

Evaluation of React Native and Flutter for cross-platform mobile application development

Elin Hjort

Master's Thesis in Computer Engineering

Faculty of Science and Engineering

Åbo Akademi University

Supervisors: Annamari Soini and Dragos Truscan

November 2020

Abstract

There is an increasing demand for mobile applications requiring software companies to deliver high-quality products at a fast pace while maintaining reasonable costs. Moreover, an application is often created for multiple platforms to reach as many users as possible. Development of native mobile applications is considered to be challenging due to the use of specific development environments and technologies for each platform. Cross-platform development can be a solution to this problem. This thesis evaluates React Native and Flutter, two of the most modern cross-platform frameworks, to determine which of them is best suited for mobile application development at the Finnish software company Gambit. The evaluation is done with regard to development for Android and iOS, since they are the most commonly used mobile platforms. In order to compare the frameworks a set of criteria has been constructed, where each criterion has a weight to define its significance for the evaluation. The criteria and their weights have been tailored to the needs at Gambit. The evaluation shows that mobile application development with React Native produced the best results, however, both cross-platform frameworks proved to be viable options to native development.

Table of Contents

List of Abbreviations	1
1 Introduction	2
2 Mobile application development	5
2.1 Native applications	5
2.2 Cross-platform applications.....	7
3 Cross-platform frameworks.....	11
3.1 React Native.....	12
3.1.1 The bridge and platform APIs	13
3.1.2 Components	13
3.1.4 Layout and styling.....	16
3.1.5 The Virtual DOM	16
3.2 Flutter	18
3.2.1 Dart.....	19
3.2.2 Platform channels.....	19
3.2.3 Widgets.....	21
3.2.4 Layout and styling.....	22
3.2.5 The widget tree and the element tree	23
3.3 Summary.....	24
4 Evaluation criteria	25
4.1 Development perspective.....	26
4.2 Application perspective	28
4.3 Weights	29
5 Prototype application	31
6 Evaluation results	34
6.1 React Native.....	34
6.1.1 Development perspective	34
6.1.2 Application perspective.....	37
6.2 Flutter	44
6.2.1 Development perspective	44
6.2.2 Application perspective.....	46
6.3 Summary.....	51
7 Conclusion	54
Swedish summary	58
References	58

List of Abbreviations

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
OEM	Original Equipment Manufacturer
PWA	Progressive Web Application
UI	User Interface
XML	Extensible Markup Language

1 Introduction

Mobile applications are becoming an essential part of our daily lives. People want to do the same things they do on their desktops. According to studies on web traffic, the number of site visits from mobile devices has been increasing in recent years and is now over 50% compared to desktops [1].

An application must be available on multiple platforms to reach as many users as possible. Google's Android and Apple's iOS are by far the two most popular platforms in use, Android holds 75% of the market share and iOS 23% [2]. They have their own markets through which the apps can be distributed, Google Play Store for Android and App Store for iOS. How to deal with multi-platform applications is a common dilemma for many companies. An application can be created either natively, which means having to create a native application for every platform, or with a cross-platform framework producing one application targeted for multiple platforms.

Native applications are defined as software programs developed for a specific platform [3]. A native application has to be developed using the tools and language of the target operating system and cannot be installed on any other platform. It gives the best user experience in terms of user interface and performance and can make use of the latest features and technology of the target platform. Native applications have always been the superior choice for mobile application development. However, for many companies it is considered to be too expensive having to develop and maintain multiple code bases for one application.

Cross-platform frameworks try to bypass the difficulties of native development. A cross-platform application can target multiple platforms or operating systems, from one single code base, and is developed using a cross-platform framework. Generally, there are a few drawbacks to this type of application. The look and feel of the application are often not as appealing to the end user as those of a native application, and the application can

suffer from performance issues and bugs [4]. Cross-platform applications often depend on third-party libraries maintained by voluntary developers for additional functionality, which increases the risk of bugs in the application. Various cross-platform frameworks have tried to tackle these issues, but most of them have not yet been good enough to compete with the native frameworks.

This thesis is a collaboration with Gambit Labs Oy, hereafter referred to as Gambit, a Finnish software company with 35 employees founded in 2011. Gambit helps customers digitize their business by creating highly customized web and mobile applications. Gambit's customer base is very diverse, and project resources are limited by time and budget. To be as efficient as possible, it is important to use appropriate tools for application development. This thesis evaluates two cross-platform frameworks for mobile development, React Native and Flutter. The goal is to determine which one is best suited for developing applications at Gambit, in cases where native development is not an option. The framework needs to have access to a wide range of functionality to adapt to each customer's specific needs. Furthermore, the framework should be easy to start using with existing knowledge in web and mobile development and have long-term viability. The look and feel of the resulting application are also important, and the application should resemble a native application as much as possible to appeal to the end users. Both the technology and the result of each framework will be analyzed. This is done by implementing a prototype application in both frameworks and evaluating the results according to the list of criteria presented in chapter four. Since Android and iOS are by far the most popular operating systems for mobile devices, this thesis focuses on development for those two platforms.

The second chapter gives an overview of both native and cross-platform mobile application development along with the different approaches to cross-platform development that exist today. The third chapter studies React Native and Flutter, the two cross-platform frameworks that will be compared in this thesis, and the technology behind them. The list of criteria

used for evaluation is defined in chapter four, and chapter five describes the prototype application that will be created using each framework. The results of the evaluation are presented in chapter six.

2 Mobile application development

Mobile applications can be created in several ways. There are native applications and a few different types of cross-platform applications. The cross-platform applications can be divided into four categories: web, hybrid, interpreted, and cross-compiled, depending on their technology. Figure 1 provides an overview of the different approaches to mobile application development that will be discussed in this chapter. Web applications and native applications are on opposite ends of the spectrum. Web applications and PWAs are run in a web browser and have no connection to the native environment, while the other cross-platform approaches are somewhere in between web and fully native applications. Web technologies are often used also for hybrid and interpreted applications. All types except Web applications and PWAs can be distributed through Google Play or AppStore. Interpreted, cross-compiled and native applications do not have to be run in a browser or web view, and therefore have better performance than the others [4].

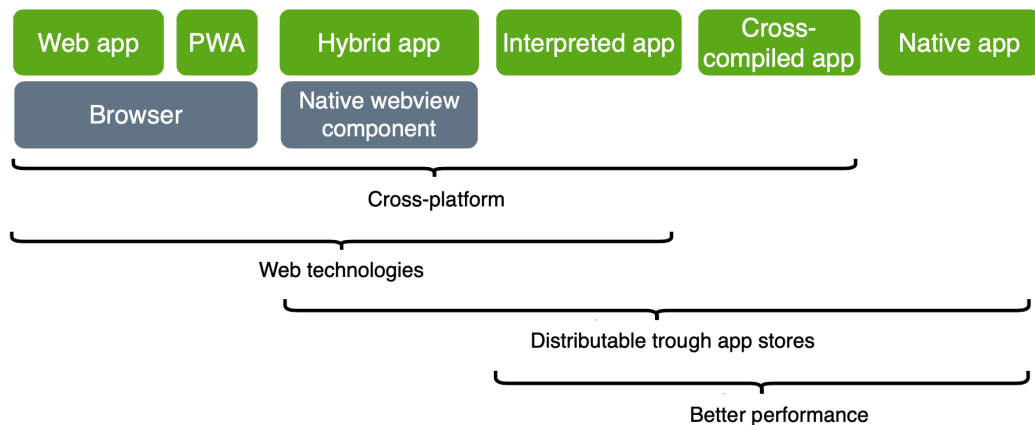


Figure 1 - Overview of mobile application development frameworks.

2.1 Native applications

Native mobile applications are built using the official framework and Software Development Kit (SDK) of the operating system. They follow platform-specific guidelines regarding appearance, structure, and navigational flow. Users that are familiar with a platform intuitively know how the platform standard controls work. Native applications are fast and

responsive as they communicate directly with the UI components, or original equipment manufacturer (OEM) widgets, as seen in Figure 2. Since they are tailored to their platform, native applications can make full use of the hardware and software of the device. This is why native applications give the best user experience and performance.

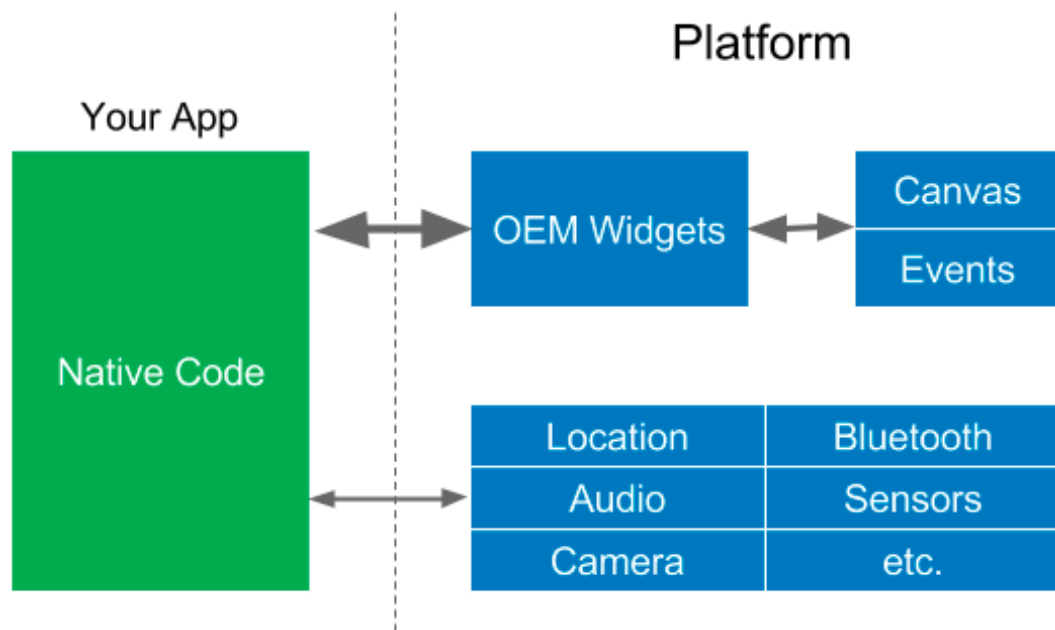


Figure 2 - Native application runtime architecture [4].

Android's official IDE is Android Studio, and the development language is Java or Kotlin. Android Studio comes with everything needed to develop an Android application and has an emulator that can be used for testing. There is a layout editor for building the UI. Native Android UI elements can be chosen from a palette and the element attributes can be set from within the editor [5]. Developers are encouraged to follow Google's Material Design guidelines [6] to achieve appearance and behavior that is familiar to Android users. When the application is ready, Android Studio bundles the application to a single file that can be uploaded to Google Play for distribution.

The IDE for iOS development is called Xcode and the supported languages for application development are Swift and Objective-C. Similar to Android Studio, it has an interface builder containing a library of native iOS UI

components that can be used in the applications. It includes visible elements such as navigation bars and switches, but also gesture recognizers for handling taps, swipes, and other touch events [7]. For iOS, it is Apple's human interface guidelines [8] that define the general appearance of an iOS application. Xcode is used for development as well as for submitting the applications to the App Store.

When developing a mobile application, the developer needs to be aware of the differences between platforms. Most things, such as following the design guidelines for each platform, come naturally when developing an application in Android Studio or Xcode. For example, an Android date picker looks completely different from an iOS date picker. A date picker component from the native framework will always have the right look, but with a cross-platform framework these differences have to be taken into consideration.

2.2 Cross-platform applications

Cross-platform frameworks exist because native application development is considered challenging and expensive. It requires time and different skills for each platform. In contrast, many cross-platform frameworks targeting mobile devices use the same technologies used in web development. The most fundamental are HyperText Markup Language (HTML) [9], Cascading Style Sheets (CSS) [10], and JavaScript [11]. HTML defines the content of a webpage. It uses predefined tags to set the layout and elements of the page. CSS is used to style an HTML document by declaring rules for how selected elements should be displayed. JavaScript is a client-side scripting language, meaning that the source code is not processed on a web server, but instead by the client's web browser [12]. Javascript is used to manipulate dynamic web content through the Document Object Model (DOM) of the browser. The DOM represents an HTML document as a node tree, where each node represents a part of the document. Nodes can be created, changed, or removed by JavaScript calls to the DOM API [13]. These common web technologies are used, to varying extent, in the cross-

platform development approaches presented here. Below, the main types of cross-platform applications are discussed.

Web applications target a browser rather than a specific platform. They can be accessed from any platform running a browser, both mobile and desktop. Web applications require an internet connection and cannot be installed on the device, nor do they have access to any device-specific features.

Progressive Web Applications (PWA) are web applications enhanced with native-like features. A PWA is created by adding a web application manifest including the application name, URL, and icons, among other things. A PWA can be added to the device home screen, and features such as offline service and push notifications are available through web APIs [14]. PWAs can mimic native functionality but they do not communicate with the native APIs of Android or iOS, everything happens in the browser.

Hybrid applications are essentially web applications wrapped in a native web view component. A web view is comparable to an embedded browser. It can display web content within the application but does not have all the features of a standard browser application. A hybrid development framework enables the use of device functionality through a bridge between the source code and the native API (Figure 3). The native APIs for Android and iOS are written in Java and Objective-C, so the bridge translates JavaScript calls to the language of the target platform [15]. The execution of the application is, however, still done in the browser engine, so performance can be poor.

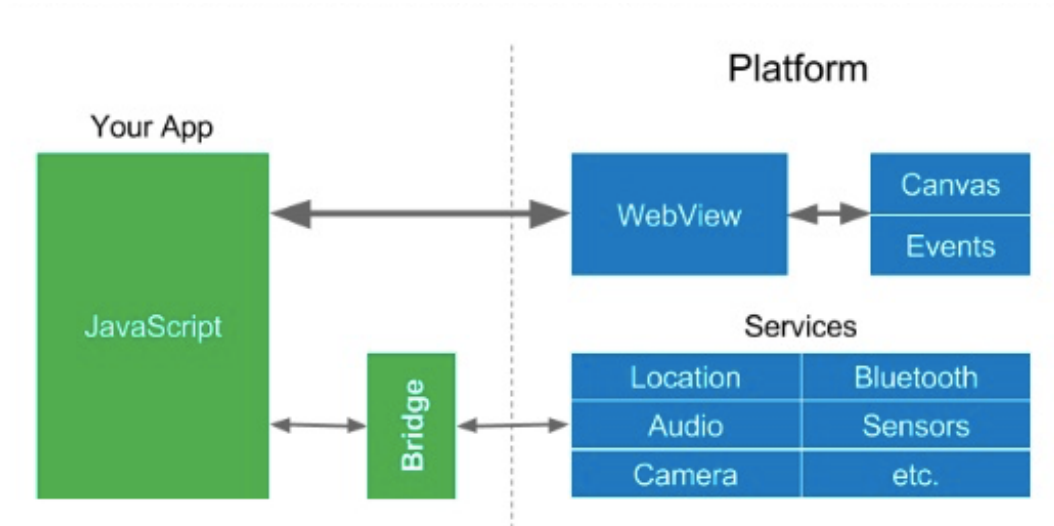


Figure 3 - Hybrid application runtime architecture [4].

