Tampereen yliopisto

Joni Tuominen

EVALUATION OF FLUTTER AS A MIGRATION TARGET

Faculty of Information Technology and Communication Sciences
Master's thesis
January 2021

ABSTRACT

Joni Tuominen: Evaluation of Flutter as a migration target
Master’s thesis
Tampere University
Master's Programme in Computer Science
January 2021

This thesis evaluated Flutter, a cross-platform framework, as a potential migration target for mobile application development. The evaluation was done from the perspective of Piceasoft Ltd, a client of the thesis. While evaluation was done from Piceasoft’s perspective, the overall perspective remained generic. The evaluation was based on selection criteria discovered in existing related research, in addition to requirements set by Piceasoft. The evaluation emphasized communication with natively developed libraries, but other aspects of the framework were also examined.

Motive for the evaluation was to determine Flutter's suitability for long-term cross-platform mobile application development. Flutter’s common codebase for Android and iOS applications is expected to simplify the development process, improve maintainability, and reduce workload of developers.

The outcome of the evaluation was that Flutter is fulfilling requirements set by Piceasoft for most aspects. Flutter contains all required aspects to develop versatile applications that are capable of utilizing native Android and iOS libraries. In addition to the capability to utilize native libraries, Flutter provides a rich set of user interface components and a relatively easy-to-learn development language. Flutter’s common codebase, user interface components, and resource/localization management would potentially lead to reduced workload, increased maintainability, and increased similarity on both platforms. While fulfilling most of the requirements, a long-term evaluation of Flutter’s operating system support could be recommended. Overall, Flutter can be recommended for application development for Piceasoft, but it could be safe to start development with Flutter on smaller applications or projects.

Keywords: Mobile application, Mobile development, cross-platform, Flutter, Android, iOS

Table of contents

1 Introduction

Mobile devices are widespread and come in different forms. A large proportion of mobile devices are mobile phones, but often also tablet computers and, to some extent, smart watches and wristbands. There are 5.25 billion (rounded) unique mobile devices globally (GSMA Intelligence, 2021). Based on registered subscriptions, the number of devices is higher when devices without a subscription are counted (GSMA Intelligence, 2021). Mobile phones and tablets today are divided mainly into Android and iOS operating systems. The Android operating system covers about 71.99 percent of all mobile devices, and iOS devices account for about 27.42 percent of mobile devices (Statcounter GlobalStats, 2021).

Methods, technologies, and tools used for developing mobile applications depend somewhat on the mobile device's operating system. Mobile operating systems have their own development methods and development languages. Native Android applications are programmed with Java or Kotlin, while iOS applications are written in Swift or Objective-C; both platforms can also be developed with C and C++ languages. Significant differences in these development tools bring challenges to application development if the same application is to be developed for both platforms.

In many cases, mobile application developers publish their applications on both platforms via their respective application stores. Sometimes, these applications are developed using some cross-platform framework that allows the application to be developed using the same technologies and tools for both platforms. For example, the mobile applications of Facebook, Instagram, Discord, and Baidu have been developed using cross-platform frameworks/development tools such as React Native or Flutter (Flutter, 2021a. Facebook Engineering, 2016. Discord, 2019. Instagram Engineering, 2017).

Cross-platform frameworks and toolkits are technologies that depending on the technology, provide the functionality, methods, and tools to implement applications on multiple platforms using the same source code. Several different frameworks are available for mobile platforms that can be used to develop an application across Android and iOS platforms. Commonly used and well-known frameworks for mobile platforms currently are React Native, Flutter, Apache Cordova, Ionic, and Xamarin, according to a developer survey by Statista (Statista, 2020).

These frameworks provide tools for user interface development. They variously include libraries and functionality for other software development needs, such as network requests, file management, additional UI components, and application dependency management. Development with application frameworks is often done using a programming language different from the language used in native development. For example, frameworks mentioned earlier utilize JavaScript, Dart, or C # depending on the framework (React Native, 2021. Flutter, 2021b. Microsoft, 2021). In addition, some frameworks use markup languages, such as HTML and CSS, as in the case of Apache Cordova (Apache Cordova, 2021).

This thesis evaluates suitability of Flutter cross-platform framework as a potential migration target. Flutter is evaluated for a scenario where native mobile application development is intended to be replaced partially. This thesis examines development restrictions and requirements for cross-platform development and their fulfillment with Flutter. Research methods for this thesis are testing/prototyping with Flutter application development, examination of literature, research, and documentation related to Flutter, and cross-platform mobile application development.

The client for this research is Piceasoft Ltd. Piceasoft, a company founded in Tampere in 2012 that produces solutions focused on the lifecycle management of mobile devices. The company produces software products that can perform mobile device diagnostics, device erasure, data transfer, and implement mobile device resale. Some of the issues regarding the suitability and feasibility will be discussed from Piceasoft's perspective, but the overall perspective will remain generic.

Research questions for Flutter evaluation are following:

Is Flutter cross-platform framework suitable for long-term application development with existing native libraries?

In this case, suitability is affected by multiple factors, for example, availability of required functionality, supported operating system versions, licensing. One of the main factors is Flutter's ability to communicate with native libraries since Piceasoft uses multiple in-house developed native libraries. Long-term suitability is affected by the frequency of updates for the framework. Criteria for suitability are discussed in the criteria and requirements for the framework selection chapter.

Does it reduce workload of development?

Reduction of workload is one of the main research questions since the potential reduction of workload with a cross-platform framework is one of the expected advantages of cross-platform development. The actual reduction of workload may be difficult to measure in the short term, but shared codebase, resource management, continuous integration, and testing are expected to reduce workload. Therefore, reduction is examined via these attributes with Flutter.

What are the challenges and restrictions to cross-platform development with flutter?

In addition to resolving known challenges and restrictions related to migration to Flutter framework, the aim is also to discover new issues if such exists. Therefore, it is essential to understand all potential issues related to the development with Flutter framework before starting technology adoption. Discovery of issues can affect potential migration or the actual development process with the framework.

This thesis consists of 9 chapters. The first chapter contains an introduction to the thesis. The second chapter examines motivation and different kinds of approaches for cross-platform application development. Approaches are identified from previous research. The third chapter examines selection criteria for cross-platform framework selection based on existing research. Selection criteria are examined from Piceasoft's perspective while maintaining a certain degree of genericity. The fourth and fifth chapters contain a brief introduction to both native and Flutter application development. Chapters 6 and 7 discuss Flutter framework evaluation results based on research questions, motivations, and selection criteria. Chapter 8 briefly mentions potential topics for further research. Chapter 9 contains a list of referenced studies, articles, and documentation.

2 Motivation and approaches for cross-platform development

2.1 Motivation

In native mobile application development, the development is often done separately for both Android and iOS platforms. Development for both platforms can result in a time and money-consuming development process since the application is required to be implemented separately for both platforms. Cross-platform development can potentially have a positive impact on issues mentioned earlier. The following rationale for cross-platform development is Piceasoft specific but can also apply generally.

Improvements to development process

A cross-platform framework is expected to improve development process mainly by improving maintainability through a common codebase for the UI part of the application. User interface codebase can potentially comprise a considerable part of the application. While it may not be feasible to have a common codebase for all parts of the application, having a common codebase on the UI part on both platforms could have a positive effect on the development process. Testing is also expected to be simplified by having one codebase and user interface to be tested.

Storyboards on the iOS platform are XML files that contain user interface implementation of natively developed iOS applications. These storyboard files are edited via Xcode IDE's Storyboard editor with a graphical user interface, and changes in the editor are reflected in the Storyboard file. While the Storyboard file can be displayed as an XML file, it may be hard to develop a user interface by editing XML since Apple does not provide any documentation for the keys and values used in the Storyboard file. In addition, Xcode also modifies some values of the Storyboard file automatically. Automatic changes, hard-to-read XML, and multiple developers working with the same Storyboard file can result in version control conflict, which can be difficult to merge. A cross-platform framework is expected to reduce these conflicts on the iOS platform via a completely different UI development method. At Piceasoft, this has been an issue, with multiple developers working on a single Storyboard file; while it does not always result in a conflict, it happens often enough to be considered an issue. On Android, this issue does not exist since it uses XML files for UI development with distinct tags and attributes with extensive documentation. Both Android and iOS UI development will be further discussed in the native mobile application development section.

Improvements to overall quality of software

Developing the same application separately for multiple platforms can introduce several issues to the development process. When an application is developed with the same architecture for both platforms, there is a risk of the disparity in the codebase of applications. The disparity may not have any effect on the functionality of the developed application. However, it complicates the maintenance of the codebase by having two different kinds of implementations. With one architecture approach, it is preferable to aim for similar implementation on both platforms to improve maintainability and simplify further development.

With separate projects for both platforms, there is a possibility of mismatches in localization texts and other resources. Different platforms may use different formats for localization string. By managing multiple localization files and formats, there is a possibility for a mismatch in localization keys and values. Managing multiple localization resources in multiple places can also result in outdated or missing localisations by mistake. A similar issue can happen with other resources, such as images and colour codes.

Other issues related to maintaining similarity may arise from platform-specific user interface components. For example, Android has built-in implementation for check box UI element, while iOS does not and prefers to the use of switch UI elements instead. While having different kinds of UI elements for the same tasks may not impact functionality, it impacts user experience and usability. Having the same look and feel for the application on different platforms with native UI elements may not be possible for some companies. Maintaining the same user interface can be an important requirement for application development.

Reduction of work times and development costs

Development of application separately for both Android and iOS platforms can be costly and time-consuming, depending on the size and complexity of the project. Costs tend to be higher because development needs to be done twice; this could be reduced by migrating to a cross-platform framework to remove the need to develop separately for both platforms. When an application does not need to be developed twice, resources could be allocated to other tasks or reduce the workload of developers and testers. Common codebase could reduce times to fix bugs if bugs are present on both platforms. Changes to the user interface could also be implemented in a shorter timeframe. Having only one UI implementation could also simplify implementing UI testing, such as automation tests on the user interface.

2.2 Approaches

Cross-platform development refers to a development method in which application development is done with a single codebase that is used on several different platforms. Cross-platform development approaches often focus on user interface development, providing various libraries, user interface components, or APIs for user interface development. However, depending on the implementation, the cross-platform frameworks may provide functionality other than user interface components or related interfaces, such as functionality related to mobile device features or components. Device-related functionalities can include, for example, APIs to sensors or other device features such as a camera.

In their survey of cross-platform mobile development, Biørn-Hansen, Grønli, and Ghinea (2018) provide a taxonomy of different approaches used to implement cross-platform solutions. Approaches mentioned in the article are hybrid, Interpreted, cross-compiled, model-driven, and progressive web applications. This thesis examines cross-compiled approach with Flutter framework as a potential migration target.

Cross-platform frameworks can be divided into the approaches mentioned above. Approaches can differ significantly in terms of their implementation and functionality. There may be differences in performance as well as usability.

2.2.1 Hybrid approach

The hybrid approach uses traditional web technologies such as HTML, CSS, and Javascript. With these technologies, hybrid cross-platform frameworks implement their user interface and operational logic. Parts implemented with web technologies are bundled into a native application and then displayed in the native web view. Web view refers to a UI component that displays a website in an application without the need to open a browser application. Some frameworks provide two-way communication with the native parts of the application; this is called bridging, allowing the use of native features. (Biørn-Hansen, Grønli, Ghinea, 2018)

2.2.2 Interpreted approach

The interpreted approach builds a native interface utilizing interpretable languages. While the interpreted approach may take advantage of web development-specific tools, such as JavaScript, it does not need a web view for the presentation of the application. Instead, interpreted approach renders a native interface based on the source code written in the interpretable language with an on-device interpreter, such as a Javascript interpreter. Although JavaScript is a common language in the interpreted approach, it is not the only

option. The Interpreted approach also uses a bridging technique so that native functionalities can be used in the application. (Biørn-Hansen, Grønl, Ghinea, 2018)

2.2.3 Cross-compiled approach

In the cross-compiled approach, the application is written in a common language for both platforms, and this common source code is translated into native byte code. As the application is compiled into a native application, there is no need to use a web view or interpreters, like in hybrid and interpreted approaches. The cross-compiled approach also does not need a bridging layer to utilize native device features or functionalities since these features are exposed via the framework's SDK, with implementations that are mapped to the native SDK's implementations. The cross-compiled approach also generates components from byte code that are rendered as native components. (Biørn-Hansen, Grønli, Ghinea, 2018)

2.2.4 Model-driven approach

The model-driven approach differs significantly from the above. It utilizes domain-specific languages in the development and does not require the developer to know the programming languages used by the platforms. In the model-driven approach, the application is modelled through user interface models and business logic. After modelling, the model can be transformed into an application for different platforms. The models are translated into native applications in application generation, i.e., the generator develops the application model. (Biørn-Hansen, Grønli, Ghinea, 2018)

2.2.5 Progressive web application approach

The progressive web application approach uses web technologies for application development with enhanced capabilities. Applications or websites can be developed to have a native-like look and feel with HTML and CSS languages. Web sites can be used via an internet browser. Progressive web applications can be installed on the device similar to regular applications, and these applications download all the resources required for offline use during installation. The progressive web application approach uses service workers introduced in iOS 11.3 version to enable these applications to function with additional capabilities compared to traditional web apps. Service workers were introduced in iOS 11.3 version, and on Android, they are included in API level 24. With progressive web application approach, applications are opened on the device as artifact-less browser windows with downloaded resources, without browser's address bar, setting icons, or other browser-related UI elements.

