TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Roman Ismagilov 172665IVSM

Migrating an existing Android application to a cross-platform

Master's thesis

Supervisors: Juhan-Peep Ernits (PhD)

Oleg Petšjonkin (MCs)

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Roman Ismagilov 172665IVSM

Olemasoleva Androidi rakenduse üleviimine platvormisõltumatuks

Magistritöö

Juhendaja: Juhan-Peep Ernits (PhD)

Oleg Petšjonkin (MCs)

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Roman Ismagilov

10.05.2021

Abstract

First aim of this thesis is to give an overview and compare existing technologies that provide possibility to create cross-platform mobile applications. Second aim is to create a novel architectural approach by using which it would be possible to migrate an existing Android application to cross-platform with iOS. The developed approach is based on Model-View-Presenter design pattern and Hexagonal architecture using Kotlin/Native and Kotlin Multiplatform technologies.

The results are validated in a case review, which is a real-world application named Lokimo, which was successfully rewritten using the developed architecture. Since the application revenue model is based on paid digital content, and application stores user progress, the migration includes seamless transition for users that would update the application on their phones.

This thesis is written in English and is 62 pages long, including 5 chapters, 38 figures and 3 tables.

Keywords

Android, iOS, Kotlin, Swift, mobile development

List of abbreviations and terms

UI	User Interface
UX	User Experience
DI	Dependency Injection
LLVM	Low Level Virtual Machine
LLDB	Low Level Debugger
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
JIT	Just-in-time
CPU	Central Processing Unit
XML	Extensible Markup Language
API	Application Public Interface
QR	Quick Response
RAM	Random Access Memory
SDK	Software Development Kit
HTTP	Hypertext Transfer Protocol
OOP	Object Oriented Programming
SQL	Structured Query Language
BPMN	Business Process Model and Notation
IDE	Integrated Development Environment
IBOutlet	Interface Builder Outlet
GUI	Graphical User Interface
ARC	Automatic Reference Counting

Table of contents

List of figures

List of tables

1 Introduction

Nowadays, smartphones are becoming an integral part of everyday life. One of the main consequences of this is the fact that mobile application development is now one of the most demanded fields in commercial software development. It happens because of the vast potential creditworthy audience. However, the main problem in that kind of development is a strong fragmentation of the mobile operating system market. Even when modern mobile operating system market is narrowed down to only two of them, they still have a lot of fundamental differences in behaviour and tools used for the development. The problem with developing separate native applications is that it is more time-consuming, and it is reported [1], that developers usually manually compare versions of their applications for different operating systems in order to find rough edges where logic is different, or some features are missing. Such an approach is laborious and error-prone, so it is not surprising, that companies tend to use cross-platform frameworks.

Several technologies provide the possibility to write code for Android and iOS platforms simultaneously, but there is a noticeable issue with performance. Also, it is not possible to adapt an existing application, already developed for one platform, developers would have to reimplement everything from scratch. There are some tools that could be used to have a shared codebase for both Android and iOS while having UI written in platform-specific frameworks. This thesis aims to provide a brief overview of both approaches and compares which one of the following is better: to reimplement applications entirely using cross-platform framework, or develop missing platform independently or use a shared codebase. Moreover, it is important to note that Android and iOS have different UX and UI standards, if the application would look and work identically on both platforms, some users could find it inconsistent, applications should behave the way other application behave on each operation system.

1.1 Problem statement

The first objective of this paper is to review current academic and business-related research on how to develop cross-platform applications. To achieve this, the problem was decomposed into these research questions:

What are the existing frameworks providing the possibility to develop cross-platform mobile applications using one programming language?

What are the existing tools providing the possibility of writing shared codebase?

How could these technologies be compared? Which applications are written using them and how they could be compared to the native ones?

After making research and having selected the technology that fits the most, next objective would be to create the structural approach on how the application should be implemented in order to meet the modern mobile application development requirements. The objective could be decomposed into several questions:

What requirements does the pattern have to achieve, and why they are important?

What are the existing design patterns and how can they be applied in the scope of this research?

What would the designed structure look like and how it will fit the listed requirements?

The third objective is a real case review. Estonian company, Apico OÜ has developed Android application and there was a need to develop an iOS version. In order to make the development cheaper and faster, it was decided to migrate the existing Android application to the developed framework. This problem could be separated into several questions:

What are the steps to migrate the codebase?