Interpreted applications (also known as *web native applications*) also rely on the bridge to enable access to the target platform API. This can be seen in Figure 4. In contrast to hybrid applications, the bridge is used to render native UI components directly for each target platform, so no web view is needed [16]. End users receive an application with a native user interface, which is the major advantage of this approach.

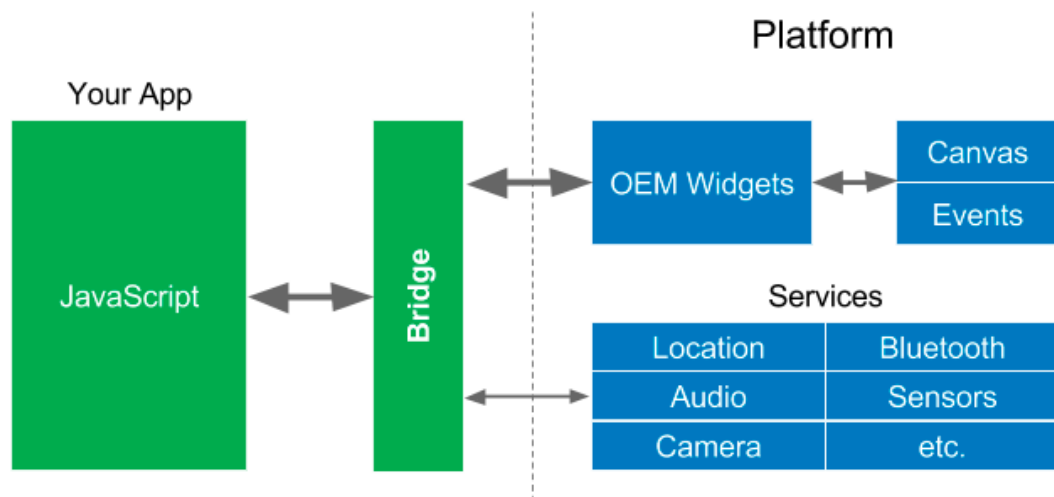


Figure 4 - Interpreted application runtime architecture [4].

Cross-compiled applications compile the source code to a native application for each platform [16]. Thanks to this, it has the same runtime architecture as native applications (Figure 2). This generally gives a better

performance than the other approaches, but since the native code is automatically generated it can be difficult to access a specific native API for a certain feature.

All these approaches to cross-platform development strive towards the same goal: to create applications for multiple platforms, without the difficulties of native application development. Every approach has its own way of handling the problem, but they have roughly the same pros and cons. They are all able to produce multi-platform applications from a single code base, but developers are limited to the features provided by the development framework or other third-party libraries. This means that support for new native features might be delayed or not implemented at all. Popular frameworks with many users tend to be quicker with new features. Cross-platform applications are distributed through the designated marketplace of each platform, although there have been cases where Apple has rejected a hybrid application from the App Store, because of too few native features [16].

3 Cross-platform frameworks

A software framework provides a toolbox for creating applications, containing built-in solutions to common problems. This can include classes and functions, but also other libraries, a code compiler, and tools to help with the deployment of the application [17]. Cross-platform frameworks enable creation of applications for multiple platforms from a single code base, aiming at speeding up the development time and lowering costs.

React Native and Flutter are currently two of the most interesting cross-platform frameworks for mobile application development. Figure 5 shows Google search trends for React Native and Flutter along with two other cross-platform frameworks, Ionic and Xamarin. The interest in React Native and Flutter is significantly higher than that for the other two, and the Flutter curve has been going steadily upwards since the release of the framework, whereas the curve for React Native seems to have plateaued. Their popularity is one of the reasons React Native and Flutter have been chosen for this evaluation. Gambit is looking for the most efficient way of creating mobile applications for their customers, using the latest technology. Both Ionic and Xamarin have previously been used at Gambit. Ionic is a hybrid solution [4] and therefore sensitive to performance issues. The framework is dependent on plugins to access native functionality. The plugins can have bugs and it might take time for new native features to be available. Xamarin is a cross-compiled solution [4], so performance is good, and the UI can either be shared between platforms or parts of it can be created separately for a platform. Unfortunately, Gambit's experience with this framework is that it suffers from a lot of bugs and delayed support for new features. The hope is that React Native or Flutter, with their updated technology, would solve these issues, or at least mitigate them.

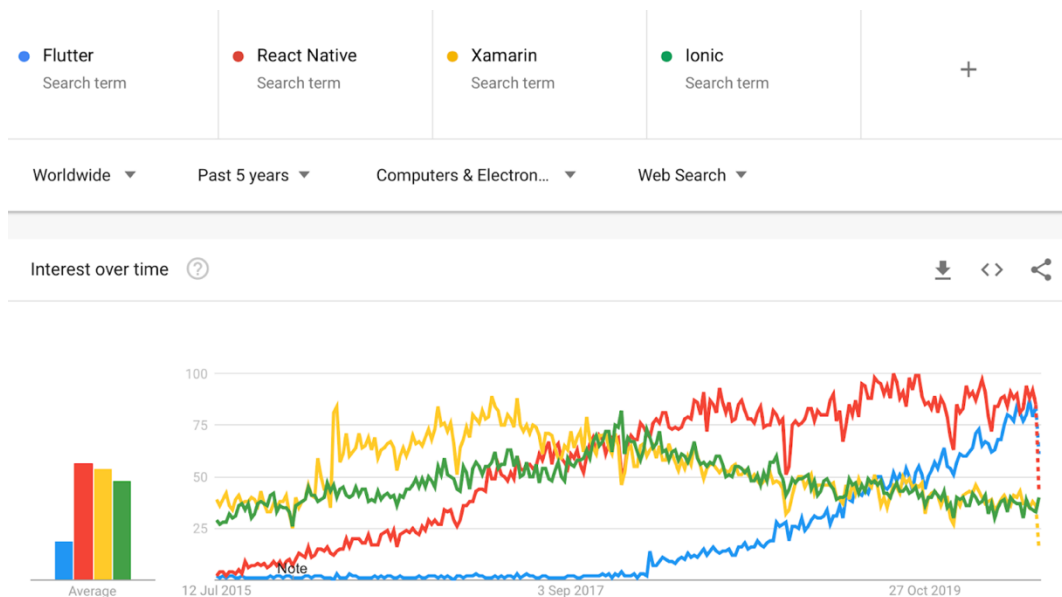


Figure 5 - Google search trends for Flutter, React Native, Xamarin and Ionic [18].

3.1 React Native

React Native is developed by Facebook. It started out as an internal hackathon where the goal was to unify the development processes for iOS and Android [15]. The initial version was released in 2015. React Native is an open source framework, so not only Facebook is developing it, but also individual developers and even companies such as Samsung and Microsoft have contributed [19].

React Native is categorized as an interpreted cross-platform framework, on the grounds that it renders UI components using the standard native rendering API of the target platform. Its runtime architecture can be seen in Figure 4. A React Native application can utilize platform-specific features such as the microphone or camera of the phone, by accessing the platform's API through JavaScript interfaces [15]. The use of JavaScript is without doubt one of the greatest advantages of React Native, since the language is commonly used by web developers. It makes it possible to build mobile applications with already familiar techniques instead of having to learn a new language and development environment for each platform.

React Native is based on React, which is Facebook's JavaScript library for building user interfaces. React applications, and also React Native applications, are written using a mix of JavaScript and JSX, a markup language described in section 3.1.2.

The main difference between React Native and React is that React Native targets mobile platforms instead of browsers. It has its own command line interface that is used for setting up a new project and building the application for Android or iOS. While React uses regular HTML elements to create views, React Native has its own components targeted for mobile platforms.

3.1.1 The bridge and platform APIs

React Native makes use of the rendering methods of the target platform. It invokes the native API to render UI components [15]. To do this, React Native is dependent on the bridge between the JavaScript code and the target platform. This bridge exposes the target platform API so that its native rendering methods can be invoked by the JavaScript calls. Since the APIs are written in the native language, the JavaScript calls must first be translated to the correct language, which is also done by the bridge. These asynchronous calls can be made without impact on the user experience since React Native runs on its own thread, not on the main UI thread [15].

All native APIs are not accessible through the React Native bridge, but thanks to the large community chances are that someone else has added any unsupported features and made them available as a library that is easily added to any existing project. Otherwise, it is possible to write one's own bridge.

3.1.2 Components

A React Native application is built out of dynamic components. Everything in the layout is a component. The components are written in JSX, which is a syntax extension to JavaScript and stands for JavaScript XML [20]. XML

is short for Extensible Markup Language [21] and is a more generic and flexible markup language than HTML where users can define their own tags [13]. JSX combines the control logic, markup, and styling in one file, even in the same language. Files are usually separated by technology, but in JSX, separation of concerns is more important than separation of technologies [15].

Components are essentially JavaScript functions that accept any type of input [22]. This input constitutes a component's properties or "props", which is the commonly used term in React Native. Props are a way of making the components dynamic, i.e. changing how and what they display at runtime. In addition to properties, a component can have an internal state. This is often a visual option, for example, if the component is expanded or collapsed. The state is a JavaScript object that can be assigned an initial value in the constructor of the component. To update the state, the component's *setState* method has to be used for changes to take effect in the user interface [23].

One of the most basic components is *View*, a versatile UI element which can be compared to the HTML `<div>` element. Like most React Native components it can be used for both Android and iOS. The component renders to an Android *View* or an iOS *UIView*, depending on the platform. There are also platform-specific components, such as *DatepickerIOS* that can only be used for iOS and renders to the typical iOS date picker. These components are named with a characterizing suffix (-IOS or -Android), and an attempt to use one for another platform than it was intended for results in an application crash.

In addition to the existing React Native components, new ones can easily be defined within a project to avoid repeating code. Figure 6 shows how to define a simple *Person* component.

```

class Person extends Component {
  render() {
    return (
      <View style={{alignItems: 'center', backgroundColor: 'lightblue', width: '80%'}}>
        <Text>Hi, my name is {this.props.name}</Text>
      </View>
    );
  }
}

```

Figure 6 - Defining a new component.

This component has a prop, *name*, that can be accessed through `this.props.name`. It simply renders a text on the screen containing the passed prop. The *Person* component has to be used somewhere in the application to be rendered. This is done by using a tag with the component name as shown in Figure 7. Figure 8 shows the *Person*-component rendered on a screen.

```

export default class ExampleApp extends Component {
  render() {
    return (
      <View style={{alignItems: 'center', top: 50}}>
        <Person name='Marco' />
        <Person name='Vicky' />
        <Person name='Debra' />
      </View>
    );
  }
}

```

Figure 7 - Using the newly defined component.

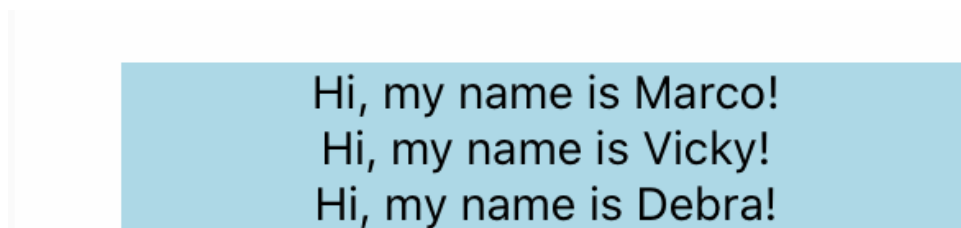


Figure 8 - The rendered *Person*-component.

If needed, it is possible to have platform-specific versions of components by separating them into different files, e.g. `component.ios.js` and `component.android.js`, and the right file for each platform will automatically be used [15]. There are some components that exist only for one platform. Examples of this are *TabBarIOS* and *SwitchAndroid*. It does not make

sense having versions for other platforms of these components since they wrap a specific functionality that only exists on that platform. An example of this is the component *ToolbarAndroid*, for which there is no counterpart on iOS. Just as components can be platform-specific, their props can be as well [15].

3.1.4 Layout and styling

When working with native mobile applications each platform has its own way of handling layout and styling. For the web, CSS is most commonly used for styling elements. React Native cannot use CSS directly, but it uses a limited subset of the available CSS styles. The aim is to keep styling simple, but still expressive. An example can be seen in Figure 7 where the CSS properties *alignItems* and *top* have been used to style the *View* element. This CSS implementation is part of the bridge between React and the target platform, so that it can be translated to the styling of the native components [15].

3.1.5 The Virtual DOM

When rendering the views, React works by creating and maintaining an internal representation of the view state. This representation includes all elements in the view and is commonly known as the Virtual DOM, referring to the document object model of the browser. The Virtual DOM acts like a description of the current UI state that can be given to a renderer in order to draw the UI elements on the screen. The purpose of the Virtual DOM is to avoid unnecessarily creating and accessing DOM nodes since this can increase the application's memory footprint and slow down the application. The Virtual DOM is simply a node tree containing elements with their properties represented by data objects, just like a browser DOM. This node tree is created by the React *render* method and updated every time there is a change in state or props [24]. When this happens, React compares the new version of an element with the one that was previously rendered, to decide if the DOM should be updated or not. If they are not equal, an update is necessary, otherwise not [22].

When working with React Native we do not have a DOM, but the process is the same. Instead of converting the content of the Virtual DOM to HTML elements (as is done for the web), React Native turns it into native UI elements.

Updating the UI is done in three steps (Figure 9) [24]:

1. A change in a component's state or props results in a re-render of the Virtual DOM representation.
2. The new Virtual DOM is compared to the previous one and the difference between them is calculated.
3. The real DOM is updated with the calculated change.

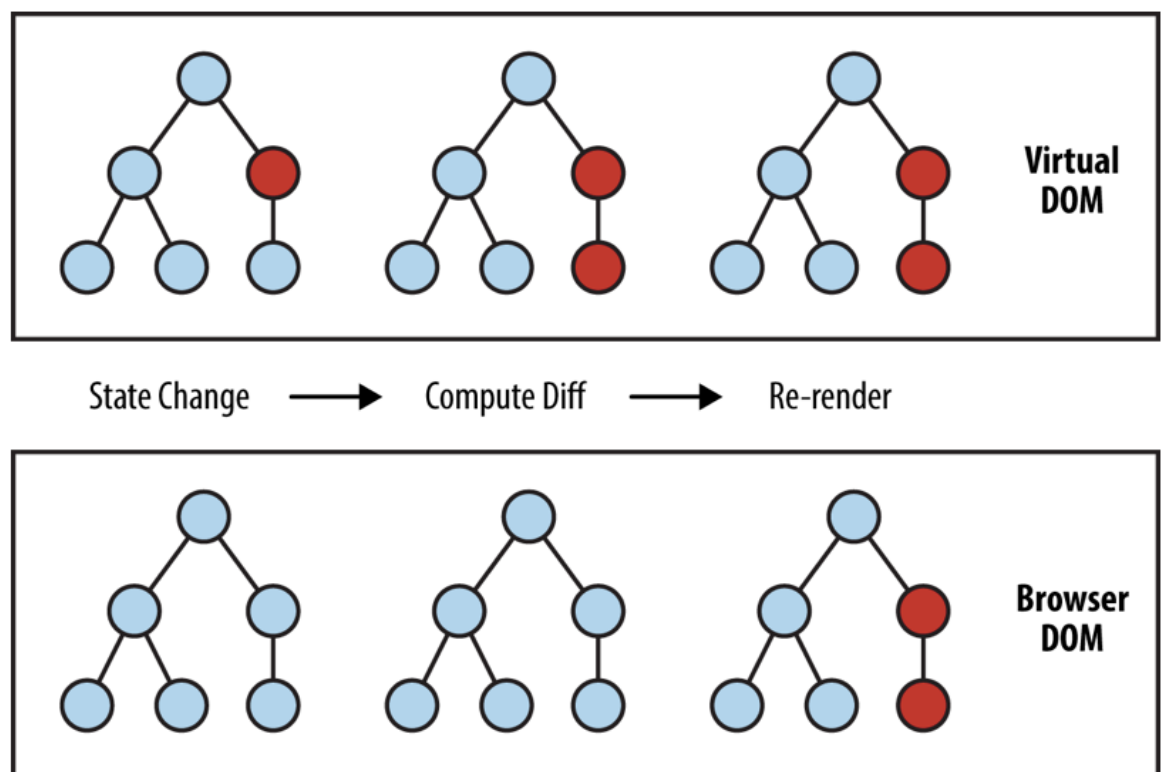


Figure 9 - The update process of the Virtual DOM [15].