3 Criteria for framework selection and evaluation

Adopting a new technology always requires that the requirements related to the new technology are adequately recognized and understood. The requirements for technology often create constraints that need to be prepared for before the migration or adoption for production to run smoothly. If the new technology does not meet the requirements or creates too many constraints for the development, then the migration or adoption may not be worthwhile.

Choosing the correct framework is influenced by several different things. When choosing a framework, the constraints on the development set by the framework must be considered. In addition to the limitations, it is good to assess the requirements for the software and the product requirements set for it. In the article How to Pick the Right Mobile Development Approach? Haire proposes the following criteria for the selection of the framework:

1. Does it require hiring additional skills?
2. Walled garden or open ecosystem?
    - Walled garden refers to low-code development platforms, where "everything you need is contained within the platform and controlled and licensed by the vendor." (Haire, 2021) For example, Kony, Mendix, and Outsystems platforms. In the walled garden approach, application development is usually done via a platform-specific graphical editor, which only contains functionality implemented by the vendor.
    - Open ecosystems refer to platforms that provide SDKs, which can be used to implement functionality without boundaries.
3. What kind of partner are you looking for?
    - Does it provide support for development?
    - Does it provide long-term versions?
4. Importance of design and UX consistency across teams and projects?
5. Evolution of ecosystems?

The article Framework Choice Criteria for Mobile Application Development examines the essential criteria related to selecting a cross-platform framework and discusses the fulfillment of the criteria by approaches of cross-platforms frameworks. The article examines the criteria proposed by Andrew Haire and suggests the following additional questions as criteria for choosing a framework:

1. Does it need access to native mobile features?

2. Does it need high performance?
3. Will it be published as a free app or under a license?
4. What are the targeted platforms?
5. Does it need support after publishing it?
6. Does it need a native look and feel?
7. Does it need to consume little power from the phone battery?
8. Does it have to implement an advanced security standard?
9. Does it need to be scalable?

Flutter was chosen as the cross-platform UI framework for this thesis. Flutter was selected for a variety of reasons, including extensive documentation and similarity of Dart programming language to languages used in native mobile application development. Also, UI development was considered similar to Apple's SwiftUI. Piceasoft requires open ecosystems for software development without limitation to development, and Flutter is seen as a potential migration target from this perspective. This thesis focuses on the mobile development aspect; therefore, UX consistency is between mobile projects and projects on other platforms is not discussed. UX consistency between mobile platforms on the same project is expected to be a feature of Flutter.

From the long-term partner perspective, Google is seen as a reliable developer for such a framework due to the size of the company and Android development. Ecosystem evolution is somewhat hard to predict in the mobile device industry, and huge changes can happen in a fairly short time. Good examples of huge changes are the development, decline, and discontinuation of Symbian and Windows Phone operating systems.

3.1 Native mobile features

Piceasoft requires access to native mobile features, such as APIs for various mobile device components. Piceasoft develops applications that perform mobile device diagnostics; therefore, it is a significant factor in framework selection. Access to native mobile features is one of the main requirements for migration. Native mobile features are generally an important factor to consider. If an application is not initially planned to use these features, it is a possibility that they are required later for the implementation of some feature. Although depending on the cross-platform framework, there may not be a framework-specific approach to access certain platform-specific APIs, evaluation of API needs of the application is essential when choosing a framework.

3.1.1 Communication with natively developed libraries

Piceasoft's main requirement for migration is communication with existing natively developed libraries. Some libraries can have features that prevent translation of library into Dart programming language. For example, a library can utilize a platform-specific feature that does not have API implementation in Dart. While some libraries cannot be migrated to the target framework's language, they may still be crucial for the functionality of an application. Therefore, the cross-platform framework is required to have the functionality to integrate these libraries as they are.

In the development of mobile applications, there is sometimes a need to implement platform-specific functionalities, in which case the development often must be done with native methods and languages. For example, the platform-dependent implementation can utilize hardware-related programming interfaces, and these interfaces may not be accessible without native implementation. These hardware-related interfaces may provide functionality related to, for example, the sensors of the device. If an application needs to handle the interfaces of device components frequently, then it may be worthwhile to isolate this functionality into a separate software library.

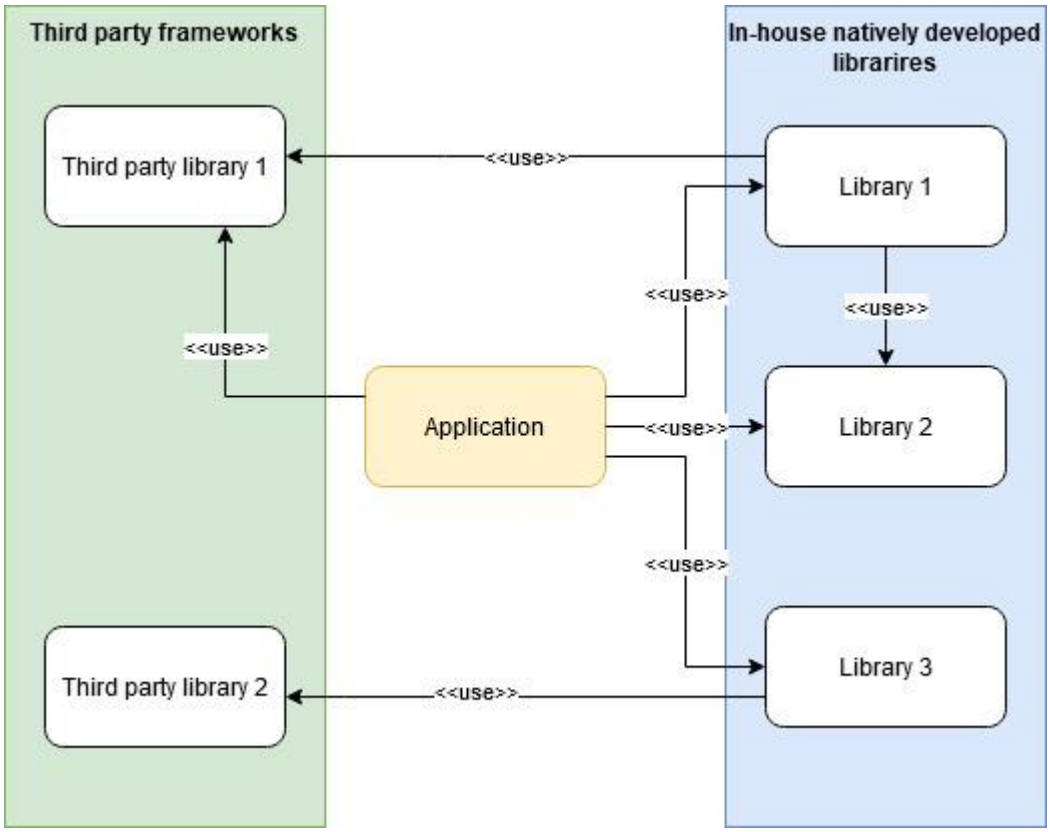


Figure 1. High-level architecture example for cross-platform application framework selection.

Figure 1 demonstrates a potential development use case for the selected cross-platform framework. In figure 1, the application is a part of the software that manages most of the application's user interface and should be implementable with Flutter. The application part depends on in-house developed libraries that are platform-specific. Libraries provide most of the functionality for the application that is relevant from the product requirements perspective. Flutter application is required to exchange values with libraries. Libraries are not to be changed, and application is required to communicate with libraries as they are. Libraries can be interacted with method calls and return values, or in some cases, with interface/delegate listener pattern. One of the internally developed libraries has a user interface required to be accessible from Flutter application user interface.

3.2 Performance

Performance is an important factor to consider since it may have a direct impact on user experience. For example, slow view transition and responsiveness negatively impact user experience; therefore, the cross-platform application is required to have equal or better performance compared to natively developed application. Poor performance also impacts negatively on battery consumption, causing the battery to drain faster.

3.3 Licensing

Licensing of the selected cross-platform framework is an essential factor for selection since the license can directly affect a company’s possibility to publish and monetize the application. License controls how software can be used, copied, and distributed. Some licenses can prevent commercial distribution completely if the application utilizes frameworks with a license that does not allow commercial use. Different licenses also affect possibilities to modify the framework to suit better into developer’s needs. For example, if a developer needs to customize the framework, then the license of that framework must allow it. In Piceasoft’s case, a license of the framework is required to allow commercial use.

3.4 Target platforms

Cross-platforms can differ greatly in their support for different platforms; therefore, target platforms are a major factor in framework selection. This criterion may not be relevant when targeting only mobile platforms since most mobile cross-platforms support Android and iOS, which comprise roughly 99% of the mobile devices (Statcounter GlobalStats, 2021). However, if the developer is required to support other platforms, such as desktop

or web, this criterion is essential. Targeting specific platforms may restrict approaches to cross-platform development.

3.5 Support

New versions of the operating system may bring changes to the user interface components or their functionality. In the case of significant changes, developers often must respond to these changes for the application to be compatible with the new version of the operating system. Operating system versions may also affect the performance of cross-platform frameworks with the new version, depending on the approach. The availability of a cross-platform framework for the latest operating system version is vital if the application is required to support the latest operating system versions and devices at their release.

In addition to always supporting new operating system versions, it may be important for the developer to support older versions. Supporting older versions to some degree is usually beneficial since some mobile devices have only older operating system versions available. Some applications, by nature, may be required to support older devices for software that performs diagnostics on the device.

3.6 Look, feel, and usability

Look and feel can be important for companies. Some companies are very strict in maintaining their specific appearance and brand in their products. Look and feel also contribute to the usability and user experience. For example, a simple and distinct look for a button clarifies that it is interactable by pressing. Frameworks implement user interface components and their management differently; some maintain a more native look and feel while others do not. While frameworks may not restrict companies from implementing their own specific-looking UI, some frameworks can simplify the development by providing some functionality of UI automatically, such as view transition animations.

Article A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX by Angulo and Ferre examines UX of cross-platform frameworks using Titanium framework. The research was conducted by developing a similar application with native methods and with Titanium and by evaluating implementations. The evaluation was done with laboratory and longitudinal studies in which performance and UX was measured with 37 participants. Results favored the native approach slightly without any extreme differences. A longitudinal study displayed that the iOS users had a stronger preference for the native version of the application. While the results of the study may

not be entirely applicable for Flutter development, it shows that look and feel should be considered in framework selection.

3.7 Battery consumption

Battery consumption in mobile devices is a common problem that can be difficult to address. Most mobile devices use lithium-ion battery technology and contain batteries with an average capacity of 3,500mAh (Android Authority, 2018). Battery life can be difficult to determine based on capacity alone, as consumption is also affected by other components of the device and the processes performed by the device and their number. The processes and applications performed on the device can differ significantly from user to user, and in addition, the applications consume battery power according to their own requirements. Although average battery capacity has increased somewhat over the past five years, this may not directly improve the battery life of mobile devices. Despite the increase in battery capacity, the development of more efficient components and communication methods may negatively affect battery consumption, reducing device lifespan (Android Authority, 2018).

Battery consumption can be influenced in application development by producing optimized applications by eliminating unnecessary computational processes. Often, graphics processing is costly to perform in terms of battery consumption, which is why battery consumption is often clearly noticed in mobile games (Singhai, Bose, 2013). The operating systems themselves also tend to save resources with various features such as Android App standby buckets, which aim to prioritize application resources (Google Developers, 2021a). The importance of battery consumption of application varies depending on the nature of the application. For example, optimization of battery consumption of applications that are used several times during a day could be beneficial compared to applications with less usage. Applications with continuously running background processes could also be optimized to reduce battery consumption. On the other hand, applications that are used infrequently, such as application performing diagnostics, may not require significant measures to reduce battery consumption because the operations it performs are infrequent and likely to last a short time.

When choosing a cross-platform framework, attention should be paid to the battery consumption of the Framework. Battery consumption is affected by the performance overheads of the frameworks, which can vary greatly depending on different cross-platform development approaches. The hybrid approach should be avoided if battery consumption is a significant selection criterion (Biørn-Hansen, et al. 2020). Battery consumption

should always be considered when developing an application, as battery life is an important feature for consumers of mobile devices (Global Web Index, 2019).

3.8 Security

Security is expected to be a built-in feature of the potential migration target framework; it is required to be secure and robust. While security-related implementation can be implemented with native programming languages and APIs, it is preferred that the framework and its programming language could provide security-related functionality as well. Applications are, in some cases, required to access or manage sensitive data such as device-related identifying data, customer data, or location data; therefore, the framework is required to provide encryption and authentication-related functionality.

3.9 Scalability

Scalability is an important attribute for software development since it directly affects the application's ability to grow or change based on user's or customer's demands. Therefore, it is preferred that the framework provides the necessary tools and functionality for developing scalable applications, even though scalability may not be a crucial factor for framework selection. In many cases, scalability can be implemented through application design and correct architectural choices.

3.10 Maintainability

Most software requires maintenance in the forms of bug fixing, new feature implementations, improvements in usability and performance. Maintainable code is easy to read, contains documentation and comments. While maintainability is affected mainly by the developer's choices, it is preferred that the syntax of the programming language of the cross-platform framework is easy to read. Defects are very common in software, and in most cases, these defects need to be fixed. Therefore framework should also provide tools to detect defects and monitor performance to make it easier to detect and fix defects. Also, technology changes are extremely common and affect existing software very often; therefore, maintainability is essential so that the software can be easily adapted to these new changes.

The framework is expected to increase the maintainability of the application by having only one implementation compared to native development, with its separate implementations on both platforms. The common codebase is expected to affect maintainability positively since changes and bug fixes are required to be implemented only once. In addition,

common localization files and resource management are also expected to increase maintainability since all localisations, and resource changes and additions are implemented to only one project instead of two separate projects.