What are the unforeseen problems the developers could face when doing the refactoring?

What are the effects on the team performance, and how it could be managed in the best way?

In other words, the goal of this paper is to create an architectural pattern based on the technology that fits the best. After that, there will be done a review of migration issues and made measurements of application performance and tracked the effort put on the development. It is not possible to track accurately in numbers how much this method would be more efficient rather than creating separate applications, but at least it would be possible to count man-hours.

With all the question stated, it is possible to come up with the **hypothesis**, that applying specific architectural approach on Android project could significantly reduce efforts for developing a similar native iOS application. On the other side, it would be said that efforts put on developing an iOS application will be the same, regardless for the architectural approach applied on the Android application. This will be the **null-hypothesis** of this research and it will be necessary to prove it wrong it by answering the listed questions step-by-step.

2 Background

2.1 Currently existing frameworks providing the possibility to develop a cross-platform mobile application using one programming language

This market is overly saturated with technologies. Competitors develop different technologies, and there is no dominant framework in this field. The overview of several of them is provided below.

Adobe Air. One of the oldest competitors in this market. All code is written in ActionScript. The framework has noticeable imperfections that are inherent for all similar technologies, and the technology is outdated. This solution could not be seriously considered as a framework for a new project, but it would be interesting to apply the imperfections on other technologies based on the same idea in order to detect possible problems.

To run an application based on non-native components, there should be some kind of an intermediate layer. The ActionScript is a dialect of ECMAScript, and it is running in the Adobe Integrated Runtime. AIR uses the Flash technology which is really slow and outdated. To run the application the user needs to download standalone package, and it is not obvious to user, why one more application should be installed. On the one hand this approach decreases the application size, on the other hand it could be very inconvenient, especially because this technology is not widespread nowadays. Because of bulky intermediate tools, the speed of the applications is much lower compared to the native ones. Moreover, modern third-party libraries are not available or they are released with a long delay. [2]

React Native. One of the most popular cross-platform frameworks. It uses JavaScript to compile it to platform code using target platform widgets and libraries. It is an interesting product, but opinions about it vary greatly. React Native code, as well as original React, is written in the extended version of JavaScript: JSX, which helps to write stateless UI widgets in the functional programming style [3].

Flutter. It is a new technology on the market. The first stable version was released only on the 4th of December 2018. It uses the Dart programming language and it is compiled to platform applications with non-native widgets and libraries, rendering everything using OpenGL (Skia graphics engine). The approach is similar to Adobe Air, with the difference that it produces native ARM code packaged with Dart runtime, instead of having a standalone runtime application, and optimizations could be done on the compilation phase, removing unused parts. It is intriguing to compare the performance. Flutter has an architecture that includes widgets that are claimed to look and feel as if they are native to the operating system, moreover, they are richly customizable and reusable, declared to be fast and extensible [3].

2.2 Existing tools providing the possibility of writing shared codebase

The idea behind this approach is to write common business logic, domain models, and everything which is not connected to the specific platform once, and then use it in natively written mobile applications as libraries, while having only UI and some platform-dependent code in those native implementations. In theory, this could provide a more flexible development process compared to the one investigated in the first question. For example, it would be possible to use libraries written specifically for iOS, without having to somehow port it to the Android version. In an effort to use different platform libraries in the shared codebase, it is possible to write common abstractions, where implementation is platform-specific. Also, it would be possible to implement some different functionality for each platform because the market policies of the Apple App Store and Google Play are different. Moreover, there are some restrictions in operating systems, for instance background functionality in iOS is considerably more restricted.

This niche is evolved to a limited degree, but there are two interesting technologies already available. Both of them also support platform-specific development using the language provided by the framework instead of native ones.

Kotlin/Native. The shared codebase is written in Kotlin and compiled using LLVM. This technology has newly arrived, there is only a beta version available at the moment of writing. The appealing fact is that Kotlin could be used for the native Android development as well, and it would run seamlessly, however it will be required to compile the Swift-compatible library for iOS. Also, it is possible to write iOS platform code using Kotlin with access to Foundation classes, which is beneficial when it comes to customizing concurrency.