3.2 Flutter

Flutter is an open source framework developed by Google and described as a “toolkit for building beautiful, natively-compiled applications for mobile, web, and desktop from a single code base” [25]. Flutter was initially released in 2017, but the first stable version came in December 2018 [26]. In this short time, it has become one of the most popular alternatives for cross-platform development. Flutter applications are written in Dart, a programming language also developed by Google. Because the source code is compiled to the native language, Flutter has great performance. It does not use native UI components, but renders its own components, Material widgets and Cupertino (iOS style) widgets, that are native looking. As seen in Figure 10, there is no communication layer between the view and the code to access the native rendering API of the target platform, which can otherwise slow the application down or cause bugs as the target platform evolves. Since the components are already styled it is easy to achieve a native look and feel for the application. A downside of Flutter is that developers need to learn a new language to use it. On the other hand, Dart has many similarities to C++, Java, and JavaScript, so it should feel familiar to most programmers.

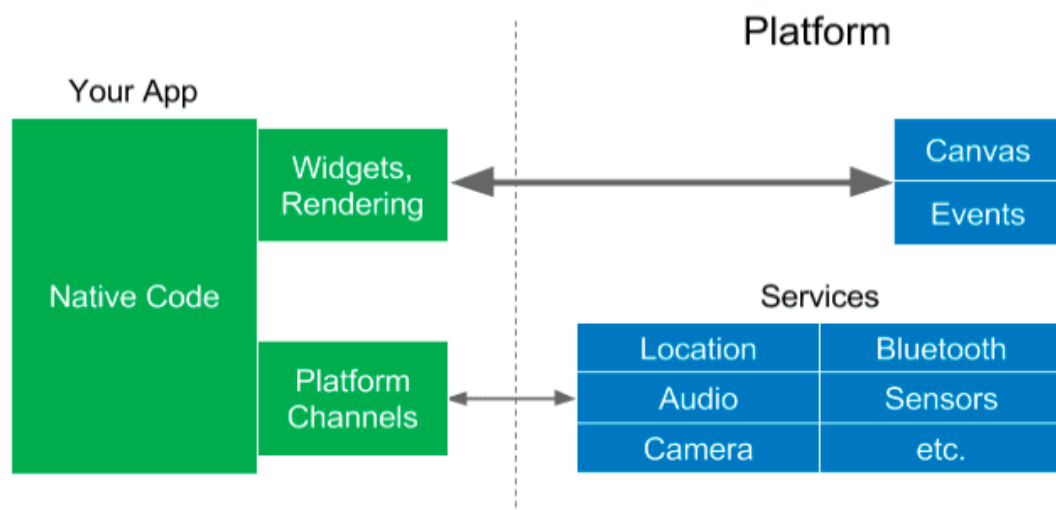


Figure 10 - Flutter runtime architecture [4].

3.2.1 Dart

Dart is a programming language that was released for production in 2013 [27]. The primary objective of Dart is to enable development of complex, yet highly performant and scalable web applications for the modern web. This also includes tablets and smartphones. Dart is designed to be an end-to-end solution, it is suited for both frontend and backend development since it can run in a web browser as well as on a server [28].

JavaScript is currently the most common language for web development, and while the web world has advanced enormously during the last 20 years, JavaScript has not progressed very much. It was not designed for building the sort of applications that are required today [29]. This is why Google decided to come up with something better, something that could handle their massive web applications and the performance they require. Dart is more applicable to large-scale applications than JavaScript, and when run on its own virtual machine it is also more performant than JavaScript [27].

Dart is a type-safe language [30], which JavaScript is not. JavaScript has very little declarative syntax and it allows spelling mistakes, wrong types, and even altering the functionality of built-in core objects. Potential programming errors can only be found at runtime. Because of this, JavaScript can be unpredictable, especially in complex applications [27]. Dart can be ahead-of-time compiled to native code for Android and iOS. This is used when compiling a Flutter application for release, resulting in an application with excellent performance. During the development process Dart can also be just-in-time compiled to quickly reflect any changes made in the code [31].

3.2.2 Platform channels

To be able to access the functionality of the target platform, such as camera and geolocation, Flutter needs to communicate with the platform's native API. While React Native uses the JavaScript bridge for this purpose, Flutter has platform channels through which messages are sent between the

Flutter application and the native platform. Messages with the method call are sent from the Flutter-side using an instance of the *MethodChannel* class. Android has a *MethodChannel* and iOS has a *FlutterMethodChannel* class as seen in Figure 11. These can receive method calls, execute them, and return the result. The user interface can then be updated accordingly. Method calls can also be sent the other way, from the target platform to the Flutter application for executing Dart methods [25]. Just like in React Native, the messages are asynchronous to avoid blocking the UI thread and causing the application to become unresponsive [31]. It is also possible to write one's own platform channels for functionality that is not already provided by the Flutter framework.

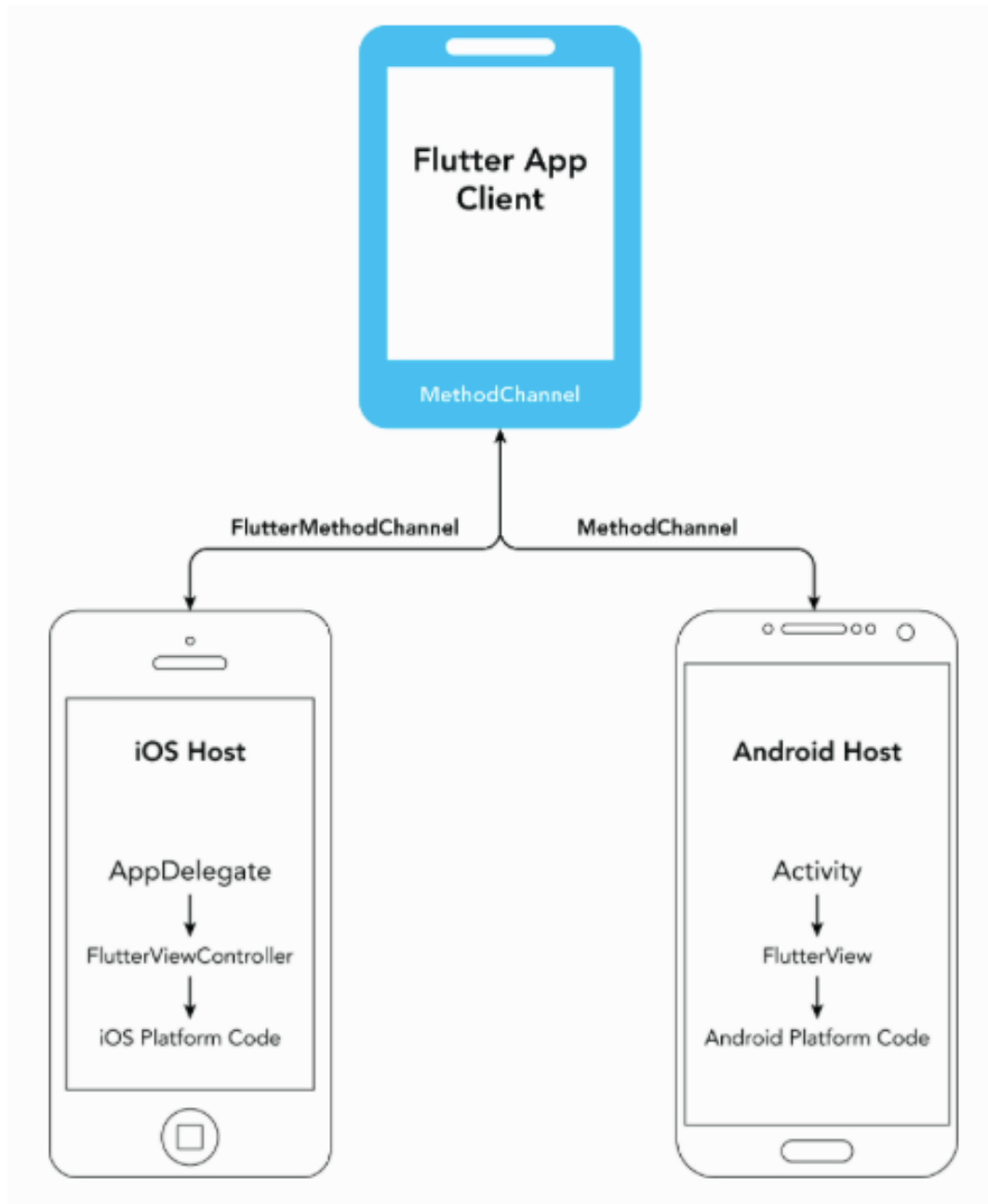


Figure 11 - Platform channels [31].

3.2.3 Widgets

Flutter's UI components are called widgets. Each widget holds instructions on how to render a part of the user interface. Everything in the user interface of a Flutter application is a widget. Not only structural elements such as buttons and views, but also padding, alignment, and other layout-related characteristics are defined by widgets. Simple single-purpose widgets can be nested to form more complex ones. Just like React Native, Flutter favors

composition over inheritance. A Flutter application consists of a root widget that contains other widgets. Properties are inherited from parent widgets to child widgets [25].

Widgets themselves are immutable, meaning that their properties are final and can only be set upon initialization. However, they can be either stateful or stateless. A stateful widget has a mutable state object that can be changed during the lifetime of the widget. This is similar to the state of a React Native component. When the state changes the immutable widget is replaced by a new one, to reflect the new state. This is suitable when a widget describes a part of the user interface that can change dynamically. Stateless widgets do not have a mutable state and are therefore static [32].

3.2.4 Layout and styling

The layout of a Flutter application is created entirely out of widgets. Most widgets are invisible components like rows, columns, and grids that determine the placement of other, visible, widgets [25]. Widgets can be customized by setting margin, color, or other attributes. Different widgets have different capabilities. One highly customizable widget is the *Container* class. A child widget is often wrapped in a *Container* widget to achieve a certain appearance. CSS is not used in Flutter, all styling is done by setting appropriate attributes for the widgets. A *Text* widget can be styled by giving it a *TextStyle* child specifying the font and color. Figure 12 shows an example of a *Text* widget inside a blue *Container* widget, the rendered widget can be seen in Figure 13.

```
Widget _textWidget() {
  return Container(
    color: Colors.lightBlueAccent,
    padding: EdgeInsets.all(20),
    child: Text("Lorem ipsum",
      style: TextStyle(
        color: Colors.black,
        fontSize: 16,
        fontWeight: FontWeight.w400,
      ) // TextStyle
    ) // Text
  ); // Container
}
```

Figure 12 - A Flutter widget.

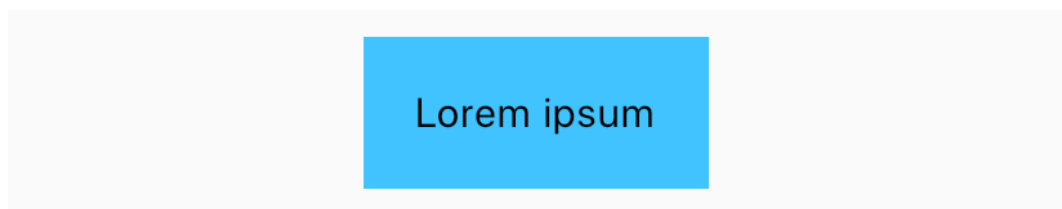


Figure 13 - The rendered widget.

3.2.5 The widget tree and the element tree

A *widget tree* is formed when composing widgets together to create a user interface. The widget tree is processed by the Flutter framework, and for each widget an element, which is an instantiation of a widget, is mounted on the screen. The currently rendered elements create the *element tree* [33]. There is also a third tree, the *render tree*, consisting of the rendered objects for each element. The render objects contain information objects for how the elements should be drawn on the screen, such as size and color [33].

When a widget is changed, it is replaced by a new widget in the widget tree. If the new widget is of the same type as the old one (e.g. they are both *Text* widgets) the reference of the existing element can just be updated to the new widget, and the corresponding render object can be updated with new values. If the new widget is of another type the old element and render

object have to be removed from their trees and new ones created [31]. The widget tree is similar to the Virtual DOM in React Native, both concepts improve performance by computing the change that needs to be made in the UI and only redraw the affected elements.

3.3 Summary

Table 1 shows a summary of the characteristics that have been discussed in this chapter. React Native has the advantage of using JavaScript, while Flutter uses Dart, a language that for many developers was unheard of before it was introduced as the language used with Flutter. On the other hand, Flutter has an advantage when it comes to performance. React Native uses the interpreted approach to cross-platform development. The JavaScript calls have to be translated to the native language at runtime, which is a drawback to the performance. Flutter generates fully native applications for each target platform, promising an application with roughly the same performance as a native application. Both frameworks are open source and created by giant technology companies, giving them credibility on the market.

Table 1 - Characteristics of React Native and Flutter.

	React Native	Flutter
Release	2015	2017
Created by	Facebook	Google
Language	JavaScript	Dart
Supported platforms	Android, iOS	Android, iOS
Open source	Yes	Yes
Cross-platform approach	Interpreted	Cross-compiled
Performance	Good	Close to native
UI components	Native	Proprietary

4 Evaluation criteria

Comparing cross-platform frameworks is a challenging task. There is no one-size-fits-all solution, but the choice of framework is dependent on available resources and requirements of the end product. In addition to this, cross-platform frameworks are constantly evolving, and so are the mobile platforms and their users' expectations. Many have presented different sets of criteria for evaluating cross-platform frameworks. Heitkötter et al. are considered to be pioneers within the topic of systematically evaluating cross-platform frameworks. Their commonly cited paper [34] presents a list of 14 criteria categorized into an infrastructure and a development perspective. This list often serves as a foundation for related research. Xanthopoulos and Xinogalos [16], along with Sommer and Krusche [35] have a similar approach to the evaluation process but with fewer, more generalized criteria. Sommer and Krusche also add weights to each criterion to define their importance.