3.11 Availability of third-party libraries

Application may be dependent on some third-party implementation. These implementations are integrated into mobile applications as libraries so that their functionality can be utilized within the application that is being developed. The functionality provided by the libraries may be essential for development if the application which is being migrated has been developed initially to take advantage of these functionalities. Replacing or implementing the functionality of third-party implementations may be too large for the company in terms of workload, and abandoning them in the event of migration may not be an option. If the functionality cannot be abandoned or replaced by another library, then the availability of that library for the new technology must be verified before the migration.

3.12 Localization management

Applications commonly have support for multiple languages besides English or the native language of the developer. When an application is developed separately for multiple platforms, then localization files need to be managed separately for each project for each platform. Managing multiple localization files is slow and exposes the application to bugs. Bugs related to localizations are usually misspellings or incorrect localization strings in UI elements. Bugs may appear when localization files use different formats, and original localization strings from the translator are being formatted to project-specific formats.

3.13 Resource management

Like localization management, if an application is developed separately for multiple platforms, it requires managing different resources separately on those platforms. Resource management includes managing images, colour constants, videos, audio files, and other resources. Resources commonly are referenced with keys or names corresponding to the resource. For example, an image with a warning sign could have a key “icon_warning” that could be used to retrieve that specific resource if needed. Managing resources for different applications on different platforms increases workload and potential issues, like missing resources, misspellings in keys, and mismatches in colour resources.

3.14 Testing

Testing is an important part of the software development process. Correctly used, testing can be an efficient way to discover defects early. Early discovery of defects can significantly reduce the cost of fixing them (Wiegers, Beatty. 2013). If a defect is found long after it has been created, fixing it could have an impact on other implemented features as well, and those features may require changes as well. Fixing defects may also require an additional release to distribute the fix.

The cross-platform framework should have testing tools and testing-related libraries available to enhance the discovery and prevention of defects. Testing tools should provide the necessary functionality to implement unit and integration testing.

3.15 UI development version control conflicts

While version control and conflicts are not essential factors in cross-platform selection and development, they should still be considered. Version control conflicts are common, especially if multiple developers are working on the same codebase. Usually, this is not an issue, but on iOS storyboard-based user interface development, this can be a hindrance. iOS storyboards are preferred to be developed by the Xcode storyboard design tool; this tool generates XML based on the developed user interface, which is usually hard to read. It is common to encounter conflicts with these storyboard files, and these conflicts can be difficult to resolve into a working Storyboard file. Again, while this may not be a crucial criterion for framework selection, it can be beneficial if the framework uses an approach that makes conflicts related to user interface development easier to resolve.

4 Native mobile application development

4.1 Android

Android is a Linux-based mobile operating system developed by Google that is commonly used on smartphones, tablets, televisions, smart wearables, and some other devices. Android was initially developed by Android Inc., but ownership passed to Google after Google's acquisition of Android Inc. in 2005. Android is an open-source operating system on which multiple device manufacturers have developed their own Android variations for their devices. Android development tools are available for Windows, Various Linux distributions, macOS, and Chrome OS via the Linux beta feature (Google Developers, 2021b).

Native Android development often takes place with Android Studio, an IDE developed by Google and JetBrains. There are alternative IDEs for development, but this thesis focuses on development only from the perspective of Android Studio. Android Studio comes with a tool that can be used to download Android software development kit, including the libraries, debugger, emulator, documentation, and other resources needed to develop Android. Android uses Gradle as the build Automation tool and Java or Kotlin as its primary development language.

4.1.1 Programming languages

Android's native libraries are written using C and C++ languages, which both can be used for application development as well. The use of C/C++ requires use of native development kit, also known as NDK. While development with NDK is possible, it may not be required in many cases since Google has exposed some of the native functionality via Java API framework (Google Developers, 2021c).

Java has been the primary development language of Android until Google announced to switch primary development language to Kotlin in 2019 (Google Developers, 2021d). While Android uses Java language, it does not use Java byte code or Java virtual machine; instead, it is compiled to Dex byte code with Android Runtime or Dalvik runtime environments (Google Android source, 2020). Dex files containing byte code are packaged into APK archive file for distribution, along with other resources of the application (Google Developers, 2020a).

The current primary development language is Kotlin, developed by the JetBrains company that has created IntelliJ Idea IDE, which Android Studio is based on (Google De-

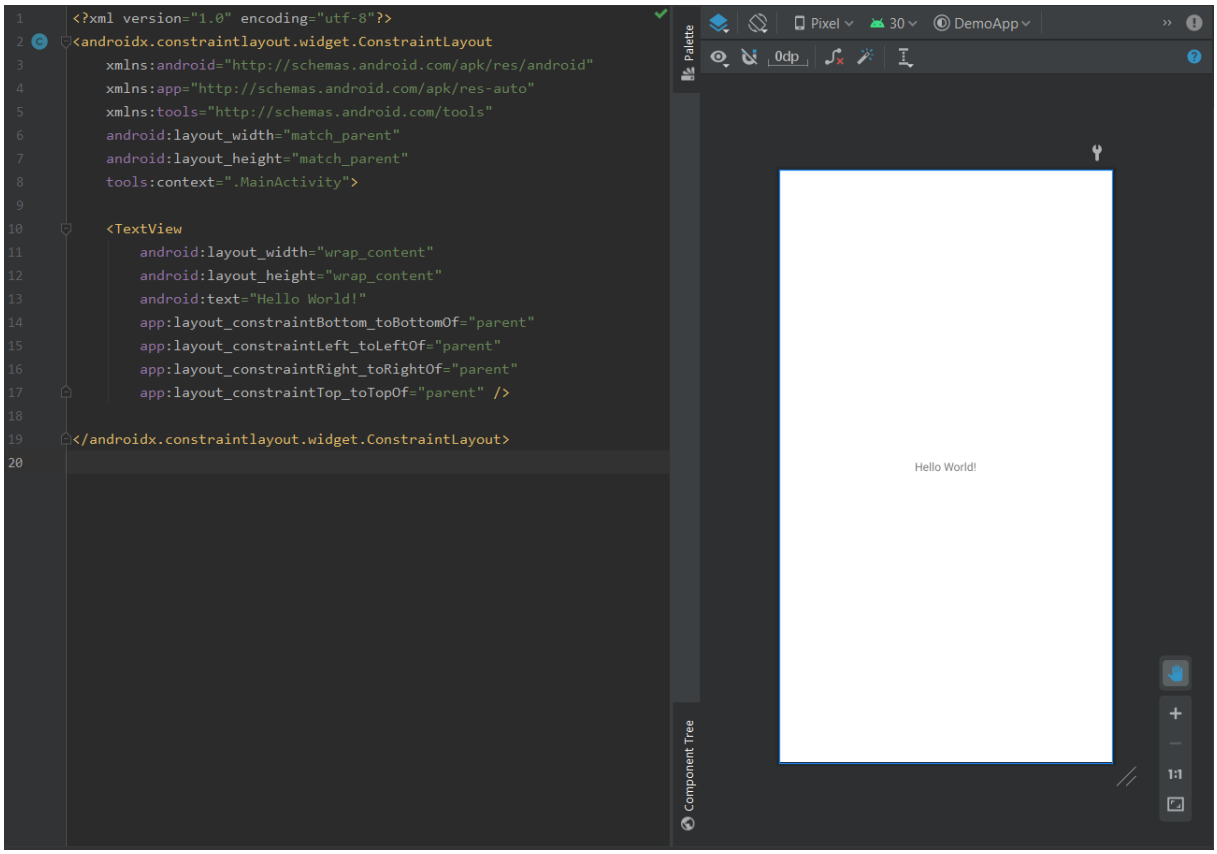
velopers, 2021e). Kotlin was adopted as the primary language in 2019, and since the Android documentation and tutorials have been written primarily using Kotlin (Google Developers, 2021d). Kotlin is an object-oriented programming language designed to work with Java virtual machine. In addition to working with Java virtual machine, it can be compiled to JavaScript, extending Kotlin language's use cases. JavaScript compilation could be used for cross-platform development on iOS and Android platforms, but it would only be limited to business logic (Kotlin, 2021).

4.1.2 User interface development

User interface development on native Android applications can be done in three different ways: user interface editor, editing XML files, and programmatically. Android Studio contains a graphical layout editor for editing Android application's user interfaces. Editing of user interfaces happens by drag and drop actions on different kinds of components. In addition to adding components, attributes of those components can be edited via layout editor. Editing via XML files is similar to editing any XML file; it contains tags for UI components, which have properties of the related components as attributes. Implementing the user interface programmatically is done by creating *View* objects in the source code. View objects' look and behavior can be changed by manipulating and accessing their properties via mutators and accessors and by adding these components to the *Activity* objects or *ViewGroup* objects as child *Views*.

Android SDK contains many different kinds of UI components for all kinds of purposes, for example, buttons for interacting and layouts for grouping components. All native Android application user interfaces are built with *View* and *ViewGroup* objects, *ViewGroup* objects being layouts containing child *View,* and *View* objects being actual UI components (Google Developers, 2020b). All native user interface components extend the *View* class. For example, a button is a *View* object with button-specific implementations and the ones inherited from *View* (Google Developers, 2020b). *ViewGroup* classes contain functionality to arrange, organize and manage views within the *ViewGroup*. Also *ViewGroup* class extends the *View* class (Google Developers. 2020c).

XML files represent layout or graphical user interface constructed with *View* and *ViewGroup* objects, with their attributes. Layouts specified in the XML files are commonly set as views for *Activity* objects. *Activity* is a class that contains activities that the application's user can interact with and see. The *activity* class contains lifecycle methods related to the layout it displays, and it can also reference layout components and contain background logic for UI elements (Google Developers, 2020d). For example, *Activity* processes button events or gestures.




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Image 1. Screenshot of Android studio, with layout editor displaying user interface XML containing “Hello, world!” text field.

4.1.3 Dependency management

In Android development, Gradle build automation tool is used for dependency management (Google Developers, 2021f). Gradle is a build tool that runs on Java virtual machine, and it can be used to manage a project’s dependencies. Dependencies can be local or remote modules and binaries (Gradle inc, 2021). In addition to managing dependencies, Gradle manages the build process of Android applications. Gradle can be used to run custom build scripts that can be written in Groovy or Kotlin (Gradle inc, 2021). Dependencies are managed by editing the project's *build.gradle* file, located in the Android app module.

Android application project contains multiple Gradle files, for example, one containing top or project-level configurations for the project and one for the app module. The project-level build file is located in the project directory. The module build file is located in the app module directory named “app” by default. The project-level build file contains configurations that are common for all modules. A project can have multiple modules con-

taining different Gradle configurations with varying dependencies for each module. Android Studio has a panel for examining and interacting with Gradle configurations and scripts for the project and each app module. (Google Developers, 2021f)

4.1.4 Resource management

An Android project can include different kinds of resources ranging from localization strings to images. Resources of Android projects are located in a directory called "res," which is located in the app module folder in the project (Google Developers, 2021g). The resource directory contains all static content of the project. These resources can be accessed from the application code by using IDs generated automatically during the build process to the project's R class (Google Developers, 2020e). Resource directory contains predefined subfolders for different types of resources; supported folder names for resources are the following: animator, anim, colour, drawable, mipmap, layout, menu, raw, values, xml, and font (Google Developers, 2020e).

The type of contents within subfolders vary depending on the names of those folders. For example, the layout folder contains XML files describing the user interface layout, and the drawable folder contains image resources. String resources are defined in XML files into the values directory (Google Developers, 2020f).

