutter. Dart is an excellent language that takes the best from other more popular languages, while maintaining a simple, clean and expressive syntax. It is no coincidence that from the surveys carried out annually by StackOverflow, Dart appears in sixth place in the ranking of the most popular programming languages of 2020 [21]. Dart is positioned behind Kotlin, who occupies the fourth place, but ahead of Swift, eighth place, and Javascript, ninth place. However, Dart is far ahead in this ranking compared to Java and Objective-C, considered by many experts languages that will disappear over time in favor of more modern solutions.

In addition, Dart provides a set of development tools, in addition to Flutter's tools, which create a complete development environment. In the sections 4.1.2, 4.1.3 and 4.1.4 I have analyzed in detail some of the essential tools for developing code in an enterprise environment. I started by analyzing the test suite I described as comprehensive to cover the main types of testing - unit tests, component tests, and e2e tests. The result of the analysis is positive because Flutter introduces high-level methods to test the UI and, in addition, having a single codebase, incentives to better test the code. Testing is evidently one of the advantages of Flutter. Even from the analysis on the editors that support the features of Dart and Flutter, no criticalities appear.

Instead, for the quality of the code, Dart does not yet have tools as advanced as other languages. I pointed out in subsection 4.1.4 that there are libraries to guarantee style rules and syntax uniformity, but there are no libraries for advanced static code analysis. Furthermore, in the enterprise environment it is common to rely on advanced tools for automatic documentation, performance analysis and continuos integration. Dart is still a marginal language on the world scene and has received less attention than more famous languages such as Javascript. This is certainly a disadvantage right now, but the situation is likely to improve quickly in the future.

In section 4.2 I analyzed the performances of the Flutter framework by dividing the work into two main parts: in the first section, subsection 4.2.1, the analysis concerns the graphic performance of the framework, while in the second section,

subsection 4.2.2, I analyzed Flutter's computational performance.

The results were obtained by combining the results coming from experiments that in the past years have tried to evaluate the performance of Flutter. The results show an overall positive picture. In Jozef Petro's work, which compared the average FPS for the same app between Flutter and React Native, the results show that Flutter manages to both hit the 60 FPS first and that the average FPS is constant throughout the test. Then I integrated the work of InVerita who did a performance analysis comparing the native development to Flutter and React Native with two test applications. The applications served to stress specific graphic aspects. The results show that Flutter performs comparable to the native albeit lower, while outperforming React Native in all tests. Finally, the work of Andrea Bizzotto, carried out on a chronometer application, showed that Flutter's performances, compared to the native ones, suffer from the continuous updates of the UI - every 30ms - which penalize the results.

Overall, Flutter's performance has always been close to native development, but never on par. The only case that proves the opposite is Matilda Olsson's thesis, where she shows that only in a very simple application Flutter's performance are actually identical to native ones. For Flutter there is room for improvement and to optimize performance. Surely the young age of the framework is a symptom that there is still work to be done to actually match the native performance. Despite this, in no test has Flutter done worse than React Native. This therefore demonstrates that the Flutter framework is the first of the frameworks for cross-platform development that does not suffer from compromises on execution performance.

After the analysis made in this thesis, I can affirm that Flutter is a valid choice as a framework for the development of cross-platform applications. Using Flutter means for a company to be able to alleviate the problems of native development. In particular, the use of a single codebase helps reduce development and update times. Furthermore, a single codebase requires less manpower to develop and maintain and therefore reduces the overall cost of development. In reality these are the advantages already present in other cross-platform frameworks, but Flutter achieves these goals with fewer compromises. Specifically, Flutter emphasizes three peculiarities. First, Flutter performs better than its direct competitors. Second, Flutter is a flexible framework because it is completely independent of the operating system below. Third, Flutter has extremely neat widget graphics similar to the native look thanks to the work of the Google team. Thus it is possible to create applications that are identical in appearance to the native. These three peculiarities of Flutter are almost unique in the landscape of cross-platform frameworks and are advantages that even a large company must consider. The common feeling is that cross-platform development is still intended as a secondary solution to native development and therefore only suitable for small contexts. Flutter in this scenario stands as a competitor to native development. Even large companies can enjoy

the benefits of a single codebase without sacrificing performance, flexibility and native graphics. These results are important for Flutter which shows how even in the mobile environment we are approaching typical solutions of the desktop environment. In fact, it is considered common for traditional systems to write multi-platform applications from a single code.

