F

Jakhongir Fayzullaev

Native-like Cross-Platform Mobile Development

Multi-OS Engine & Kotlin Native vs Flutter

Bachelor's thesis
Information Technology

2018

South-Eastern Finland
University of Applied Sciences

XAMK
South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Jakhongir Fayzullaev	Bachelor of Engineering	May 2018

Thesis title	
Native-like Cross-Platform Mobile Development Multi-OS Engine & Kotlin Native vs Flutter	62 pages 0 pages of appendices

Commissioned by

Xamk

Supervisor

Timo Mynttinen

Abstract

The goal of this thesis was to study and demonstrate a relatively new way and tools of cross-platform mobile development and to research such technologies as Multi-OS Engine, Kotlin/Native and Flutter.

The thesis contains both theory and practice about how Multi-OS Engine, Kotlin/Native and Flutter work. The thesis discussed the theory of native development vs cross-platform development as well as other cross-platform development options. As these technologies are relatively new there are not many previous studies about the topic. The thesis aimed to provide illustrative examples to demonstrate how tools could be used, their features and a work example.

The thesis concludes with a case study where three sample applications were made using different tools. The main goal of the case was to showcase how these technologies could be used in real world. The case study serves as a demonstration of the possibilities of these three tools. At the end the advantages and disadvantages of both tools were found and the real use case of them.

Keywords

Android, iOS, Flutter ,Kotlin, Java, Dart, cross-platform, programming,

CONTENTS

1 INTRODUCTION 5

2 NATIVE VS CROSS-PLATFORM DEVELOPMENT 7

2.1 Why cross-platform? 7

2.1.1 Benefits of cross-platform development 9

2.1.2 Drawbacks of cross-platform development 9

2.2 Alternative tools 9

2.2.1 Xamarin 10

2.2.2 React Native 10

3 MULTI-OS ENGINE 11

3.1 Features 11

3.2 NatJGen 13

4 KOTLIN/NATIVE 14

4.1 Mission 15

4.2 Limitations 16

4.3 Working principles 16

5 FLUTTER 17

5.1 Widgets 17

5.2 Layout 19

5.3 Under the hood 20

6 APPLICATION 24

6.1 Technologies, tools and languages 24

6.1.1 Gradle 24

6.1.2 Android Studio 25

6.1.3 CLion 25

6.1.4 Xcode 25

6.1.5 AppCode 26
6.1.6 Languages 26
6.2 APIs and libraries..27
6.3 Implementation 27
6.4 iOS UI 29
7 MULTI-OS ENGINE APPLICATION 30
7.1 Setup Android part..31
7.2 Setup common part 32
7.3 Setup Multi-OS Engine 33
7.4 Configuration 34
7.4.1 Bindings..36
7.4.2 Custom bindings 37
7.4.3 Library bindings 40
8 KOTLIN NATIVE..41
8.1 Setup 41
8.2 Working principle 43
9 FLUTTER APPLICATION..44
10 COMPARISON 49
10.1 Multi-OS Engine & Kotlin/Native vs other cross-platform dev tools.......................49
10.2 Multi-OS Engine vs Kotlin/Native..49
10.2.1 Visual code difference..50
10.2.2 Compilation difference 53
10.3 Platform interaction differences 54
10.4 Future Plans 57
10.5 Result 57
11 CONCLUSION..60
REFERENCES 62

1 INTRODUCTION

The topic in this thesis is about relatively new ways of cross-platform mobile development. In this project three new technologies Multi-OS Engine, Kotlin Native and Flutter are studied. These tools enable cross platform mobile development and are mostly useful for Android developers as they are close to Java and Android world.

The current problem of mobile development is illustrated in Figure 1.

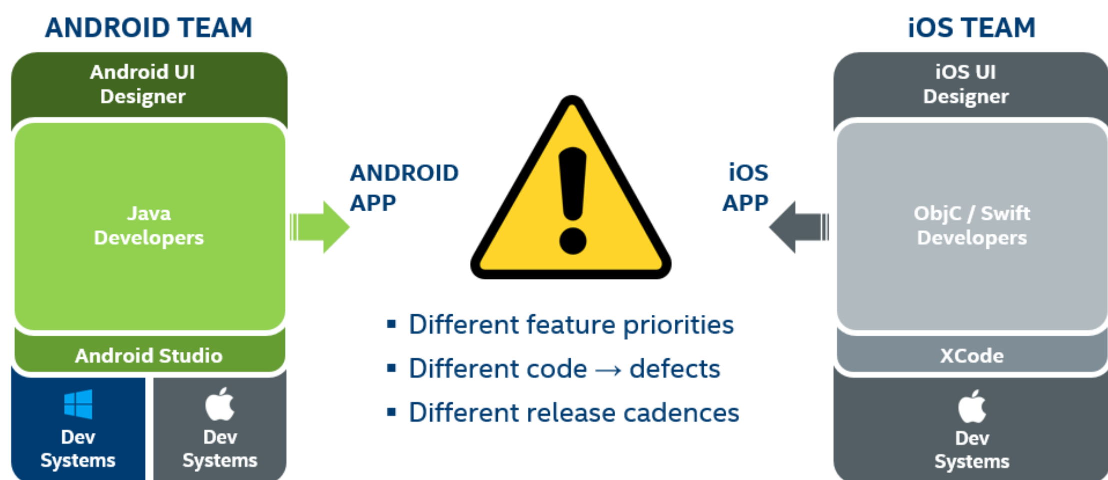


Figure 1. Current problem of mobile development (Intel Corporation 2016)

Figure 1 shows a problem of big difference of the same application on different platforms. It would be much better, if we could use one tool for both platforms. There are a lot of tools for this, like Apache Cordova and React Native. However, they mostly use a WebView which is often very slow and also doesn't look native to the platform. Furthermore, it is not always attractive and lacks some platform specific functionality.

The better alternative would be to have one shared code base which also preserves native look of apps. This is what Multi-OS Engine and Kotlin/Native aim to do (Figure 2).

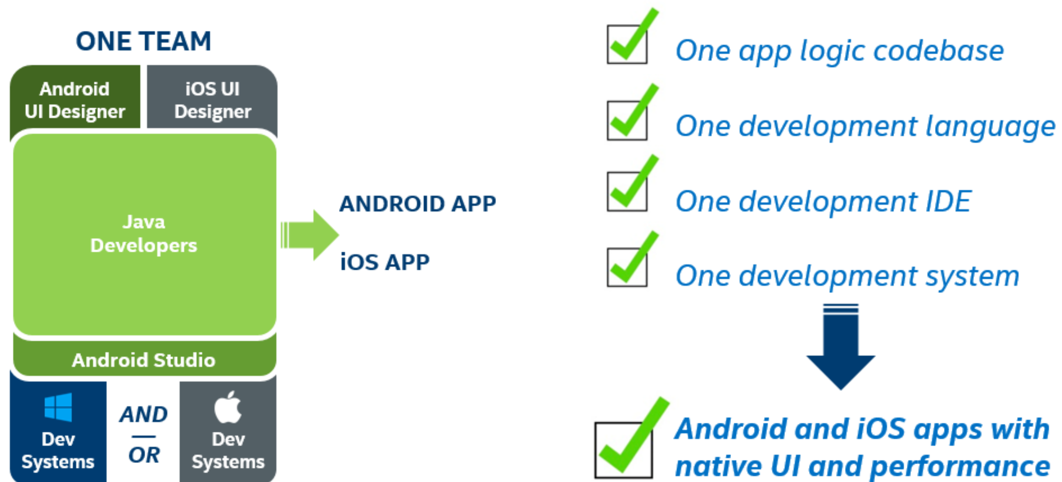


Figure 2. Solution for the problem of mobile development (Intel Corporation 2016)

Both tools give a possibility to develop native mobile applications for iOS and Android with only one language expertise, without compromising on the native look-and-feel or performance. This allows the re-use of as much of common code as possible and add platform-specific UI code for each platform.

The first one is Intel Multi-OS Engine which is now open source and located at their home page. It bases on Android and allows writing apps for iOS in any Android-compatible language. Multi-OS Engine lets develop native mobile applications for iOS and Android in Java or Kotlin language on Microsoft Windows and/or Apple macOS development machines without compromising on the native look-and-feel or performance. The biggest plus is that application have almost one code base for both platforms. However, user interfaces are written using default (native) tools.

The second one is new and even in early preview state technology called Kotlin/Native. Kotlin itself is a JVM compilating language which works well with Java. On the other hand, Kotlin/Native is a technology for compiling Kotlin to native binaries that run without any VM.

On the other hand, there is getting popular tool called Flutter. Flutter mobile SDK and UI framework for crafting high-quality native apps for iOS and Android which is supported by Google. It provides widgets and tools, that gives developers an easy and productive way to build and deploy beautiful mobile apps for both

platforms. Flutter uses Dart language for both UI declaration and code. The greatest advertised feature is Hot Reload of already running app. This allow to see code change almost immediately reflected in the running application. Flutter and Dart are open source and free to use. There is also a rumor that Flutter will be a main development tool for the next OS by Google called Fuchsia. So, it is worth investing time in it. (Flutter 2018.)

The aim of this project is evaluating these three technologies, comparing them, finding pros and cons of each of them and comparing with native development.

To reach this aim I'm planning to build three small apps for Android and iOS with same functionality using these three technologies.

2 NATIVE VS CROSS-PLATFORM DEVELOPMENT

Native application or a cross-platform one? This is the question which is often asked in developers' heads and by company managers. Both approaches have their own benefits. However, cross-platform development can actually divide into subclasses: near-native (or native like) cross-platform applications and hybrid ones.

2.1 Why cross-platform?

Nowadays, there are only two major mobile OS players on mobile market, Google and Apple. And their operating systems, Android and iOS respectively, are installed on almost all current smartphones. According to research firm Gartner (2018), 99.9% of smartphones sold worldwide last year were based on Android or iOS. So, often when one thinks about building an app, the first decisions person needs to make is to choose whether to start with Android or iOS. Even if the main goal may be to launch on both platforms eventually, it is risky and expensive to build an app for both iOS and Android simultaneously.

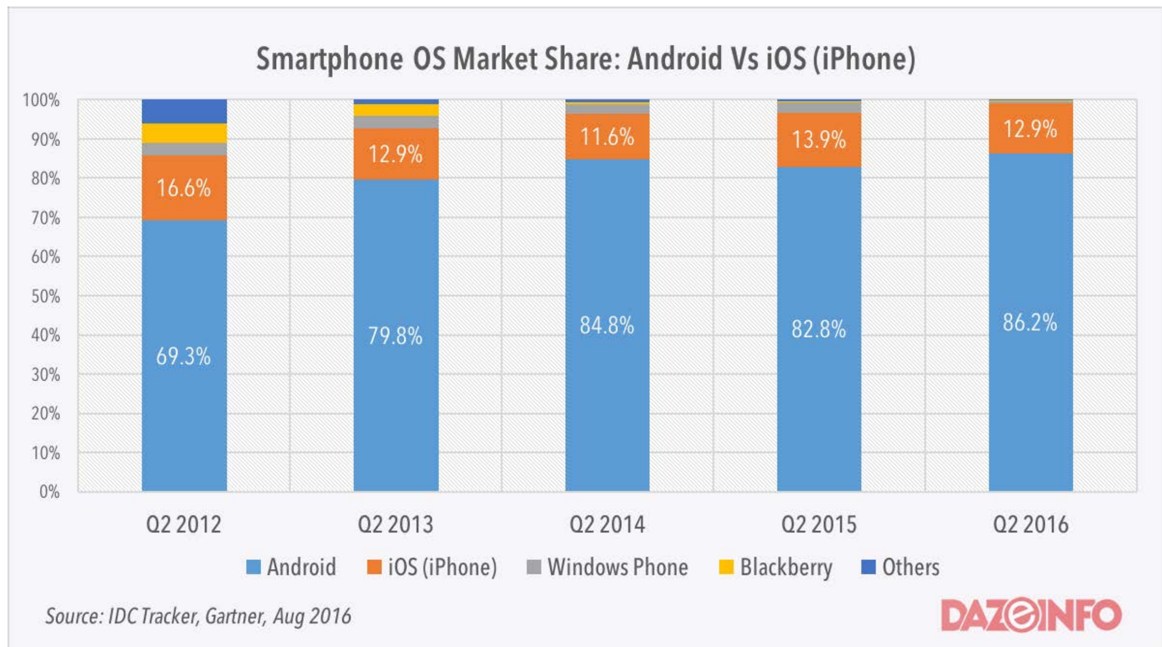


Figure 3. Worldwide mobile OS market share

Looking at Figure 3 may give a wrong feeling that developing just for Android is not such a bad option. However, if app monetization is one of the prime goals, there is something else to look at. As can be seen in Figure 4, Android users tend to be less willing to pay for apps than iOS users.

Figure 4. Downloads and revenues of app stores and forecast (App Annie 2017)

In fact, Apple App Store in 2016 generated almost twice as much revenue as Google Play, while having half of Google Play downloads number. Therefore, making an app for both platforms can be crucial. However, developing an app cab be expensive, and if each platform needs its own app it doubles the price, and so hybrid apps can be a solution.

2.1.1 Benefits of cross-platform development

Just the idea of cross-platform development sounds good, as it has a lot of benefits and some of them are:

- Shorter development time
- Cost-effectiveness
- Exposure to a larger number of users
- Updates synchronization

The shorter development time can be achieved choosing the right tech stack and thoroughly planned architecture of the project, so that it would be possible to re-use up to 80% of the original codebase (Intel Corporation 2016). So, instead of writing new code for every platform, developers can reuse the same code across all platforms.

Building a native mobile application is not cheap. Moreover, if application should be for both platform the price is at least doubled. As it was obvious from previous topic, developing an application that runs both on iOS and Android gives the added advantage of entering into a greater market and most cross-platform development tools allow developing for both Android and iOS or even more platforms. Furthermore cross-platform solution may require less developers and in combination with code reuse the application may reach the market sooner.

2.1.2 Drawbacks of cross-platform development

Even if cross-platform development has a lot of benefits it’s also has a lot of disadvantages and the most noticeable is performance and non-usual user interface compared to native applications. Often cross-platform solutions are slow, look bad and don’t have all the benefits and features of native apps. It could be said that such apps are easily spotted as look foreign to the platform.

2.2 Alternative tools

There are a lot of different tools for cross-platform development. The closest to Multi-OS Engine and Kotlin Native is Xamarin, not Xamarin.Forms which instead of being close to native development try to hide native development issues. However, Xamarin use C# language which is not native to neither Android or iOS.

The closes to Flutter tool is React Native, which however tries to be close to platform and uses native platform drawing mechanics, while Flutter uses fully brand-new rendering engine for drawing content.
The next sections describe Xamarin and React Native features shortly.

2.2.1 Xamarin

Xamarin platform allows to develop applications for iOS, Android and Windows Phone using C# language and .NET framework. Xamarin tools are available for Visual Studio on Windows and Mac. Xamarin is supported by Microsoft Corporation.

Xamarin platform allows writings apps not only in one language but also provides possibility to write cross-platform UI. There are actually two different ways to build the UI. It's possible to use the original native way of building the UI or another option is to use Xamarin.Forms. Xamarin.Forms gives a possibility to build UI for different platforms all at once. There is almost 100% code sharing if one decides to choose Forms over Native UI Technology. (The Windows Club 2017.)

2.2.2 React Native

React Native is JavaScript framework supported by Facebook that uses the JavaScript syntax to build mobile apps. It uses the same design as JavaScript React framework, which is quite popular among Web developers. The biggest plus is that react Native has a native fluent performance as it uses the same fundamental UI widgets as native iOS and Android apps. This is all possible because of the "connector," which provides React with an interface into the host platform's native UI elements. React Native currently supports iOS and Android. Because of the abstraction layer provided by the Virtual DOM, React Native could target other platforms too. The only thing is that there needs to be a written connector. (Subham A. 2017.)

React Native now is quite popular, as it is fast and look native to platform. However, the tool is still young, and haven't reached the stable state.

3 MULTI-OS ENGINE

Intel’s Multi-OS Engine Technology gives possibility to use Java capabilities to develop native mobile applications for Apple iOS and Android devices providing the native look, feel and performance. This technology provides a stand-alone plug-in that integrates into Android Studio on Windows and Apple macOS development machines.

An application starts as an Android project in Android Studio. The Multi-OS Engine configures the project to build and run as an iOS app on the iOS simulator that can be invoked from Android Studio or on a real device. The development process is illustrated in Figure 5.