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="example">This is example.</string>
4  </resources>
```

Image 1. Example of string resource file, containing one example string value.

String resources are defined as in Image 1. Defining values into XML files is not restricted to string type only; files can be used to store *boolean*, *colour*, *dimension*, *ID*, *integer*, and *array* values as well. These values can be referenced via the *R* class. For example, the value in Image 1 could be referenced in code with *R.string.example*. In other XML files, Image 1 example string could be accessed using *@string/string_name* reference. (Google Developers. 2020)

Localization strings are stored in a similar manner to other string resources but by using alternative resources feature. Alternative resources for localizations can be defined by creating additional *values* directories into resource directory with a language-region combination. For example, the *res/values-fi/strings.xml* file would contain Finnish localiza-

tion strings for the application. If values directory with required language-region combination does not exist in the application, the application uses default localization. (Google Developers, 2019)

4.2 iOS

iOS is an operating system based on an open-source operating system called Darvin (GitHub: Apple, 2021). iOS is developed by Apple Inc. and is used as an operating system in Apple's iPhones and iPods. Apple's tablet, iPad, also previously used iOS but was changed to use iPadOS in 2019. iPadOS is variant of iOS with additional features (Apple, 2021a). Even though Darvin is open source, iOS is closed source only available on devices developed by Apple. Unlike Android applications, iOS applications cannot be developed on operating systems other than macOS (Apple, 2018a).

iOS development is usually done with Xcode, An IDE developed by Apple. Xcode is used for development for all platforms developed by Apple in addition to iOS (Apple, 2021b). Xcode provides tools for code and UI editing. In addition to editors, Xcode also provides debugger, simulators, documentation, and software development kits for Apple's products. Xcode. iOS applications can be developed using alternative development tools to some degree. For example, the developer can write Objective-C or Swift code with other IDEs or editors. However, development with alternative tools usually requires use of Xcode since there are no alternative tools for Xcode's Storyboard editor. iOS uses Objective-C and Swift as main development languages (Apple, 2014. Swift, 2021a). iOS project settings and build process is managed with Xcode's project settings tool.

4.2.1 Programming languages

Native iOS development can be done using C, C++, Objective-C, or Swift languages. iOS applications can be usually written mainly using Objective-C or Swift since most objects iOS developer works with are provided by Cocoa Touch framework (Apple, 2014). Cocoa/Cocoa Touch API provides functionality from system services like threading to UI development with UIKit (Apple, 2018b). Some of the APIs provided by the Cocoa Touch may require use of C (Apple, 2018b). While Cocoa Touch API uses mainly Objective-C language, it can be utilized by Swift language since Swift contains full support for Objective-C interoperability (Apple. 2021).

Objective-C is a programming language designed to enable object-oriented programming for the C language. As a superset of C language, it inherits functionality from C language while providing its own features such as blocks, *NSObject* root class, reference counting,

and other features. Apple acquired Objective-C language along with NeXT company in 1996, and since it has been used for iOS development along with other Apple's platforms (Hsu, 2017). In 2015, Apple published Swift language, which is intended to replace C, C++, and Objective-C on Apple's platforms (Swift, 2021a).

Swift is an open-source programming language developed by Apple. Swift has cross-platform support; therefore, it can be used to develop applications on Apple's platforms, Linux, and Windows (Swift, 2021b). Swift introduces multiple improvements to iOS development, such as better null safety with additional flow control mechanics (Swift, 2021a).

4.2.2 User interface development

On iOS development, there are multiple ways to approach UI development. One way for UI development is to use Storyboards. Storyboards are XML files that are edited via Storyboard editor, a graphical user interface editor (Apple, 2021c). Storyboard files can contain multiple *UIViewController* objects, which can contain UI components, such as buttons and labels (Apple, 2021d). A storyboard can also contain navigation-related objects, like *UINavigationControllers* and *Segues,* to implement navigation between views objects (Apple, 2021d). While Storyboard files can be displayed and edited as XML files, it can be difficult to make working changes to the file since Apple does not provide any documentation to the XML structures. Also, compared to Android's XML files, iOS's XML files are more difficult to read since some of the tags are not self-explanatory. Storyboard editor contains a side panel similar to Android's layout editor, enabling the developer to change UI components' properties. Adding new components happens via drag and drop action from the UI component selection menu. *UIViewController* class is somewhat similar to Android's *Activity*, and it commonly holds references to UI components and handles events on the user interface. Similar to Android, iOS UI components extend one super class called *UIView*, excluding some components like *UIBarButtonItem* (Apple, 2021e). The user interface could also be created programmatically, for example, initializing UI components in the *UIViewController* class's implementation.

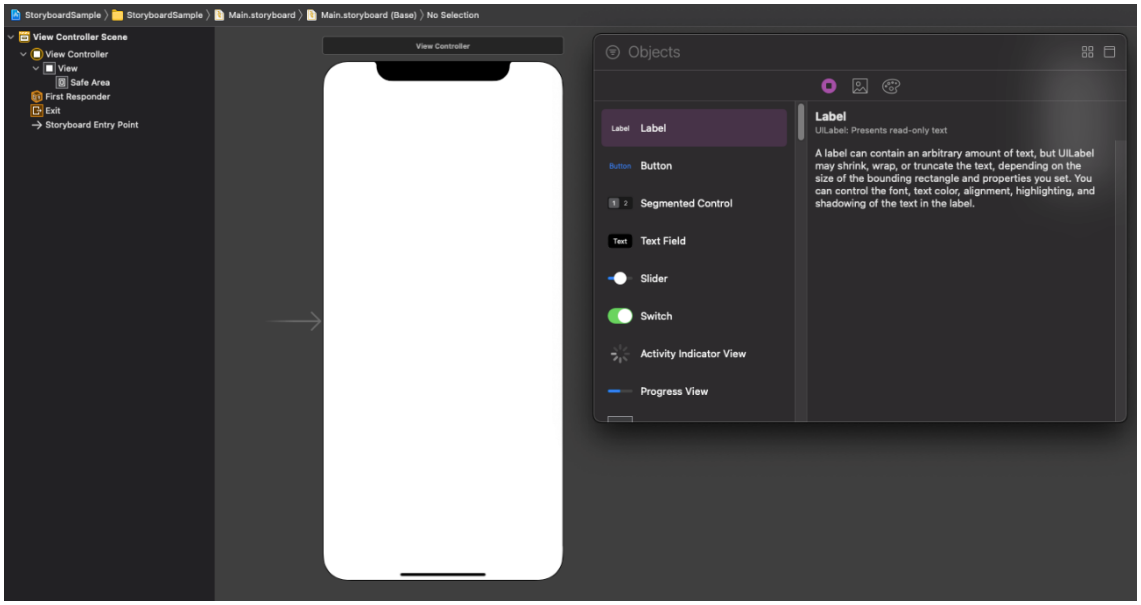


Image 2. Screenshot of Xcode's Storyboard editor, with UI element selection menu open.

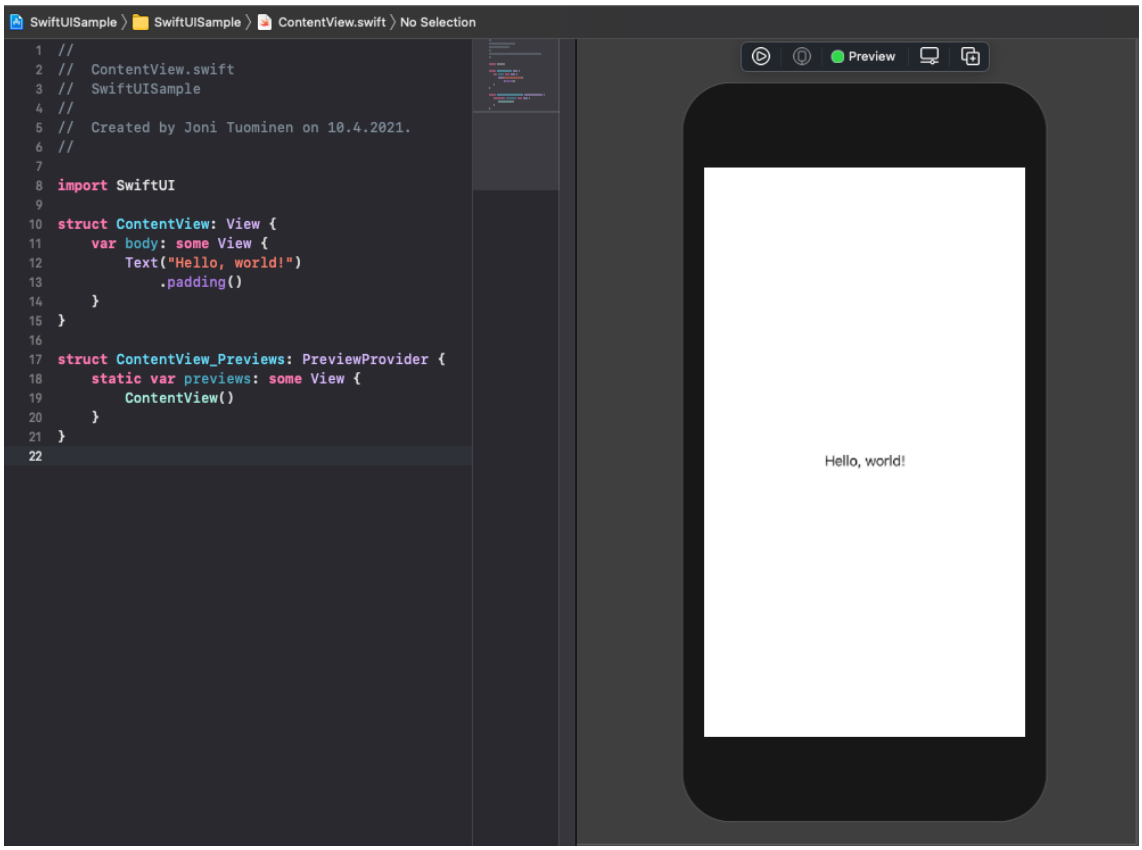


Image 3. Screenshot of Xcode, with SwiftUI preview displaying user interface code containing "Hello, world!" text field.

At Apple's Worldwide Developers Conference 2019, Apple announced SwiftUI as a new UI development toolkit. SwiftUI differs from Storyboard development, as SwiftUI development is done programmatically instead of using a graphical user interface editor. With

SwiftUI developer adds components to the *View* struct by declaring them programmatically by calling the constructor function, like in Image 3. Adding child views for components also happens by using the constructor of the components in the trailing closure of the parent, like in the *View* on line 11 in Image 3. Properties of a component can be modified by using functions of the object returned by the constructor function. In SwiftUI, components also inherit common properties and functions, but instead of inheriting them from the super class, they are defined by *View* protocol (protocol is an abstraction, similar to Java's interface) (Apple, 2021f).

4.2.3 Dependency management

Dependency or package management of an iOS project can be a frustrating experience since Apple did not provide any decent dependency manager with remote repository support until the introduction of Swift packages. Without any third-party managers, dependency management is done manually via Xcode by adding binaries as dependencies to the project file (AIM Consulting, n.d.). Adding dependencies manually to projects can cause issues and increase workloads since every dependency must be downloaded and added manually (AIM Consulting, n.d.). Furthermore, in the case of updating, the dependency management process is done multiple times for a single dependency.

For iOS dependency management, there are popular third-party tools, like CocoaPods and Carthage. CocoaPods is an open-source Ruby library that manages the project's dependencies, and it can automatically add, remove, and update dependencies by using a single command (GitHub: CocoaPods, 2021). With CocoaPods, dependencies are declared in the Pods file of the project, which determines what packages are downloaded and with what version. CocoaPods supports multiple version control systems for remote repositories (CocoaPods, 2021).

Like CocoaPods, Carthage is an open-source third-party dependency manager for Xcode projects. Unlike CocoaPods, Carthage is written mainly by using Swift programming language. Carthage uses a file named *Cartfile* to declare dependencies for Xcode projects. Also, Carthage supports multiple different remote repositories and can be configured to use specific versions of dependencies. (GitHub: Carthage, 2021)

Along with Swift 3.0 version, Apple released Swift package manager (Swift, 2021c). It is integrated into the Swift build system and has built-in Xcode support since version 11 (Swift, 2021c. Apple, 2021g). Xcode support makes dependency management much more manageable without third-party solutions. Swift package manager supports linking dependencies from remote repositories, similarly to CocoaPods, Carthage, and Android's

Gradle (Swift, 2021c). Swift packages are modules that contain code and *Package.swift* manifest file, containing the configuration of the package (Swift, 2021c). Each Swift package is organized into modules with its own namespace and access control to enforce visibility of functionality outside of the package (Swift, 2021c). Even though Swift package name indicates points at Swift language, packages can also contain Objective-C, Objective-C++, C, and C++ code (Apple, 2021h).

4.2.4 Resource management

iOS uses an asset catalog for resource management (Apple, 2018c). An asset catalog is basically a directory that contains assets or resources, such as images, for the project. The asset catalog directory is identified by giving the directory *xcassets* extension. Apple instructs to create asset catalogs via a new file menu. However, they can be created manually by creating the asset directory with a file manager and by adding a reference to it into the Xcode project file (Apple, 2020a).

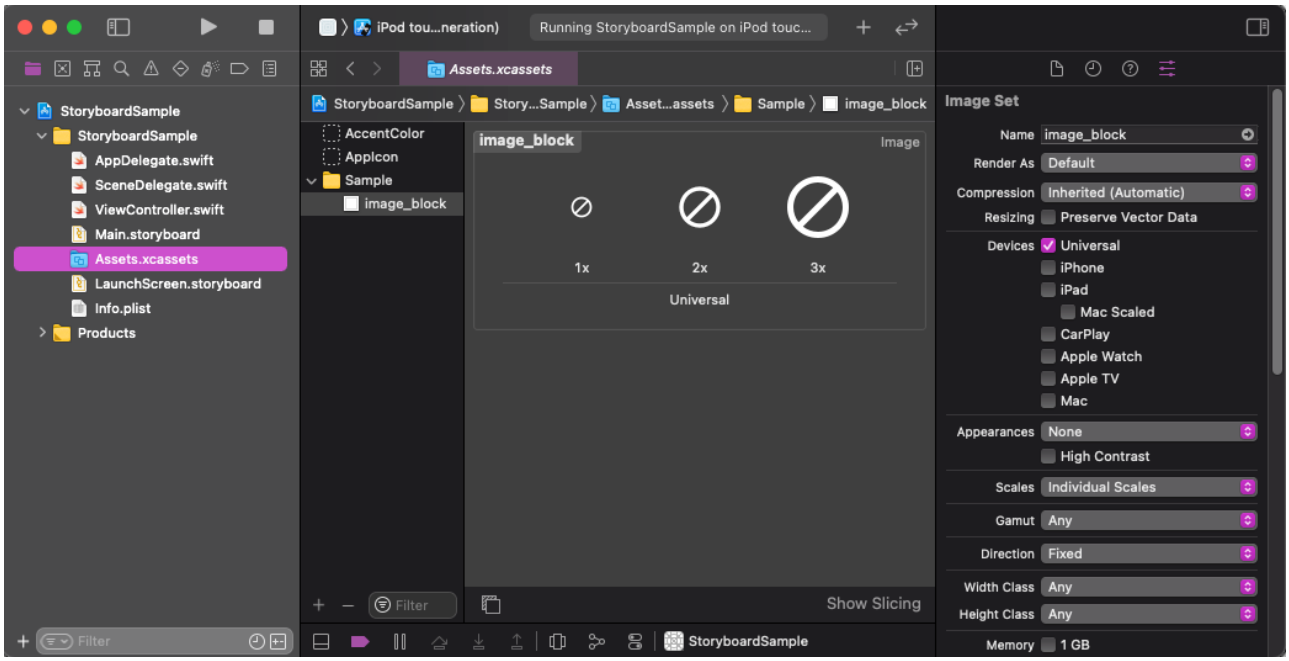


Image 4. Screenshot of Xcode with image asset visible.

An asset catalog can contain groups that are directories used for organizing resources (Apple, 2018c). Unlike the catalog directory, the group directory does not have an extension (Apple, 2018c). Assets can be placed directly into catalogs or groups. Also, the asset itself is a directory that contains the actual resource files, and these files are called content files (Apple, 2018c). Asset directories have extensions that describe the contents of the asset directory. For example, an asset containing an image has *.imageset* extension (Ap-

ple, 2018d). Asset directory can have variations of the same content file for different environments, scenarios, and devices. Image 4 demonstrates an image asset containing multiple content files with different dimensions for various scaling options. Xcode also provides tools to inspect and manipulate attributes of the resources, as seen on the right side of Image 4. Each directory in the asset catalog also contains a JSON file that encodes the attributes of resources (Apple, 2018d). Content JSON files are hidden in Xcode but can be found with a file manager. Assets in the asset catalog are referenced in the code by using the asset name. For example, asset visible in Image 4 is referenced with *image_block* string. While asset catalog can be used for managing different types of resources, it may have restrictions on some resource types on different iOS versions. For example, colour (also known as named colour) assets can be used with iOS 11 or above versions (Apple, 2021i).