In contrast to the more generalized approach, Christoph Rieger and Tim A. Majchrzak [36] present a detailed catalogue of 33 criteria that can be used to evaluate cross-platform frameworks not only for mobile devices, but for all app-enabled devices. They, too, suggest using weights for each criterion to make the framework adaptable to different needs. They have identified many criteria that can be used when evaluating a framework, but do not provide any clear way of how to evaluate them, and the distinction between them is not always unambiguous.

The criteria for the present evaluation of React Native and Flutter have been selected together with Gambit with inspiration from previously mentioned work on the topic, mainly Heitkötter et al. [34] and Christoph Rieger and Tim A. Majchrzak [36]. The criteria have been developed based on Gambit's experience on what is important when developing a mobile application for their customers.

Customer satisfaction is crucial and there are always restrictions regarding budget and time. In accordance with this, relevant criteria have been chosen. The criteria listed below are divided into two categories – development perspective and application perspective. The criteria in the development perspective focus on aspects concerning the development process, while the application perspective concentrates on factors affecting the end product.

4.1 Development perspective

D1 License and cost: The license of a cross-platform framework is important because it can restrict the usage of the developed application. The license can be prohibiting or liberal regarding modification of the framework itself. The costs of using a framework also have great impact when choosing a cross-platform framework. This criterion examines the license under which the framework is distributed and the costs for developing and distributing an application using the framework. The most favorable would be a free open source framework distributed under a permissive license [37].

D2 Supported platforms: At least two different platforms have to be supported by a cross-platform framework for it to have any purpose. Android and iOS are vital when targeting mobile platforms. Apart from which platforms are supported, this criterion also considers how easy it is to add multiple platforms and how much platform-specific code is required. Ideally, it should be possible to build the application for another platform without extra manual configuration.

D3 Distribution: The application created with the framework should be distributable through the standard application market for the platform. Users are used to downloading applications from Google Play and App Store and can feel safe doing so. These applications have gone through reviews to control that the platform's guidelines are followed and ascertain they do not contain any harmful or inappropriate content.

D4 Long-term feasibility: The long-term feasibility is significant when committing to a framework. It takes considerable effort to change the framework for an already existing application, and it might not even be possible due to technological lock-in. It is impossible to foresee the future, but there are factors that can help predict the long-term feasibility of a framework. A stable framework is characterized by regular bug fixes, commercial supporters, and a large and active community. Other aspects of interest are the age of the framework and that it supports the latest versions of the target platform operating systems [34].

D5 Development environment: This criterion examines the development tools meant to support development with the framework. A full kit containing IDE (including visual tools for UI development), debugger, and device simulators as provided by the native Android and iOS frameworks enables efficient development. A hot reload feature is valuable since it updates the application immediately to reflect any changes to the source code while preserving the state of the application [25] and, thereby, speeds up the development.

D6 Preparation time: This criterion examines the time and effort needed to start using the framework. If the framework requires knowledge of programming languages or additional tools that are not commonly used, the development process will be slowed down. Familiar concepts, good documentation, and available guides mitigate this. The learning curve will always be relevant if new developers are added to the team.

D7 Maintainability: Applications usually need to be updated and maintained to keep up with evolving technology or to add new features. A simple way of measuring maintainability is by lines of code [36], assuming that source code with fewer lines of code is easier to understand and maintain. Since this might not be very accurate, the modularity of the source code will be the mainly considered factor to evaluate this criterion [35]. If the code can be separated into independent modules that can be reused within the application, it is easier to test and refactor.

4.2 Application perspective

A1 Access to device hardware: Mobile applications often make use of device hardware such as camera and GPS, hence it is crucial for a cross-platform framework to support this functionality. Other hardware features that are examined are the microphone, the accelerometer which is often used in fitness applications to measure velocity, and the gyroscope, used to keep track of the device orientation.

A2 Access to platform functionality: Access to platform functionality is equally as important as the hardware access. Having access to the filesystem of the device is part of the native functionality as well as push notifications and biometrics. Push notifications are a convenient way of communicating with end users, while a biometric login can simplify the login process and by that increase the usage of the application. Maps are commonly used in mobile applications. Android and iOS both have their own map implementations, which should also be supported by a cross-platform framework. The possibility to use a web view will be checked for as well. This enables showing web content within the app instead of switching to a browser application.

A3 Internationalization: In order to appeal to as many users as possible the application needs to be localized. This includes translation of the UI as well as cultural aspects, such as date and number formats. There might even be a need for different content in different countries or for different languages. Ideally, the framework has built-in support for this functionality.

A4 Look and feel: End users generally prefer applications that look and feel native. Hence, it is important that the framework supports a native appearance. A cross-platform application should comply with the design principles of each platform. The cross-platform framework should provide similar UI components as the native frameworks, and the UI components should adopt the native look of the target platform without requiring much manual styling or platform-specific code.

A5 Performance: How fast the application responds to user interaction and loads content while the user is navigating within the application is significant for the end user. Performance issues will often become evident while scrolling through a view. The user may notice stuttering, or that parts of the UI completely freeze for a period of time. Both Android Studio and Xcode have profiling tools that can be used to monitor CPU activity, memory allocation, and energy usage. These metrics, along with manual inspection of potentially visible performance issues, will be used to evaluate the frameworks' performance.

4.3 Weights

Each criterion will be given a weight between 1 and 5 to show its significance for the evaluation. 1 is least significant and 5 is most significant. The weights for each criterion have been selected together with Gambit and are presented in Table 2. Most emphasis is given to platform support, distribution, and look and feel. Complete support for both Android and iOS, as well as having the apps distributed through Play Store and App Store, is often required by Gambit's customers. They also want the apps to have a native look and feel but might not always be willing to pay for the development of native applications. Lowest weight is given to the development environment and maintainability. A good development environment is beneficial, but not imperative for a good end result. Maintainability is difficult to measure accurately, and it is partly dependent on the developer writing the code.

Table 2 - Weighted criteria

Criteria	Weight
D1 License and cost	4
D2 Supported platforms	5
D3 Distribution	5
D4 Long-term feasibility	4
D5 Development environment	2
D6 Preparation time	3
D7 Maintainability	2
A1 Access to device hardware	4
A2 Access to platform functionality	4
A3 Internationalization	3
A4 Look and feel	5
A5 Performance	3

To evaluate the frameworks, prototype applications will be created with each framework, serving as a basis for the evaluation. This is done primarily for evaluating the criteria in the application category. The native development process and applications are for many criteria the considered point of reference. Both frameworks will be given a textual evaluation as well as a numerical value for each criterion. The value can range from 1 to 5, where 1 means “very poor” and 5 means “very good”.

5 Prototype application

This chapter describes the prototype applications created in each framework. The prototype is designed mainly to test the functionality mentioned in the application category. Experience gathered from the development of the prototype applications will also be used to evaluate the criteria in the development category. The prototype is built with React Native version 0.63.2 and Flutter version 1.22.3, and tested on Android 10 and iOS 14.

The foundation of the prototype application is a bottom tab navigator. This is a very common navigation system for both Android and iOS applications. Figure 14 displays the bottom tab navigator and the first view for each framework and platform with default styling. The first tab will have views for testing the hardware (A1). The application needs to be able to take a photo with the camera, and record audio using the microphone. The application has to retrieve the location of the device using the GPS and display it on a map. The map view is also included when evaluating the access to device functionality (A2). The application should be able to detect movement by utilizing the accelerometer and gyroscope of the device.

Similarly to the hardware, device functionality is tested in the second tab. The user should be able to access the filesystem of the device from the application and select a file for use in the application. The application has to check if and what types of biometric authentication are available, and also prompt the users to authenticate themselves. The framework should support usage of the platform standard map. The application should include a web view, enabling the user to view and interact with a web page at a predefined URL without leaving the application. The application should also be able to receive push notifications.

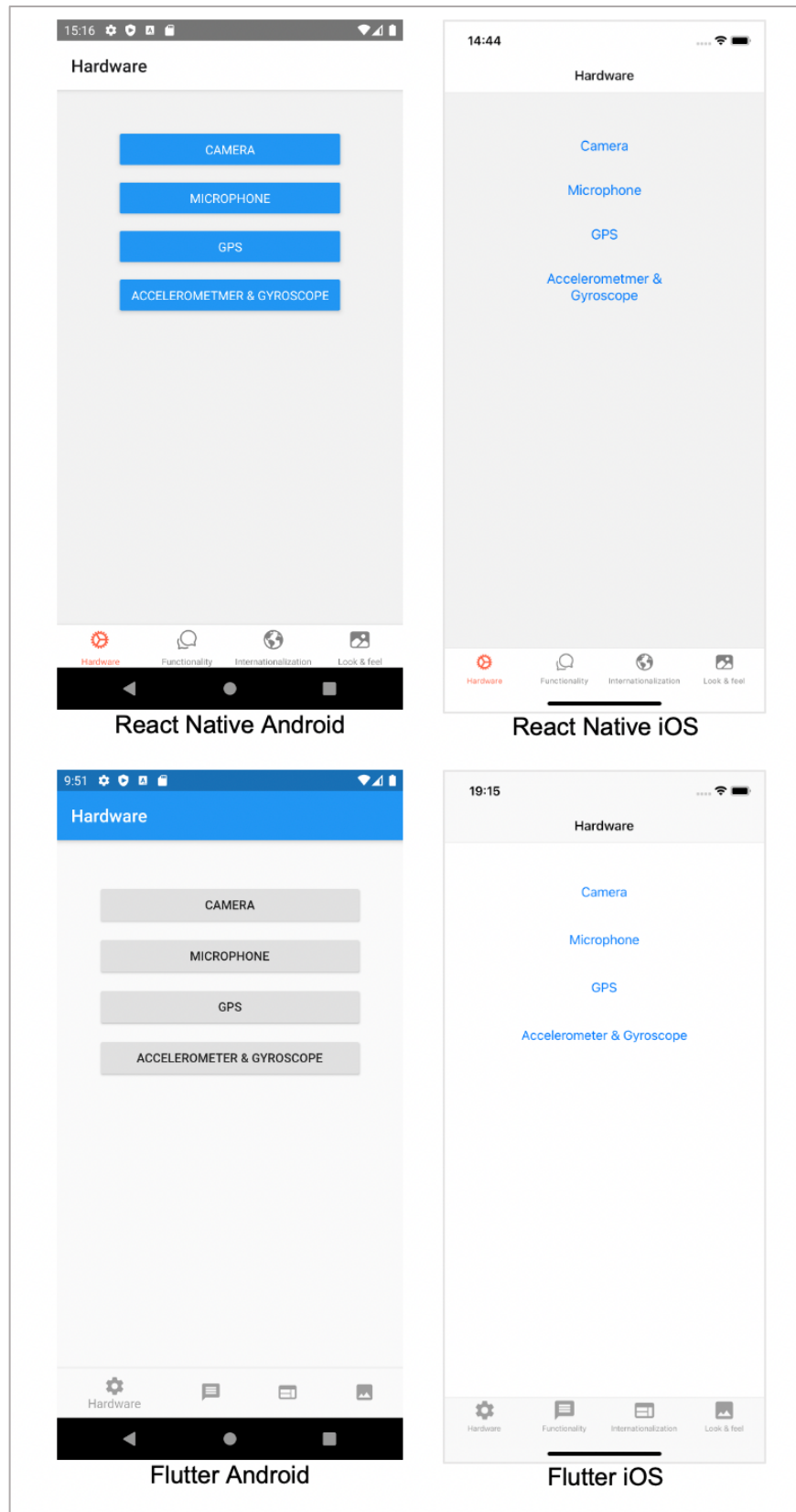


Figure 14 – Comparison of the bottom tab navigator and first view for each framework and platform with the framework's default styling

Push notifications are implemented with Firebase, which is commonly used at Gambit in both native and cross-platform applications. Firebase is a platform for mobile and web application development created by Google. There are quite a few different services available, including cloud messaging, crash logging, and app usage analytics [38]. It is also the officially recommended way of implementing push notifications with Flutter [39][40]. React Native does not seem to have a recommended way of implementing push notifications. The official documentation points to the community's library for push notifications, but only for iOS. There is no information on what to use for Android. However, according to developer forums Firebase seems to be the most popular way of implementing push notifications also for React Native. The notifications can be sent using the GUI in the Firebase console, or through an API.

To evaluate internationalization (A3) translations are implemented in the application. The application should be able to detect the language and country of the device and translate the content accordingly. Another valuable feature is to be able to change the language of the application independent of the device language.

The overall appearance and behavior of the prototype application will be taken into account for evaluation of the look and feel of the framework (A4). In addition to this, a view containing a selection of common UI components with different appearance on Android and iOS will be implemented to assess how well they resemble their native counterparts.

To evaluate the performance (A5), a separate application is created to ensure that anything else is not affecting the performance. This application will only have one view with a scrollable list containing 10 000 items. The CPU activity, memory allocation, and energy usage will be recorded for 20 seconds while manually scrolling through the list at a rapid pace. For comparison, similar native Android and iOS applications are also created. The results from the native applications will serve as a baseline for the evaluation.

6 Evaluation results

This chapter presents the evaluation of the frameworks. The frameworks have been scored based on how well they fulfil each criterion, with reference to the prototype application where applicable. The final result is then calculated as the weighted average score for each framework.

6.1 React Native

6.1.1 Development perspective

D1 License and cost

React Native is free of charge and open source. The framework itself, along with its dependencies, is distributed under the MIT license [41]. This is a very permissive software license which allows private and commercial use, copying, modification, and distribution of the software.

Score: 5

D2 Supported platforms

React Native supports Android and iOS, the two important mobile platforms. Support for other platforms is under development by the community [42]. This could become a valuable feature for streamlining the development process in projects where both mobile and web applications are requested. A React Native application can by default run on both Android and iOS without any manual setup.

Score: 5

D3 Distribution

Applications created with React Native can be distributed through Google Play and App Store. No reports of problems with the distribution caused merely by the application having been created with React Native were found. The Android APK file can be generated and signed with command-line tools, then uploaded to Google Play Store. The iOS version can be

opened in Xcode to archive and upload the application directly to the App Store, just like natively created iOS applications.

Score: 5

D4 Long-term feasibility

React Native has been around since 2015 and can be considered sufficiently mature and stable for commercial production. The use of JavaScript impacts the long-term feasibility positively since it is so widely used in different frameworks, for both mobile platforms and the web. This eliminates the risk of technological lock-in. The framework is backed by Facebook as well as a very large and active community. As of October 2020, there are 88 thousand questions tagged with React Native on the programmer forum Stack Overflow [43], reflecting the substantial user base that the framework has generated over the years. React Native aims at keeping the core library, which is managed by the Facebook team, as lean as possible to better manage it and keep it up to date. This means letting the community take care of extractable modules, such as UI components. It reduces the complexity of the core repository and makes it easier for the community to grow and develop new libraries [44]. The React Native core library is updated frequently, even daily at times [45], and there are plenty of libraries available for functionality that is not included in the core, both by the official community [46] and third-party contributors. These are all indicators that the framework will be actively maintained for a long time. On the other hand, the dependency on voluntary contributors is a weakness, especially if developers start to decline.