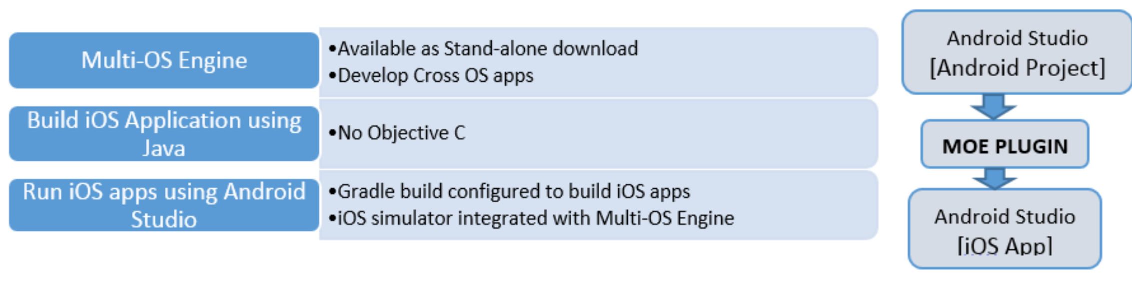


Figure 5. Development process with Multi-OS Engine (Intel Corporation 2016)

3.1 Features

The key features of Multi-OS Engine are:

- Java support on iOS devices
- Developing iOS apps in Java instead of Apple Objective-C
- Direct access to platform-specific UI components for either of the supported platforms
- Native multithreading support
- Debugging apps on real devices or the iOS simulator integrated with Android Studio
- Running/deploying apps from Apple App Store

Multi-OS Engine Runtime is based on the modern Android ART, which is the runtime component of Android that runs Java apps.
ART has a list of features that provide optimal performance for apps on iOS devices:

- Ahead-of-time (AOT) compilation, which can improve app performance
- Use of the same Java runtime libraries as Android, which simplify cross-platform app development
- Enhanced memory management and garbage collection

A compiled Multi-OS Engine app contains the following components (Figure 6):

- Compiled Java sources
- Resources
- Standard (iOS) library bindings
- Third party native libraries and bindings
- Nat/J native library for the Java to native binding that enables the implementation of native classes and functions in pure Java and makes them available to the native side
- The specialized ART virtual machine (VM) with Multi-OS Engine ART enhancements

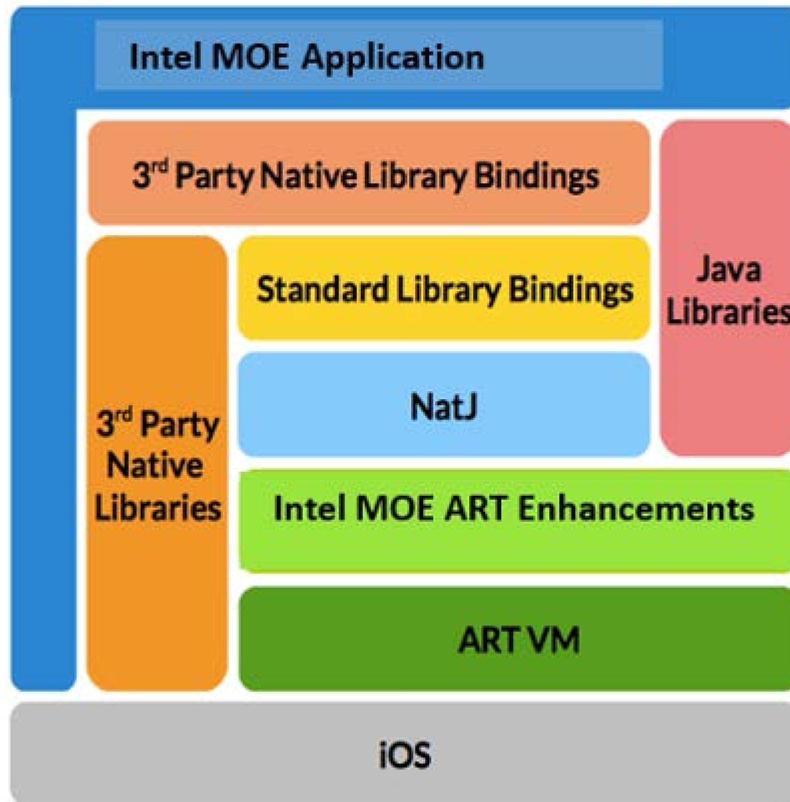


Figure 6. Multi-OS Engine app components (Intel Corporation 2016)

When an iOS app launches, it starts the ART VM and executes the pre-compiled code on it.

3.2 NatJGen

One of the useful features of Multi-OS Engine is the generation of Java bindings. NatJGen is a command line tool which implements this feature. The NatJ library provided with the Intel's Multi-OS Engine is a bridge between the native code and Java code. Using NatJ, it is possible to easily bind or even extend native implementations in Java code.

NatJGen allows generate Java bindings based on special configuration file. NatJGen is a low-level tool and its configuration is quite complicated so why wrapnatjgen wrapper was created to simply the tool using.

Wrapnatjgen is a wrapper of NatJGen which provides a useful command line interface and avoids the need of writing configuration files with special format. Wrapnatjgen tool helps to generate Java bindings for selected native header files. Tool also allows for frameworks or libraries to create JAR files with precompiled Java bindings, resources (if needed), additional linker flags and frameworks or libraries itself. This approach is very useful because once compiled, such JAR files can be used in different projects on different machines. Build procedure in Android Studio allows to easily link frameworks/libraries contained in JAR to Xcode project and also all the specified resources necessary for the framework of libraries will be copied to the final app file.

Intel's Multi-OS Engine provided tool called WrapNatJGen which can help to efficiently use native methods in Java-based applications. The features of WrapNatJGen tool are:

- Generate Java bindings for selected native header files.
- Generate JAR files that include precompiled Java bindings for frameworks or libraries, as well as additional resources or linker flags if required.
- Generate JAR files based on CocoaPods specs.

Wrapnatjgen may be used as command line tool with appropriate arguments or as a context menu item in Android Studio where all the functions of the tool are integrated. (Intel Corporation 2016.)

4 KOTLIN/NATIVE

Kotlin/Native is a relatively new technology, which is currently in development state and available as early preview. Kotlin itself is a JVM compilating language which is almost 100% interoperable with Java. That means that one can use Kotlin and any Java written libraries, or even continue writing any Java project using Kotlin without breaking things and compatibility. However, Kotlin/Native is a technology for compiling Kotlin to native binaries that run without any Virtual Machine (VM). It comprises a LLVM-based backend for the Kotlin compiler and a native implementation of the Kotlin runtime library. So Kotlin/Native loses all benefits of a Java world, which means no Java libraries available during development, and this is for now a big minus as there are a lot of awesome or even crucial libraries written in Java. On the other hand, according to JetBrains (2017) Kotlin/Native can be compiled to the next platform targets, giving an opportunity to share the same code base:

- Windows (x86_64 only at that moment)
- Linux (x86_64, arm32, MIPS, MIPS little endian)
- MacOS (x86_64)
- iOS (arm64 only)
- Android (arm32 and arm64)
- WebAssembly (wasm32 only)

Multiplatform Kotlin project (Figure 7) is composed from different types of modules. Kotlin specification defined following types of modules:

- A common module contains code that is not specific to any platform, as well as declarations without implementation of platform-dependent APIs. Those declarations allow common code to depend on platform-specific implementations.
- A platform module contains implementations of platform-dependent declarations in the common module for a specific platform, as well as other platform-dependent code. A platform module is always an implementation of a single common module.
- A regular module. Such module targets a specific platform and can either be dependency of a platform module or depend on platform module. (JetBrains 2018.)

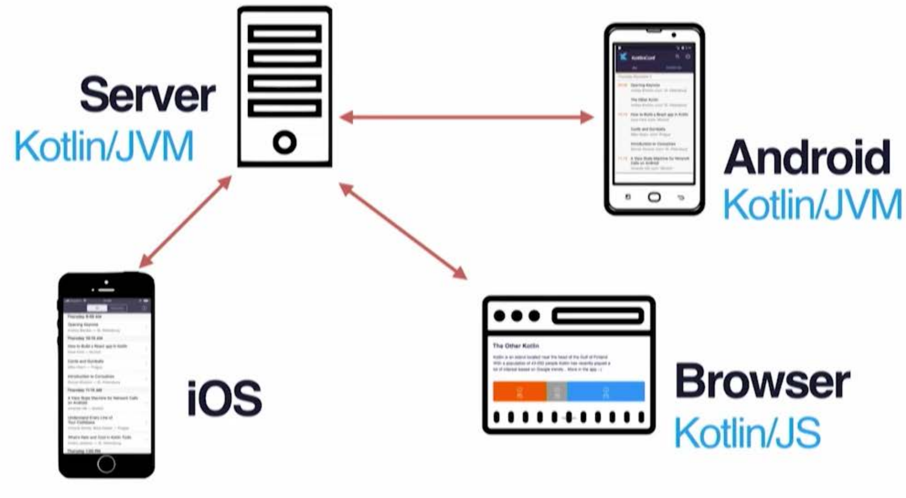


Figure 7. Multiplatform vision using Kotlin Native (JetBrains 2018)

Since Kotlin version 1.2 cross-platform shared code along all these platforms is allowed. (JetBrains 2017.)

4.1 Mission

Kotlin/Native is another step toward making Kotlin usable throughout a modern application. Eventually, it will be possible to use Kotlin to write every component, from the server back-end to the web or mobile clients. Sharing the skill set is one big motivation for this scenario. Another is sharing actual code. (JetBrains 2017.)

Kotlin team sees inter-platform code reuse as follows: one can write entire modules in Kotlin in a platform-independent way and compile them for any supported platform (currently these are Kotlin/JVM, Kotlin/JS and the upcoming Kotlin/Native). These modules are called common modules. Parts of a common module may require a platform-specific implementation, which can be developed individually for each platform. Common modules provide a common API for all clients, but other (platform-specific) modules can extend this API to provide some exclusive capabilities on their platform.

Even so that Kotlin is a JVM compilating language, the Kotlin team is not planning to make arbitrary Kotlin/JVM programs runnable on Kotlin/Native or

Kotlin/JS. The reason is that making this is equivalent to implementing another JVM, which is both a lot of work and a lot of limitations for the runtime. For now Kotlin team sees their product as a common language for all platforms while enabling creation of common libraries through seamless interoperability with platform code.

4.2 Limitations

As mentioned above, Kotlin/Native, at the moment of writing, is far from complete, it is a technology preview which has a number of limitations that will be eliminated at later stages:

- No performance optimization has been done yet, so benchmarking Kotlin/Native makes no sense at this point.
- The Standard Library and reflection support are far from complete, more APIs will be added later.

However, majority of limitations should be resolved by the time of a stable release. Now the product is early access state and reached version 0.6. But there are still well implemented tools are missing. (JetBrains 2017.)

4.3 Working principles

Kotlin Native is also uses some similar approach as Multi-OS Engine NatJGen tool for generating Objective-C code.
First headers for using Objective-C via Kotlin is generated with a tool called cInterop. The rules for generating these headers are stored on a so called ".def" file format.
Then, created Objective-C code is need to be compiled to LLVM byte code (.bc) using LLVM clang compiler.
After compiling to LLVM byte code, Objective-C header and Kotlin source files are send to Kotlin konanc compiler, which also generates LLVM byte code.
In the end all byte code is merged into one using LLVM-LTO.
Kotlin Native allows to have projects written in Kotlin and using Objective-C files and libraries and also allows to be used as a compiled library, so it can be used in any iOS project which uses Objective-C.

5 FLUTTER

Flutter started as an experiment performed by members of the Chrome browser team at Google. They wanted to see whether it is possible to build a fast rendering engine while ignoring the traditional model of layout. In a few weeks, significant performance gains were achieved and that is what was discovered:

- Most layout is relatively simple, such as: text on a scrolling page, fixed rectangles whose size and position depend only on the size of the display, and maybe some tables, floating elements, etc.
- Most layout is local to a subtree of widgets, and that subtree typically uses one layout model, so only a small number of rules need to be supported by those widgets.

After investigating all this information Flutter team come up with an idea that the layout can be simplified significantly if changed heavily:

- Instead of having a large set of layout rules that could be applied to any widget, each widget would specify its own simple layout model.
- Because each widget has a much smaller set of layout rules to consider, layout can be optimized heavily.
- To simplify layout even further, almost everything was turned into a widget.

5.1 Widgets

Widgets are the basic building blocks of a Flutter application user interface. Each widget is an immutable declaration of part of the user interface. Unlike other frameworks or native platform tools that separate views, view controllers, layouts, and other properties, Flutter has a consistent, unified object model - the widget
A widget can define:

- a structural element (like a button or menu)
- a stylistic element (like a font or color scheme)
- an aspect of layout (like padding)
- and so on…

In other words, in Flutter everything is widget!

Widgets are the elements that affect and control the view, feel of application and its look. It is not an overstatement to say that the widgets are one of the most important parts of a mobile app.

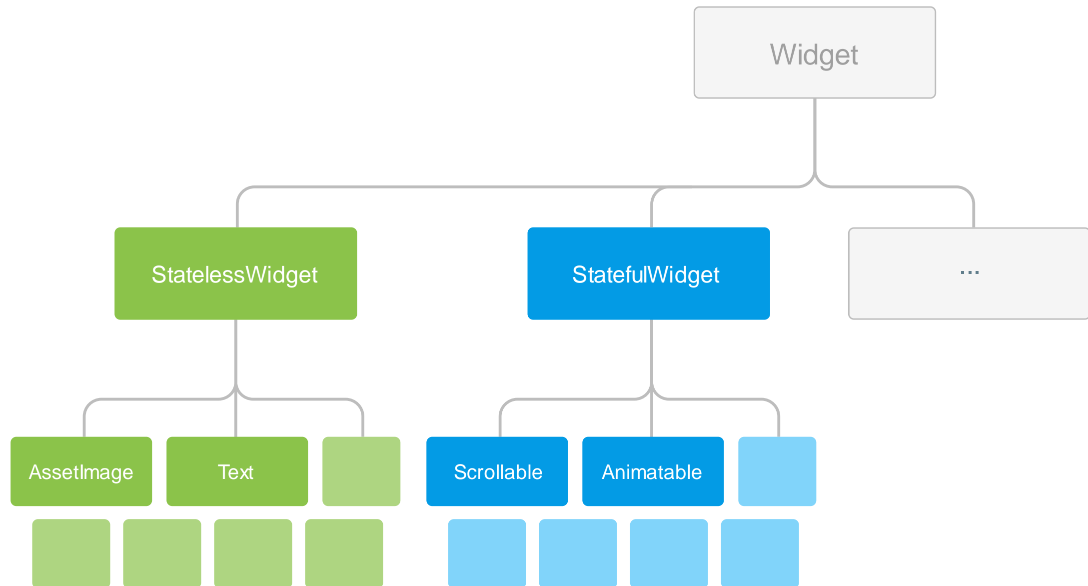


Figure 8. Flutter Widget class hierarchy

As widgets are so important, they need to look good, including on various screen sizes. They also need to feel natural. Moreover, widgets must perform as fast as possible. Creation of the widget tree, inflating the widgets (instantiating their children), displaying them out on the screen, render them, or animating widgets, all of this need time to be done, and it should be as low as possible to have consistent 60FPS experience.
For modern apps, widgets should be extensible and customizable. Developers want to be able to add delightful new widgets and customize all widgets to match the application/company brand.

Flutter has a new architecture that includes widgets that look and feel good, are fast, and are customizable and extensible. The main point is that Flutter does not use the platform or OEM widgets, it provides its own widgets (Figure 8).

5.2 Layout

One of the biggest improvements in Flutter is how it does layout. Layout determines the size and position of widgets based on a set of rules.

Traditionally, layout uses a large set of rules that can be applied to (virtually) any widget. The rules implement multiple layout methods. To take as an example Android XML. It has a lot of properties and attributes, which are applied to all view elements. Each widget may have their own attribute. Moreover, parent layout models are already predefined, and you need to follow it rules. This result in less space for optimization and a lot of hacks as writing own layout parent is problematic and may not worth it.

Another problem with traditional layout is that the rules can interact (and even conflict) with each other, and elements often have dozens of rules applied to them. This makes layout slow. Even worse, layout performance is typically of order N-squared, so as the number of elements increases, layout slows down even more.

Flutter is simple and reader friendly. Quite simple widget tree is presented in Code1.

```
Card(child: ListTile(
      leading: new CachedNetworkImage(
        placeholder: new Icon(Icons.attach_money),
        imageUrl: currency.getImageUrl(),
      ),
      title: new Text(currency.getText(),
      style: new TextStyle(fontWeight: FontWeight.bold)),
    ));
```

Code 1. Example widget tree layout.

This code semantic is enough that to easily imagine what it will produce, the result can be seen on Figure 9. In this code everything is a widget, except TextStyle. The Card widget wraps its child inside card. The ListTile layout widget arranges its children so that leading widget is drawn on the left and after it the Title widget is displayed.

1 EUR = 1.5197 CAD

Figure 9. The result of running code from Code 1

In Flutter, centering and padding are widgets. Themes are widgets, which apply to their children. And even applications and navigation are widgets.

Flutter includes quite a few widgets for doing layout, not just columns but also rows, grids, lists, etc. In addition, Flutter has a unique layout model we call the “sliver layout model” which is used for scrolling. Layout in Flutter is so fast it can be used for scrolling. Think about that for a moment. Scrolling must be so instantaneous and smooth that the user feels like the screen image is attached to their finger as they drag it across the physical screen.
By using layout for scrolling, Flutter can implement advanced kinds of scrolling with lots of animation.

5.3 Under the hood

Flutter is built with C, C++, Dart and Skia graphics engine (Figure 10).