While Flutter isn't perfect and has flaws, most of them exist mainly because Flutter is a young framework and because Dart was a very rare language before Flutter - Dart was released in 2011 and Flutter in 2018. Considering these two reasons, it's plausible to think that in the next few years the deficiencies found will be resolved. In addition, there is still a need for optimization for performance and it is necessary for the Flutter team to improve integration with the hardware.

Even if in section 4.3 I have pointed out that it is possible to have up to 98% of shared code in Dart, for now it is unthinkable to be able to completely give up the native code. To build enterprise-level applications, a company needs to invest to create native development knowledge, in parallel with Dart's knowledge. Flutter still has its limitations, especially when it comes to integrations with mobile ecosystems - subsection 3.5.2. In several situations it is necessary to rely on native code to implement what is missing. However, with Flutter it is possible to solve the two main limitations of native development: cost and development time. The ability to use shared Dart code almost entirely allows developers to really increase productivity and reduce development costs. Flutter does better than other frameworks and is a viable alternative to native.

However, compared to native programming, developing in Flutter requires a greater investment in terms of personnel and time. By now, Dart is still a little known language and it is easier to find competent developers with native languages or Javascript. This is because there are few job offers for Dart or Flutter developers and in general the average salary is lower than in other languages [22]. Therefore, the investment that a company must make lies in training developers to use Flutter and to adapt to its programming style.

Furthermore, it must be considered that Google is behind Flutter: Google is a company famous for introducing great new features in the field of software and then withdrawing them if they do not respect the high standards of use set. There are countless services created by Google that over the years have quickly gone into the background for the corporate business and soon abandoned. The Google cemetery [1] site gives you an idea of what I'm talking about. Flutter may have the same treatment in the future, especially considering that Google also promotes Kotlin as a cross-platform language. The Kotlin Multiplatform SDK shares several fundamental architectural choices with Flutter with the result that

[1] `https://killedbygoogle.com/`

there are discussions on the web about why these two systems promoted by Google are so similar [2]. Investing in Flutter could therefore pose a risk.

However, the results on Flutter's analysis can only be positive and the prospects for the future are many. In chapter 7 I offer some ideas for reflections on possible developments.

6.2 Development experience

In this section I report my development experience with the Flutter framework. I started developing in Flutter in March 2021, in parallel with the begin of my analysis. I started programming right away to grasp the potential of the framework. My previous mobile experiences were based on university jobs and small personal projects. So I wanted to understand what I could do with Flutter and then direct my work. For this reason I started from the official Flutter documentation. I enjoyed the flow with which it is thought and I found the topics well introduced and explained. The interactive component contributed to the assimilation of concepts. It took me a few hours to complete the built-in tutorials and redo some of the proposed examples. I did not include the time it takes to complete the setup of Flutter and Android Studio. The two software are well integrated with each other and the setup is frictionless.

I entirely support Flutter's policy of promoting logic and graphics in the same file. I have had backend development experiences using web development frameworks. I have often noticed how easy is to lose focus when programming following the file division promoted by a framework. For example, the controller must have its file separate from the logic and the UI template, the HTML and CSS are split and away from the logic. Certainly in the long run this approach promotes the quality and maintainability of the code. However, it often introduces too much complexity when not needed and extends development time.

Combining logic and graphics allows you to iterate over the code very quickly, focusing productivity on the short term. In the medium term, however, it is easy to refactor the code to organize it better, always trying to have small files. Surely the most difficult aspect was the management of the state and shared data. Some frameworks, such as Spring and Angular, have injection mechanisms that Flutter lacks. To better manage the state and shared data, it is necessary to design classes and flows. Despite this, it is very easy to start creating interfaces. It is intuitive to use the basic components to achieve alignment, padding, element distribution, lists and images.

[2] `https://discuss.kotlinlang.org/t/google-flutter-what-could-be-their-reason-for-choosing-dart-over-kotlin/21179`

Overall, it took me just over a week to complete the first of the mockups I presented in the section 5.1 in Flutter. This was enough time to introduce the internal app navigation, the data list and the custom card-shaped widget within the dynamic user experience bar. My experience has therefore been positive and I confirm that I have encountered few obstacles. I confirm this experience also with the second project presented in section 5.2. This was of course more difficult in complexity than in the first case, but the increase in difficulty was proportional to the problem and not to the code. After understanding the Flutter mechanisms, it is easy to increase the complexity of the code and keep up to date with the innovations introduced.