Score: 4

D5 Development environment

Android applications can be developed on Windows, Linux, or macOS, as long as Android Studio and the Android SDK are installed. A Mac with Xcode installed is required for iOS development. A new React Native application is created with the React Native CLI (command line interface) [47], a tool that needs to be installed on the development machine. The CLI is also used for building and running the application and can help with

configuration of the development environment. It also includes a hot-reloading feature. The device simulators that come with Android Studio and Xcode can be used to run and test the application. These IDEs are, however, ordinarily not used for writing the source code. There is no officially recommended code editor for React Native, but one of the most popular ones is Visual Studio Code. This is a very versatile and light-weight code editor and was chosen for writing the prototype application. It has plugins to support React Native development and JavaScript code in general, such as code linters (code analysis tools for improving the quality of the source code) and debuggers, but this is not something provided by React Native. There is no visual tool to aid UI development, but the hot reloading is certainly helpful in this aspect. Altogether the above-mentioned tools make for a good development experience, but it is not comparable to the fully integrated native development environments.

Score: 3

D6 Preparation time

The React Native official documentation includes a very helpful getting-started guide listing all the tools needed for development. There are minor variations depending on the development machine and target platform. Downloading the required tools can take a few hours but setting up the development environment is easy with the step-by-step guide. The use of JavaScript and HTML- and CSS-like technologies is once again a huge advantage for React Native. Most developers at Gambit are familiar with web development so React Native would be a good fit in this regard. There is user-friendly documentation for each component explaining the component's props, and most importantly: examples on how to use them. The same goes for the components provided by the community. Additionally, the examples in the core documentation can be seen rendered on each platform, so the developer does not have to implement the component in order to see what it looks like. On the other hand, a lot of content is deprecated or outdated, giving the impression that the core documentation is not very well maintained.

Score: 4

D7 Maintainability

The source code of a React Native application is highly modular. An application is made out of many components where each component is responsible for a lesser part of the functionality or UI. A component can be reused within the application and contain other components. A self-contained component can conveniently be updated without affecting the rest of the application. Third-party libraries are also modules that can be switched out if needed. React Native makes it easy to organize the source code, but in the end, it is up to the developer to write maintainable code. The prototype application contained 1081 lines of code. This exclusively includes code that was written for the application, not code generated by the framework.

Score: 5

6.1.2 Application perspective

A1 Access to device hardware

The prototype application is able to take pictures with the camera, record audio using the microphone, and retrieve the current coordinates of the device from the GPS. These are all libraries provided by the official community. Other sensors can be accessed through third-party libraries. The accelerometer and gyroscope were successfully tested in the prototype application. APIs for using the magnetometer and barometer are available as well, provided they exist on the device.

Score: 5

A2 Access to platform functionality

The web view, image picker, and map view are all libraries from the official React Native community. The image picker can use the camera to take a new photo or open the device gallery to select a photo from there. The map library utilizes Google Maps for Android and iOS MapKit for iOS. The library used for accessing the microphone in A1 makes use of the speech recognition service of the device, so it can be used for converting speech into text. Biometric authentication is available by a third-party library, as well

as a file picker. Push notifications were implemented with Firebase and can be sent from the Firebase console. The Firebase module is provided by a third-party company and is smoothly integrated into the application but does require some manual configuration for each platform.

Score: 5

A3 Internationalization

The React Native official community has a library for localization. It has APIs for fetching the user's preferred locales from the device settings, as well as information on time zone, time format, and if the metric system is used, along with a few other localizable units. By detecting the country and language of the device, different content can be shown for different regions. Text translation is unfortunately not included. Instead, the library's readme suggests using a third-party internationalization library for JavaScript [48]. With this, the developer can define translation strings for text content in the application. The locale to be used is set programmatically and the correct translation is selected accordingly. The locale can be set and changed at runtime, either based on the device language or the user's choice. Natively rendered components such as the date picker are automatically translated to the language of the device by React Native, if nothing else is specified.

Score: 4

A4 Look and feel

Because the React Native components render their native counterparts on the screen, they adapt to the target platform's native style very nicely. All UI components in the prototype application are from React Native's core library or from the official community. Figure 15 shows the main view for testing UI components. There is for the most part no need for adding separate components or styling for each platform. Naturally, if a platform-only component, such as *ToastAndroid* or *ActionSheetIOS*, needs to be implemented it requires some platform-specific code. The segmented control is an iOS-only component and the date/time picker seen in Figure 16 and 17 requires some platform-specific logic. Other than this, the UI components view is implemented with the same code for both platforms.

The whole application looks and behaves as if it were a native application on both Android and iOS. iOS 14 was released in September 2020, and with that came a few significant design changes to some of the UI components. Affected components in the prototype application are the activity indicator, and date/time picker. Since React Native calls native APIs for rendering UI components they are automatically updated to the new design when updating the iOS version of the device, which definitely enhances the native feel of the application.

Score: 5

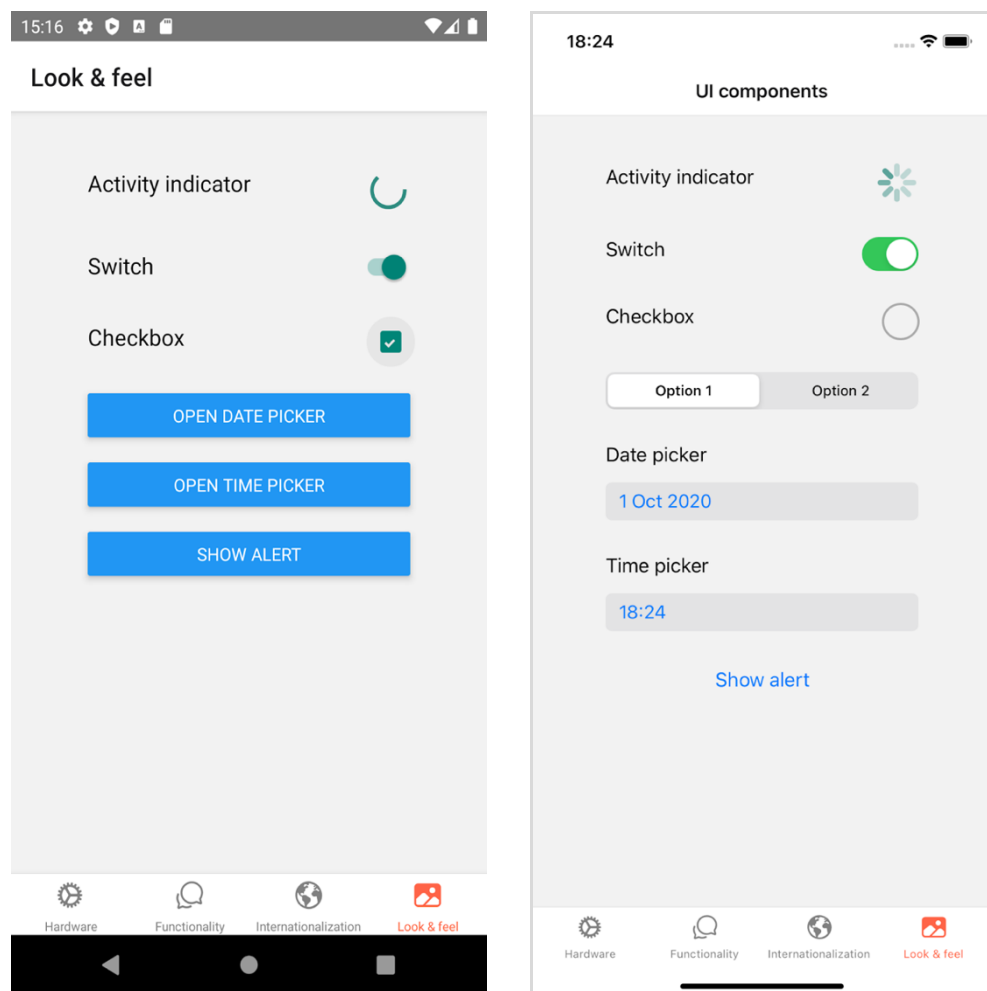


Figure 15 - React Native UI components for Android (left) and iOS (right).

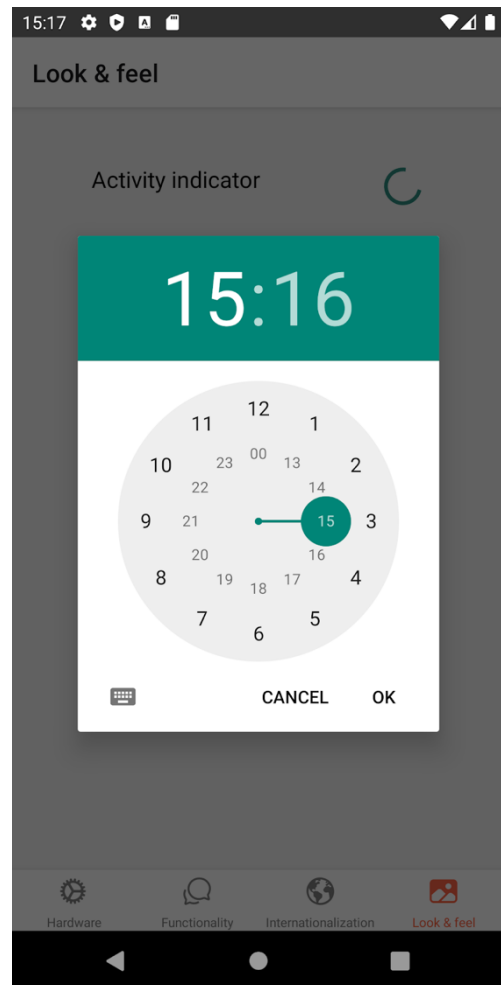
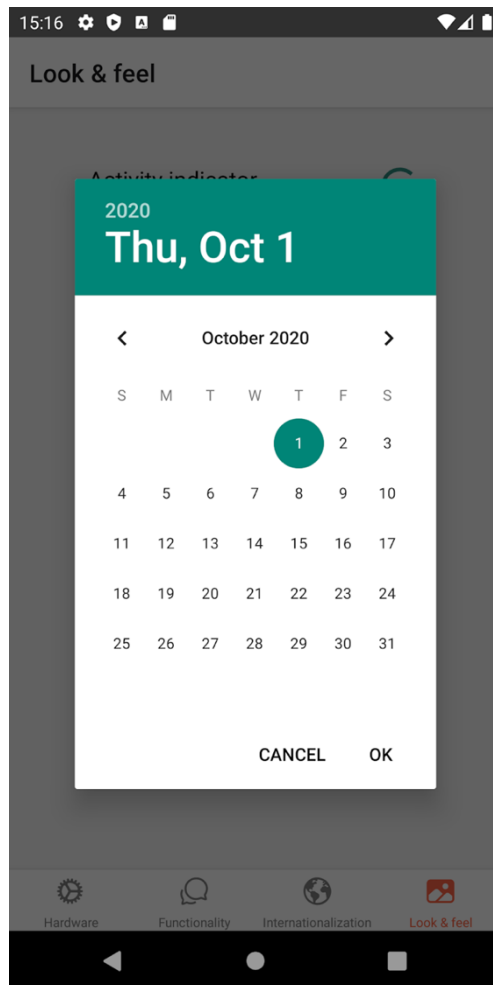


Figure 16 - React Native Android date/time picker.

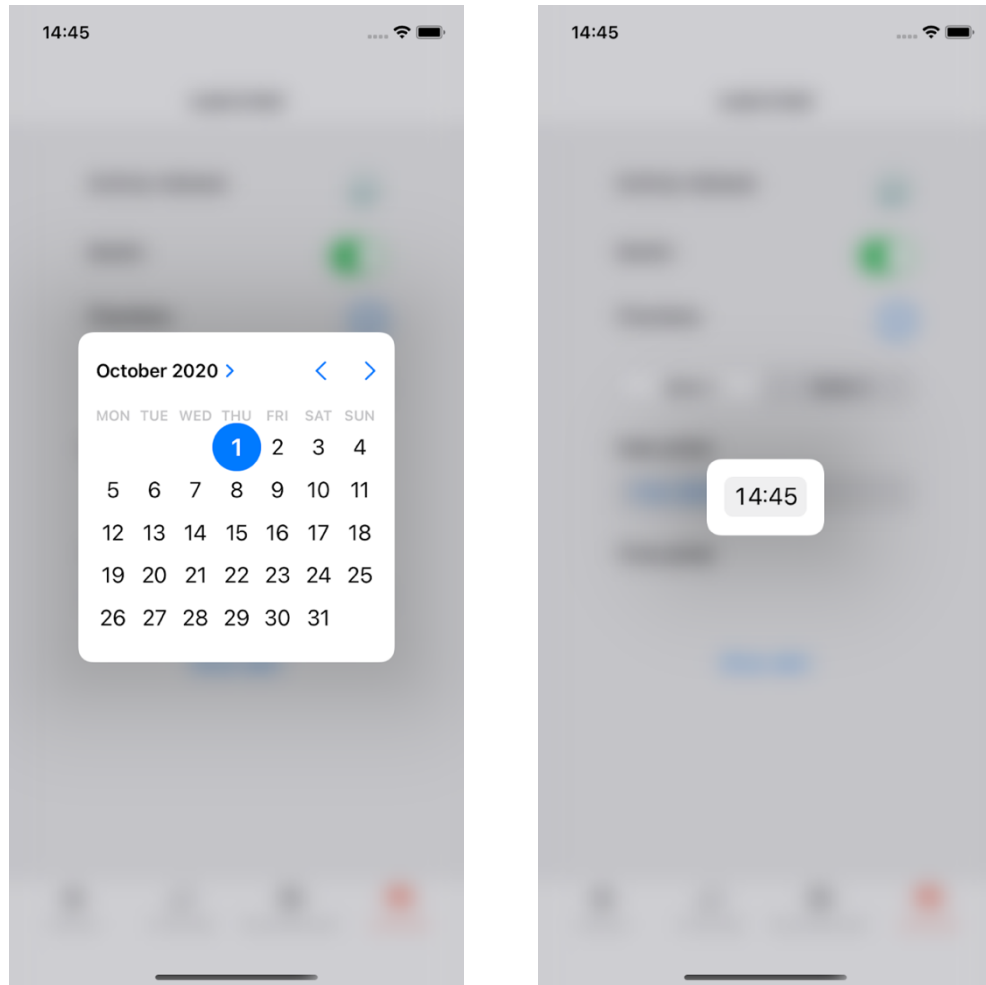


Figure 17 - React Native iOS date/time picker.

A5 Performance

No performance issues were detected with a visual inspection of the prototype application. Navigation was smooth and the application had a fast response time on user interaction. A separate application containing nothing but a scrolling *FlatList* component [49] was used for profiling. The performance testing has been done with developer mode turned off, as is advised by React Native [50]. While scrolling through the list it becomes obvious that the framework cannot quite keep up with the content that has to be rendered. The result is a lagging scroll and sometimes even a completely white screen, on both Android and iOS. The profiling results from the Android application can be seen in Figure 18. The CPU activity is on average around 30% and the average value for memory allocation is 110MB. The profiler does not give a numeric value for energy usage, but it

is classified as 'Light'. Compared to the native Android application (Figure 19) the results are not very good. The native application has a CPU activity of 6% on average, a memory allocation of 75MB, and certainly less energy usage than the React Native application.