```
{
  "images" : [
    {
      "filename" : "outline_block_white_20pt_1x.png",
      "idiom" : "universal",
      "scale" : "1x"
    },
    {
      "filename" : "outline_block_white_20pt_2x.png",
      "idiom" : "universal",
      "scale" : "2x"
    },
    {
      "filename" : "outline_block_white_20pt_3x.png",
      "idiom" : "universal",
      "scale" : "3x"
    }
  ],
  "info" : {
    "author" : "xcode",
    "version" : 1
  }
}
```

Image 5. Content JSON file of image asset displayed in image 4.

While asset catalog can be used to manage multiple types of resources, it is not used to manage localizations, like Android app uses XML resource management system for localizations as well. On iOS, application localizations are added via project file, and localized strings are defined into language-specific *.strings* files (Apple, 2020b). Strings files use a key-value pair system for localized strings. Values can be accessed by calling the

NSLocalizedString macro function by passing it the key of the localized string as a parameter (Apple, 2020b. Apple, 2021j). Other resources can also be localized. For example, images could be shown with regionalized content (Apple, 2020c). Xcode generates directories with *.lproj* extension to contain language-specific versions of resources (Apple, 2020c). *.lproj* extension is preceded by a language identifier using ISO 639-1 standard. For example, Finnish localization resources would be placed into *fi.lproj* directory (Apple, 2020d).

5 Flutter

Flutter is an open-source, cross-platform framework developed by Google (GitHub: Flutter, 2021). Flutter was introduced in 2015 at the Dart developer summit. Flutter is designed to enable application development for multiple platforms, using the same code base for all supported platforms. Flutter currently supports development on mobile, web, and desktop platforms (Flutter, 2021c). Flutter uses Flutter engine to host its applications. Flutter engine provides all that is required for Flutter application development, from 2D graphics rendering with Skia graphics library to Flutter core libraries and Dart runtime (GitHub: Flutter engine, 2021). Google provides Flutter development plugins for Android Studio and IntelliJ IDEA integrated development environments; plugins are also provided for Visual Studio Code and Emacs editors.

5.1 Dart

Dart is a programming language that is designed to be a client-side language. In its documentation, it is described as optimized for UI. Syntactically Dart bears similarity to languages used in native mobile application development, excluding Objective-C. Dart contains multiple features for managing asynchronous operations and events in the form of *Future*, *async*, *await* functionalities. *Future* is an object that represents potential value or error, which will be available later; it uses call-back functions to complete events (Dart, 2021a). *Async* keyword is used to mark that function is asynchronous, and it changes the function's return type into *Future* class (Dart, 2021b). *Future* class supports genericity; therefore, if an asynchronous function has a return value with a specific type, then the generic type of *Future* is replaced with the return value's type (Dart, 2021b). *Await* keyword is used to mark that the code needs to wait for *Future* completion (Dart, 2021b). Dart also uses isolate-based concurrency, meaning that separate threads have their own memory heaps and event loops; these are called isolates (Kathy Walrath, 2019. Dart, 2021c). Since threads are isolated from one another, their variables cannot be mutated from outside of the isolate. While isolates do not have access to each other directly, they can communicate via a message system, where the receiver handles the message in its own event loop (Kathy Walrath, 2019).

In addition, features related to threading, Dart contains multiple libraries to provide core features of the language and basic functionality like file handling and web requests. Dart also supports null safety; it uses question marks similarly to Kotlin and Swift to mark optional or nullable types. (Dart, 2021d)

5.2 Project structure

In the context of mobile development, Flutter project basically consists of three projects, Flutter project itself and two sub-projects for Android and iOS platforms. Android project is contained in a directory called “android”, and it contains Android-specific source code along with other Android-specific files. Like the Android sub-project, the iOS project and its files are contained in a directory called “ios”. Flutter application itself has its source code in a directory called “lib”. The project folder also contains other directories and files like optional “test” and “assets” directories and *pubspec.yaml* file. “test” and “assets” directories contain unit tests and assets for the project, and *pubspec* file contains project configurations and metadata.

5.3 User interface development

Flutter uses Dart language for user interface development. The user interface usually consists of Widget objects, which are building blocks of the user interface in Flutter; they are used to represent almost anything from containers to buttons (Flutter, 2021d). Unlike on Android and iOS, Flutter does not have a layout or user interface editor, and its user interface is implemented programmatically. Flutter user interface is implemented by creating a class that inherits from the *Widget* class, which has a *build* function that builds widget objects with all of its children and properties. *build* function has a tree-like structure since it can contain multiple child widgets and their properties, as seen in Image 6.

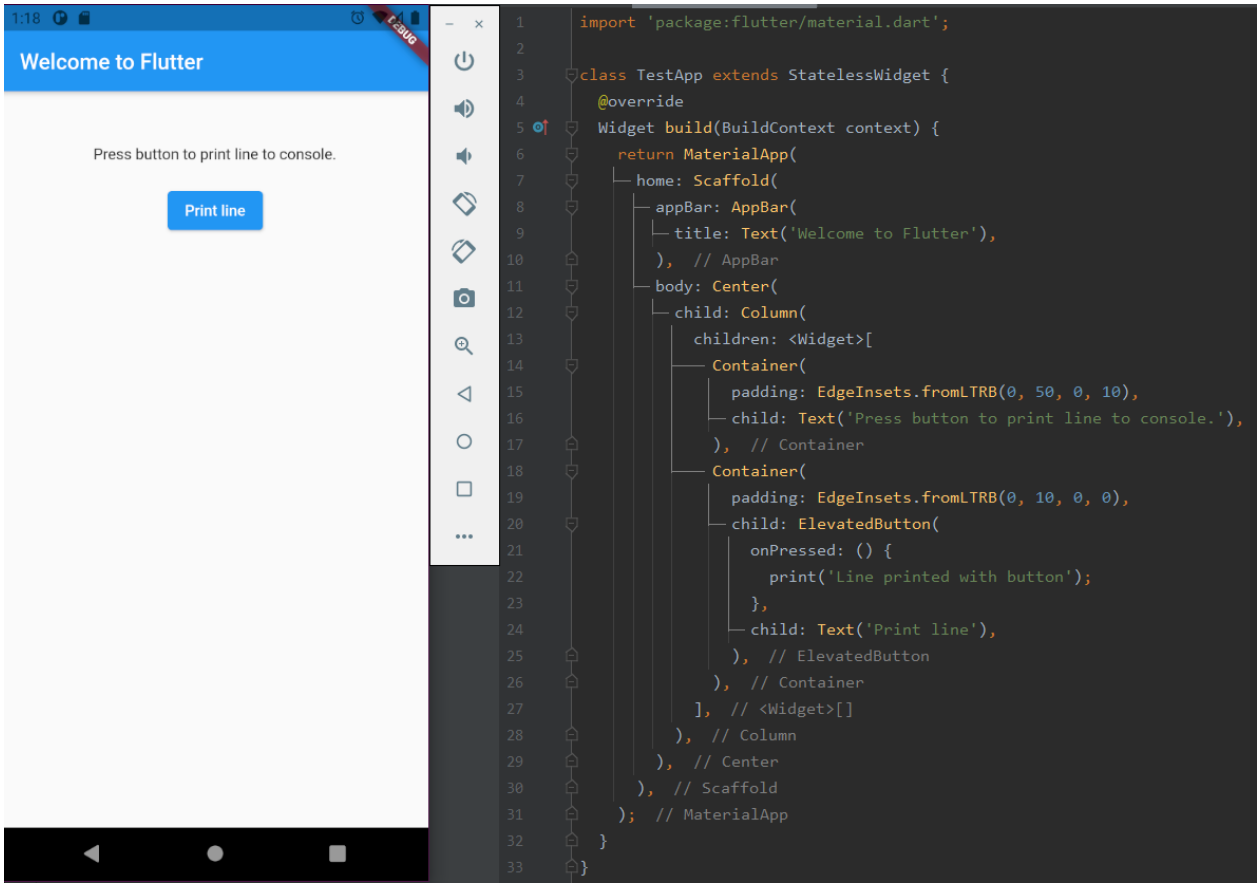


Image 6. Screenshot of Android emulator running Flutter application with user interface declared on the right side of the screenshot.

There are two types of Widgets in Flutter in the sense of state and interactivity, and these Widgets are stateful and stateless. *StatelessWidget* class is used to declare parts of the user interface that are not required to change during runtime. *StatefulWidget* class has a state which can be changed during runtime. Widget's state enables changes to the user interface components based on different events, such as completing HTTP requests. (Flutter, 2021e)

Since Flutter does not have a graphical layout editor, the changes to the user interface are hard to detect without running the application on an emulator or actual device. Flutter provides functionality and tools like Hot reload and Flutter Inspector for experimenting and detecting modifications to the user interface. Hot reload enables developers to refresh the user interface of running applications to see changes without re-running and building the whole application again (Flutter, 2021f). Hot reload injects updated source code to Dart virtual machine, and then Flutter framework proceeds to rebuild widget tree by calling *build* functions (Flutter, 2021f). Hot reload preserves the application's state. For example, data received with HTTP requests do not reset or change (Flutter, 2021f). If a state

is required to reset, the application must restart with either Hot restart or Full restart functionality. On the other hand, Flutter inspector is used to visualizing and inspecting widget trees and their properties. Flutter inspector can detect errors in layouts or visualize changes to layouts by testing different alignments for Widgets (Flutter, 2021g).

5.4 Dependency management

Every Flutter project contains a YAML file called "pubspec", this file contains project-specific configurations, including dependencies of the project (Flutter, 2021h). Flutter supports adding Dart packages as dependencies using *pubspec* files (Flutter, 2021i). These packages can be targeted for specific platforms, meaning they can contain implementation only for a specific platform (Flutter, 2021j). Packages can be published via Google's *pub.dev* service. Flutter projects can also use private packages, enabling developers to create Dart packages without publishing them (Flutter, 2021j). Flutter support adding local dependencies with a file path. It also has support for Git repositories (Flutter, 2021i). Since Flutter supports file paths and Git repository URLs for dependency management, private packages can be declared into *pubspec* files using either option. Additionally, *pubspec* file can have configurations for packages to use only specific version(s) of the packages (Flutter, 2021i).

Flutter also supports the use of native libraries/frameworks as dependencies. Native dependencies are declared in the platform-specific sub-projects. Android-specific dependencies can be declared in the Gradle file located in the app module within the Android sub-project. iOS dependencies can be declared with the iOS project file located in the directory of the iOS sub-project. Native dependencies are further examined in chapter 6.1.

5.5 Resource management

Flutter resource/asset management is done with *pubspec* YAML file. *pubspec* file is used to declaring and referencing asset files of the project. Actual asset files are in directories that are located in the root directory of the project. These resource directories can be freely named, and they can have sub-directories to contain different kinds of assets. For example, Flutter project could have an "assets" directory, which could have directories called images and sounds for image and audio files. Asset files with their paths need to be declared in *pubspec* file. For example, it could contain declaration like *assets/images/example_image.jpg*, for some image asset. Flutter supports multiple types of assets ranging from JSON files to multiple types of image files. Flutter resource/asset system does not support declaration of colour defines or string assets similarly to Android, although

JSON/XML assets could be used to create somewhat similar functionality. (Flutter, 2021k, Flutter 2021i)

Accessing assets can be done by using the resource path of an asset that is declared in *pubspec* file. For example, an image could be instantiated using *AssetImage* class' constructor and passing the asset path for it as a parameter. Flutter also provides functionality to use these resources within native sub-projects. For example, assets declared in *pubspec* file can be referenced in Swift/Objective-C/Kotlin/Java classes. Resources and assets are further examined in chapter 6.13. (Flutter, 2021k).

Flutter supports localizations via an additional Flutter package called *flutter_localizations*. To localize application, the *flutter_localizations* package is required to be set as a dependency into the *pubspec* file. Flutter uses ARB file format, which is similar to JSON, to store localized texts. Dart file containing localizations is generated from these ARB files, which can be used to access localizations from the code. Each localization string has a key that is used to reference the localization. Each language has a separate ARB file which contains keys for specific localizations and their localized strings. For example, a Flutter project can have two ARB files, *app_en.arb* and *app_fi.arb* files, then these files could contain language-specific strings with the same key. Flutter also provides an additional plugin for the development tools to provide syntax highlighting and automatic formatting for ARB files. (Flutter, 2021l)

6 Results

6.1 Native mobile features

Flutter has a functionality called method channel for communication with native features. Method channel works by initializing *MethodChannel* class with name parameter defining the communication channel's name, and this needs to be done in Flutter and iOS sides of the project. Since Flutter projects contain native sub-projects for Android and iOS and automatically creates Flutter-related *Activity* and *AppDelegate* classes for both sub-projects, these classes can be used to implement native features and register *MethodCallHandlers*. *MethodCallHandler* is an interface that defines only *onMethodCall* function for handling method call via *MethodChannel*. *MethodCallHandler's* only function, *onMethodCall,* has two parameters, *Call* and *Result* objects, containing information about the method call and implementation to respond to a method call. *MethodCallHandler* is created by calling *MethodChannel's setMethodCallHandler* function on the native side and passing *MethodCallHandler* as an anonymous class as a parameter for *setMethodCallHandler* function call. When Flutter side has registered *MethodChannel,* and the native side has set *MethodCallHandler*, then *MethodChannel* can be used to invoke methods using *MethodChannel's invokeMethod* function to call native functionality specified by *invokeMethod's* method name parameter. In order to *invokeMethod* to work, it is required to have implementation within the anonymous implementation of the *MethodCallHandler* interface. On invocation of the method, the *MethodCallHandler's onMethodCall* implementation receives a *Call* object with properties to differentiate which functionality is being called. After the required native functionality has been run, the caller can be notified about the result of the method call with the *Result* object by calling its functions. *Result* object can be used to return values to Flutter implementation by passing values as parameters for *Result's* functions. (Flutter, 2021m)