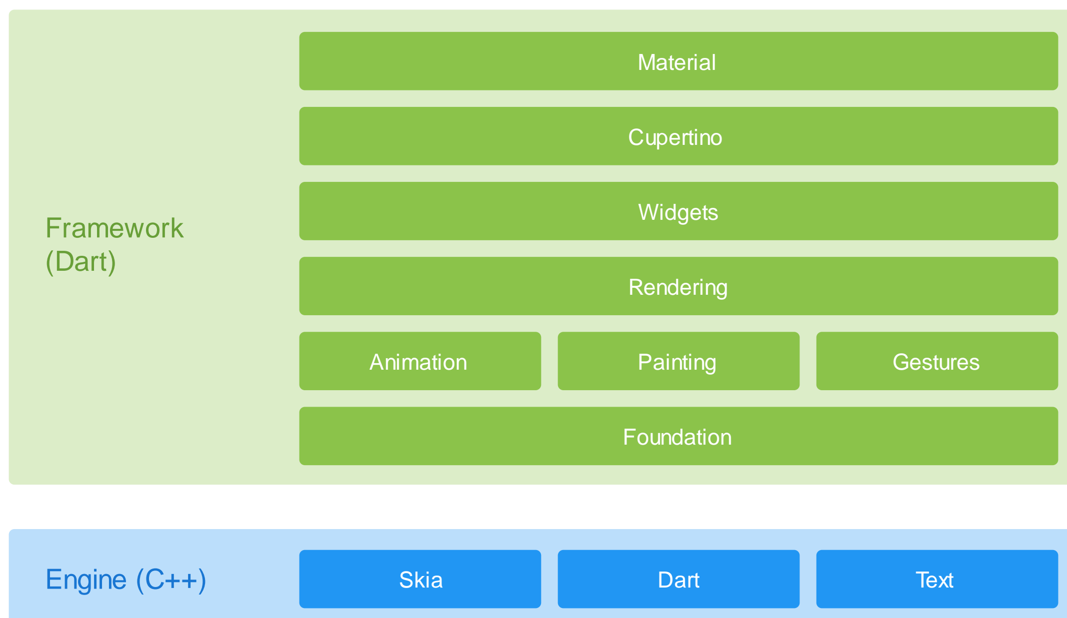


Figure 10. Flutter framework and engine contents

Flutter uses Dart for building components and under the hood uses Skia 2D graphics engine to bring code to life. Flutter also includes a modern react-style framework. The content of the framework is illustrated in Figure 10. At lowest level all the UI code uses Skia to render the application UI (Figure 11). Flutter runs most of its framework and application code inside a lightweight Dart VM. The framework code is written in Dart whereas the rendering engine is implemented in C++.

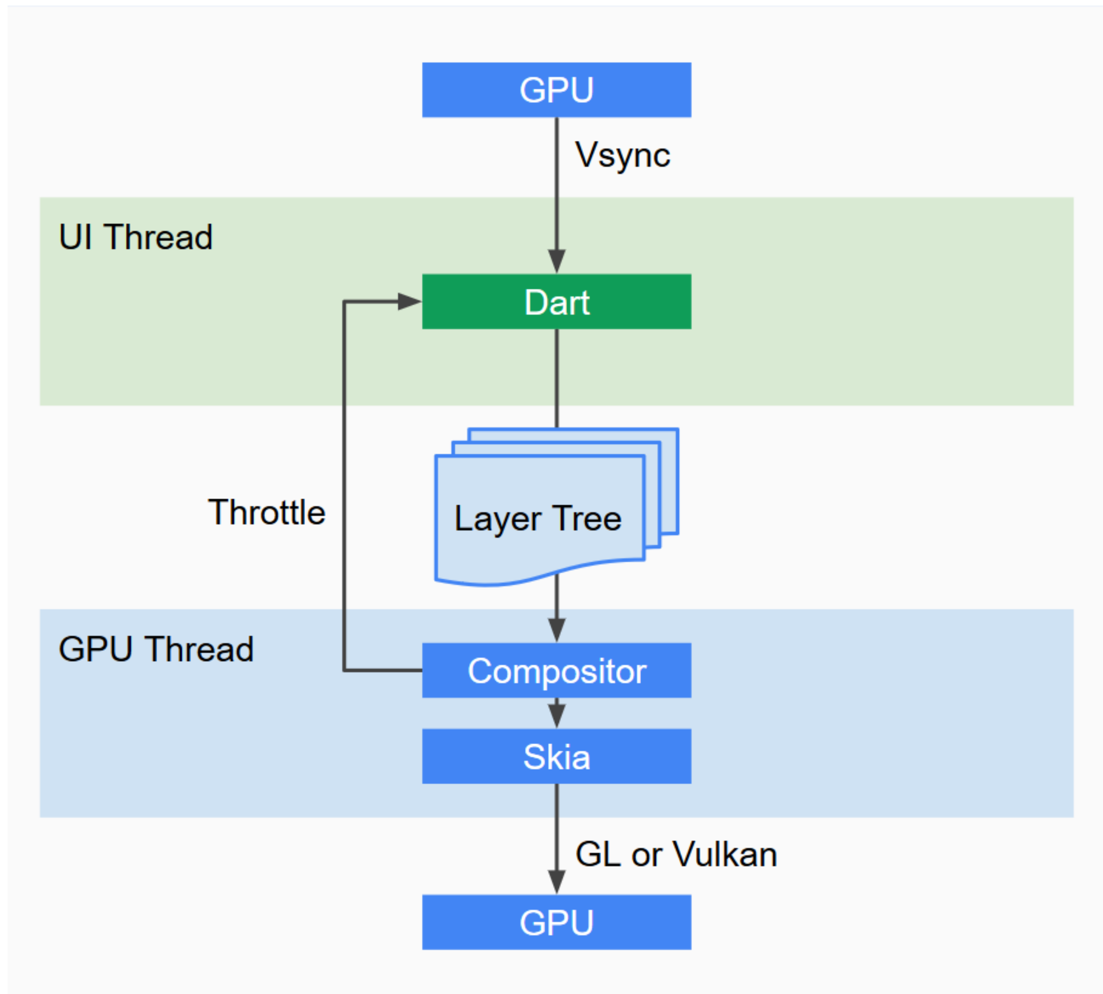


Figure 11. Flutter drawing pipeline

The Dart source code is compiled to native code using Dart's AoT (Ahead of Time) compilation feature. However, it still needs the Dart VM (Virtual Machine) to run. On Android the engine’s C/C++ code is compiled with Android’s NDK, and all Dart code is AOT-compiled into native code. On iOS the engine’s C/C++ code is compiled with LLVM (Low Level Virtual Machine), and all Dart code is also

AOT-compiled into native code. The app runs using the native instruction set in both cases.

Flutter can also access all platform services like sensors and storage. Flutter already provides a wide number of platform services and APIs via packages. However, if there is a need in additional native functionality it is possible to use Flutter services library, using which a platform channel can be implemented and this can be used to call platform specific functions from Dart and vice versa (Figure 12). For example, on Android it's possible to access Java functions and on iOS it is possible to have access to Objective-C functions. Flutter also supports building custom plugins that allow to call out to native platform code. (Flutter 2018.)

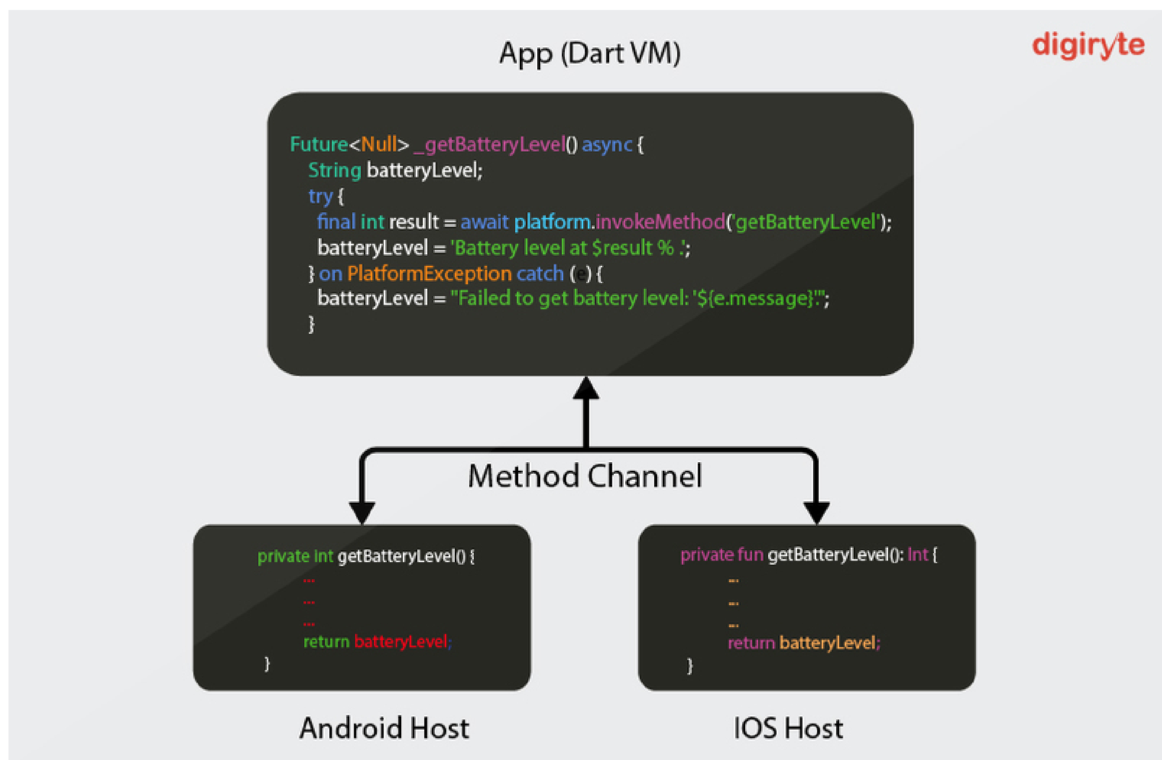


Figure 12. Example of platform features access using Flutter

All platform features are accessed asynchronously and so doesn't slow down the UI.

6 APPLICATION

The application is called Currency Observer and represent a simple exchange rate checker app, which shows world currency and cryptocurrency exchange rates.
The main work of application is to fetch data using REST API and displaying it nicely via a platform native UI.

6.1 Technologies, tools and languages

All manipulations were done on macOS High Sierra version 10.13.3.
The Xcode version was 9.3 and Android Studio 3.0.1
Multi-OS Engine SDK version 1.4.2 was used.
Initially, Kotlin/Native SDK version 0.6.2 was planned as the latest stable version, however due to some issues I had to use development build version 0.7-dev-1440. By the end of the thesis there was a new stable release of version 0.7, and project was updated accordingly.

6.1.1 Gradle

Gradle is an open-source build automation system. It is built upon the concepts of Apache Ant and Apache Maven. Gradle combines the good parts of both tools and provides additional features and uses Groovy as a Domain Specific Language (DSL), also support for Kotlin as DSL was added. It has power and flexibility of Ant tool with Maven features such as build life cycle and ease of use. (Wikipedia, 2018)

Kotlin/Native multiplatform projects use Gradle as a build tool. For this a Kotlin/Native Gradle plugin, called konan, required. It is also open source and available at JetBrains GitHub.

Multi-OS Engine also uses Gradle as a build tool. This Gradle plugin called MOE and version 1.4.3 was used for this project. The plugin available at GitHub.

6.1.2 Android Studio

Android Studio is the official integrated development environment by Google, as stated by its name it's primary target use is Android app development. It's built on top of JetBrains' IntelliJ IDEA IDE.
Even so it's intended for Android development it can perfectly fit for Multi-OS Engine as it also have a Java code and build using Gradle system. Moreover, it allows writing both Android and iOS code in single IDE, familiar for Android developers. To enable Multi-OS Engine only additional MOE plugin required.

6.1.3 CLion

CLion is a C/C++ IDE built on top of the IntelliJ platform. Kotlin support in CLion is provided via a couple of plugins, the core support for the Kotlin language is provided by the Kotlin plugin, and to provide functionality for native the Kotlin/Native plugin is used.
It is possible to write Kotlin/Native code in any editor like any other language, however, there is a real fully-fledged IDE experience which JetBrains provides support via CLion. However, CLion is more intended for native development and so it fits better for development to desktop, embedder and similar platforms.

6.1.4 Xcode

Xcode is an integrated development environment (IDE) for macOS containing a suite of software development tools developed by Apple for developing software for macOS, iOS, watchOS, and tvOS. The Xcode IDE is the center of application development for Apple devices. (Apple Inc 2018.)

Both Multi-OS Engine and Kotlin Native don't support building iOS UI and only Xcode have Interface Builder. Moreover, both tools require Xcode to modify settings of iOS project. AppCode can work with Xcode project options but it's capabilities are quite limited, and it's suggested to use Xcode for such kind of tasks.

6.1.5 AppCode

AppCode is an integrated development environment (IDE) for Swift, Objective-C, C, C++, and JavaScript development built on top of JetBrains' IntelliJ IDEA platform. AppCode is an attempt by JetBrains to replace Xcode, it uses the same project model and project file, and synchronizes all changes with Xcode. As not features are always available in AppCode, it's possible to work simultaneously in both IDEs.
I was not planning to use AppCode, but as of April 11, 2018 Kotlin/Native support is available in AppCode via plugin and so I decided to use it for Kotlin/Native development purpose instead of CLion as it's better suited for iOS development.

6.1.6 Languages

Java is general purpose language which derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them. Java is default language for Android app development. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

Kotlin is a general purpose, open source, statically typed “pragmatic” programming language for the JVM and Android that combines object-oriented and functional programming features. It is focused on interoperability, safety, clarity, and tooling support. Kotlin is designed to be an industrial-strength object-oriented language, and a "better language" than Java, but still be fully interoperable with Java code, allowing to make a gradual migration from Java to Kotlin or coexisting.

Objective-C is the primary programming language you use when writing software for OS X and iOS. It's a superset of the C programming language and provides object-oriented capabilities and a dynamic runtime. Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods. (Apple Inc 2014.)

Swift is a modern, powerful and intuitive programming language for macOS, iOS, watchOS and tvOS. Swift includes modern features which developers love much as writing Swift code is concise and expressive. Moreover, Swift code is safe by design, and according to Apple Inc (2018) produces software that runs lightning-fast.

Dart is an object-oriented, class defined, single inheritance language using a C-style syntax. It is used to build web, server and mobile applications, and for Internet of Things (IoT) devices. Dart compiles to ARM and x86 code and so Dart mobile apps can run natively on iOS, Android, and more. For web apps, Dart transcompiles to JavaScript. It is open-source software under a permissive free software license (modified BSD license).

6.2 APIs and libraries

To provide currency exchange rates, the data from Fixer.io API was used, it's available for public usage through https://api.fixer.io/.
The cryptocurrency data would be gathered with a help of Coinmarketcap.com API, available for public use through https://api.coinmarketcap.com/.
Cryptocurrency icons used were kindly provided by github.com/cjdowner, and are free to use for personal and commercial purpose. They could be found at https://github.com/cjdowner/cryptocurrency-icons/.

6.3 Implementation

To make comparison more objective and the work easier, both Multi-OS Engine and Kotlin Native versions of the application is written using only Kotlin language.

The user interface would have unified style across all app versions. It would consist of a screen with “bottom navigation” pattern (Figure 13).

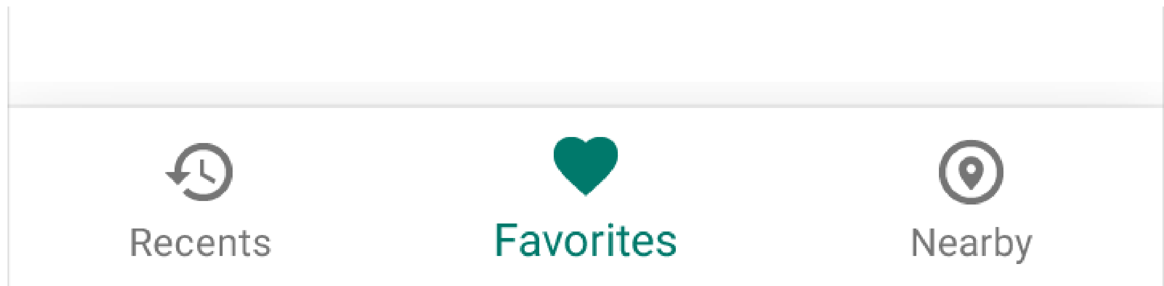


Figure 13. Bottom navigation example

There are three navigation options:

- Currency
- Cryptocurrency
- Settings

The Currency tab would have a list of world currencies exchange rate compared to selected currency, which in my case is Euro.

The Cryptocurrency tab would show a list of most popular cryptocurrencies, their exchange rates compared to selected currency. Moreover, there would be indication of exchange rate changed in percentage for 1 hour, 24 hours and 7 days.

In the settings tab there would be options to change default currency to which all others would be converted.

As application is using native UI for both Android and iOS it looks different, but native to each platform. However, it is following the same semantics and uses appropriate or close UI component to mimic the same look on both platforms. To implement bottom navigation pattern Android version uses BottomNavigationView from Android Design Support library while iOS version utilizes Tab Bars controlled by UITabBarController.