Compilation logic:

- iOS: Kotlin classes are compiled to C-compatible binaries using LLVM. The output is a .framework file which is a native library extension for iOS and macOS. When using Kotlin for iOS platform code, one has to use specific annotations, but all the classes are accessible by their original names, as it is shown in the Figure 1:

```
@ObjCOutlet
lateinit var textField: UITextField
```

Figure 1. iOS native UI element outlet. Kotlin

- Android: There are 2 possibilities to use a shared codebase: First is generating JVM6 bytecode. Since Kotlin and Java are the official languages for Android development, and they are fully interoperable, while compiled to JVM, it is the best developer experience. The second approach is generating LLVM binary in the similar way it does with iOS.

- Other platforms: it is also possible to compile Windows, macOS, Linux, WebAssembly binaries.

However, there are some limitations in the technology, namely, it is not possible to use libraries written in Java or any other JVM-language, except for Kotlin in the shared codebase. Since *Kotlin/Native* uses the LLVM compiler instead of JVM for iOS, it would be impossible to compile any Scala/Java/Clojure library. But it is possible to use them in platform-specific code.

Xamarin. The shared codebase is written in C#. This technology was presented in 2011 and it is still receiving updates from the Microsoft. It is possible to do anything a developer would do in Objective-C, Swift or Java but in C#. Using Xamarin insights tool allows crash and issue reporting as well as user sessions monitoring [5]. The platform supports the so-called Portable Class Library which contains C# classes compiled into selected platforms. It supports iOS, Android and Windows. Also, it is possible to store shared resources (for example, JSON assets) in the shared library. Unfortunately, the commercial usage is not free.

Compilation logic:

- iOS: C# classes are compiled to ARM assembly language with a lightweight version of the .NET framework implementation included. For a platform-specific code written in C#, it is possible to use Apple's CocoaTouch SDK classes.
- Android: shared code is compiled into the Common Intermediate Language package with embedded MonoVM and JIT. Android-specific code has the possibility to use any Google's Android SDK packages as namespaces.
- Windows: compiled into the Common Intermediate Language and ran by built-in runtime.

For both Android and iOS implementations, .NET/Mono frameworks are reduced at compilation time by removing unused classes in order to minimize the installation file size. However, the support of C# classes from standard library is limited. Also, it is impossible to use dynamic .NET languages such as IronPython, IronRuby.

Since Kotlin shares many language features with Swift, C#, and Java, there would be no particular benefit to using them, but there would be many difficulties when dealing with platform-related problems since solutions accessible on common internet platforms (Stackoverflow, Github issues) are mostly about native language implementations. Also, official documentation and tutorials for iOS are written in Obj-C/Swift, and the developer would have to rewrite them in Kotlin/C#. Likewise, Android has all the official documentation provided with examples in Java/Kotlin. The advantage of using non-native language for native development is insignificant compared to corresponding problems and limitations. Instead, it would be much more justified to use one language for platform-independent code, particularly in the case when there is a tangled business logic, a sophisticated domain model, complex computations, and/or if there is a need to continuously update listed features for the reason that it would require two teams to implement same logic twice and cover it with two times bigger number of tests.

Also, it is possible to use both Kotlin/Native and Flutter in the same project. The first technology would be responsible for the platform-independent code, while the second one would only render UI. In addition, there would be a platform-specific code layer. This approach is unusual and could require a specific case to be used for. It requires a team to use 4 frameworks and write code in 3-4 languages (depends on which language would be used for the Android platform: Java or Kotlin). Moreover, there will be notable inconveniences in communication between Flutter and other layers due to contracts of Method Channels, which is illustrated in Figure 2, which is adapted from the research made in OLX [7].