I personally also enjoyed Flutter's integration with Android Studio. There are two shortcuts for generating a new widget template, which are `stless` and `stful`. In addition, a context menu that can be called up with `alt + enter` allows you to add or remove a widget from the tree by generating or adapting its keywords. Productivity is high when writing in Flutter on Android Studio. Finally, it's easy to set up the debugger and run code on a physical device or emulator. Building an application in Flutter means seeing it evolve on the phone display. It is very convenient with the hot-reload to instantly reload minor changes and evaluate their effectiveness. In general, Flutter applications seem to be more aesthetically pleasing thanks to this ease of developing them in real time.

Chapter 7

Conclusions and future work

In recent years the Flutter Framework was born and spread. Thanks to its immediate success it is seen by many experts as a valid alternative to native development. Flutter is proposed as a solution to improve developer productivity and reduce development costs, while maintaining execution performance close to native. This thesis was born with the aim of evaluating these claims and contextualizing Flutter from a business point of view. The analysis I made was divided into 3 main areas: first, understanding the type of cross-platform approach chosen to create Flutter; second, to analyze the technical and architectural aspects of Flutter; third, include the development environment in the analysis and evaluate Flutter's performance.

From the analysis carried out in this thesis, I was therefore able to highlight the potential of this framework and to find its weaknesses. I got a positive answer to the main question of the thesis: Flutter is suitable as a main choice for developing mobile applications in a business context.

The results are positive for two reasons in particular: the first is the expressiveness of the framework and of Dart, which allows rapid programming of complex elements of the interface. The second is the few disadvantages introduced by Flutter, especially on the performance side. Although I have shown that the performance of the framework rarely and in simple cases equate the performance of native applications, Flutter applications are generally very smooth and fast. This is still true even in the case of complex and hardware-intensive operations. Flutter's performance surpasses React Native in all tests, and this is remarkable due to the importance of React Native, which is considered one of the best cross-platform frameworks.

In conclusion to this thesis, Flutter is a well thought and well done framework.

It is a good choice for both small and large companies. Flutter allows companies to reach both mobile operating systems - Android and iOS - without the need for dedicated teams. The development environment is complete with the main tools needed by developers and the very comprehensive documentation reduces learning time. Sure, Flutter can capitalize on the benefits of cross-platform development - speed up software development and keep costs down - without introducing disadvantages over native programming.

However, Flutter is still a young framework with several aspects that can be improved upon. One of the possible future works is to follow the development of the framework trying to include more of the mobile ecosystem among the supported devices. Right now the main lack is the impossibility of easily implementing applications for wearables, or the integration with TV systems and with Android and iOS home widgets. In addition, the development team has already planned some updates to further improve the performance of the framework and simplify the use of the most common structures of each mobile application. So another future work is to delve further into the performance analysis. It would be necessary to create a more complex application in parallel between native and Flutter to collect some metrics: how long the development took, how much space the final application occupies, what the execution performance is and what users think of it as user experience. Unfortunately, these metrics were impossible to collect in this thesis alone and therefore a new experiment would be interesting to further explore the performance aspect.

The future developments I've described may help to better understand Flutter and cross-platform development in general. In the desktop and web environment it is common to develop a single codebase to cover multiple devices, while in the mobile environment this trend is less widespread. Flutter seems to be the beginning of a change in trend and it is necessary to follow its developments to better understand the phenomenon. Of course, it will also be Google's job to continue to support and promote Flutter so that it can establish itself as an industry standard.

Bibliography

[1] Andreas Biørn-Hansen, Christoph Rieger, Tor-Morten Grønli, Tim A. Majchrzak, and Gheorghita Ghinea. «An empirical investigation of performance overhead in cross-platform mobile development frameworks». In: *Empirical Software Engineering* (July 2020). DOI: 10.1007/s10664-020-09827-6 (cit. on p. 11).

[2] Dave Francis. *Is Drifty's Ionic dead?* https://foresightmobile.com/blog/ionic. Accessed: 2021-06-29. Dec. 2020 (cit. on p. 15).

[3] Dave Francis. *Everything that is wrong with Xamarin and why it is bad for you.* https://foresightmobile.com/blog/2020/09/15/isxamarindead. Accessed: 2021-06-29. Sept. 2020 (cit. on p. 16).