6.4 iOS UI

The final look of project Storyboard is illustrated in Figure 14.

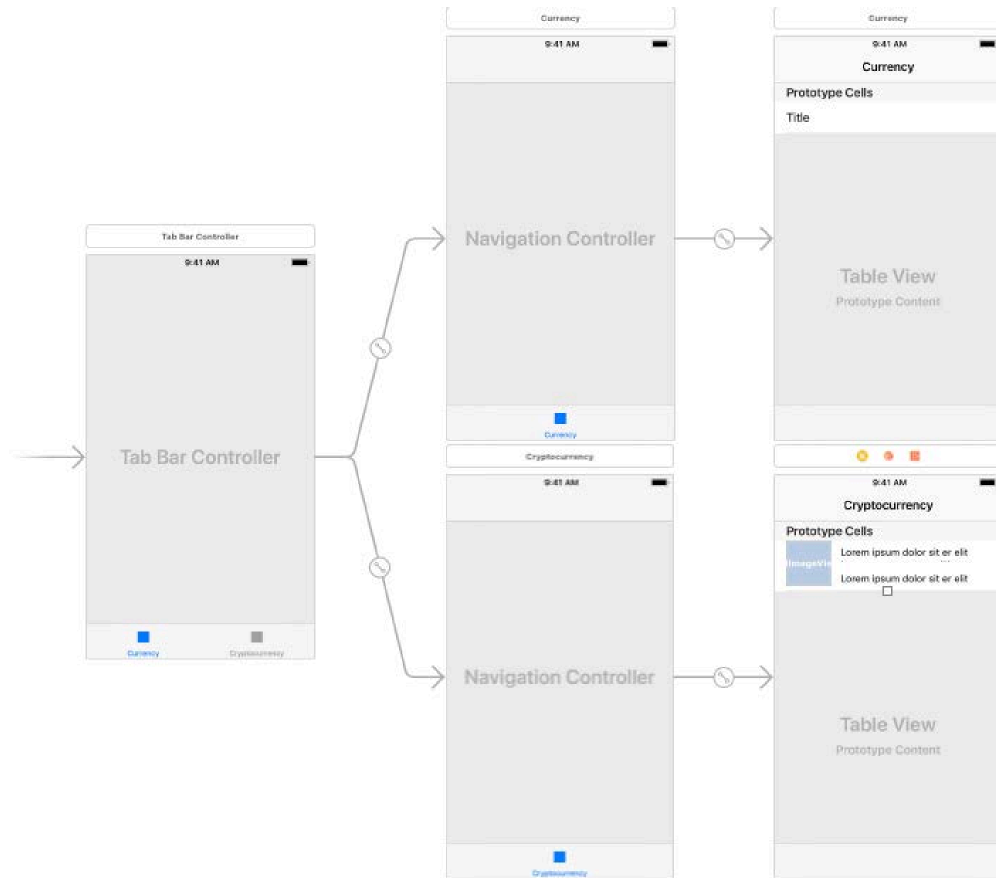


Figure 14. iOS Storyboard final look

The iOS part of the application consists of one main Tab Bar screen which have options for selecting either Currency view or Cryptocurrency one. Tab Bar is controlled by UITabBarController and as the default behavior of it suits the app, there is no need to use custom UITabBarController. The controller responsible for Currency tab is called CurrencyTableViewController which is subclass of TableViewController. It uses the default Basic cell for Table View. Another controller called CryptocurrencyTableViewController is responsible for presenting cryptocurrency data. It is also subclass of TableViewController and have custom cell called CryptocurrencyCell which contains ImageView for cryptocurrency icon and TextViews for showing name and rates.

7 MULTI-OS ENGINE APPLICATION

Multi-OS engine project has the next semantics:
The top-level project called “Currency Observer”
Module “app”, which is Android specific application.
Module “common”, which contains shared code between platforms.
Module “ios”, which contains iOS specific application code.

The shared library dependencies of Multi-OS Engine application are:

- Retrofit
- Kotlin coroutines
- Kotlin stdlib jdk7

Retrofit is a type-safe HTTP client written in Java and is used for data fetching using REST API, which is implemented to be easy and straightforward to use.

Coroutines are used for simpler asynchronous programming. The logic of the program can be expressed sequentially in a coroutine, and the underlying library will figure out the asynchrony for us. The library can wrap relevant parts of the user code into callbacks, subscribe to relevant events, schedule execution on different threads, and the code remains as simple as if it was sequentially executed.

Kotlin stdlib is the Kotlin Standard Library. It provides necessary essentials for everyday work with the Kotlin language.

To work with Multi-OS Engine from Android Studio the MOE plugin is required. For this the next steps are needed:

In Android Studio go to Preferences/Settings → choose Plugins and in a search field type Multi-OS and install it.

As it was mentioned earlier, the application consists of three modules. The next parts describe the process of creating each of them and their main purpose.

7.1 Setup Android part

The first module is called “app” and it is a regular android module. To create this module, it is necessarily to just follow the regular steps of creating Android app project in Android Studio and use Bottom Navigation Template (Figure 15).

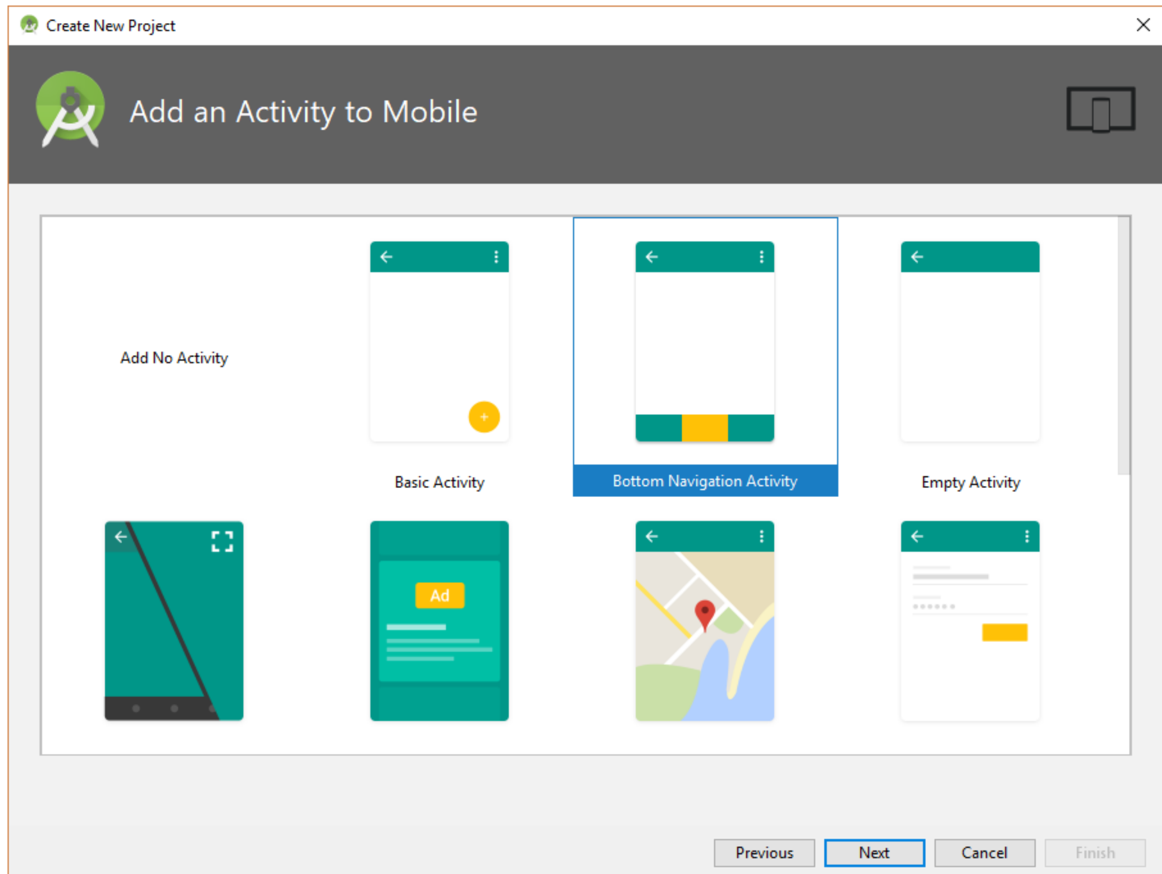


Figure 15. Choosing template in new Android project

The Android part code is basic, in simple words, it just gets data and binds to UI. But actually, it requests data asynchronously form API using common module methods and then represents it to native UI. The Android part consists of Main Activity with bottom navigation which uses CurrencyFragment for presenting currency data and CryptocurrencyFragment responsible for presenting cryptocurrency rates. Also, some helpers and utilities classes for decoding SVG images and utils implementing expected behavior by ‘common’ module.

7.2 Setup common part

The second module called “common” contains all the application’s shared logic. It’s the main part of the project where all business logic is located. In my case this is the code responsible for communicating with REST API.

To create “common” module we need to follow the next steps:

1. Right click on the main project directory and select New → Module
2. Select “java library”
3. Change module name to “common” and change package name to the package used in Android part or any other.

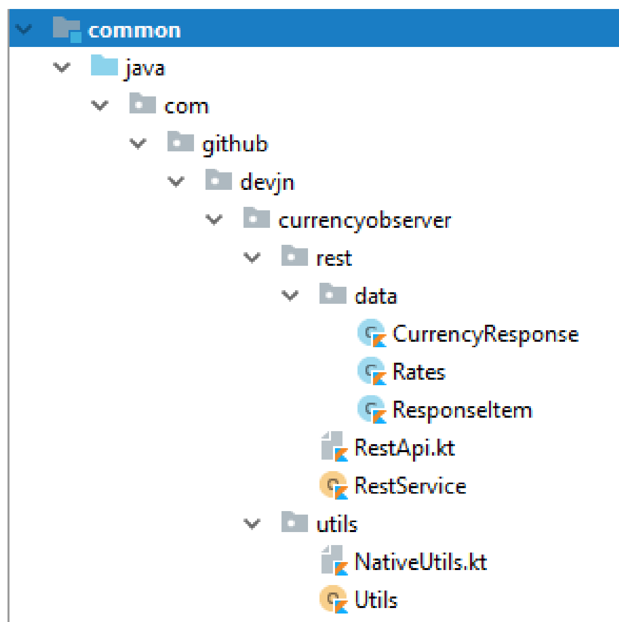


Figure 16. The ‘common’ module source code structure

Th code hierarchy is listed in Figure 16. I have top-level packages called “rest” and “utils”. The “rest” package contains all code responsible for REST services communication and presentation. The sub-package called “data” contains code which represents API data in Kotlin objects.

The responsibilities of classes are:

- Rates class represents all currencies response.
- CurrencyResponse represent currency data returned by API.
- ResponseItem represent cryptocurrency returned data returned by API.
- RestApi class contains Retrofit Get request API requests
- RestServices class contains code responsible for Retrofit and HTTP client initialization.

7.3 Setup Multi-OS Engine

The last step of project setup is to create “ios” module which would contain all iOS code written in Kotlin and entire Xcode project. To generate this module, right click on project directory and select New → “Multi-OS Engine Module” (Figure 17)

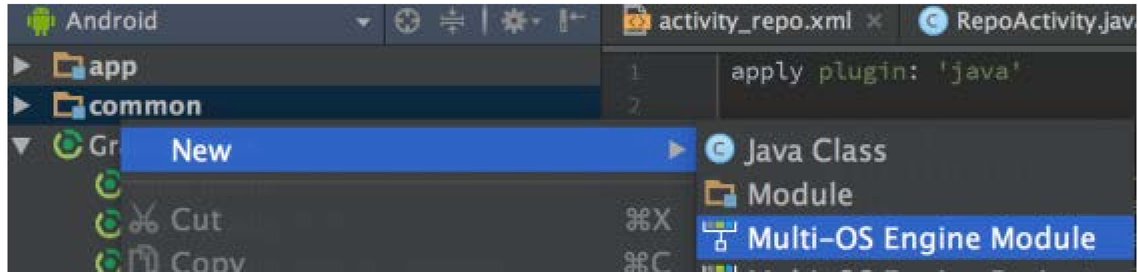


Figure 17. Creating a new Multi-OS Engine module

Then select “Kotlin Single View Application” (Figure 18) and click Next

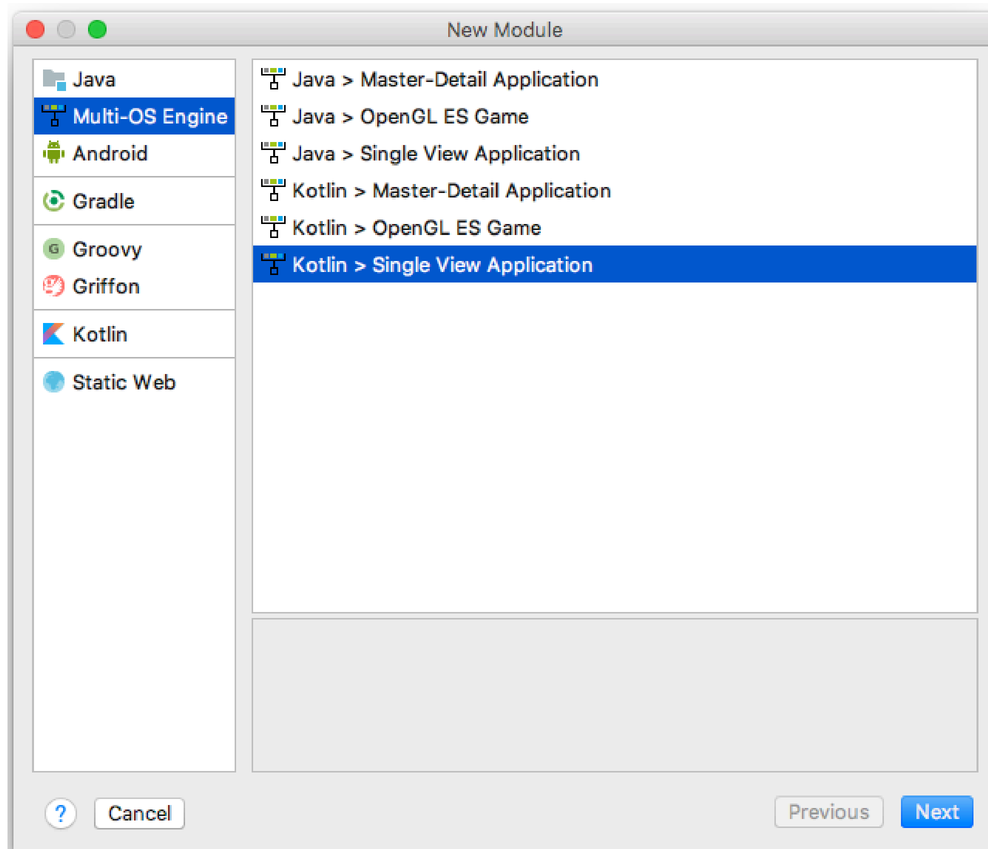


Figure 18. Process of creating new Multi-OS engine module

Then fill in all the necessary information, change module name to “ios” and click Finish. Now we have three modules, however the problem is that they are all

independents modules. To make common module to be visible for Android and iOS modules right click on common module and select Open Module Settings. Then, on the left section select “ios”, click + on the bottom of window and select “Module Dependency”, select “common” and accept (Figure 19). Next, we need to repeat the same with “app” module.

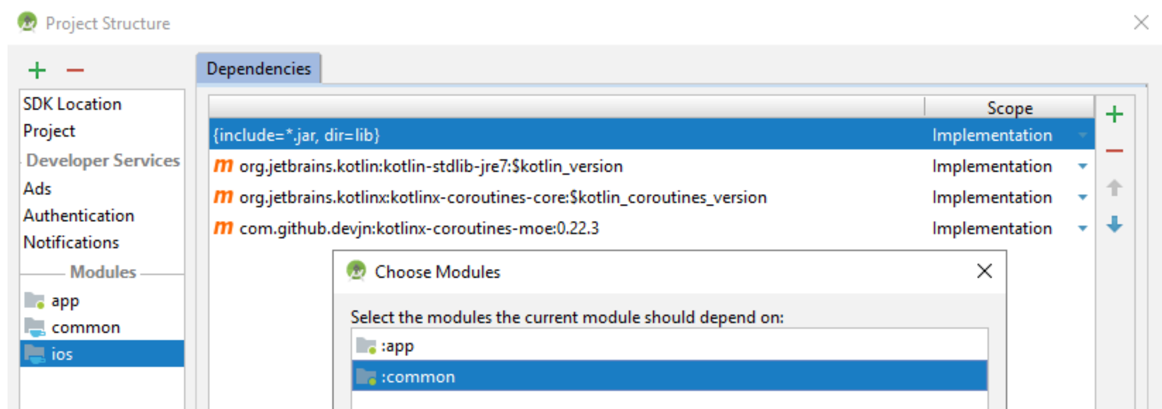


Figure 19. Adding common module as dependency for “ios” module

Now all preparation is finished, and we can start writing the code.