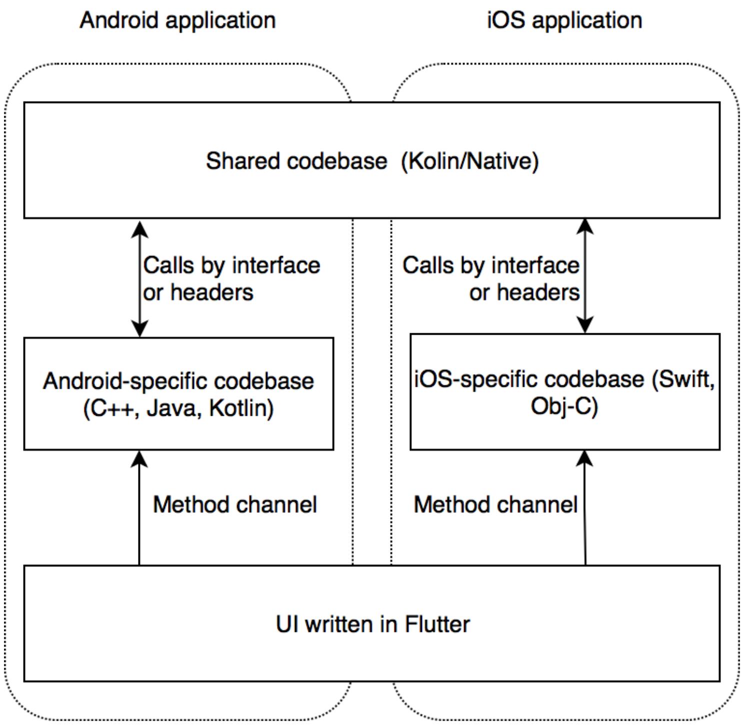

Android application
iOS application
Shared codebase (Kolin/Native)
Calls by interface or headers
Calls by interface or headers
Android-specific codebase (C++, Java, Kotlin)
iOS-specific codebase (Swift, Obj-C)
Method channel
Method channel
UI written in Flutter

Figure 2. Architecture where both Flutter and Kotlin/Native are used

2.3 Technology comparison

Table 1 is used to visually compare these frameworks based on academic studies and other reports found on the internet. The learning curve section is filled from perspective of a native Android developer without any significant experience in Web or iOS development.

Table 1. Frameworks comparison

Technology	Runtime bundling, its size	Battery/RAM/CPU impact	Learning curve	Community
Air	Standalone application	No reports were found	Pretty straightforward to start writing an	Limited number of community-

	Adobe Air needs to be installed to run compiled code bundled into apps.		application. Hard to maintain and achieve good performance. Need to learn ActionScript, which is an implementation of ECMAScript, hence it shares a lot with JavaScript.	created libraries. Stackoverflow has only 41215 questions with “actionscript” word included. Github has about 1800 repositories written in this language, even when it is a mature technology that first appeared in 1998.
React Native	Embedded instance of V8	Average 30% more CPU consumption [7], feasibly higher memory consumption. Andreas reports [6] that React Native consumes 224% energy compared to native Android applications. Furthermore, the study reports that RN applications take 25% more time to launch.	React itself is not very easy as web framework, compared to Vue.js; it is quite hard to switch to it from native Android. Could be very easy to start if developers already know React.	React Native is the 3rd most starred project in Github. Its community is really huge, having all needed libraries, and common development problems solved.
Flutter	Dart needs the Dart Virtual Machine to run. Thus, the engine is compiled into native code and bundled with the application.	The problem is this technology is very young and there are no serious investigations on the question. However, there are several non-scientific articles, based on creating a single app and consumption comparison. [7] measured that Flutter applications consume 30% more memory, but CPU	The framework is designed in React-like style. But Dart is not a well-known language, and in the most cases, developer will have to learn it from the scratch. According to TIOBE index, Dart shares 0.38% of popularity and takes 34th place.	The community is very young. There are not many libraries available for Dart. There are only 27000 repositories on Github.

		usage is close to the native one. [9] built a stopwatch application and came to the conclusion that Flutter app is twice more CPU and memory consuming compared to native iOS application when running the stopwatch.		
Xamarin	Lightweight version of .NET/Mono Framework bundled into application. Jiang [10] developed 2 native applications, their compared size to Xamarin one: native iOS was 33 Mb, native Android 64 Mb, Xamarin 150 Mb.	40% longer start time compared to the native one was reported [10].	This paradigm is very easy to learn, because it is the same style compared to the native development: OOP codebase written in C# is way similar compared to Java ones.	There are 19000 Github projects having Xamarin in their name or tags. Furthermore, there are more than 400000 repositories written in C#. Moreover, it is possible to use native libraries for platform specific code.
Kotlin/ Native	Regular application size plus .so library containing LLVM code.	No reports were found. Measurements are listed below in Computational performance overhead section.	Kotlin is very easy for developer who already know Java. It takes 1-2 weeks to make one feel comfortable with it. But it is hard to switch to Kotlin-only libraries from the Java ones. Moreover, developers need to learn the iOS platform.	All libraries written in Kotlin for Android, server and desktop development are available for Kotlin/Native. There are 37400 projects in Github written in Kotlin. But Kotlin/Native technology is very young. Also, it is

				possible to use native libraries for platform specific code.