Figure 20 displays the results gathered from the iOS application. The Xcode profiler shows a CPU usage of almost 100% on average and a memory allocation of 11MiB. The energy usage was classified as 17/20. The values were very high compared to those of the native iOS application (Figure 21). The CPU usage was 40%, memory allocation at 5MiB, and the energy usage was for the most part at 0 or 1. The performance differences are clear also when using the applications. Scrolling through the list in the native applications is incredibly smooth without any lagging.

Score: 2

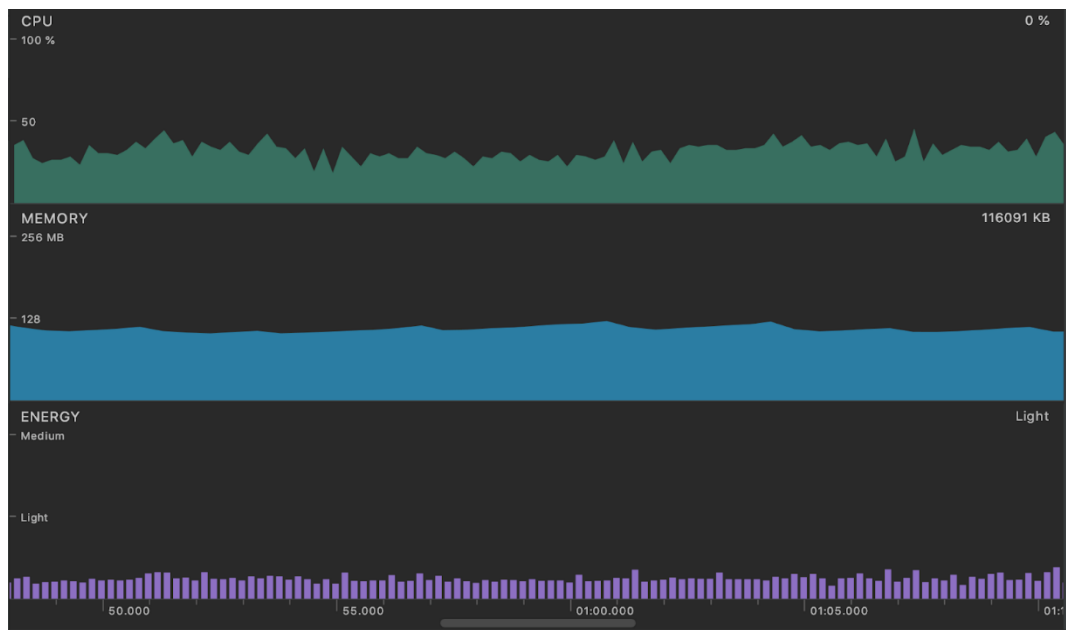


Figure 18 - Profiling results for React Native Android.

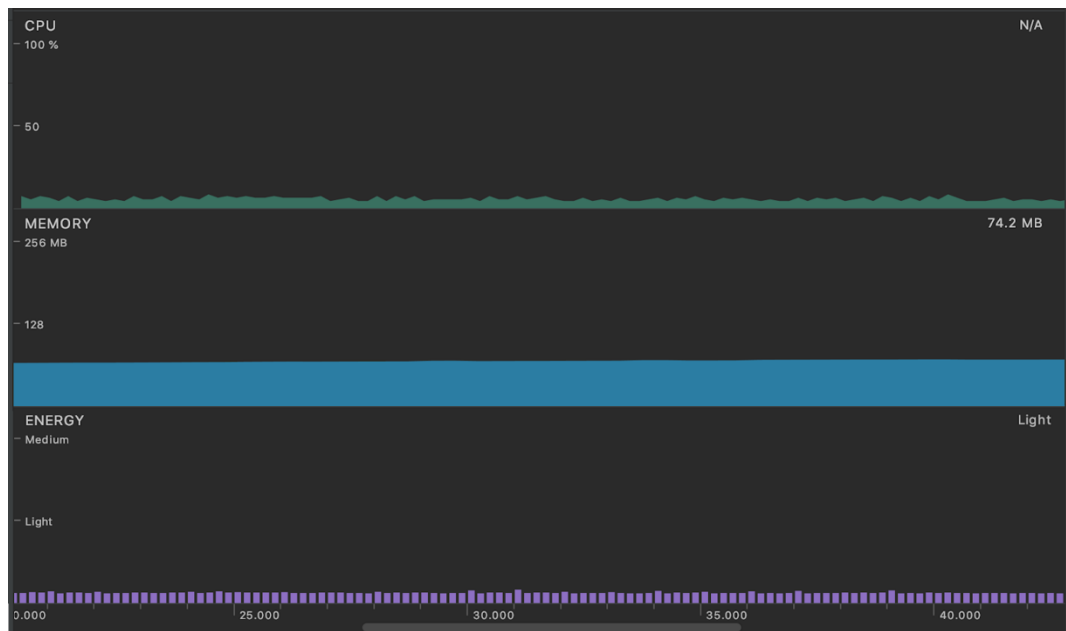


Figure 19 - Profiling results for native Android.

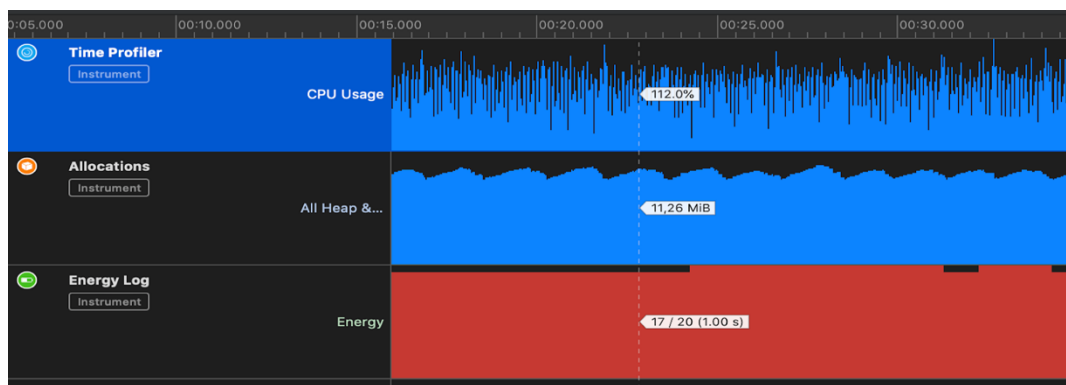


Figure 20 - Profiling results for React Native iOS.

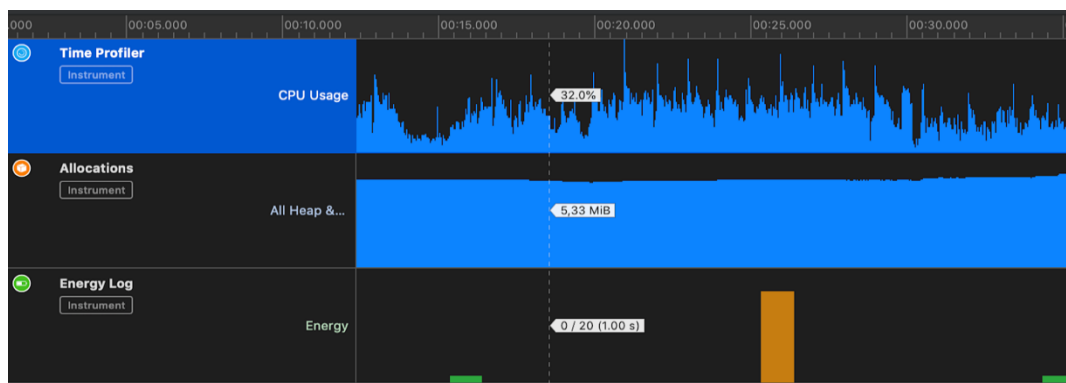


Figure 21 - Profiling results for native iOS.

6.2 Flutter

6.2.1 Development perspective

D1 License and cost

Flutter is free of charge and open source. The framework itself is distributed under the BSD-3 license [51], also known as the “New BSD” license. It is a permissive software license which allows private and commercial use, modification and distribution, provided that the original copyright and license requirement is maintained. It is very similar to the MIT license.

Score: 5

D2 Supported platforms

Just as React Native, Flutter supports the important mobile platforms Android and iOS. Support for the web, as well as desktop applications, is in progress. Flutter applications created with the Flutter CLI include support for both Android and iOS and without any manual configuration required.

Score: 5

D3 Distribution

Flutter applications can be distributed through Google Play and App Store. The process is the same as for React Native. A release version of the application is then generated with the Flutter CLI and the resulting APK-file is uploaded to Google Play Store. The iOS version is opened in Xcode and from there published to the App Store. No reports of problems with the distribution caused merely by the application having been created with Flutter were found.

Score: 5

D4 Long-term feasibility

Flutter has been on the market since 2017. It is a relatively new framework, but created by Google, a company that presumably has the resources to keep the framework running as long as there are users. Flutter and its community have evolved at a fast pace and it has gained an impressive

user base in the relatively short time it has existed. Stack Overflow has 64 thousand questions tagged with Flutter [52] (October 2020), indicating that the ecosystem is not yet fully as established as React Native's. Flutter comes with a very large set of UI widgets, but users are very much dependent on the third-party plugins to get the same functionality as offered by the native frameworks. Both Flutter itself and the community plugins are still under active development, there are updates to the Flutter source code every day [53]. Due to the use of Dart, a relatively uncommon programming language, the risk of technological lock-in is significantly higher for Flutter than for React Native. In the case of Flutter, it is not just the functionality that needs to stay up to date with platform changes. Since Flutter has its own UI components these too need to be maintained. This is a weakness that React Native does not have. Bugs and outdated versions of UI components are especially undesirable, as they are quickly noticed by the end user.

Score: 2

D5 Development environment

Flutter's development environment is very similar to that of React Native. A Mac with Xcode is required for iOS development, and Android applications can be developed on Windows, Linux, or macOS, with Android Studio and the Android SDK installed. Flutter has its own CLI used for creating, building, and running a Flutter application. It also has a hot reloading feature, but no visual UI editor. The simulators that come with Android Studio and Xcode can be used for testing the application. Officially recommended IDEs to use with Flutter are Android Studio, IntelliJ, Visual Studio Code, and Emacs. They all have plugins for Flutter and Dart. Android Studio was used for creating the prototype application. Since it is also created by Google it provides a completely integrated IDE with the use of Flutter and Dart plugins. In practice this means that the developer can manage most development related tasks from within the editor instead of the CLI.

Score: 4

D6 Preparation time

Just like React Native, Flutter has its own step-by-step guide for installing needed tools and creating a new application. Flutter also requires roughly the same tools to be installed for development as React Native. Since Dart is not a common programming language some time might be needed to grow familiar with it. Dart has not previously been used at Gambit so this would mean adding a completely new system to their technology stack. The Flutter documentation is clear and well organized, but somewhat technical and not very beginner friendly. There are examples, and sometimes even interactive displays for the Material widgets, but not for the Cupertino ones.

Score: 3

D7 Maintainability

Similar to React Native's components, the Flutter widgets are the modules out of which a Flutter application is built. Everything, even layout attributes such as padding and margins, can be an independent widget with its own area of responsibility. Widgets can be reused and stacked or nested together to create larger widgets. This means parts of the application can be changed without affecting anything else. The same goes for third-party libraries used in the application. 1045 lines of code were written for the prototype application, which is very similar to the count for React Native.

Score: 5

6.2.2 Application perspective

A1 Access to device hardware

All hardware is successfully tested also in the Flutter version of the prototype application. The camera integration is part of Flutter's own image picker plugin, while the microphone and GPS are accessed using third-party plugins. Flutter provides a plugin for utilizing the accelerometer and gyroscope, which can be used to detect movement of the device. There were no issues with implementation or usage of any of the hardware tested in the prototype, and they worked as expected on both Android and iOS.

Score: 5

A2 Access to platform functionality

The Flutter prototype application implements a web view and an image picker both from Flutter. The image picker has access to the camera as well as the gallery. Flutter has a library for implementing a map view, but it only supports Google Maps, not Apple Maps. Moreover, this functionality is still in “developers’ preview” and might not be suitable for production. There is, however, a platform-agnostic third-party library that is able to use both Google Maps and Apple Maps, but it still builds upon the Flutter map library which is not yet stable. Flutter also provides a library for biometric authentication that works with face and fingerprint identification. Push notifications can for Flutter as well be implemented with Firebase. Firebase is created by Google, so it comes as no surprise that they are compatible. The Firebase documentation contains a guide for integration with Flutter applications.

Score: 4

A3 Internationalization

Flutter has its own way of doing internationalization. It requires more setup but is perhaps more robust than the React Native way. As of 2020, Flutter has support for over 70 languages [54] for their widgets. Supported locales need to be defined in the application and based on the device language the closest matching locale will be selected for the application. Translations for text strings can be managed through the functionality of the *localizations* package, but also by using any other i18n-library. The second alternative is often considered to be simpler. Based on the device language, one of the defined supported locales for the application is automatically chosen. There seems to be no way of retrieving any locale information from the device, other than what is the selected locale for the application.

Score: 4

A4 Look and feel

Flutter’s Material and Cupertino widgets are designed to look and behave as their native counterparts but can technically be used on both Android and iOS. Achieving native-looking elements requires implementing

appropriate widgets for each platform. The UI components view implemented in Flutter is shown in Figure 22, and the Android date picker, almost identical to the native one, can be seen in Figure 23. The Flutter widgets do not adapt to their target platform the same way as React Native components do. The Flutter team constantly needs to update their widgets to stay up to date and will always be slightly behind with changes. This became evident with the release of the iOS 14, where Flutter’s Cupertino widgets in version 1.22.3 of Flutter still follow the old design (Figure 24). There was no iOS-style checkbox widget available for Flutter, which is why it is missing from the iOS-version of the prototype application. Other than that, all tested UI components look and behave as expected.