7.4 Configuration

The code can be written in Kotlin, however with UI there is something to keep in mind. In iOS world there are two options for creating UI, dynamically via code, or using storyboards. Storyboards were introduced in iOS 5. Storyboards help to create all the screens of an application and interconnect the screens under one file. Before the Storyboard file format was introduced, developers had to create XIB files for each view controller and programmed the navigation between each view manually. Using a Storyboard lets the developer to define both view controllers and the navigation between them on a design surface. So basically speaking, the Storyboard is a visual representation of the appearance and flow of iOS application.

Unfortunately, Multi-OS Engine stopped supporting editing/creation of Storyboards, and so it is necessarily to use Xcode. The main Storyboard is called Main.storyboard. iOS uses “ViewControllers” for controlling views, and to write our custom logic we had to first create Objective C file, and after that create a Java binding for it.

To do this we first need to enable Xcode workspace from Gradle script, for this we need to modify “moe” properties of Gradle script as shown in Figure 20 so that Xcode workspace settings is synchronized.

```
moe {
    xcode {
        project 'xcode/Currency Observer.xcodeproj'
        mainTarget 'Currency Observer'
        testTarget 'Currency Observer-Test'

        // Uncomment and change these settings if you are working with a Xcode workspace
         workspace 'xcode/Currency Observer.xcworkspace'
         mainScheme 'Currency Observer'
         testScheme 'Currency Observer-Test'
    }
}
```

Figure 20. Gradle setting of Multi-OS Engine part.

After that, in order to be able to use Xcode for modifications and adding UI we first need to “Inject/Refresh Xcode Project Setting” (Figure 21) and then we can “Open Project in Xcode.

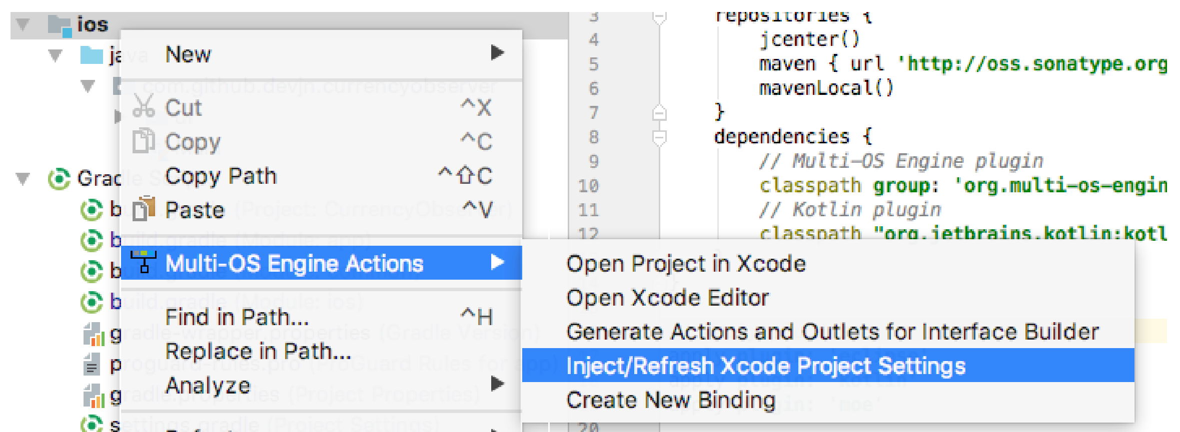


Figure 21. Multi-OS Engine Actions in Android Studio

Now we can open project in Xcode and start creating Storyboard UI.
The process of creating the Storyboard is the same as any regular iOS application development. Even more, it is still required to create Cocoa Touch classes for extending views and then use NatJGen tool as a bridge to connect view to Java code. The process of connecting Outlets and Actions of Storyboard view is illustrated in Figure 22.

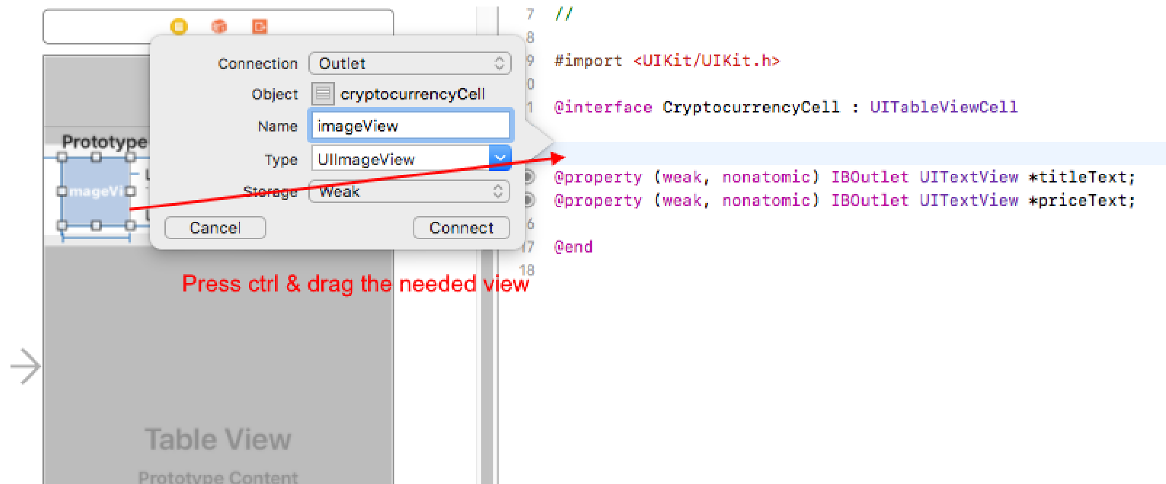


Figure 22. Process of connecting Storybvoard View to code Outlet

7.4.1 Bindings

As it was mentioned before, to use and manipulate with anything which requires access to Objective-C code we need to bind it to appropriate Java/Kotlin class. This is also true for the views of Storyboard. To bind the resulted UI to the rest of the app written in Kotlin it's necessary to use Android Studio Multi-OS Engine plugin.

Binding configurations are stored on a special file with ".nbc" extensions and Multi-OS Engine plugin provides good user interface for modifying it more intuitive (Figure 23).
I decided to store all binding files in separated folder called "bindings". The one binding file can contain several bindings, so I separated them by three major categories:

- CellBindings
- Controllers
- Libraries

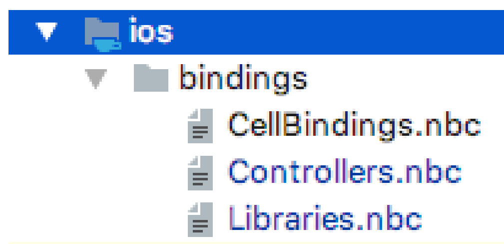


Figure 23. bindings folder content of the application

The CellBindings file contains all binding configuration responsible for handling binding for iOS custom UITableViewCell views. In my case it's "CryptocurrencyCell".

The Controllers file contains bindings for iOS custom UIVIewControllers.

The Library file contains bindings responsible for giving access to Objective-C libraries and frameworks through Java code.

7.4.2 Custom bindings

To have access for our views in Storyboard we first need to create binding to our header files using. Header file are files with ".h" extension in Xcode project (Figure 24).

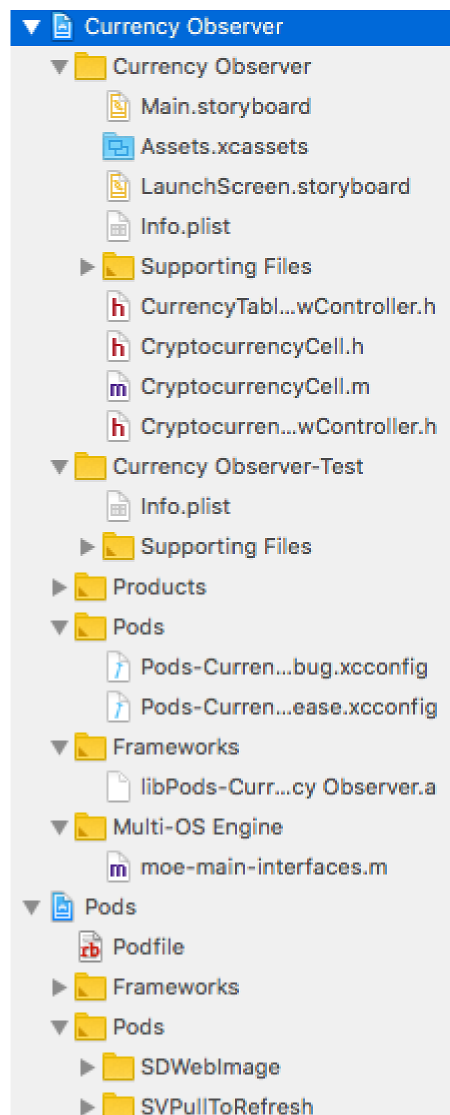


Figure 24. Xcode project hierarchy

To create bindings, right click on “ios” module, select “Multi-OS Engine Actions” → “Create New Binding” give it a name, like “Controllers”. To add bindings simply select green plus “+” button, select Header option and name (Figure 25).

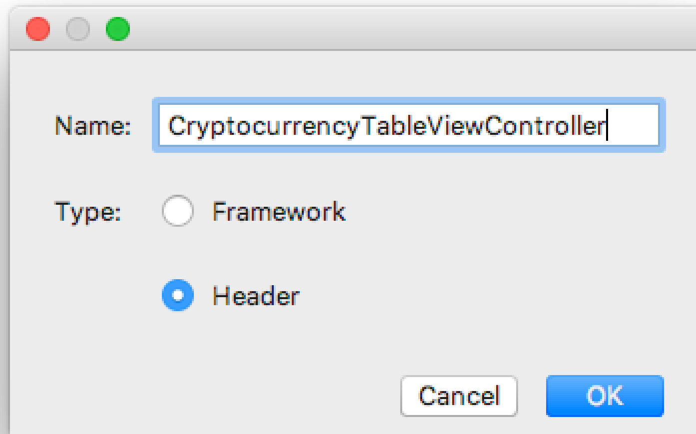


Figure 25. New binding file creation

After the file is created, it is time to configure it. Figure 26 illustrate how CrypocurrencyTableViewController binding is configured. The “Header path” is where tool will look for header (.h) files listed in “Import headers”. “Base package name” is java package of generated binding files. In my project it is named as “org.moe.bindings” to easier identify binding files. In “Import headers” section all Objective-C header files which are needed to be generated are listed. In this particular option it’s named as “#import <CryptocurrencyTableViewController.h>”.

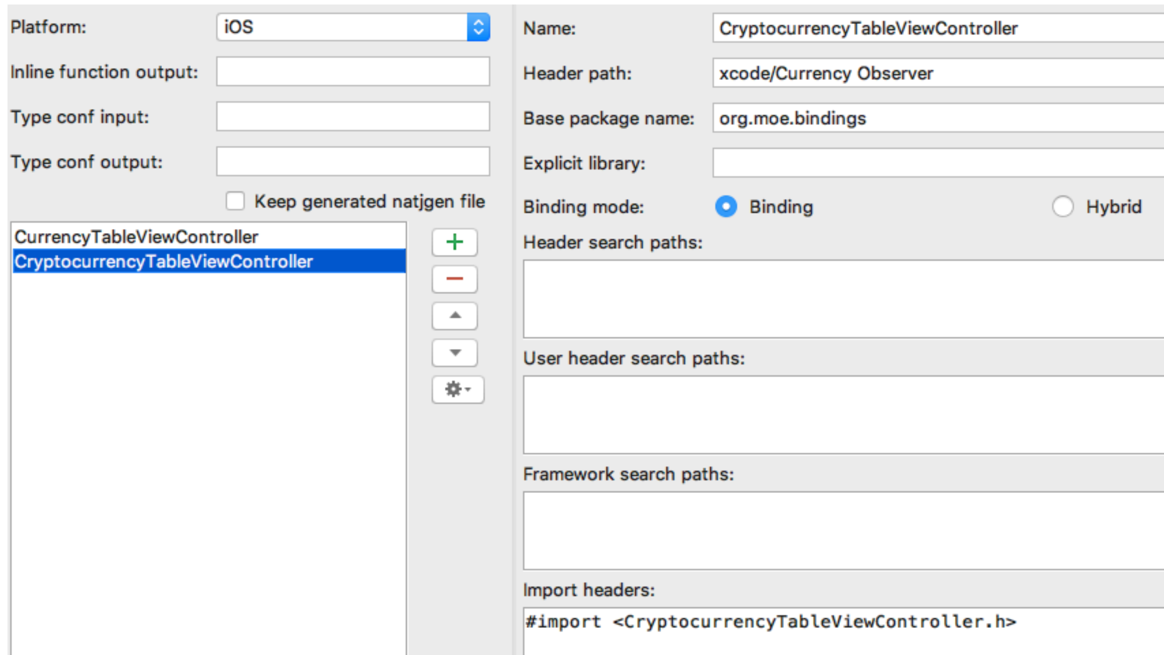


Figure 26. Example of binding file

To generate binding simply click settings icon and choose “Generate Bindings”. Now the tool will generate all the code to selected “Base package name”, in the project it is “org.moe.binding”. However, all generated code is in Java language, so in order to use it I used built in capabilities of Kotlin plugin to convert Java code to Kotlin and moved to the appropriate package.

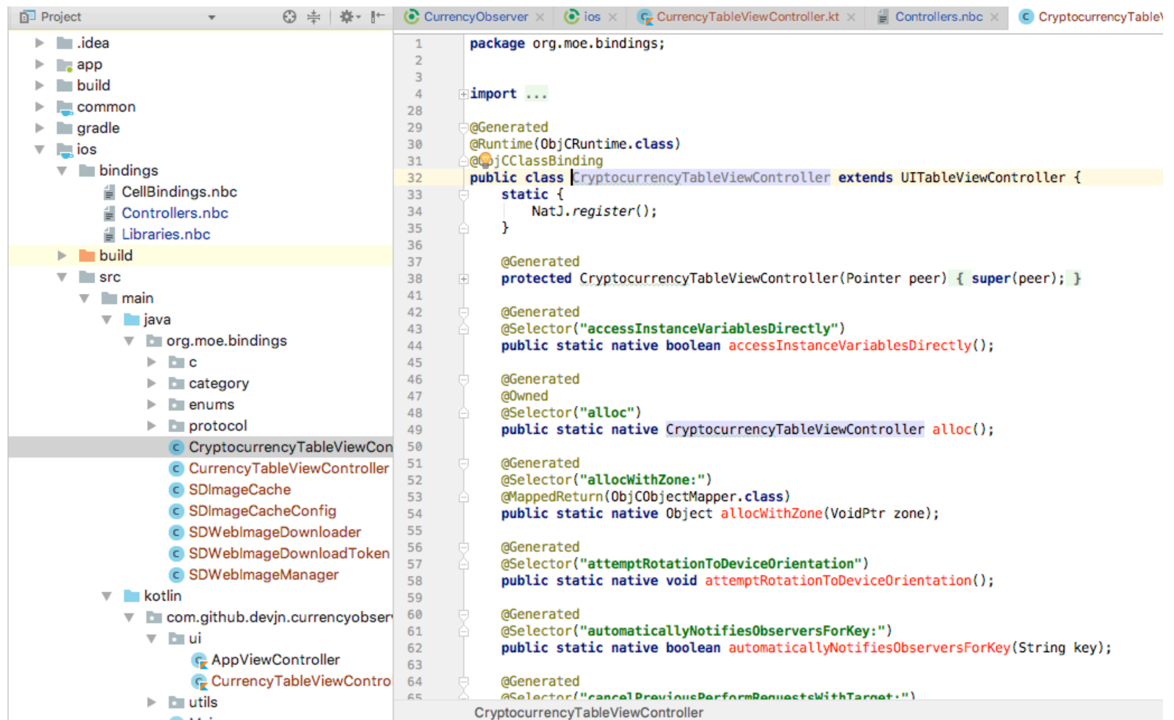


Figure 27. Example of generated binding Java code by NatJGen

As can be seen in Figure 27 the generated code looks quite “eye catching” and some parts are shown as errors, but it’s okay. The generated code includes methods that are required by the Objective-C runtime (and/or Nat/J)

7.4.3 Library bindings

Binding library and frameworks can be a bit trickier. Because some libraries presented just as header while other ones are frameworks. I am using only one library called SDWebImage used for image loading. It’s not a framework so it can be imported easily using the previous method (Figure 28).

Figure 28. Multi-OS Engine binding screen

All generated binding code from libraries is left without modifications in Java language.

8 KOTLIN NATIVE

The Kotlin Native project only consists of iOS project, because for now there is no much to share.

8.1 Setup

Kotlin/Native uses Gradle as a build tool. To make a project with Kotlin/Native compliable from Xcode, you need to add to the project a Run Script phase that invokes building of Kotlin/Native code with Gradle. Setting up a project like that from scratch can be a bit tricky, so the Kotlin/Native plugin comes with several iOS and macOS templates which simplify this process. To install Kotlin/Native in AppCode go to Preferences/Settings → Plugins and in a search field type Kotlin/Native and install it (Figure 29).