After reviewing the market of technologies, it becomes clear that the best solution to migrate an existing project would be Kotlin/Native, because developers would not have to rewrite the entire codebase in a new language. Moreover, this approach will make the UI part of the application fully native, and the importance of this is described in the research made by S. Xanthopoulos [11].

Although the team will have to either learn iOS platform or hire new developers in order to make the UI and platform-dependent part.

3 Proposed development pattern

3.1 Overview of existing patterns

To begin with, this section reviews popular patterns for iOS and Android to decide which one would be taken as the basic for the research. The decision will be based on the implementation difficulty and technical restrictions of the platform. This can probably make some strengths of the pattern unnecessary or dysfunctional. Based on several researches [14, 15, 16], the most popular design patters for Android and iOS could be listed as:

Android:

- **MVC** (Model — View — Controller)
- **MVP** (Model — View — Presenter)
- **MVVM** (Model — View — ViewModel)

iOS:

- **MVC** (Model — View — Controller)
- **MVP** (Model — View — Presenter)
- **MVVM** (Model — View — ViewModel)
- **VIPER** (View — Interactor — Presenter — Entity — Routing)

Table 2. Comparison of selected development patterns

Pattern	Advantages in the current scope	Disadvantages in the current scope
MVP	View is as simple as possible. View does not know about Model. Presenter and View could be platform-independent. It is reported that this pattern is the most memory-efficient [14].	Quiet a lot of boilerplate code: each View and Presenter should have a contract.

	According to a study made by M. Potel [16] it is more modern version on MVC, which is adapted to event-driven systems.	
MVC	Default development pattern in iOS development. Would make it easier for iOS team to read the core code. The most well-known pattern according to research made in the University of Technology of Troyes [17].	Views are not that simple as in MVP, because they know both about Model and Controller. UI logic is not limited to a single class. Also, default iOS MVC implementation encourages developers to make ViewControllers instead of separate Views and Controllers, so that they will have a lot of logic, which does not apply to our concept. Worst in terms of memory and CPU usage.
MVVM	Fewer interfaces to declare. Easier view state management. Reduced complexity and improved reusability according to the International Journal of Computer Science [18]. Furthermore, it is reported that MVVM increases data independence and improves application logic encapsulation.	Difficult to test an application when having complex View Model. Harder to separate domain level in Android. Google's implementation of LiveData and ViewModel could not be used in the core because they are platform-dependent. Furthermore, Lou. T. reports that it becomes more difficult to debug Android applications using MVVM compared to other patterns [14], because of moving presentation logic to XML files.
VIPER	Same advantages as in MVP. High level of code decoupling. Allows code reusability.	Not a common approach in Android development.

After looking through the most popular patterns, it becomes apparent, that the suitable ones in this scope are MVP and MVVM. The biggest advantage of MVVM, good support in Android, is nullified by the fact that platform libraries could not be complied on iOS. Nevertheless, it could be implemented again from the scratch, but it will be very development-intensive task to bind Android XMLs and iOS storyboards in the same way. In this case the best choice would be Model-View-Presenter.

3.2 Requirements for the solution

Now, when we have selected MVP as the basis for the pattern, the next step is to organize the code structure. Then, it is important to determine, which part of the code will be in the core and in the platform. Finally, it is necessary to assess how it will look like. It is obvious that the more code is presented in the shared codebase the better, but the architecture should help developers achieve some goals, which could be listed as:

- **Testablity**

From this point of view, the approach will not differ from regular MVP a lot. Presenters and business logic should be covered with unit tests, while platform modules should be covered with automation tests. Moreover, the tests of Kotlin Multiplatform core modules could be running with JVM, which means that all the popular and time-proven tools, such as JUnit, Mockito, etc. could be used, and developer is not obliged to learn new ones written in Kotlin.

- **Extensibility**

It should not be a big problem for a developer to add a new feature when the project is already well-developed and has massive code-base. The common solution for that is separating code into feature-based units, which could be packages or separate modules. It will make easier to navigate through the code when all the parts of a single feature are located close to each other and irrelevant parts are isolated or incapsulated. In our case it is also possible to significantly improve build speed using separate modules because Kotlin/Native supports incremental builds. It means that when some parts of the code were changed since the last build, only those modules, that have their parts edited, will be recompiled. So, the average build time would not be significantly different when the project has only 1000 lines of code or 50000. Peculiarities of adding new modules are reviewed below in Module structure section.