Score: 3

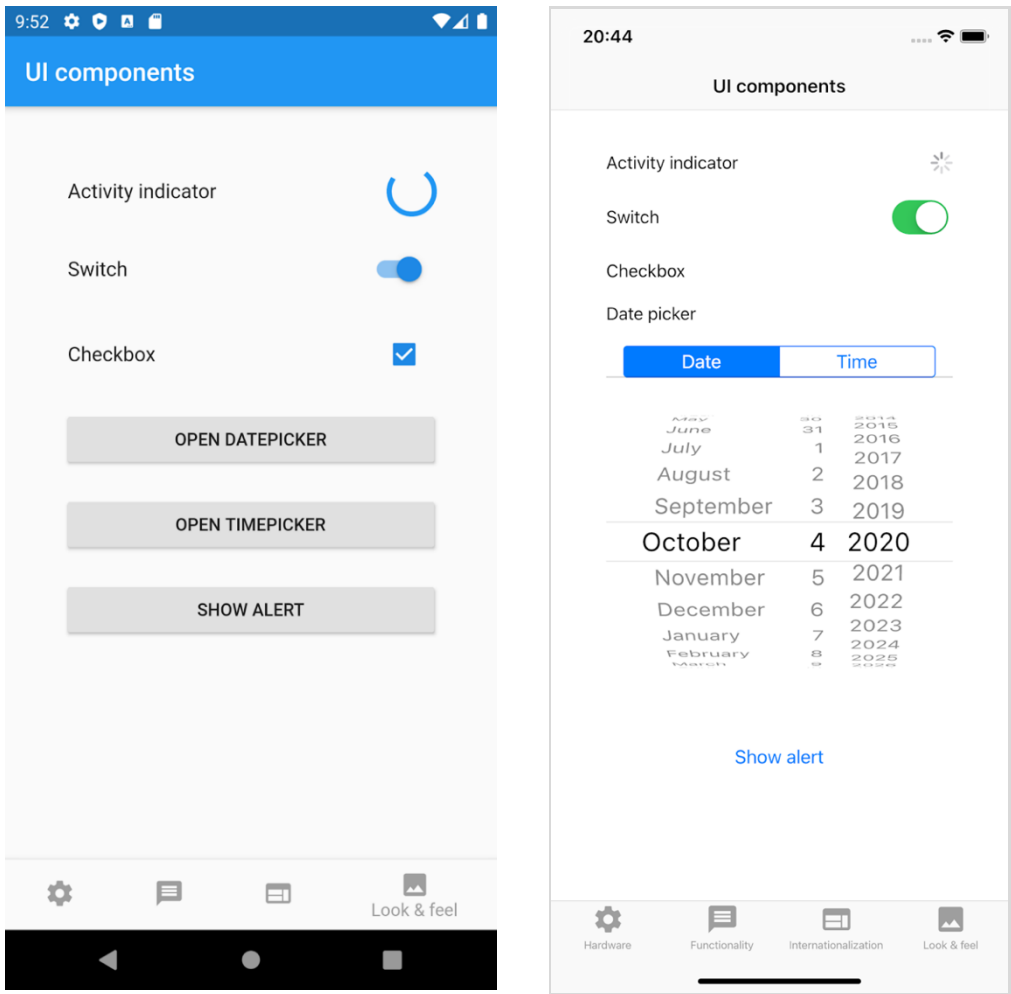


Figure 22 - Flutter UI components for Android (left) and iOS (right).

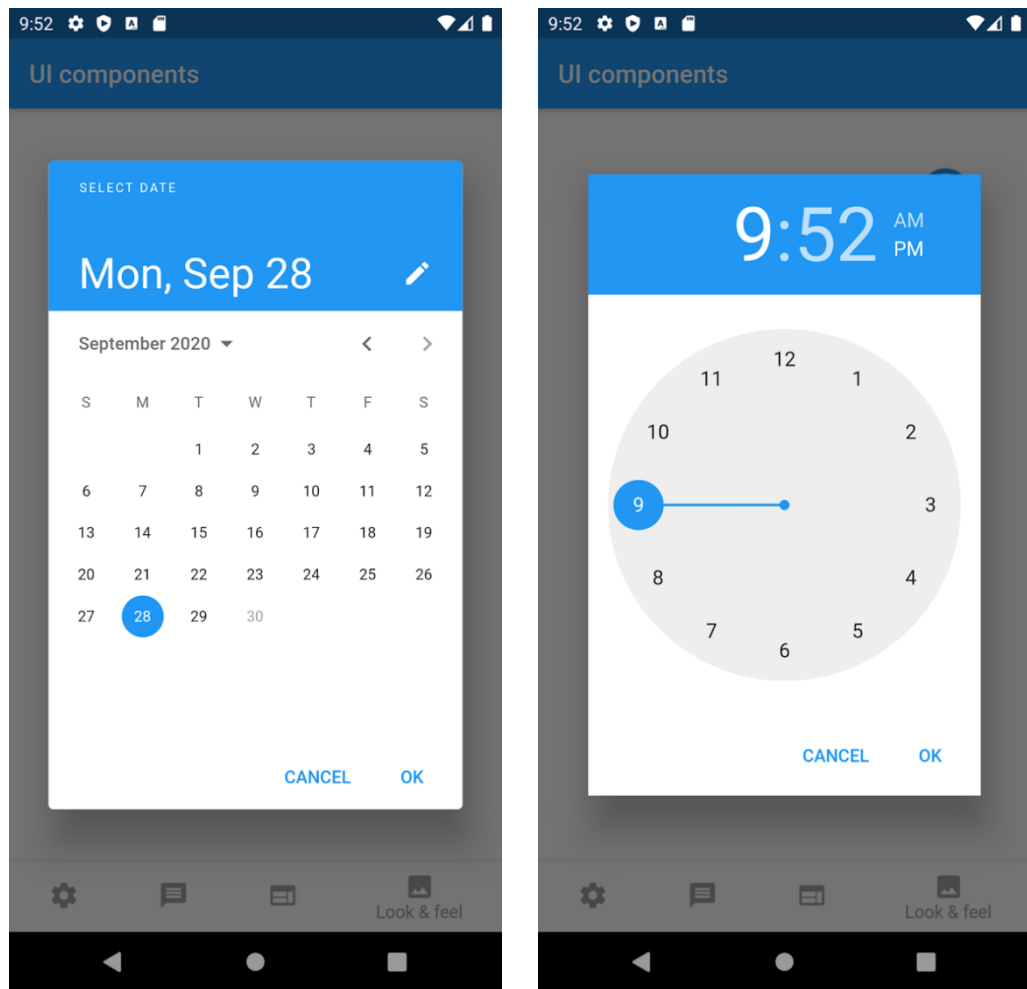


Figure 23 - Flutter Android date/time picker.

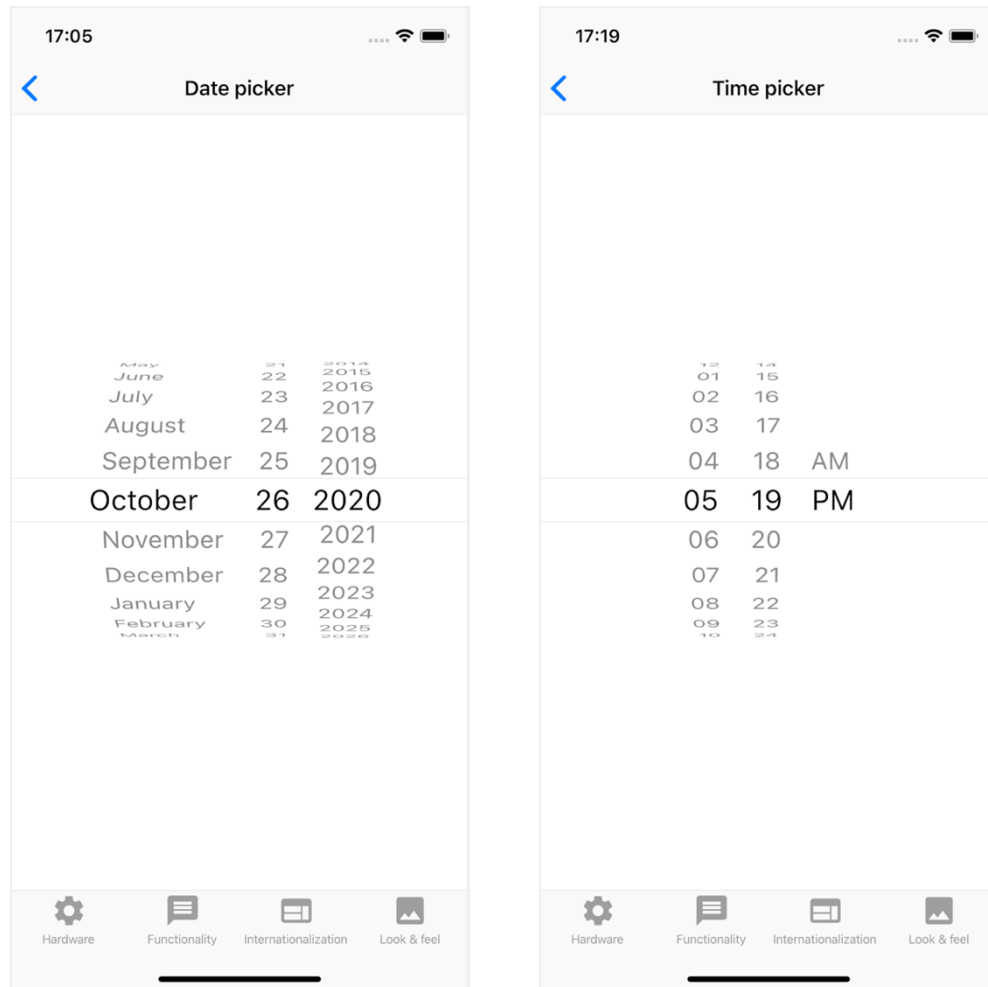


Figure 24 - Flutter iOS date/time picker.

A5 Performance

No performance issues could be seen in the prototype application. To test the performance a separate Flutter application with a *ListView* widget [55] was used. There is a special build mode to be used while testing the performance of a Flutter application [56]. This profile mode was used for running all performance tests. No performance issues were noted during the visual inspection of the application, neither in the Android nor the iOS application. Scrolling was perceived to be just as smooth as in the native applications. The results of the profiling on Android can be seen in Figure 25. The average value for CPU activity was around 9%, and memory allocation at 99MB. The CPU activity was impressively close to that of the native application, but the memory allocation was higher. The energy

consumption was slightly higher than for the native application, but lower than for React Native.

The results from profiling the iOS application can be seen in Figure 26. The recorded CPU usage was on average 40%, memory allocation 33MiB, and power requirement 7/20. The CPU usage is similar to that of the native application, but the memory usage is surprisingly high compared to both the native and the React Native application. The power requirement is higher than that of the native application as well.

Score: 4

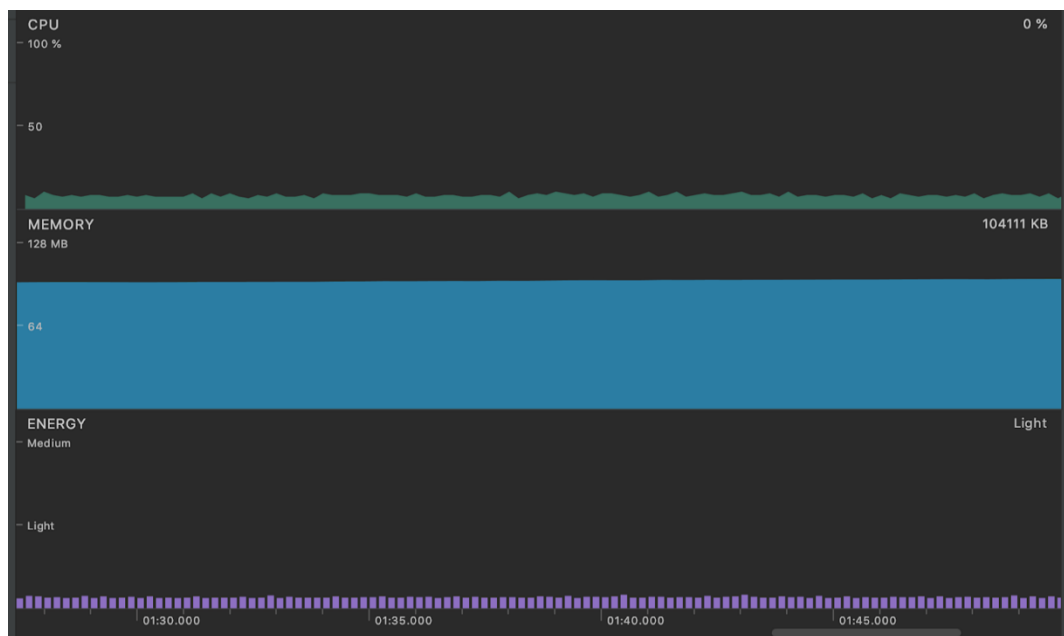


Figure 25 - Profiling results for Flutter Android.

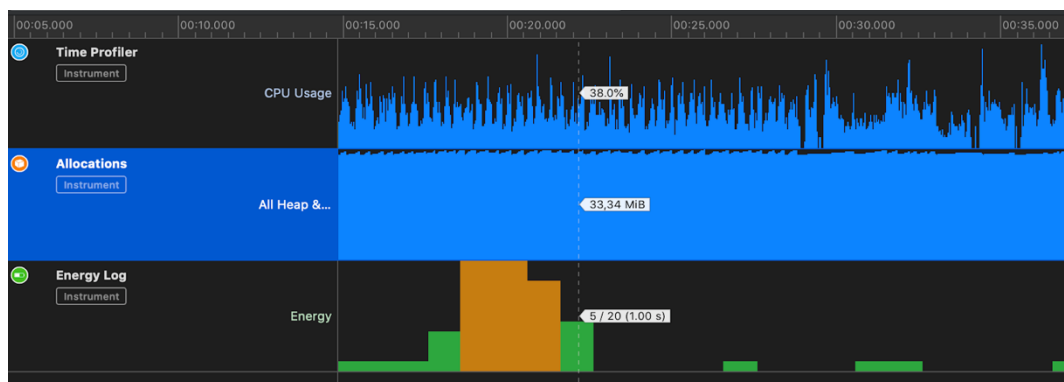


Figure 26 - Profiling results for Flutter iOS.

6.3 Summary

To determine the results of the evaluation the weighted average score is calculated for each framework. The final results are presented in Table 3. Both frameworks did well in the evaluation. React Native received a score of 4.5 and Flutter received a score of 4.1. The frameworks received similar scores for many of the criteria, but it also becomes clear in which areas they differ from each other. Flutter excels in development environment and performance. The fully integrated IDE that is available with Android Studio makes for a more native-like development experience, and the results of the performance testing prove that the performance of Flutter is very good. This is, however, not enough to overcome the advantages of React Native. One of the major factors contributing to React Native's success is that the framework builds upon mature, well-known technologies. This impacts the long-term feasibility and preparation time positively. Another strength of React Native is the usage of the target platform's native rendering APIs, thus, the UI components will always look and feel like their native counterparts. The cost of this is performance, as every API call has to go through the JavaScript bridge.

Table 3 - Evaluation results

Criteria	Weight	React Native	Flutter
D1 Licence and cost	4	5	5
D2 Supported platforms	5	5	5
D3 Distribution	5	5	5
D4 Long-term feasibility	4	4	2
D5 Development environment	2	3	4
D6 Preparation time	3	4	3
D7 Maintainability	2	5	5
A1 Access to device hardware	4	5	5
A2 Access to platform functionality	4	5	4
A3 Internationalization	3	4	4
A4 Look and feel	5	5	3
A5 Performance	3	2	4
Result		4,5	4,1

7 Conclusion

This thesis presents an evaluation of two cross-platform frameworks, React Native and Flutter. The goal was to determine which framework works best for mobile application development at the software company Gambit. The evaluation was done based on a set of weighted criteria derived from typical requirements of mobile applications and the development process of them. The results reveal that React Native is the most suitable framework. React Native got high scores for most of the criteria and can therefore be considered fitting for many different types of applications. Flutter, on the other hand, could be a better option for certain applications where performance is of the essence and a native look and feel is less important. This approach to evaluate frameworks with a set of criteria with weights is highly customizable and can be used for similar evaluations in the future.