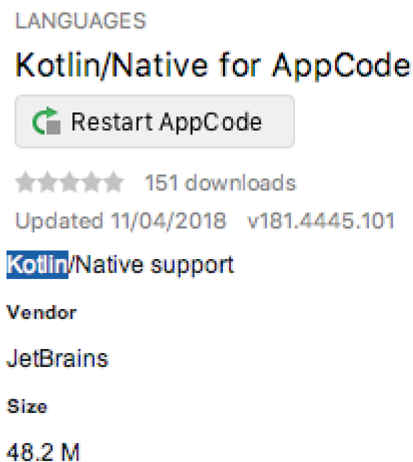


Figure 29. Kotlin/Native plugin for AppCode

The process of a creating new Kotlin Native includes three options (Figure 30):

- Application
- Framework
- App with Kotlin Native Framework

“Application” is fully a normal native Xcode iOS project but written in Kotlin. This is the option which I selected.

“Framework” is a type of Kotlin Native project which enables an iOS framework written in Kotlin, which will result in an Objective-C framework available for any iOS project and not only Kotlin Native one.

“App with Kotlin Native Framework” option creates a regular iOS Xcode project in selected language (Swift or Objective-C) and adds a Kotlin Native framework module. So basically, it’s just a normal native iOS project which includes framework based on Kotlin.

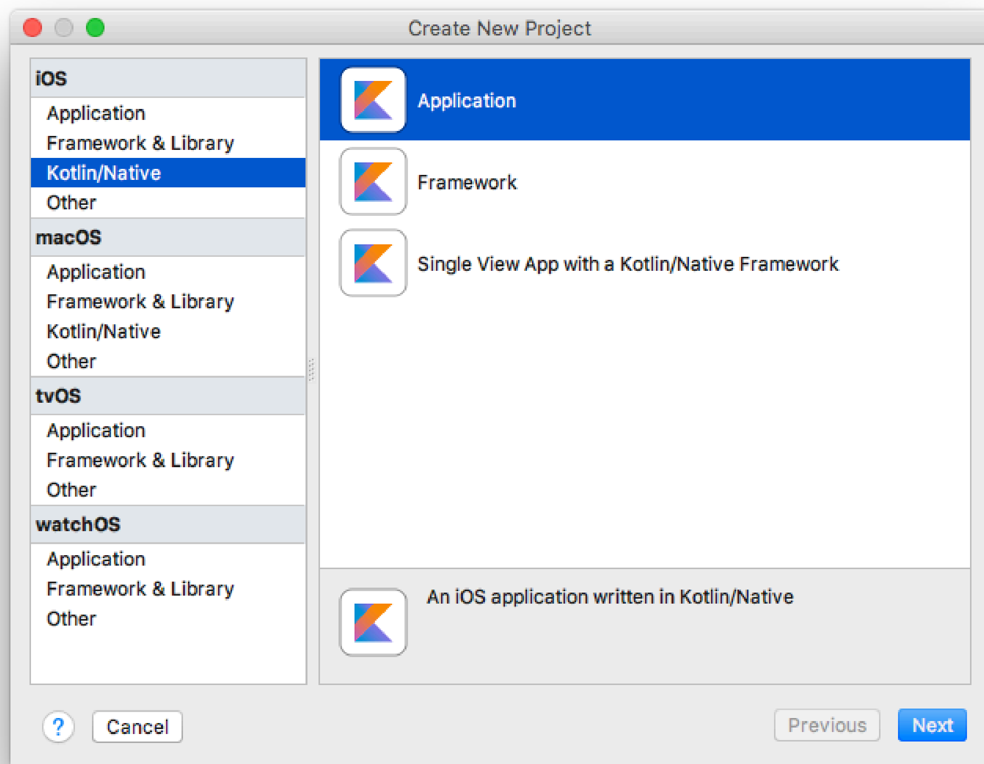


Figure 30. Creating a new Kotlin Native project in AppCode

After the project is created (Figure 31) it has almost the same semantics as an ordinary Xcode project.

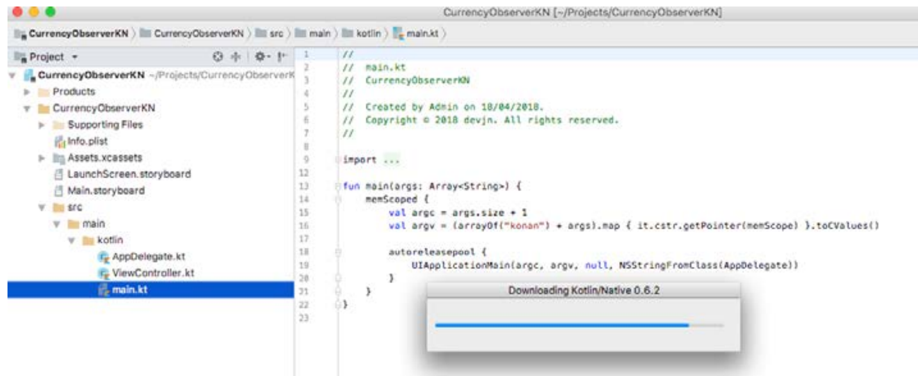


Figure 31. Kotlin Native project initial structure

8.2 Working principle

Basically, this Kotlin Native project is still a 100% Xcode Objective-C project. The only differences happen in the building phase. Kotlin native will generate the code using LLVM via Gradle and swap the Objective-C counterparts with it (Figure 32). The code should be pretty straightforward. It fully follows Apple's convention, but with a different language. And the binding "magic" between Objective-C and Kotlin happens at annotation decorators such as `@ObjCAction` and `@ObjCOutlet`.

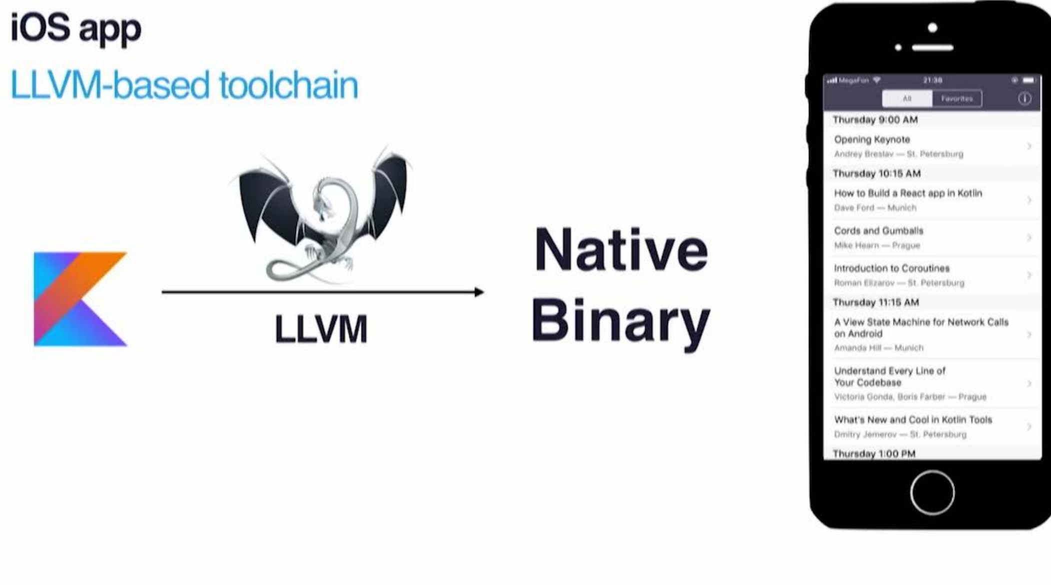


Figure 32. How Kotlin Native compiles to iOS

So, in simple words it can be said that Kotlin Native just translates Kotlin language to Objective-C.

9 FLUTTER APPLICATION

Flutter setup process is handy and a bit different from previous tools.
First, we need to download the installation bundle which contains Flutter SDK and Dart Runtime. After that Flutter need to be added to the PATH environment variable.
Flutter relies on the Android Studio IDE to supply its Android platform dependencies. So almost all the development was done using Android Studio.
To work with Flutter Android Studio requires two plugins:

- The Flutter plugin which powers Flutter development workflows (running, debugging, hot reload, etc.).
- The Dart plugin which offers Dart code analysis, assistance, etc.

To create project, I used Android Studio 3.1 (Figure 33), but it's also possible to create it using command line tools.

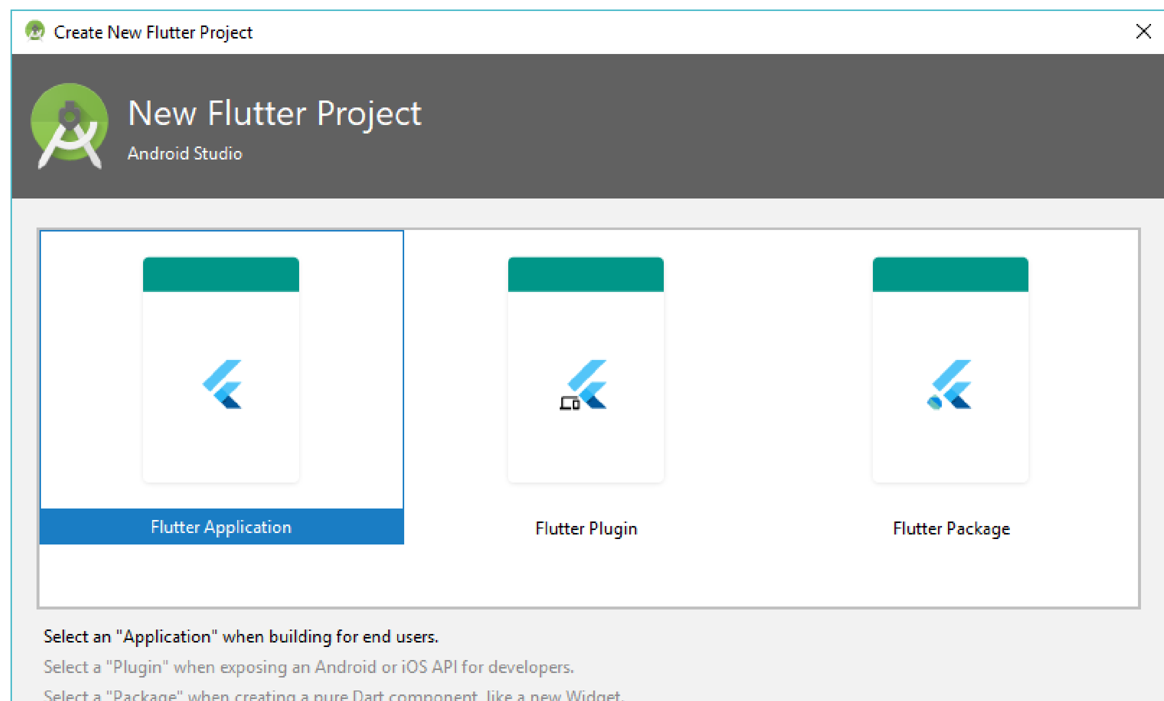


Figure 33.Creating a new Flutter application

The generated project consists of three main folders:

- android
- ios
- lib

android and ios folders contains platform specific code, and project configurations like any native platform application. The generated data consist of the settings and project files and the code which uses native code to start Flutter part.

The most interesting is lib folder (Figure 34), this is where all Flutter application code written in Dart language is located.

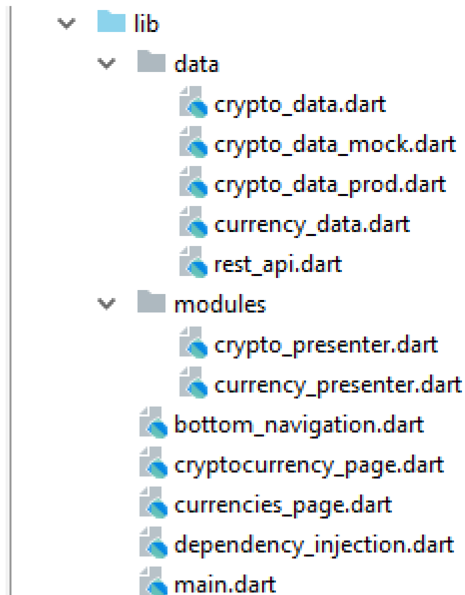


Figure 34. Final Flutter project structure.

The application is written using MVP architecture in mind. The data folder contains models and related code, modules folder contains presenters.
The main entry point of flutter application is class “main.dart” and method *void main().* The application content of this main class is illustrated in Code 2 and Code 3.

```
void main() async {
  Injector.configure(Flavor.PROD);
  runApp(new MyApp());
}
```

Code 2. Main.dart file main method

The main method is responsible for configuring Injector, this is custom dependency injection class, so that easier to test app. Using Flavor.MOCK allows easier testing as it generates the data instead of getting from internet.
In the main method the app is start running by supplying the *runApp* method the root widget, in my case it is MyApp (Code 3).
MyApp is a StatelessWidget which builds material app and uses appropriate theme on iOS and Android.

```
class MyApp extends StatelessWidget {
  final ThemeData iOSTheme = new ThemeData(
    primarySwatch: Colors.pink,
    primaryColor: Colors.blue[300],
    primaryColorBrightness: Brightness.light,
  );

  final ThemeData defaultTheme = new ThemeData(
    primarySwatch: Colors.purple,
    accentColor: Colors.pink,
  );

  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      theme: defaultTargetPlatform == TargetPlatform.iOS
           ? iOSTheme : defaultTheme,
      home: new MainNavigation(),
    );
  }
}
```

Code 3. MyApp widget of main.dart class

Flutter benefit is that application with same code can work on both android and iOS and have the same feel and look, however as default application uses Material theme widgets which looks good and native on Android but cheap on iOS platform.

Flutter has already pre-built widgets called Material which follow Material Theme guidelines, which is now de facto on Android platform. However, on iOS this doesn't look good and doesn't look native at all. For this, Flutter team prepared iOS widgets called Cupertino, which are unfortunately not as complete as Material ones. However, it's not possible to just use these different themes widgets using only one codebase. So, I decide to make platform aware widgets. To do this I created a base PlatfromWidgetCreator (Code 4) which checks for a platform and builds respective widget. So, in order to use I had to create respective platform-aware widgets creator for all necessarily widgets. This is done by extending PlatfromWidgetCreator and providing with required widgets for output.

```
abstract class PlatformWidgetCreator<I extends R, A extends R,
    R extends Widget> {
  R create() {
    if (Platform.isAndroid) {
      return createAndroidWidget();
    } else if (Platform.isIOS) {
      return createIosWidget();
    }
    // platform not supported returns an empty widget
    return null;
  }

  I createIosWidget();

  A createAndroidWidget();
}
```

Code 4. Base platform aware widget creator.

This was easy with simple widgets, as they mostly have same behavior and actions. For example, platform-aware AppBar (Code 5), this is the top most view in application. It extends PlatformWidgetCreator and creates CupertinoNavigationBar for iOS version and AppBar for Material one. They both have leading, and title can be set on both.

However, some widgets are handier and not so easy to implement common class. Hopefully in future Flutter team will present some combined solution which will be native for both platform out of the box.

```
class PlatformAppBar extends PlatformWidgetCreator<CupertinoNavigationBar,
    AppBar, PreferredSizeWidget> {
  final Widget leading;
  final Widget title;

  PlatformAppBar({
    this.leading,
    this.title,
  });

  @override
  CupertinoNavigationBar createIosWidget() => new CupertinoNavigationBar(
        leading: leading,
        middle: title,
      );

  @override
  AppBar createAndroidWidget() => new AppBar(
        leading: leading,
        title: title,
        elevation: 5.0,
      );
}
```

Code 5. Platform-aware AppbBar source code.

The resulted application is illustrated in Table 1.

As can be seen, applications look native for platform, but still have same functionality and alike look and furthermore, this is all done using almost one codebase.

10 COMPARISON

Multi-OS Engine and Kotlin/Native are quite close to how they solve the cross-platform problem. On the other hand, there is Flutter which almost allows to abstract from platform and build independent UI, but it is also possible to build native looking user interface and still have single codebase unlike these two tools above.

10.1 Multi-OS Engine & Kotlin/Native vs other cross-platform dev tools

The main difference of these tools with others, like React Native or Cordova, is that these tools are not designed to have their own user interface. Instead the main focus is on code sharing. This approach has several
benefits and drawbacks. Let's start with advantages:

- Faster speed
- Native UI
- Easy access to platform specific functions and libraries

As an application is compiled to native code it has faster speed. Using native UI also makes the application more responsive, and native UI is more familiar for users. Some people oppose cross-platform tools mostly because they have a nonstandard UI, which is sometimes also slow.

Sharing code allows easier code reuse and not writing the same code again, which not only increases speed of the development but also makes apps less error prone.

10.2 Multi-OS Engine vs Kotlin/Native

Kotlin/Native is still in early access preview state, so a lot of things may be improved closer to the release of stable version. However, some points still can be made.

As it can be seen from the completed project, both tools allow writing cross-platform code in the Kotlin language. Both require working with Xcode and

building native UI using platform default tools. But what is the difference? In short, I can say that Multi-OS Engine has benefits of Java world, while Kotlin/Native is Kotlin only. In practice, for Kotlin Native it means that it is not possible to use Java libraries, like retrofit and in conjunction of how young Kotlin language is, it is a big disadvantage. Not being able to use Java libraries, of which in the world there is a great variety, does not allow writing a really cross-platform, complex apps. While using Multi-OS Engine it is possible to still use the same application architecture, reuse most of the code of Android part and benefit from superb Java libraries with a big history.