- **Flexibility**

It is not a secret that in modern world the requirements for the application can change throughout the whole development process. If it is a startup, the initial business model is continuously adjusted, new hypothesizes are tested on real users, third-party services used by application are changed. For example, what will happen if management decide to change analytics platform because pricing is better or work is more stable? All third-party services should be encapsulated into separate module, providing contract, that could be used from the core.

- **Reusability**

What if it will be necessary to develop another application for the project that will have a lot of similar features in the future? For example, after year of developing application for content consumers, managers will require a separate application for a content developer or content moderator? It will be a resource waste to develop them from scratch or copy-paste existing code to a new project.

All of these goals should be considered while developing the architectural pattern with a focus on supporting code quality throughout the whole lifespan of the application. All those requirements are perfectly solved with Kotlin/Native.

3.3 Clean Architecture

There are a lot of different approaches on an architecture of systems, but Robert C. Martin figured out similarities between them and summarized them into the Clean Architecture concept. According to his study [14], although the particulars of these architectures differ, they all are working for the same goal, which is the separation of concerns. They all accomplish this by layering the applications. It has at least one layer dedicated to business rules and another to interfaces.

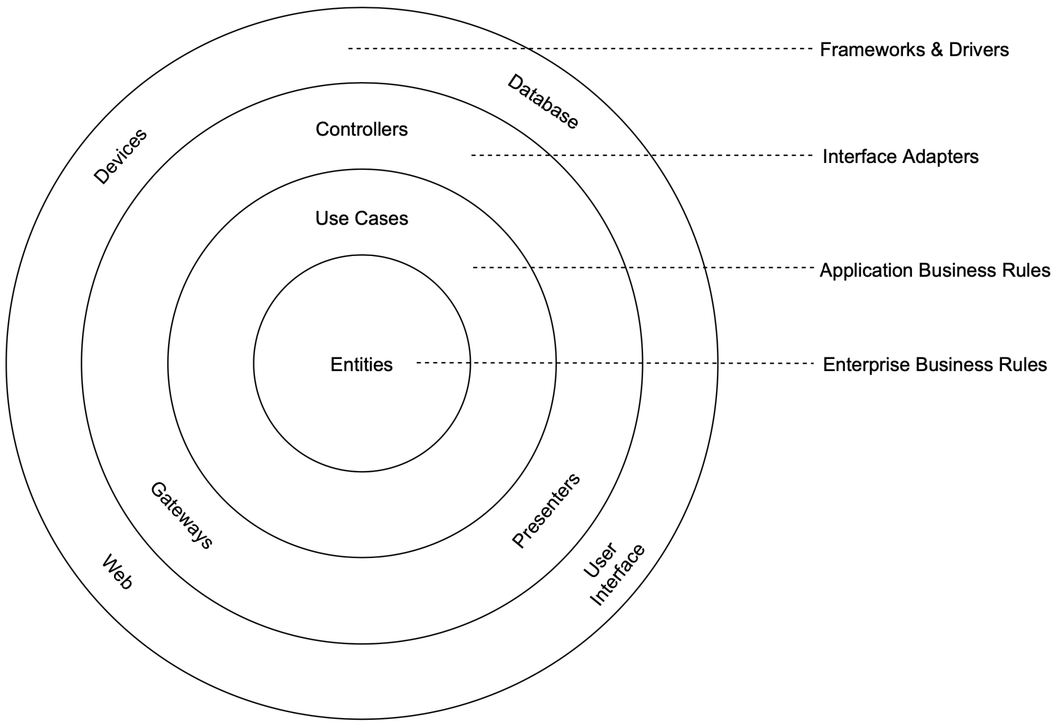


Figure 3. Simplified scheme of Clean Architecture approach

As shown in Figure 3, there are concentric circles that represent layers where inner circles do not have access to outer ones. In other words, code in the Enterprise Business Rules section can't mention class names from any other circles, whereas it is possible to use everything in the Frameworks & Drivers section. One can benefit from applying these rules in having external parts of the system easily changed, and making the system abstracted from third-party libraries, external API specifications, UI frameworks. This in all fulfils the requirement of **flexibility**, and such a system becomes intrinsically **testable**. Thus, it is important to apply this recommendation to the desired architecture. Having MVP in mind, we can initially separate the system into 3 layers: contract, having all the interfaces, possible navigation transitions, data classes and models; core, which will have all the business and presentation logic; and the platform, that will have all the context-aware code.