MethodChannel uses serialization and deserialization for values that are passed between the native and Flutter sides of the application. Values that can be passed are required to be supported by Flutter's *StandardMessageCodec* class, which encodes and decodes passed messages. Supported values are described in TABLE 1. (Flutter, 2021m)

Dart	Java	Kotlin	Objective-C	Swift
null	null	null	Nil (NSNull when nested)	nil
bool	java.lang.Boolean	Boolean	NSNumber numberWithBool:	NSNumber(value: Bool)

int	java.lang.Integer	Int	NSNumber numberWithInt:	NSNumber(value: Int32)
Int, if 32 bits not enough	java.lang.Long	Long	NSNumber numberWithLong:	NSNumber(value: Int)
double	java.lang.Double	Double	NSNumber numberWithDouble:	NSNumber(value: Double)
String	java.lang.String	String	NSString	String
Uint8List	byte[]	ByteArray	FlutterStandardTypedData typedDataWithBytes:	FlutterStandardTypedData(bytes: Data)
Int32List	int[]	IntArray	FlutterStandardTypedData typedDataWithInt32:	FlutterStandardTypedData(int32: Data)
Int64List	long[]	LongArray	FlutterStandardTypedData typedDataWithInt64:	FlutterStandardTypedData(int64: Data)
Float64List	double[]	DoubleArray	FlutterStandardTypedData typedDataWithFloat64:	FlutterStandardTypedData(float64: Data)
List	java.util.ArrayList	List	NSArray	Array
Map	java.util.HashMap	HashMap	NSDictionary	Dictionary

Table 1. Table of conversion of types between languages. (Flutter, 2021m)

6.1.1 Accessing natively developed libraries

MethodChannel can be used to invoke native functionality via Activity or *AppDelegate* classes, and it can be used to invoke the functionality of natively developed libraries. Since Flutter projects contain Android and iOS projects as sub-projects, it is possible to add dependencies for those sub-projects via the project settings file on iOS or Gradle on Android. Both sub-projects can access the functionality of dependency libraries from Activity or *AppDelegate*; therefore, *MethodChannel* can be used to call functions from those libraries.

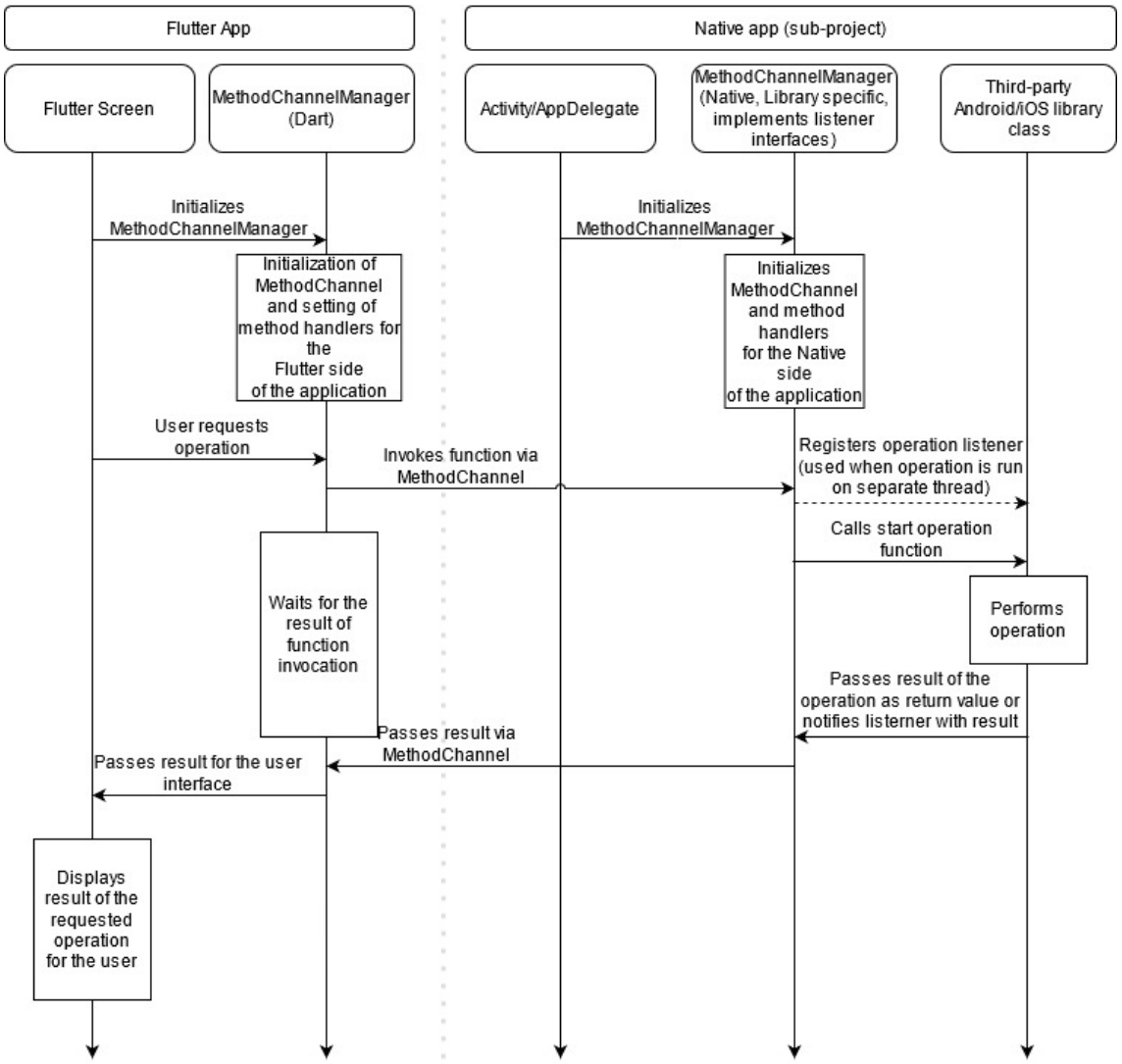


Figure 2. Diagram demonstrates invocation of operation from a third-party native library via *MethodChannel*.

MethodChannel uses *Future* objects to manage flow in the Dart implementations. The *Future* class represents the potential result that can be available at some point after a function call. *Future* uses callback functions to handle future results, and it provides functions like *then*, *timeout,* and *whenComplete* for the developer to implement actions based on the result.

Call of native library function can be invoked from Dart class by calling *MethodChannel* instance's *invokeMethod* function, which returns *Future* object. When the *invokeMethod* function is called, it is received on the native side's *MethodChannel's* method handler. After receiving method invocation, the method handler proceeds to run the implementation of the method; this implementation can utilize functions and classes from native libraries. If implementation from the library uses interface/protocol listener pattern to notify results, then *Activity* or *AppDelegate* class can be modified to implement required

interfaces. Results of the method invocation can be returned to the calling dart class via *Future* object once *MethodChannel* returns the result of the method invocation. If the application is required to navigate to native UI classes such as *Activities* or *UIViewControllers*, then navigation to these can also be implemented into *Activity* or *AppDelegate* classes.

While *MethodChannel* and method handlers can be created within the *Activity* or *AppDelegate* class, I would prefer to separate these implementations to *MethodChannelManager* classes, as seen in Figure 2. Separate class for *MethodChannel* related functionality prevents cluttering or bloating the *Activity* or *AppDelegate* classes. These classes have other uses as well, especially if the application communicates with multiple native libraries via *MethodChannel*. Separating *MethodChannel* implementation from *Activity* and *AppDelegate* makes the implementation easier to read. Depending on the number of native libraries, there could be multiple *MethodChannelManager* implementations for each library. Multiple *MethodChannelManager* implementations would provide uncluttered and distinct implementations for each library. On Flutter side of the application, *MethodChannel* implementation could also be separated into its own class. Isolating *MethodChannel* implementations on Flutter side would provide a simple entry point for the native implementations.

6.1.2 Pigeon

Alternatively, the developer can use Pigeon to use the native side of the application. Pigeon is a Dart package developed by Flutter's development team. Pigeon is a code generator tool that generates code for message handlers to the application. Generated message handlers can be used to send messages in either direction, from native to Flutter/Dart and vice versa. Pigeon also does not require declaring the same arguments and datatypes for messages, unlike *MethodChannel*. While Pigeon seems to simplify communication between native and Flutter code, it is still in pre-release stage, and future releases may introduce breaking changes. (Flutter, 2021m. Pub.dev, 2021a)

6.2 Performance

Jarkko Saarinen examines and evaluates cross-platform application performance in his thesis, Evaluating cross-platform mobile app performance with video-based measures. Saarinen used video-based measurement to measure performance of cross-platform frameworks. Cross-platform frameworks were tested with different tasks on similar user interfaces created with tested frameworks. Tasks included testing of button reaction delay, performance of lists, heavy computation, and vibration functionality. Flutter on both

Android and iOS platforms performs well compared to native and other frameworks on most. Compared to native, Flutter does not seem to have significant advantages or disadvantages in the sense of performance. (Jarkko Saarinen, 2019)

6.3 Licensing

Piceasoft develops and publishes commercial software, and therefore license of selected framework needs to permit this. Flutter is licensed under the BSD 3-Clause "New" or "Revised" License, and it allows commercial use, distribution, modification, and private use. The license requires BSD copyright and license notice to be included in the product. Since the license allows commercial use, distribution, and modification, it can be used to develop Piceasoft's applications.

6.4 Target platforms

The main targets of Piceasoft's mobile products are Android and iOS platforms. Flutter is designed for mobile platforms, so this requirement is met without any problems. Since Piceasoft's mobile applications are not targeted on other platforms, Flutter works in this situation. Flutter is also developing support for web as well as desktop platforms, and there are experimental versions of these available. Although web and desktop applications are not criteria for cross-platform framework selection, it is good to be aware that development on those platforms could be possible at some point.

6.5 Support

Flutter supports the iOS platform from the iOS 9 version and Android from API level 19. Flutter development team states that they are committed to providing full support for the latest features of Android and iOS platforms. In addition to the commitment to support the latest features, they try to support older versions as long as they can. Flutter development team monitors API changes and announcements from the platform developers to quickly react to the upcoming changes. (Ray Rischpater, 2014)

Flutter also states in their frequently asked questions site that Flutter's interop and plugin system allows developers to access new features immediately without the need to wait for Flutter team to expose them (Flutter, 2021n). Long-term support of Flutter is challenging to evaluate, and it would require examination on a more extended period. Now Flutter seems to have very active development on its GitHub repository, and it has had frequent releases in 2021 (GitHub: Flutter, 2021).

6.6 Look, feel, and usability

For user interface development, Flutter provides sets of widgets. These sets implement widgets using Material and Cupertino (iOS style) design languages, the latter referencing Apple's design language (Flutter, 2021n). Flutter encourages to use other than the Cupertino widget set if the application is designed to work on platforms other than iOS (Flutter, 2021o). Using common set for UI components reduces differences of the application on different platforms; therefore, the look and feel of the application remain similar. Common UI elements also converge usability on different platforms when components are designed to behave similarly. Flutter application can use separate widget sets based on the target platform with *defaultTargetPlatform* property.

Flutter uses the Skia graphics engine to render its own UI components instead of implementing an abstraction layer to use native UI libraries. Using its own components and separate graphics engine ensures that the components are always rendered the same way regardless of the platform. Flutter also provides a Dart package for animations, therefore providing developers the possibility to use the same animations on both platforms without additional implementations. (Flutter, 2021p. Pub.dev, 2021b)

6.7 Battery consumption

Battery consumption of an application is usually related to its hardware and performance requirements. For example, an application with heavy computation requires more from the device's CPU, resulting in increased power consumption. Battery consumption depends significantly on the nature of the application; how much the application consumes the device's hardware resources? And for how long?

An Empirical Investigation of Performance Overhead in Cross-Platform Mobile Development Frameworks article examined performance overheads in cross-platform frameworks. The investigation concluded in the article compares the performance of cross-platform frameworks on tasks utilizing file system, contact lists, GPS, and accelerometer. Flutter required slightly more CPU usage than native implementation in all tasks, excluding file system tasks. Overall, Flutter also consumes more memory than native implementation. While CPU and memory use are not the only factors of the power consumption, it may indicate that Flutter application may consume more power on some features than the native application.

6.8 Security

Dart provides *http* package for HTTP communication in Flutter applications; this package also supports HTTPS (Pub.dev, 2021c). Flutter, by default, has forced HTTPS connections since Android 28 and iOS 9 versions. Clear text connections require changes to *AndroidManifest.xml* and *Info.plist* files (Flutter, 2021p). Dart also provides a *crypto* package for hash functions (Pub.dev, 2021d).

In addition, to secure network communication, Flutter has the *local_auth* package to enable development on biometric authentication APIs, TouchID on iOS, and fingerprint APIs on Android (Pub.dev. 2021e). In addition to packages developed by the Dart or Flutter teams, Flutter has multiple security-related third-party packages covering authentication, encryption, iOS keychain, and Android Keystore (Pub.dev, 2021f. Pub.dev, 2021g. Pub.dev, 2021h).

Flutter uses a branching model with stabilization period beta and stable branches on the development prior to release. The stabilization period on beta and stable branches is used only to introducing fixes, if needed, to stabilize the release. Flutter's branching model enables testing feature locked branch before release, therefore increasing the stability of the released product. Flutter also receives regular releases to introduce new features and to fix found defects. (Tim Sneath, 2020)

Since Flutter is a product of Google, it follows Google's security philosophy, including a vulnerability disclosure policy and reward program. (Flutter. 2021q, Google, n.d.)

6.9 Scalability

As stated in the 5.9 section, scalability is an important factor but may not be crucial in selecting a cross-platform framework for mobile application development. Flutter and Dart offer very similar architectural and application design options to languages and libraries used on native application development. Scalability in the sense of application's ability to adapt to changes can be achieved with well documented, tested, structured, and modular codebase in addition to design and architectural choices. On mobile development, scalability, or the application's ability to adapt to demands set for it, can also be affected by factors, such as the device's limited hardware resources.

6.10 Maintainability

Maintainability of the application can be examined on multiple different aspects of the application, such as extensivity of documentation, readability, modularity, and structure

of the code. All attributes mentioned above affect how easy it is to expand, maintain, and update the application, especially for newly hired developers.