But there are more differences under the hood. Multi-OS Engine as its name explains, is an attempt to put another OS, specifically Android OS, inside iOS platform. So basically, it’s an Android runtime running on top of iOS platform. On the other hand, Kotlin Native approach is to give the ability for Kotlin compiler to compile to output standalone native executables that can be run without using a virtual machine (VM). So, even it seems that Multi-OS Engine and Kotlin Native are quite similar, they are totally different ideas. Multi-OS Engine embeds Android system into OS, while Kotlin Native position itself as a new language for native platforms, like C++.

10.2.1 Visual code difference

Even so that Multi-OS Engine and Kotlin Native both allow writing in Kotlin language there is still difference in the format.

```
import kotlinx.cinterop.*
import platform.UIKit.*

class AppDelegate : UIResponder(), UIApplicationDelegateProtocol {
    companion object : UIResponderMeta(), UIApplicationDelegateProtocolMeta {}

    override fun init() = initBy(AppDelegate())

    private var _window: UIWindow? = null
    override fun window() = _window
    override fun setWindow(window: UIWindow?) { _window = window }
}
```

Figure 35. Main entry point in Kotlin Native application

```kotlin
@RegisterOnStartup
class Main protected constructor(peer: Pointer) : NSObject(peer), UIApplicationDelegate {

    private var window: UIWindow? = null

    override fun applicationDidFinishLaunchingWithOptions(application: UIApplication?, launchOptions: NSDictionary<*, *>?): Boolean {
        NativeUtils.registerResolver(IOSUtils)
        return true
    }

    override fun setWindow(value: UIWindow?) {
        window = value
    }

    override fun window(): UIWindow? = window

    companion object {

        @JvmStatic
        fun main(args: Array<String>) {
            UIKit.UIApplicationMain( argc: 0,  argv: null,  principalClassName: null, Main::class.java.name)
        }

        @Selector( value: "alloc")
        @JvmStatic
        external fun alloc(): Main
    }
}
```

Figure 36. Main entry point in Multi-OS Engine application

Figures 35 and 36 illustrate how the code of the main enter point of applications differ. Even when using the same language, there are some differences which catch the eye.

The next Figures 38 and 39 show the same class code generated by using different tools for “CryptocurrencyCell”. If we compare the converted Kotlin class which is generated by the NatJGen tool of Multi-OS Engine (Figure 39) to the Objective C header file (Figure 37), it seems to have too much code, 420 lines of code to be precise. If we compare it with Kotlin Native version, which is illustrated in Figure 38, we can see how simple it is. Multi-OS Engine team claims, that this generated code includes methods that are required by the Objective-C runtime. However, I was able to shrink it by removing some of the code which I think is not required. The updated version of the class can be seen in Figure 40. It is now cleaner but still is not as reader-friendly as Kotlin Native version (Figure 38).

```objc
11  @interface CryptocurrencyCell : UITableViewCell
12
    @property (weak, nonatomic) IBOutlet UIImageView *iconImage;
    @property (weak, nonatomic) IBOutlet UITextView *titleText;
    @property (weak, nonatomic) IBOutlet UITextView *priceText;
16
17  @end
```

Figure 37. CryptocurrencyCell header file

```
1   import ...
8
9
10  @ExportObjCClass
11  class CryptocurrencyCell(aDecoder: NSCoder) : UITableViewCell(aDecoder) {
12
13      override fun initWithCoder(aDecoder: NSCoder) = initBy(CryptocurrencyCell(aDecoder))
14
15      @ObjCOutlet
16      lateinit var iconImage: UIImageView
17
18      @ObjCOutlet
19      lateinit var titleText: UITextView
20
21      @ObjCOutlet
22      lateinit var priceText: UITextView
23
24  }
```

Figure 38. “CryptocurrencyCell” class, Kotlin Native version

```
@Runtime(ObjCRuntime::class)
@ObjCClassName( value: "CryptocurrencyCell")
@RegisterOnStartup
class CryptocurrencyCell protected constructor(peer: Pointer) : UITableViewCell(peer) {

    @Generated
    @ProtocolClassMethod( value: "appearance")
    @MappedReturn(ObjCObjectMapper::class)
    override fun _appearance(): Any {
        return appearance()
    }

    @Generated
    @ProtocolClassMethod( value: "appearanceForTraitCollection")
    @MappedReturn(ObjCObjectMapper::class)
    override fun _appearanceForTraitCollection(trait: UITraitCollection): Any {
        return appearanceForTraitCollection(trait)
    }

    @Generated
    @Deprecated( message: "")
    @ProtocolClassMethod( value: "appearanceForTraitCollectionWhenContainedIn")
    @MappedReturn(ObjCObjectMapper::class)
    override fun _appearanceForTraitCollectionWhenContainedIn(
            trait: UITraitCollection,
            @Mapped(ObjCObjectMapper::class) ContainerClass: Any,
            vararg varargs: Any): Any {
        return appearanceForTraitCollectionWhenContainedIn(trait,
                ContainerClass, *varargs)
    }

    @Generated
    @ProtocolClassMethod( value: "appearanceForTraitCollectionWhenContainedInInstancesOfClasses")
    @MappedReturn(ObjCObjectMapper::class)
    override fun _appearanceForTraitCollectionWhenContainedInInstancesOfClasses(
            trait: UITraitCollection, containerTypes: NSArray<*>): Any {
        return appearanceForTraitCollectionWhenContainedInInstancesOfClasses(
                trait, containerTypes)
    }
```

Figure 39. CryptocurrencyCell Multi-OS Engine screen

```
@RegisterOnStartup
class CryptocurrencyCell protected constructor(peer: Pointer) : UITableViewCell(peer) {

    @Selector( value: "init")
    external override fun init(): CryptocurrencyCell

    @Deprecated( message: "")
    @Selector( value: "initWithFrame:reuseIdentifier:")
    external override fun initWithFrameReuseIdentifier(@ByValue frame: CGRect, reuseIdentifier: Stri

    @Selector( value: "initWithStyle:reuseIdentifier:")
    external override fun initWithStyleReuseIdentifier(@NInt style: Long, reuseIdentifier: String): (


    @Selector( value: "titleText")
    external fun titleText(): UITextView?

    @Selector( value: "priceText")
    external fun priceText(): UITextView?

    @Selector( value: "iconImage")
    external fun iconImage(): UIImageView?


    @Selector( value: "setIconImage:")
    external fun setIconImage_unsafe(value: UIImageView?)

    fun setIconImage(value: UIImageView?) {...}

    @Selector( value: "setPriceText:")
    external fun setPriceText_unsafe(value: UITextView?)

    fun setPriceText(value: UITextView?) {...}

    @Selector( value: "setTitleText:")
    external fun setTitleText_unsafe(value: UITextView?)
CryptocurrencyCell
```

Figure 40. The optimized “CryptocurrencyCell” class, Multi-OS Engine version

10.2.2 Compilation difference

The way Multi-OS Engine runs on iOS is clever, and the build process parts are in some way similar to how it works on Android. First MOE converts Java bytecode to dex. Then proguard runs over the resulted dex, which shrinks it and runs dex2oat to convert it to ART (Android Runtime). On Android from the version 5.0 and onwards this is normally done at application install time. After the Java part is done, MOE compiles an Xcode project whose main starting point method simply runs ART on the generated ART app. So, MOE actually runs ART on iOS. This is clever because the runtime will be as solid as Intel's port of ART to iOS is. Any non-platform specific code that runs on ART on Android should run on iOS too, except of course bugs which can present in Intel's ART iOS port.

So main difference comes in that Multi-OS Engine ports Android ART runtime to iOS and tries to mimic it as much as possible so that majority part of Android

application can be ported to Multi-OS Engine iOS application. While Kotlin Native just transcompiles Kotlin language to native one.

The closes to Multi-OS Engine is Xamarin (not Xamarin.Forms). Xamarin uses C# language and on iOS it is compiled ahead-of-time (AoT) to ARM assembly language. The .NET framework, which is included in resulted project, same as Multi-OS Engine stripes out unused classes during linking to reduce the application size.

10.3 Platform interaction differences

The different tools perform and interact differently on a platform.
Using native tools, application communicates with the platform to create widgets, or access services like the camera (Figure 41). The widgets are rendered to a screen canvas, and events are passed back to the widgets. This is a simple architecture, for each platform there should be a separate app because the widgets are different.

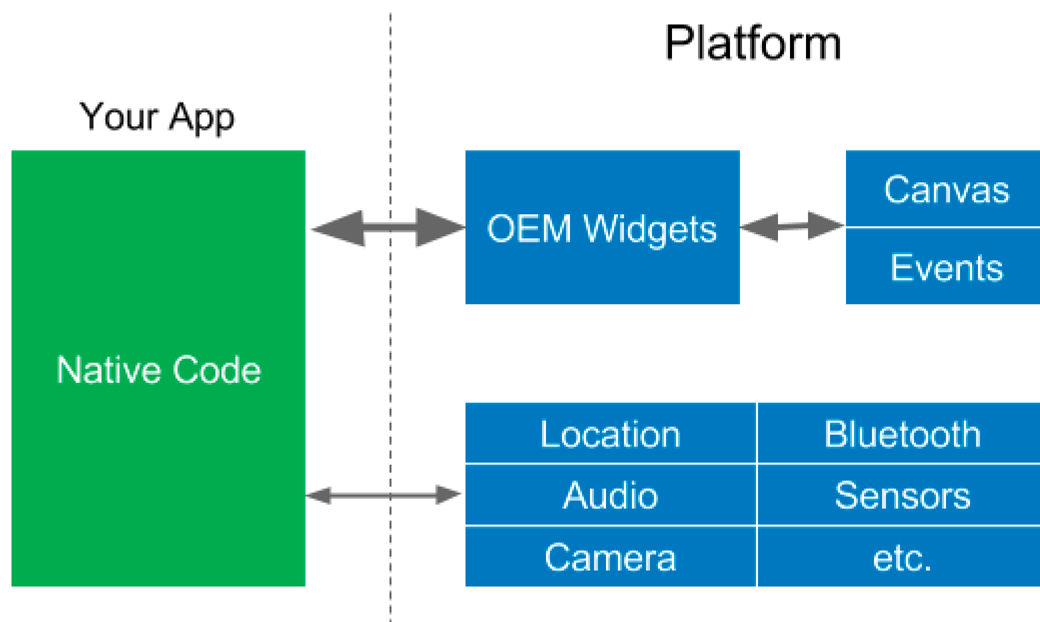


Figure 41. How native Android/iOS code interacts with the platform (Wm L. 2018)

Multi-OS Engine and Kotlin Native communicates with the platform in a similar manner, the difference is only in additional process of transferring Java/Kotlin

source code to native one (Figure 42). Everything else is the same, that why it' still required to have different code for handling UI.

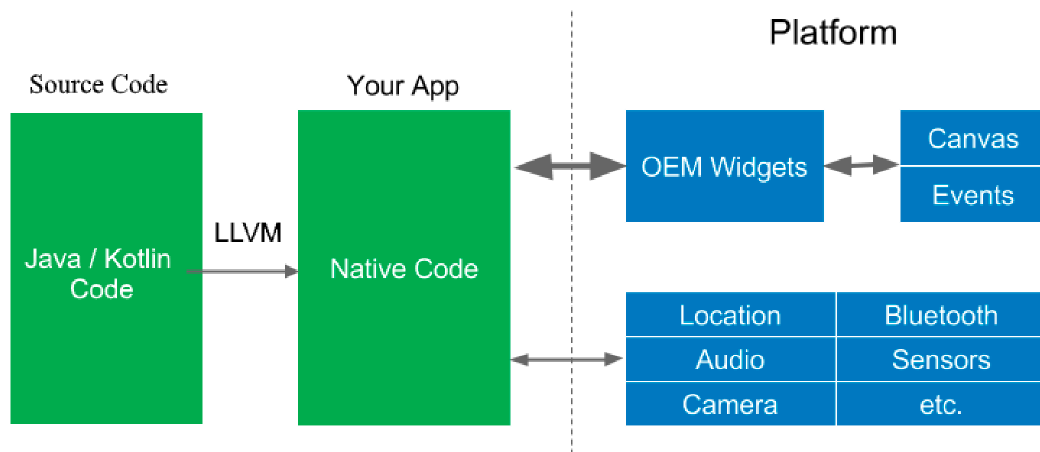


Figure 42. How Multi-OS Engine & Kotlin/Native interacts with the platform

On the other hand, Flutter communicates with a platform a bit different.

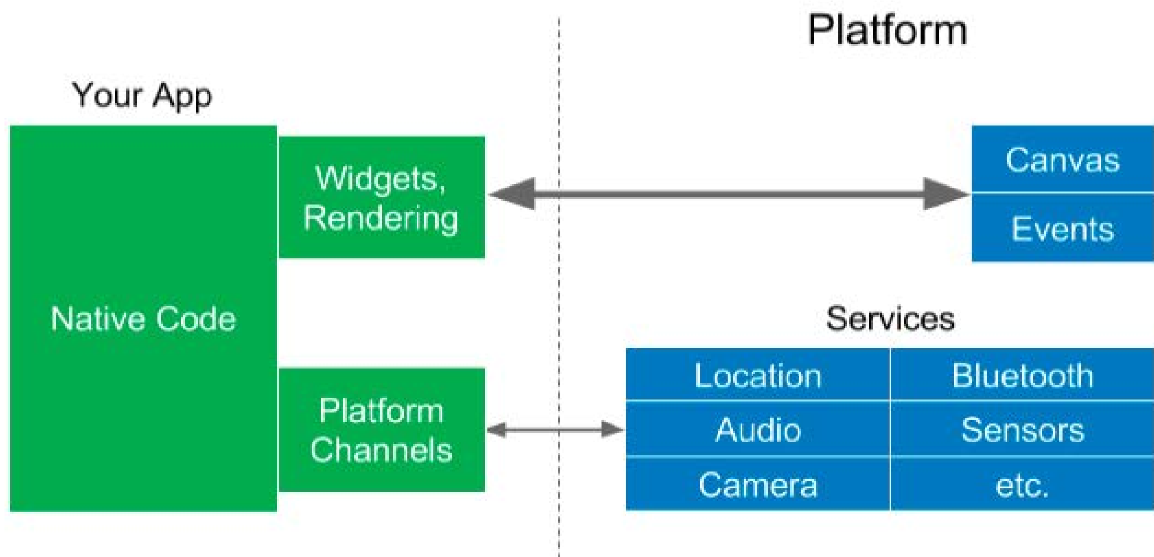


Figure 43. How Flutter interacts with the platform (Wm L. 2018)

Flutter has a new architecture that includes widgets that look and feel good, are customizable, fast and extensible. As can be seen in Figure 43, Flutter does not use the platform widgets, but provides its own widgets and rendering mechanics. Flutter moves the widgets and the renderer from the platform into the application, which allows them to be customizable and extensible. Flutter needs only platform

canvas in which to render the widgets, so they can appear on the device screen, and access to platform events and services.
The closest in speed and flexibility tool from the web world is a well-known and popular JavaScript framework React Native.

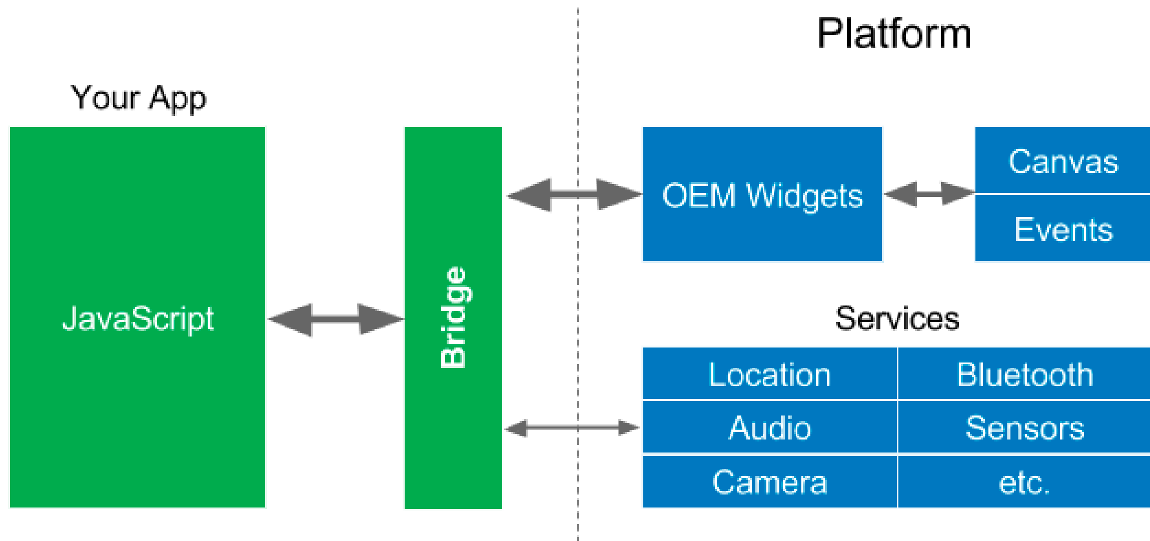


Figure 44. How React Native interacts with the platform (Wm L. 2018)

React Native uses JavaScript and instead of using slow WebView it uses a so-called bridge to accesses the native platform widgets (Figure 44). Widgets are typically accessed quite frequently, as for consistent 60 fps behavior it can be accessed for up to 60 times per second. And this is a big problem and bottleneck as it can cause performance problems.