[4] Jan Rabe. *Everything that is wrong with Xamarin and why it is bad for you.* https://medium.com/@kibotu/everything-that-is-wrong-with-xamarin-and-why-it-is-bad-for-you-a5399075e50a. Accessed: 2021-06-29. Oct. 2018 (cit. on p. 17).

[5] *Flutter developers. How big is the Flutter engine?* https://flutter.dev/docs/resources/faq#how-big-is-the-flutter-engine. Accessed: 2021-06-29. 2021 (cit. on pp. 20, 39).

[6] Dan Abramov. *Basic Reducer Structure and State Shape.* https://redux.js.org/recipes/structuring-reducers/basic-reducer-structure. Accessed: 2021-06-29. 2015 (cit. on p. 22).

[7] Ian Mundy. *Declarative vs Imperative Programming.* https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2. Accessed: 2021-06-29. Feb. 2017 (cit. on p. 23).

[8] Michael Goderbauer. *Navigator 2.0 and Router.* https://docs.google.com/document/d/1Q0jx0l4-xymph9O6zLaOY4d_f7YFpNWX_eGbzYxr9wY. Accessed: 2021-06-29. Oct. 2019 (cit. on pp. 35, 36).

[9] John Ryan. *Learning Flutter's new navigation and routing system*. https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade. Accessed: 2021-06-29. Sept. 2020 (cit. on p. 36).

[10] Mikkel Rawn. *Flutter Platform Channels*. https://medium.com/flutter/flutter-platform-channels-ce7f540a104e. Accessed: 2021-06-29. Aug. 2018 (cit. on p. 48).

[11] Souvik Biswas. *Flutter: Building Wear OS app*. https://medium.com/flutter-community/flutter-building-wearos-app-fedf0f06d1b4. Accessed: 2021-06-29. June 2019 (cit. on p. 50).

[12] Rafael Ferraz. *Flutter + Apple Watch — Swift*. https://medium.com/@ferrazrx/flutter-apple-watch-swift-b43110dc4545. Accessed: 2021-06-29. Dec. 2019 (cit. on p. 50).

[13] Jozef Petro. *Flutter and React Native performance overview*. https://sudolabs.io/blog/flutter-and-react-native-performance-overview. Accessed: 2021-06-29. 2020 (cit. on p. 62).

[14] InVerita. *Flutter vs React Native vs Native: Deep Performance Comparison*. https://medium.com/swlh/flutter-vs-react-native-vs-native-deep-performance-comparison-990b90c11433. Accessed: 2021-06-29. June 2020 (cit. on pp. 62, 64, 65).

[15] Matilda Olsson. «A Comparison of Performance and Looks Between Flutter and Native Applications: When to prefer Flutter over native in mobile application development». MA thesis. Blekinge Institute of Technology, June 2020 (cit. on pp. 66, 68).

[16] Andrea Bizzotto. *How fast is Flutter? I built a stopwatch app to find out.* https://www.freecodecamp.org/news/how-fast-is-flutter-i-built-a-stopwatch-app-to-find-out-9956fa0e40bd/. Accessed: 2021-06-29. Mar. 2018 (cit. on pp. 68, 70, 77).

[17] Ihor Demedyuk and Nazar Tsybulskyi. *Flutter vs Native vs React-Native: Examining performance*. https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980. Accessed: 2021-06-29. Mar. 2020 (cit. on pp. 70, 71).

[18] Corey Sprague and Larry McKenzie. *eBay Motors: Accelerating With Flutter™*. https://tech.ebayinc.com/product/ebay-motors-accelerating-with-fluttertm/. Accessed: 2021-06-29. Sept. 2020 (cit. on pp. 72, 73).

[19] Gabriel Basilio Brito. *React Native vs Flutter: Android Benchmark*. https://www.youtube.com/watch?v=brXMXcscrg0. Dec. 2018 (cit. on p. 73).

[20] Smart Code App. *React Native vs Flutter: Android Benchmark*. FlutterVsReactNative|PerformanceandBuildsizeBenchmark. Jan. 2021 (cit. on pp. 73, 74).

[21] *Stackoverflow developers. Most Loved, Dreaded, and Wanted*. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved. Accessed: 2021-06-29. 2020 (cit. on p. 98).

[22] *Nix developers. React Native vs. Flutter: What is Better for App Development in 2021?* https://nix-united.com/blog/flutter-vs-react-native/. Accessed: 2021-06-29. 2020 (cit. on p. 100).