Since Flutter provides a single codebase for the application, or at least for part of it, it can potentially increase maintainability by requiring changes to only one project. For example, defects are required to be fixed only in one place. Similarly, new additions to the application can be implemented only once, reducing the possible defects or dissimilarities between projects for different platforms. Maintainability of resources and localizations can also be done only for one project instead of two.

While Flutter can increase the maintainability of the developed application, it can also have a decreasing effect on some aspects. For example, maintainability can decrease along with the increase of complexity, when application relies heavily on natively developed libraries and is constantly required to utilize their functionality. With native libraries, the developer must maintain additional components responsible for communication with these libraries. Although maintaining classes' that communicate with native libraries can be a smaller task than developing two separate user interfaces for different platforms. In addition to native libraries, changes in third-party dart libraries may require additional work if they contain platform-specific functionality.

Readability, modularity, and structure are attributes where also development, architectural, and design choices greatly impact the maintainability of the application. Flutter and Dart provide similar tools and functionality to implement robust and highly maintainable code than the native tools, and programming languages have. Flutter also provides extensive documentation for the development; this is likely to have an increased effect on the maintainability of Flutter applications.

Since Flutter has its own user interface development tools with its own user interface components, Flutter removes version control conflicts related to iOS storyboard user interface development. However, iOS storyboard issues can persist if Flutter application relies on native user interface implementations for the iOS platform.

Examination of Flutter's advantages and disadvantages regarding maintainability more extensively proved to be difficult. The more extensive examination would have required examination on a more extended period with natural application development with Flutter, instead of limited testing.

6.11 Availability of third-party libraries

Flutter has a great number of third-party frameworks or libraries available via *pub.dev* site. *pub.dev* site contains over 17000 published Dart and Flutter packages usable with Flutter applications (Pub.dev, 2021i). In addition to Dart packages, Flutter can use native packages, developed for iOS or Android, via Flutter projects native subprojects and MethodChannel functionality.

6.12 Localization management

Flutter provides all the necessary tools for the localization of developed applications. The localization package provided by Flutter differs from native ones, but by having JSON-like localization files, it is relatively easy to structure localization strings into Flutter localization files. Since Flutter provides tools for user interface development for both platforms, the localization can also be implemented for both platforms only once, therefore removing the need to add/update localized strings into two separate projects.

6.13 Resource management

Similar to localization management, Flutter has the functionality to support extensively different file formats as resources or assets. Also, similarly to localization management, Flutter simplifies resource management by requiring their management only once for both platforms.

While managing resources only on a project developed with Flutter is simpler compared to managing them on two separate projects with native implementation, it is not without issues. The issue I encountered with Flutter's resource management is the burdensome reference system, where asset/resource is required to be referenced with the full path to the resource instead of a simple identifier, like on native development. Referencing a full path can be burdensome when the application has a vast number of different kinds of resources in a complex directory structure. Also, referencing the full path can cause issues when references are in multiple files when resources are relocated or refactored. Refactor can result in broken references when assets are no longer available in the same directories. However, this issue could potentially be fixed with a wrapper class that would have functions to provide resources by appending the full path to a simple file name given as a parameter.

6.14 Testing

Flutter and Dart development teams provide additional packages for implementing different kinds of testing. For unit testing, the Dart development team provides *test* package via *pub.dev* site. *test* package contains functions like *test* and *expect*, which can be used to implement unit tests with expectation evaluation. *expect* function takes anonymous function/lambda/closure as a parameter, which is used to perform the actual evaluation of the result. Tests can also be grouped with *group* function. *test* package also contains functions for test setup and tear down. Since Flutter is a cross-platform framework, the test package also contains a platform selector for running tests only on specific platforms. (Pub.dev, 2021j)

In addition to unit testing, Flutter has *flutter_test* package for widget testing. *flutter_test* package is built on top of the *test* package; it provides functionality for performing user interface component (*Widget*) and integration testing. *flutter_test* package has functions like *testWidget*, *pumpWidget*, and *find*. *testWidget* function creates the test, *pumpWidget* builds and renders *Widget* and *find* can be used to find *Widget* object from widget tree with properties like text. (Flutter, 2021r)

7 Conclusions

Mobile application development has multiple approaches, Flutter framework being one of them among other cross-platform frameworks and native development. Usually, the native development approach provides the best results since native languages' APIs tend to be more optimized for the developed platforms. Also, it is common for cross-platform frameworks to utilize the native APIs as well; therefore, native approach removes third-party-related issues. Nevertheless, implementing the same application for multiple platforms can increase workload, development costs, and mismatch between the applications.

This thesis examined Flutter cross-platform framework developed by Google. Flutter provides tools to develop applications for multiple platforms, although this thesis examined the development mostly from a mobile application development perspective. While the thesis evaluates Flutter's overall applicability for mobile application development, it strongly focuses on communication between natively developed libraries and Flutter applications. In addition to the examination of Flutter, the thesis has a brief overview of Flutter's development language Dart and native mobile application development on Android and iOS platforms.

In the thesis, Flutter was evaluated via existing research, documentation, and development of test application. The evaluation was done from the perspective of Piceasoft Ltd, a client for this research. Piceasoft develops mobile solutions utilizing in-house developed native libraries for both Android and iOS platforms, steering the focus into communication with native libraries.

As a result of the evaluation, Flutter can be seen as a potential framework for application development for Piceasoft, at least for smaller projects with a careful approach. Flutter provides all required functionality to utilize existing natively developed libraries. In addition to communication with native libraries, it provides a rich palette of different kinds of widgets to standardize user interface on both platforms, with its own rendering engine. Flutter also provides simplified management for resources/assets and localizations with reduction of workload of the mobile application development team of Piceasoft, even though resource management could require additional wrapper class implementation. Also, the common codebase results in reduced workload and arguably increased maintainability, even though communication with native libraries may require additional implementations. *pub.dev* site also provides multiple third-party libraries in addition to existing third-party native libraries if such is required in the development. Flutter also pro-

vides commonly used testing tools and functionality that can be used to increase the quality of applications and to detect defects in the application. These testing tools can likely be utilized in automated testing.

Flutter's development process and dense release frequency increase the framework's reliability, although this is one of the aspects I would recommend evaluating further. I would recommend re-evaluating iOS and Android version support after few iOS and Android version releases. Re-evaluation would provide more information about Flutter and its reaction times to operating system version changes. Gaining more information about Flutter's support for new operating system versions and API changes could help developers to adjust the development process to react to these changes. However, Flutter's interop and plugin system should provide access to new and updated features immediately, at least according to their frequently asked questions site (Flutter, 2021n). Overall, Flutter seems reliable framework by a reliable developer, and it has matured a development process. Also, Google is likely to be committed to the development of Flutter, as they released Flutter 2 in 2021, along with news of the adoption of Flutter, by companies like Toyota, Canonical, Sony, Samsung, and Microsoft (Sneath, 2021).

While Flutter provides all essential tools to implement mobile applications that can utilize natively developed libraries, it still could be useful to evaluate maintainability over a more extended period. Also, comparison to similar cross-platform frameworks could be useful. Another interesting topic would be Flutter's ability to react to operating system changes; this could provide information on how quickly Flutter applications could be updated to support new OS features or API changes. In addition to the evaluation of cross-platform frameworks, the suitability of Kotlin language for long-term cross-platform development could provide interesting alternatives.

Summarily, Flutter provides an interesting alternative to native mobile application development. Furthermore, as stated earlier, Flutter provides features to implement an application that can utilize native libraries. Therefore, I could recommend the adoption of Flutter, but I would also recommend caution with adoption, for example, by testing it on smaller products first. Flutter has the potential for versatile mobile application development.

8 References

AIM Consulting. n.d. Choosing the Right iOS Dependency Manager. Retrieved from: *https://aimconsulting.com/insights/choosing-the-right-ios-dependency-manager/* (Accessed: 15.4.2021)

Apache Cordova. Overview. Retrieved from: *https://cordova.apache.org/docs/en/latest/guide/overview/index.html* (Accessed: 3.2.2021)

Apple. 2014. Documentation archive: About Objective-C. Retrieved from: *https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html* (Accessed: 9.4.2021)

Apple. 2018a. Documentation: Start Developing iOS Apps. Retrieved from: *https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/* (Accessed: 9.4.2021)

Apple. 2018b. Documentation archive: Cocoa (Touch). Retrieved from: *https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html* (Accessed: 9.4.2021)

Apple. 2018c. Documentation: Asset catalog format overview. Retrieved from: *https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_ref-Asset_Catalog_Format/index.html* (Accessed: 15.4.2021)

Apple. 2018d. Documentation: Asset catalog types overview. Retrieved from: *https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_ref-Asset_Catalog_Format/AssetTypes.html* (Accessed: 15.4.2021)

Apple. 2020a. Documentation: Create asset catalogs and sets. Retrieved from: *https://help.apple.com/xcode/mac/current/#/dev10510b1f7* (Accessed: 15.4.2021)

Apple. 2020b. Documentation: Add a language. Retrieved from: *https://help.apple.com/xcode/mac/current/#/devafa3a605f* (Accessed: 17.4.2021)

Apple. 2020c. Documentation: Make a resource localizable. Retrieved from: *https://help.apple.com/xcode/mac/current/#/dev7c584bb2a* (Accessed: 17.4.2021)

Apple. 2021a. New features available with iPadOS. Retrieved from: *https://www.apple.com/ipados/ipados-14/features/* (Accessed: 9.4.2021)

Apple. 2021b. Documentation: Xcode. Retrieved from: *https://developer.apple.com/documentation/xcode/* (Accessed: 9.4.2021)

Apple. 2021c. Documentation: Using interface builder. Retrieved from: *https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html* (Accessed: 11.4.2021)

Apple. 2021d. Documentation: Designing with Storyboards. Retrieved from: *https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/DesigningwithStoryboards.html* (Accessed: 11.4.2021)

Apple. 2021e. Documentation: UIView class. Retrieved from: *https://developer.apple.com/documentation/uikit/uiview* (Accessed: 11.4.2021)

Apple. 2021f. Documentation: View protocol. Retrieved from: *https://developer.apple.com/documentation/swiftui/view* (Accessed: 11.4.2021)