On the contrary, Flutter still have interface between the Dart program (in green, Figure 43) and the native platform code (in blue, for either iOS or Android) that does data encoding and decoding. Even so there is an interface, it's mostly used for sending orders and so is much faster than a JavaScript bridge.

Moving the widgets and the renderer into the app does affect the size of the app. The minimum size of a Flutter app on Android is approximately 6.7MB, which is quite similar to minimal apps built with comparable tools, like for example React Native.

10.4 Future Plans

Multi-OS Engine is already presented on the development scene for good amount of time, but for now it only supports writing apps for iOS and does not have any options for shared UI. However, this might change soon. From what I know, Multi-OS Engine team is planning to also add support for targeting desktop platforms and preparing UI Java based tool which utilizes React Native under the hood. This will enable creation of fast cross-platform UI which will result in even bigger code sharing and decreased development time.

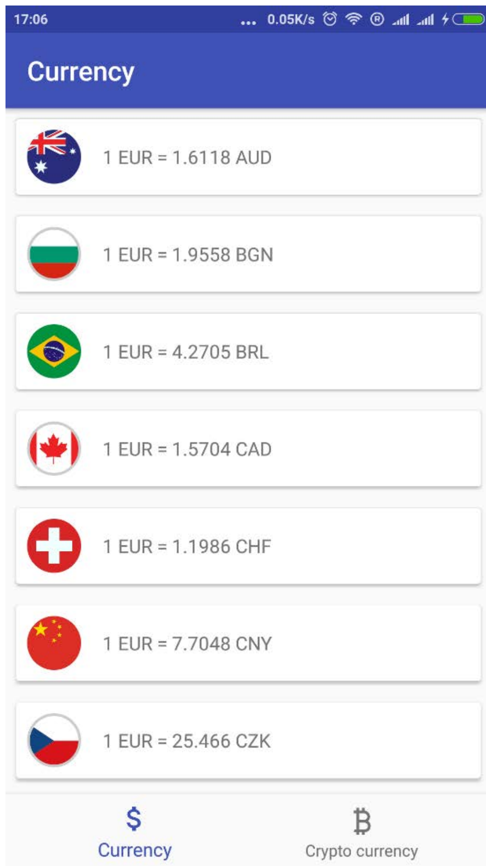
As for Kotlin Native, this tool still needs much time for development, but it’s not targeting just iOS. Kotlin Native team tries to make it so that you would be able to write in Kotlin for almost any possible platform. And it looks that they want to replace C++ in the world of native development. There is nothing bad in it, I would say that I even would like to be true. However, it is still young so not so many tools and libs are written for it. Moreover, without cross-platform UI building tool, it may not be so much popular, and I haven’t heard anything about the work towards this.

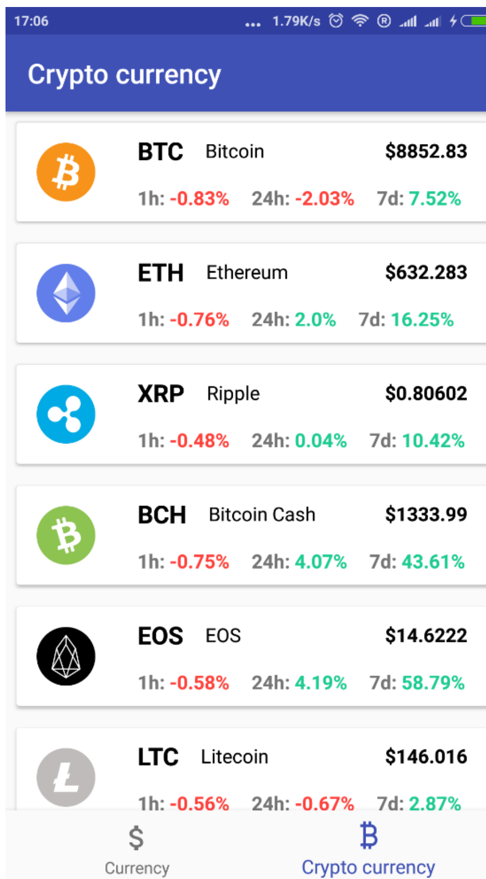
Flutter is great, but as not everything is available it would be great to try using it in conjunction with Multi-OS Engine or Kotlin Native, which is theoretically possible using Flutters services library.

10.5 Result

The resulted applications are illustrated in Table 1 and Table 2. Table 1 shows apps build with Multi-OS Engine, the Kotlin Native version is almost same, as it uses the same layout. On the left side illustrated Android version and on the left iOS one. Table 2 illustrates Flutter version with same semantics.
As it can be observed, the Flutter version is quite close to its native analogue version (Table 1), as the functionality and look is almost identical.

Table 1. The resulted app using Multi-OS Engine (Kotlin/Native one is similar)





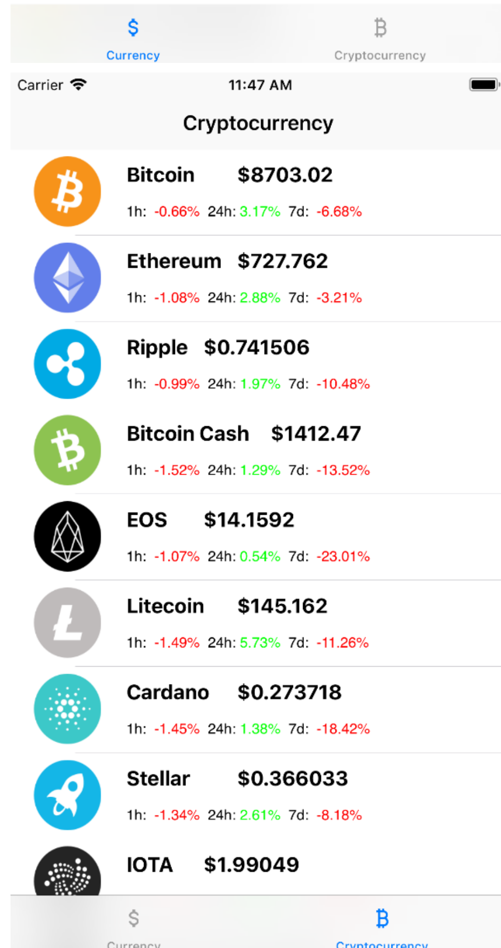
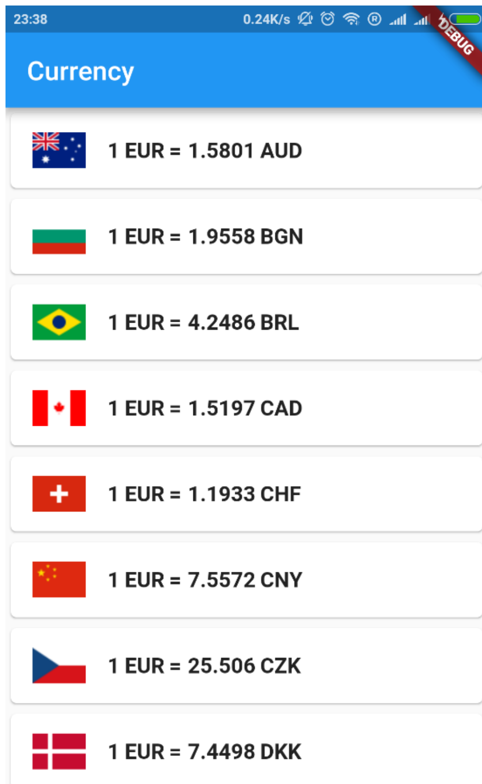
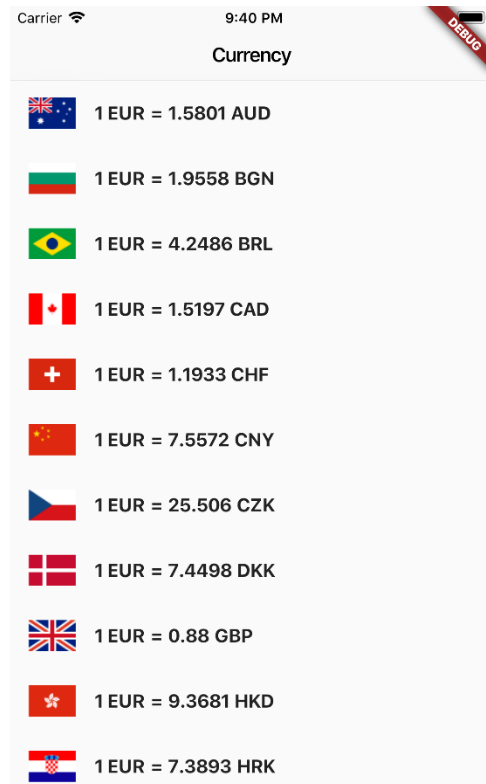
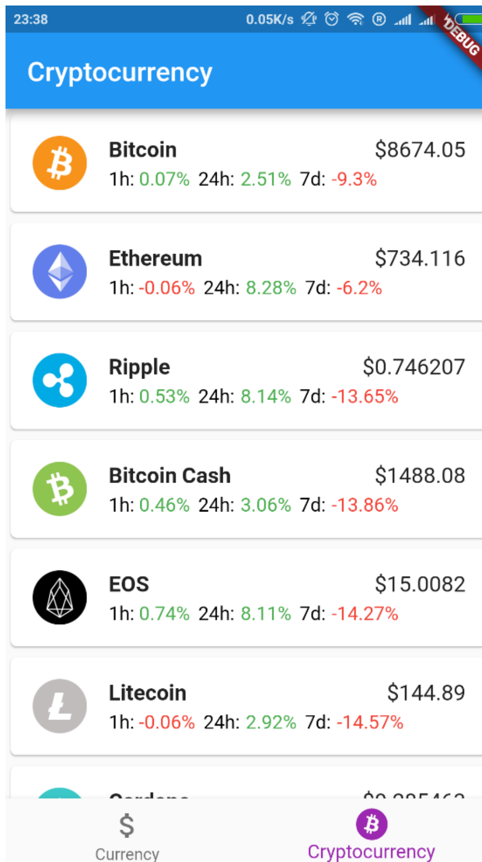
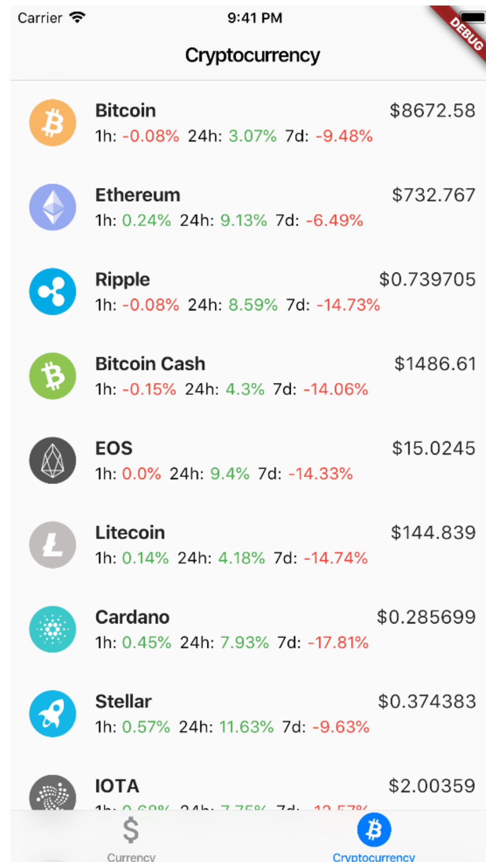


Table 2. Resulted application using Flutter (Android left, iOS right)









11 CONCLUSION

After using these cross-platform solutions I can say that they have some advantages and disadvantages not only compared to themselves but also compared to other cross-platform solutions and native development.

Multi-OS Engine is quite mature technology and gives the possibility to write real apps now for Android and iOS with a big percentage of code sharing. This particular tool can be very useful mostly for Android developers, as it is closer to them, and allows to use most of the tools and libraries they are used to.

Kotlin Native technology is still very young, but its use case is clear. As it was already mentioned, Kotlin Native team sees their product as a shared library between multiple platforms. So, its primary objective now is not to replace the currently existing native development tools, but just to give an ability to share some repetitive code. This is some business logic, models, validators, etc.

The main point is to understand that Multi-OS Engine and Kotlin Native are built not to replace the existing native development tools, but to extend and give possibility of better code sharing and reuse. To develop apps using these tools it is still mandatory to use and now the native development of both platforms. These tools just give opportunity for faster prototyping, consisting app behavior. So, for example if business logic changes it is just changed in one place and reflects on both apps. The testing becomes easier and faster as both platform should have quite same behavior. Therefor logical bugs won't be unique per platform and finding and fixing it on one platform will be reflected to another.

On the other hand, Flutter is a great tool which offers high performance and still allows to use single codebase not only for business logic but also UI. However, it's still young, though it's already in beta state and is recommended for production development, but there are not that much libraries and tools built like for Java. Flutter ecosystem is young, so you have to write a lot of things by your own, only basic and simple widgets are available. This might result in a lot of

‘reinventing the wheel’ work, which could be avoided using mature or native technologies and tools.

In the end, as these tools offer almost identical to native performance, I would like to say that these tools are some sort of a golden bullet in the battle of native development vs cross-platform one.

The full source code can be found on my GitHub, each application has its own repository. Multi-OS Engine version called CurrencyObserver, Kotlin/Native one named CurrencyObserverKN, and the Flutter version simply nominated as CurrencyObserver-Flutter.

REFERENCES

Animesh J. 2018. Why native app developers should take a serious look at Flutter. WWW document. Available at: https://hackernoon.com/why-native-app-developers-should-take-a-serious-look-at-flutter-e97361a1c073 [Accessed 1 May 2018].)

App Annie, 2017. Android to top iOS in app store revenue this year. WWW document. Available at: https://techcrunch.com/2017/03/29/app-annie-android-to-top-ios-in-app-store-revenue-this-year/ [Accessed 20 April 2018].

Apple Inc, 2014. Programming with Objective-C. WWW document. Available at: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html [Accessed 20 April 2018].

Apple Inc, 2018. Swift. WWW document. Available at: https://developer.apple.com/swift/ [Accessed 20 April 2018].

Apple Inc, 2018. Xcode IDE. WWW document. Available at: https://developer.apple.com/xcode/ [Accessed 3 March 2018].

Belov R. 2018. Kotlin/Native Plugin for AppCode. WWW document. Available at: https://blog.jetbrains.com/kotlin/2018/04/kotlinnative-plugin-for-appcode/ [Accessed 12 April 2018].

Breslav A. 2017. Kotlin/Native Tech Preview. WWW document. Available at: https://blog.jetbrains.com/kotlin/2017/04/kotlinnative-tech-preview-kotlin-without-a-vm/ [Accessed 18 February 2018].

Flutter. 2018. Technical Overview. WWW document. Available at: https://flutter.io/technical-overview/ [Accessed 18 April 2018].)

Intel Corporation. 2016. Docs Multi-OS Engine. WWW document. Available at: https://doc.multi-os-engine.org [Accessed 18 February 2018].)

JetBrains. 2017. Kotlin/Native Tech Preview. WWW document. Available at: https://kotlinlang.org/docs/reference/native-overview.html [Accessed 18 February 2018].

JetBrains. 2018. Kotlin/Native with CLion. WWW document. Available at: http://kotlinlang.org/docs/tutorials/native/kotlin-native-with-clion.html [Accessed 3 March 2018].

JetBrains. 2018. Multiplatform Projects. WWW document. Available at: https://kotlinlang.org/docs/reference/multiplatform.html [Accessed 20 April 2018].

Klubnikin A. 2016. Cross-platform vs. Native Mobile App Development. WWW document. Available at: https://medium.com/all-technology-feeds/cross-platform-vs-native-mobile-app-development-choosing-the-right-dev-tools-for-your-app-project-47d0abafee81 [Accessed 21 April 2018].)

Leiva Antonio, 2016. Kotlin for Android Developers. CreateSpace Independent Publishing Platform. Amazon.

Subham A. 2017. How React Native works. WWW document. Available at: http://www.discoversdk.com/blog/how-react-native-works [Accessed 21 April 2018].)

The Windows Club. 2017. What is Xamarin?. WWW document. Available at: http://www.thewindowsclub.com/what-is-xamarin-and-cross-platform-mobile-development [Accessed 21 April 2018].)

Wikipedia. 2018. Android Studio. WWW document. Available at: https://developer.android.com/ [Accessed 8 April 2018].)

Wikipedia. 2018. Gradle. WWW document. Available at: https://en.wikipedia.org/wiki/Gradle [Accessed 4 April 2018].)

Wikipedia. 2018. Xcode. WWW document. Available at: https://en.wikipedia.org/wiki/Xcode [Accessed 8 April 2018].)

Wm L. 2018. What's Revolutionary about Flutter. WWW document. Available at: https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514 [Accessed 1 May 2018].)