Apart from comparing React Native and Flutter the evaluation also proves how far cross-platform frameworks have come in recent years. The resulting applications are well functioning and have user interfaces that closely resemble those of a native Android and iOS applications. The cross-platform frameworks provide a simpler development process than the native frameworks, saving time and costs for development and maintenance. Unfortunately, they will always be one step behind the native frameworks with new features, and there is no guarantee for how long they will last. Therefore, in cases of applications predicted to have a long lifespan, native applications will always be a safer choice.

Swedish summary

Evaluering av React Native och Flutter för multiplattformutveckling av mobilapplikationer

Utveckling av nativa applikationer för flera plattformar är en utmaning för många företag. En nativ applikation är programvara utvecklad för en specifik plattform med plattformens egna ramverk och kan inte installeras på andra plattformar än den tilltänkta [11]. Android och ios är de två mest populära plattformarna. Android utgör 75 % av marknaden, medan ios utgör 23 % [2]. För att attrahera så många användare som möjligt behöver en mobilapplikation vara tillgänglig för båda dessa plattformar. Det innebär utveckling och underhåll av två applikationer med helt olika teknologier, något som kräver tid och kunskap. Multiplattformramverk kan vara en lösning på detta problem. Ett multiplattformramverk möjliggör utveckling av applikationer som kan köras på flera plattformar med samma källkod.

Nativa applikationer anses ofta vara det främsta alternativet för mobilutveckling. De ger den bästa användarupplevelsen med avseende på användargränssnitt och prestanda [21]. Eftersom de utvecklas med plattformens egna verktyg följer de automatiskt plattformsspecifika riktlinjer om utseende och funktionalitet. Nativa applikationer kan direkt ta i bruk eventuell ny funktionalitet som målplattformen erbjuder [11]. Utmaningen med nativ utveckling är att varje plattform har egna verktyg och miljöer för utveckling, dessutom använder de sig av olika programmeringsspråk. Därmed krävs omfattande plattformsspecifik kunskap av utvecklaren.

Med hjälp av multiplattformramverk strävar man efter att förenkla utvecklingen av mobilapplikationer för flera plattformar. Dessa ramverk använder sig ofta av webbutvecklingsteknologier som är bekanta för utvecklare sedan tidigare [37]. Med hjälp av ett multiplattformramverk skapas en applikation som kan installeras på flera plattformar och med samma källkod, vilket underlättar både utveckling och underhåll. Generellt

finns det nackdelar med dessa ramverk. De har ofta sämre prestanda än de nativa ramverken och upplevs inte lika tilltalande för slutanvändaren [40]. Vartefter målplattformarna utvecklas bör ramverken uppdateras, vilket leder till att ny funktionalitet blir tillgänglig med en viss tids fördröjning. Multiplattformverktyg är ofta beroende av tredjepartsbibliotek för funktionalitet, vilket medför en förhöjd risk för buggar.

Denna avhandling är ett samarbete med företaget Gambit. Målet är att utvärdera två multiplattformramverk, React Native och Flutter, för att avgöra vilket av dem som lämpar sig bäst för utveckling av mobilapplikationer för Android och iOS. För att utvärdera ramverkens funktionsduglighet har ett antal kriterier tagits fram. Ramverken poängsätts för hur väl de uppfyller varje kriterium. Varje kriterium har viktats för att visa dess signifikans för evalueringen. Både kriterierna och deras vikter är utformade enligt Gambits behov. Slutresultatet bestäms genom att beräkna det viktade medelvärdet av poängen för varje ramverk.

React Native är ett ramverk skapat av Facebook. Ramverket använder sig av programmeringsspråken JavaScript och JSX (JavaScript XML) [3]. JavaScript är ett ofta använt programmeringsspråk för webbutveckling, och JSX liknar märkspråket XML (extensible markup language) [7]. React Native använder sig av en JavaScript-brygga för att kommunicera med målplattformen. Således kan en React Native-applikation använda sig av funktionaliteten hos enheten, såsom kamera och GPS. Genom bryggan kan React Native också anropa målplattformens API:er (Application Programming Interface) för att rendera komponenter [3]. Användningen av målplattformens egna komponenter bidrar till en nativ känsla för slutanvändaren.

Flutter är skapat av Google och använder sig av programmeringsspråket Dart [19]. Även Dart är skapat av Google och var ett relativt okänt programmeringsspråk innan det introducerades tillsammans med Flutter. En Flutter-applikation kompileras till nativ kod, vilket medför att dessa

applikationer har bra prestanda [18]. Till skillnad från React Native använder sig ramverket inte av målplattformens komponenter, utan har egna komponenter för både Android och iOS som ser ut som sina nativa motsvarigheter [19].

För evalueringen har en prototypapplikation skapats med båda ramverken. Applikationen är utformad för att testa ramverkets åtkomst till enhetens funktionalitet. En annan aspekt är hur väl applikationen till utseendet motsvarar de nativa plattformarna. För att utvärdera ramverkens prestanda skapas en separat applikation för att säkerställa att prestandan inte påverkas av något annat.

Evalueringen visade att ramverken liknade varandra på många punkter. De är gratis att använda och har öppen källkod. Båda är också delvis beroende av frivilliga utvecklare för att upprätthålla dem. Både React Native och Flutter kunde erbjuda i det närmaste samma funktionalitet för sina applikationer som de nativa ramverken. Några aspekter som skiljer ramverken åt är utvecklingsmiljö och prestanda. Flutter-applikationens prestanda är jämförbar med de nativa applikationerna, medan React Native-applikationen hade signifikant högre CPU-aktivitet och minnesallokering. Med hjälp av insticksprogram för Android Studio (den officiella utvecklingsmiljön för Android) kan Flutter erbjuda en utvecklingsmiljö likt de nativa ramverken, vilket underlättar arbetet. React Native är beroende av kommandotolken för att exekvera instruktioner såsom att bygga och köra applikationen, men koden kan skrivas i vilket textbehandlingsprogram som helst. En av React Natives styrkor är användningen av nativa API:er för att rendera komponenter. På detta vis säkerställs att komponenterna alltid ser ut och beter sig som sina nativa motsvarigheter trots uppdateringar av målplattformen. En annan fördel med React Native är tillämpningen av välkända etablerade teknologier, vilket är positivt ur ett långtidsperspektiv och förkortar inlärningstiden för ett nytt verktyg. Resultatet visade att utveckling med React Native ger bäst resultat, även om båda ramverken är möjliga alternativ till nativ utveckling. React Native fick 4,5 poäng och Flutter fick 4,1 poäng av 5 möjliga.

References

[1] Desktop vs mobile market share worldwide

<https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide>. Obtained 15.12.2019.

[2] Mobile operating system market share worldwide

<http://gs.statcounter.com/os-market-share/mobile/worldwide>. Obtained 15.12.2019.

[3] Margaret Rouse, Alexander Gillis, Native application definition, March 2018, <https://searchsoftwarequality.techtarget.com/definition/native-application-native-app>. Obtained 11.05.2020.

[4] Jose Berardo Cunha, Mobile apps runtime architectures, 05.06.2018, <https://www.freecodecamp.org/news/a-deeply-detailed-but-never-definitive-guide-to-mobile-development-architecture-6b01ce3b1528/>. Obtained 11.05.2020.

[5] Android Studio user guide, <https://developer.android.com/studio>. Obtained 11.05.2020.

[6] Material design guidelines, <https://material.io/design/>. Obtained 12.05.2020.

[7] Apple developer portal, Xcode 11 documentation, <https://developer.apple.com/xcode>. Obtained 11.05.2020.

[8] Apple's human interface guidelines, <https://developer.apple.com/design/human-interface-guidelines/>. Obtained 12.05.2020.

[9] HTML standard, <https://html.spec.whatwg.org/multipage/>. Obtained 12.05.2020.

[10] CSS standard, <https://www.w3.org/Style/CSS/>. Obtained 12.05.2020.

[11] ECMAScript 2019 Language Specification, <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. Obtained 12.05.2020.

[12] P. Christensson, JavaScript definition, 08.08.2014, <https://techterms.com/definition/javascript>. Obtained 30.09.2020.

[13] MDN Web Docs Glossary: Definitions of Web-related terms, <https://developer.mozilla.org/en-US/docs/Glossary>. Obtained 11.05.2020.

[14] Web app manifests, 15.04.2020, <https://developer.mozilla.org/en-US/docs/Web/Manifest>. Obtained 11.05.2020.

[15] Bonnie Eisenman, *Learning React Native*. O'Reilly Media Inc., 2015.

[16] Spyridon Xanthopoulos, Stelios Xinogalos, A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications. ACM International Conference Proceeding Series, 2013.
https://www.researchgate.net/publication/258010031_A_Comparative_Analysis_of_Cross-platform_Development_Approaches_for_Mobile_Applications. Obtained 23.10.2020.

[17] P. Christensson, Framework Definition. 07.03.2013.
<https://techterms.com/definition/framework>. Obtained 30.09.2020.

[18] Google Trends, Comparison of React Native and Flutter, <https://trends.google.com/trends/explore?cat=5&date=today%205-y&q=Flutter,React%20Native>. Obtained 7.7.2020.

- [19] Facebook's React Native gets backing from Microsoft and Samsung
<https://program.developer.samsung.com/2016/04/21/facebook-s-react-native-gets-backing-from-microsoft-and-samsung>. Obtained 15.12.2019.
- [20] JSX Specification, <https://facebook.github.io/jsx>. Obtained 11.5.2020.
- [21] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau. XML specification. 26.11.2008. <https://www.w3.org/TR/xml>. Obtained 12.05.2020.
- [22] React official documentation, <https://reactjs.org/docs>. Obtained 11.05.2020.
- [23] Eric Masiello, Jacob Friedmann, *Mastering React Native*. Packt Publishing Ltd., 2017.
- [24] Chris Minnick, The Real Benefits of the Virtual DOM in React.js, 19.04.2016, <https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/>. Obtained 11.05.2020.
- [25] Flutter official documentation
<https://flutter.dev>. Obtained 06.05.2020.
- [26] Flutter 1.0 released: The first stable release of Google's mobile UI toolkit, 4.12.2018, <https://www.androidauthority.com/google-flutter-1-0-released-931190/>. Obtained 11.05.2020.
- [27] Ivo Balbaert, Dzenan Ridjanovic, *Learning Dart*. Packt Publishing Ltd., 2013.
- [28] Ivo Balbaert, Sergey Akopkokhyants, Davy Mitchell, *Dart: Scalable Application Development*. Packt Publishing Ltd., 2017.

[29] Gilad Bracha, *The Dart Programming Language*. Addison-Wesley, 2015.

[30] Dart official documentation, <https://dart.dev/guides>. Obtained 11.05.2020.

[31] Marco L. Napoli, *Beginning Flutter: A Hands On Guide To App Development*, Wrox 2019.

[32] Ed Freitas, *Flutter succinctly*. Syncfusion, Inc., 2019.

[33] Flutter official YouTube channel, How Flutter renders Widgets, 15.11.2019, <https://www.youtube.com/watch?v=996ZgFRENMs>. Obtained 11.05.2020.

[34] Henning Heitkötter, Sebastian Hanschke, Tim A. Majchrzak, Evaluating Cross-platform Development Approaches for Mobile Applications. WEBIST 2012.
<https://www.semanticscholar.org/paper/Comparing-Cross-platform-Development-Approaches-for-Heitkötter-Hanschke/4f7f0bc65c86d09e1138a4d9ffd51f165ba7acdb>. Obtained 23.10.2020.

[35] Sommer, A., Krusche, S., Evaluation of cross-platform frameworks for mobile applications 2013.
https://www.researchgate.net/publication/259852798_Evaluation_of_cross-platform_frameworks_for_mobile_applications. Obtained 23.10.2020.

[36] Christoph Rieger, Tim A. Majchrzak, "Towards the definitive evaluation framework for cross-platform app development approaches", *Journal of Systems and Software*, Volume 153, 2019.
<https://www.sciencedirect.com/science/article/pii/S0164121219300743>. Obtained 23.10.2020.

[37] Heitkötter, Henning & Majchrzak, Tim A. & Ruland, Benjamin & Weber, Evaluating Frameworks for Creating Mobile Web Apps. WEBIST 2013 - Proceedings of the 9th International Conference on Web Information Systems and Technologies. 209-221.
<https://pdfs.semanticscholar.org/59e2/950d74d231eaa6889d346b3b7ba7823446d4.pdf>. Obtained 23.10.2020.

[38] Firebase official documentation, <https://firebase.google.com>. Obtained 25.09.2020.

[39] Flutter official documentation, Flutter for iOS developers, <https://flutter.dev/docs/get-started/flutter-for/ios-devs#notifications>. Obtained 25.09.2020.

[40] Flutter official documentation, Flutter for Android developers, <https://flutter.dev/docs/get-started/flutter-for/android-devs#notifications>. Obtained 25.09.2020.

[41] React Native License <https://github.com/facebook/react-native/blob/master/LICENSE>. Obtained 30.09.2020.

[42] React Native official documentation, Supported platforms. <https://reactnative.dev/docs/out-of-tree-platforms>. Obtained 30.09.2020.

[43] Stack Overflow, Questions tagged with React Native. <https://stackoverflow.com/questions/tagged/react-native>. Obtained 2.11.2020.

[44] React Native Official documentation, Lean Core, <https://reactnative.dev/blog/#lean-core>. Obtained 30.09.2020.

[45] React Native Core GitHub page. <https://github.com/facebook/react-native/graphs/commit-activity>. Obtained 2.11.2020.

[46] React Native Community GitHub organization page,
<https://github.com/react-native-community/.github>. Obtained 30.09.2020.

[47] P. Christensson, Command Line Interface Definition. 26.08.2014.
https://techterms.com/definition/command_line_interface. Obtained 30.09.2020.

[48] Nando Viera, I18n.js Github repository. <https://github.com/fnando/i18n-js>. Obtained 30.09.2020.

[49] React Native official documentation, FlatList.
<https://reactnative.dev/docs/flatlist.html>. Obtained 30.09.2020.

[50] React Native official documentation, Profiling. <https://reactnative.dev/docs/profiling>. Obtained 30.09.2020.

[51] Flutter license. <https://github.com/flutter/flutter/blob/master/LICENSE>. Obtained 30.09.2020.

[52] Stack Overflow, Questions tagged with Flutter.
<https://stackoverflow.com/questions/tagged/flutter>. Obtained 2.11.2020.

[53] Flutter GitHub page, <https://github.com/flutter/flutter/graphs/commit-activity>. Obtained 2.11.2020.

[54] Flutter official documentation, Internationalizing Flutter apps.
<https://flutter.dev/docs/development/accessibility-and-localization/internationalization>. Obtained 30.09.2020.

[55] Flutter official documentation, ListView class.
<https://api.flutter.dev/flutter/widgets/ListView-class.html>. Obtained 30.09.2020.

[56] Flutter official documentation, Flutter's build modes.

<https://flutter.dev/docs/testing/build-modes#profile>. Obtained 30.09.2020.