Apple. 2021g. Xcode 11 release notes. Retrieved from: *https://developer.apple.com/documentation/xcode-release-notes/xcode-11-release-notes* (Accessed: 15.4.2021)

Apple. 2021h. Documentation: Swift Packages. Retrieved from: *https://developer.apple.com/documentation/swift_packages* (Accessed: 15.4.2021)

Apple. 2021i. Documentation: UIColor class init function. Retrieved from: *https://developer.apple.com/documentation/uikit/uicolor/2877380-init* (Accessed: 17.4.2021)

Apple. 2021j. Documentation: NSLocalizedString macro. Retrieved from: *https://developer.apple.com/documentation/foundation/nslocalizedstring* (Accessed: 17.4.2021)

Android Authority. 2018. Fact check: Is smartphone battery capacity growing or staying the same? Retrieved from: *https://www.androidauthority.com/smartphone-battery-capacity-887305/* (Accessed: 25.2.2021)

Biørn-Hansen Andreas, Grønli Tor-Morten, Ghinea Gheorghita. 2018. A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. ACM Comput. Surv. 51, 5, Article 108 (November 2018), 29 pages. *https://doi.org/10.1145/3241739*

Biørn-Hansen Andreas et al. 2020. An Empirical Investigation of Performance Overhead in Cross-Platform Mobile Development Frameworks. Empirical software engineering : an international journal 25.4 (2020): 2997–3040. Web.

CocoaPods. 2021. CocoaPods about. Retrieved from: *https://cocoapods.org/about* (Accessed: 15.4.2021)

Dart. 2021a. Documentation: Future class. Retrieved from: *https://api.dart.dev/stable/2.12.0/dart-async/Future-class.html* (Accessed: 7.5.2021)

Dart. 2021b. Documentation: Asynchronous programming: futures, async, await. Retrieved from: *https://dart.dev/codelabs/async-await* (Accessed: 7.5.2021)

Dart. 2021c. Documentation: Language tour, Isolates. Retrieved from: *https://dart.dev/guides/language/language-tour#isolates* (Accessed: 7.5.2021)

Dart. 2021d. Dart overview. Retrieved from: *https://dart.dev/overview* (Accessed: 7.5.2021)

Discord. 2019. How Discord achieves native iOS performance with React Native. Retrieved from: *https://blog.discord.com/how-discord-achieves-native-ios-performance-with-react-native-390c84dcd502* (Accessed: 3.2.2021)

Esteban Angulo and Xavier Ferre. 2014. A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX. In Proceedings of the XV International Conference on Human Computer Interaction (Interacción '14).
Association for Computing Machinery, New York, NY, USA, Article 27, 1–8. *https://doi.org/10.1145/2662253.2662280*

Facebook Engineering. 2016. Dive into React Native performance. Retrieved from: *https://code.facebook.com/posts/895897210527114/dive-into-react-native-performance/* (Accessed: 3.2.2021)

Flutter. 2021a. Application showcase. Retrieved from: *https://flutter.dev/showcase* (Accessed: 3.2.2021)

Flutter. 2021b. Flutter documentation. Retrieved from: *https://flutter.dev* (Accessed: 3.2.2021)

Flutter. 2021c. Documentation: supported platforms. Retrieved from: *https://flutter.dev/docs/development/tools/sdk/release-notes/supported-platforms* (Accessed: 7.5.2021)

Flutter. 2021d. Layout in Flutter. Retrieved from: *https://flutter.dev/docs/development/ui/layout* (Accessed: 14.5.2021)

Flutter. 2021e. Adding interactivity to your Flutter app. Retrieved from: *https://flutter.dev/docs/development/ui/interactive* (Accessed: 14.5.2021)

Flutter. 2021f. Hot reload. Retrieved from: *https://flutter.dev/docs/development/tools/hot-reload* (Accessed: 14.5.2021)

Flutter. 2021g. Using the Flutter inspector. Retrieved from: *https://flutter.dev/docs/development/tools/devtools/inspector* (Accessed: 14.5.2021)

Flutter. 2021h. Flutter and the pubspec file. Retrieved from: *https://flutter.dev/docs/development/tools/pubspec* (Accessed: 24.5.2021)

Flutter. 2021i. Using packages. Retrieved from: *https://flutter.dev/docs/development/packages-and-plugins/using-packages* (Accessed: 24.5.2021)

Flutter. 2021j. Developing packages & plugins. Retrieved from: *https://flutter.dev/docs/development/packages-and-plugins/developing-packages* (Accessed: 24.5.2021)

Flutter. 2021k. Adding assets and images. Retrieved from: *https://flutter.dev/docs/development/ui/assets-and-images* (Accessed: 24.5.2021)

Flutter. 2021l. Internationalizing Flutter apps. Retrieved from: *https://flutter.dev/docs/development/accessibility-and-localization/internationalization* (Accessed: 24.5.2021)

Flutter. 2021m. Documentation: Writing custom platform-specific code. Retrieved from: *https://flutter.dev/docs/development/platform-integration/platform-channels* (Accessed: 7.5.2021)

Flutter. 2021n. Frequently asked questions. Retrieved from: *https://flutter.dev/docs/resources/faq* (Accessed: 25.5.2021)

Flutter. 2021o. Cupertino library. Retrieved from: *https://api.flutter.dev/flutter/cupertino/cupertino-library.html* (Accessed: 25.5.2021)

Flutter. 2021p. Insecure HTTP connections are disabled by default on iOS and Android library. Retrieved from: *https://flutter.dev/docs/release/breaking-changes/network-policy-ios-android* (Accessed: 25.5.2021)

Flutter. 2021p. Flutter architectural overview. Retrieved from: *https://flutter.dev/docs/resources/architectural-overview* (Accessed: 25.5.2021)

Flutter. 2021q. Security. Retrieved from: *https://flutter.dev/security* (Accessed: 25.5.2021)

Flutter. 2021r. An introduction to widget testing. Retrieved from: *https://flutter.dev/docs/cookbook/testing/widget/introduction* (Accessed: 3.6.2021)

GitHub: Apple. 2021. Darwin-XNU kernel Git reporitory. Retrieved from: *https://github.com/apple/darwin-xnu* (Accessed: 9.4.2021)

GitHub: Carthage. 2021. Carthage Git repository. Retrieved from: *https://github.com/Carthage/Carthage* (Accessed: 15.4.2021)

GitHub: CocoaPods. 2021. CocoaPods Git reporitory. Retrieved from: *https://github.com/CocoaPods/CocoaPods/* (Accessed: 15.4.2021)

GitHub: Flutter. 2021. Flutter Git repository. Retrieved from: *https://github.com/flutter/flutter* (Accessed: 7.5.2021)

GitHub: Flutter engine. 2021. Flutter engine Git repository. Retrieved from: *https://github.com/flutter/engine* (Accessed: 7.5.2021)

Global Web Index. 2019. Which Smartphone Features Really Matter to Consumers? Retrieved from: *https://blog.globalwebindex.com/chart-of-the-week/smartphone-features-consumers/* (Accessed: 25.2.2021)

Google. n.d. Application Security. Retrieved from: *https://www.google.com/about/appsecurity/* (Accessed: 28.5.2021)

Google Android source. 2020. Android documentation: Android Runtime (ART) and Dalvik. Retrieved from: *https://source.android.com/devices/tech/dalvik#features* (Accessed: 25.3.2021)

Google Developers. 2019. Android documentation: Localize your app. Retrieved from: *https://developer.android.com/guide/topics/resources/localization* (Accessed: 3.4.2021)

Google Developers. 2020a. Android documentation: Build and run your app. Retrieved from: *https://developer.android.com/studio/run* (Accessed: 25.3.2021)

Google Developers. 2020b. Android documentation: Layouts. Retrieved from: *https://developer.android.com/guide/topics/ui/declaring-layout* (Accessed: 28.3.2021)

Google Developers. 2020c. Android documentation: ViewGroup class. Retrieved from: *https://developer.android.com/reference/android/view/ViewGroup* (Accessed: 28.3.2021)

Google Developers. 2020d. Android documentation: Activity class. Retrieved from: *https://developer.android.com/reference/android/app/Activity* (Accessed: 3.4.2021)

Google Developers. 2020e. Android documentation: App resources overview. Retrieved from: *https://developer.android.com/guide/topics/resources/providing-resources* (Accessed: 3.4.2021)

Google Developers. 2020f. Android documentation: String resources. Retrieved from: *https://developer.android.com/guide/topics/resources/string-resource* (Accessed: 3.4.2021)

Google Developers. 2021a. Android documentation: App Standby Buckets. Retrieved from: *https://developer.android.com/topic/performance/appstandby* (Accessed: 19.3.2021)

Google Developers. 2021b. Android studio downloads. Retrieved from: *https://developer.android.com/studio#downloads* (Accessed: 25.3.2021)

Google Developers. 2021c. Android documentation: Platform architecture. Retrieved from: *https://developer.android.com/guide/platform* (Accessed: 25.3.2021)

Google Developers. 2021d. Android documentation: Android's Kotlin-first approach. Retrieved from: *https://developer.android.com/kotlin/first* (Accessed: 25.3.2021)

Google Developers. 2021e. Android documentation: Meet Android studio. Retrieved from: *https://developer.android.com/studio/intro* (Accessed: 27.3.2021)

Google Developers. 2021f. Android documentation: Add build dependencies. Retrieved from: *https://developer.android.com/studio/build/dependencies* (Accessed: 3.4.2021)

Google Developers. 2021g. Android documentation: Project overview. Retrieved from: *https://developer.android.com/studio/projects* (Accessed: 3.4.2021)

Gradle inc. 2021. Gradle documentation: What is Gradle?. Retrieved from: *https://docs.gradle.org/current/userguide/what_is_gradle.html* (Accessed: 3.4.2021)

Gradle inc. 2021. Gradle documentation: Writing Build Scripts. Retrieved from: *https://docs.gradle.org/current/userguide/writing_build_scripts.html* (Accessed: 3.4.2021)

GSMA Intelligence. Unique mobile subscribers. Retrieved from: *https://www.gsmaintelligence.com/data/* (Accessed 3.2.2021)

Haire Andrew. How to Pick the Right Mobile Development Approach?. Retrieved from: *https://ionicframework.com/resources/articles/how-to-pick-the-right-mobile-development-approach* (Accessed: 5.2.2021)

Hansen Hsu. 2017. A Short history of Objective-C. Retrieved from: *https://medium.com/chmcore/a-short-history-of-objective-c-aff9d2bde8dd* (Accessed: 9.4.2021)

Instagram Engineering. 2017. React Native at Instagram. Retrieved from: *https://instagram-engineering.com/react-native-at-instagram-dd828a9a90c7#.3h4wir4zr* (Accessed: 3.2.2021)

Jarkko Saarinen. 2019. Evaluating cross-platform mobile app performance with video-based measurements. Retrieved from: *http://urn.fi/URN:NBN:fi:tuni-201905161720* (Accessed: 25.5.2021)

Karl Wiegers, Joy Beatty. 2013. *Software Requirements, Third Edition*.

Kathy Walrath. 2019. Dart asynchronous programming: Isolates and event loops. Retrieved from: *https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bffc3e296a6a* (Accessed: 7.5.2021)

Khachouch Mohamed Karim. Korchi Ayoub. Lakhrissi Younes. Moumen Anis. 2020. Framework Choice Criteria for Mobile Application Development. In: Proc. of the 2nd International Conference on Electrical, Communication and Computer Engineering, 12-13.

Kotlin. 2021. Documentation: Javascript overview. Retrieved from: *https://kotlinlang.org/docs/js-overview.html* (Accessed: 27.3.2021)

Microsoft. What is Xamarin?. Retrieved from: *https://dotnet.microsoft.com/learn/xamarin/what-is-xamarin* (Accessed: 3.2.2021)

Pub.dev. 2021a. Pigeon package. Retrieved from: *https://pub.dev/packages/pigeon* (Accessed: 25.5.2021)

Pub.dev. 2021b. Animations package. Retrieved from: *https://pub.dev/packages/animations* (Accessed: 25.5.2021)

Pub.dev. 2021c. HTTP package. Retrieved from: *https://pub.dev/packages/http* (Accessed: 28.5.2021)

Pub.dev. 2021d. Crypto package. Retrieved from: *https://pub.dev/packages/crypto* (Accessed: 28.5.2021)

Pub.dev. 2021e. Crypto package. Retrieved from: *https://pub.dev/packages/local_auth* (Accessed: 28.5.2021)

Pub.dev. 2021f. Oauth2 package. Retrieved from: https://pub.dev/packages/oauth2 (Accessed: 28.5.2021)

Pub.dev. 2021g. Cryptography package. Retrieved from: *https://pub.dev/packages/cryptography* (Accessed: 28.5.2021)

Pub.dev. 2021h. Flutter secure storage package. Retrieved from: *https://pub.dev/packages/flutter_secure_storage* (Accessed: 28.5.2021)

Pub.dev. 2021i. Packages. Retrieved from: *https://pub.dev/packages* (Accessed: 2.6.2021)

Pub.dev. 2021j. test package. Retrieved from: *https://pub.dev/packages/test* (Accessed: 3.6.2021)

Ray Rischpater. 2014. Providing operating system compatibility on a large scale. Retrieved from: *https://medium.com/flutter/providing-operating-system-compatibility-on-a-large-scale-374cc2fb0dad* (Accessed: 25.5.2021)

React Native. Introduction. Retrieved from: *https://reactnative.dev/docs/getting-started* (Accessed: 3.2.2021)

Singhai, Amit & Bose, Joy. 2013. Reducing Power Consumption in Graphic Intensive Android Applications. 10.1109/COMSNETS.2014.6734921.

Statcounter GlobalStats. 2021. Mobile Operating System Market Share Worldwide. Retrieved from: *https://gs.statcounter.com/os-market-share/mobile/worldwide* (Accessed: 3.2.2021)

Statista. 2020. Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020. Retrieved from: *https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/* (Accessed: 18.3.2021)

Swift. 2021a. About Swift. Retrieved from: *https://swift.org/about/* (Accessed: 9.4.2021)

Swift. 2021b. Platform support. Retrieved from: *https://swift.org/platform-support/* (Accessed: 10.4.2021)

Swift. 2021c. Package Manager. Retrieved from: *https://swift.org/package-manager/* (Accessed: 15.4.2021)

Tim Sneath. 2020. Flutter Spring 2020 Update. Retrieved from: *https://medium.com/flutter/flutter-spring-2020-update-f723d898d7af* (Accessed: 28.5.2021)

Tim Sneath. 2021. Announcing Flutter 2.2 at Google I/O 2021. Retrieved from: *https://medium.com/flutter/announcing-flutter-2-2-at-google-i-o-2021-92f0fcbd7ef9* (Accessed: 28.5.2021)