3.4 Core and Contract structure

The structure of the core and contract levels would be similar to the ones in the Model-View-Controller approach. In addition, there will be a separate module for navigation contracts for each feature. There will be one module for networking, which will have implementations of interactor contract for all the features. The same will apply to the database, analytics, etc. One can observe in the Figure 4, that it is quite straightforward to replace network (or database) module implementation with another one, in case there will be a need to change used libraries.

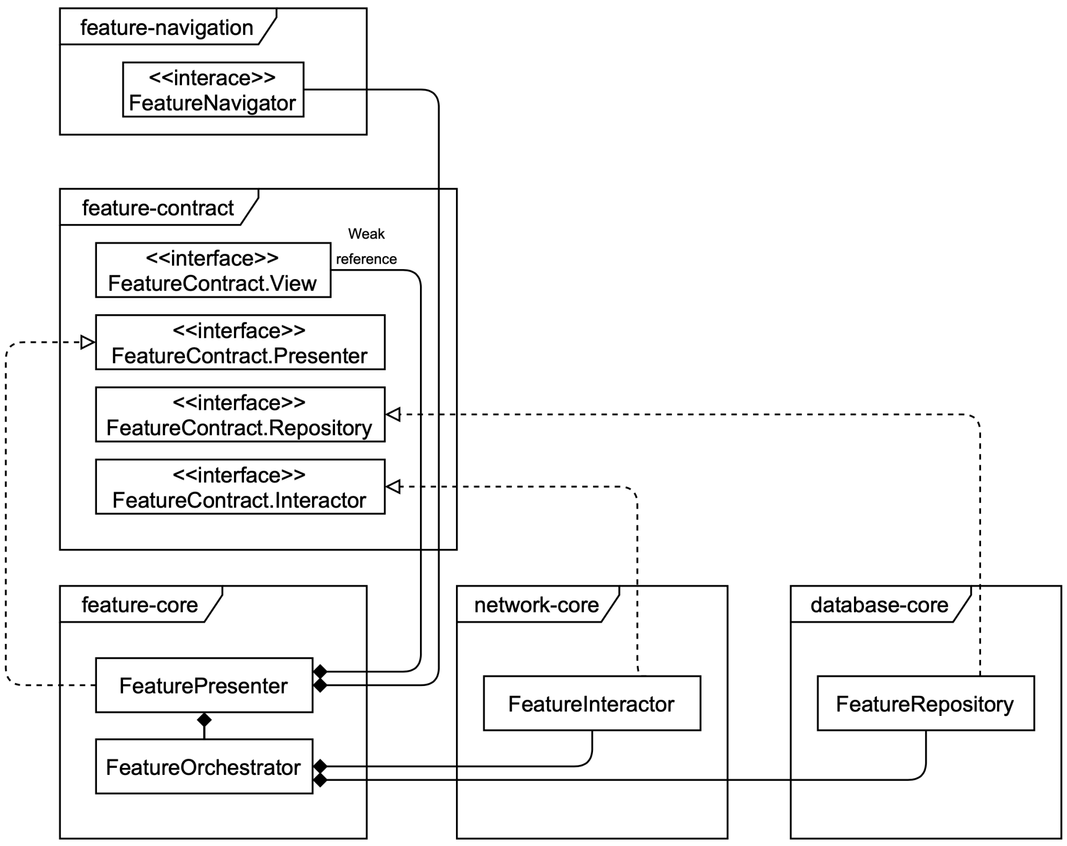


Figure 4. Schematic representation of an abstract feature with some networking and persistence functionality.

The role of Orchestrator is to encapsulate the logic of network and database method calls arrangement, so that in Presenter it will be possible just to call one method, and all the caching, mapping and error handling would be done separately. The need for a weak reference need will be justified in the Memory leaks section.

3.5 Android structure

Each feature will also have an Android module, which will contain all the platform-specific code, namely Activities, Fragments, Views, Adapters, ViewHolders, Navigator implementations, etc. It is important to decouple Views and the native Android components, such as Activities, for the purpose of single responsibility: View is handling the layout updates and the user events listening, whereas Activity takes care of the component lifecycle and the system events and broadcast messages.

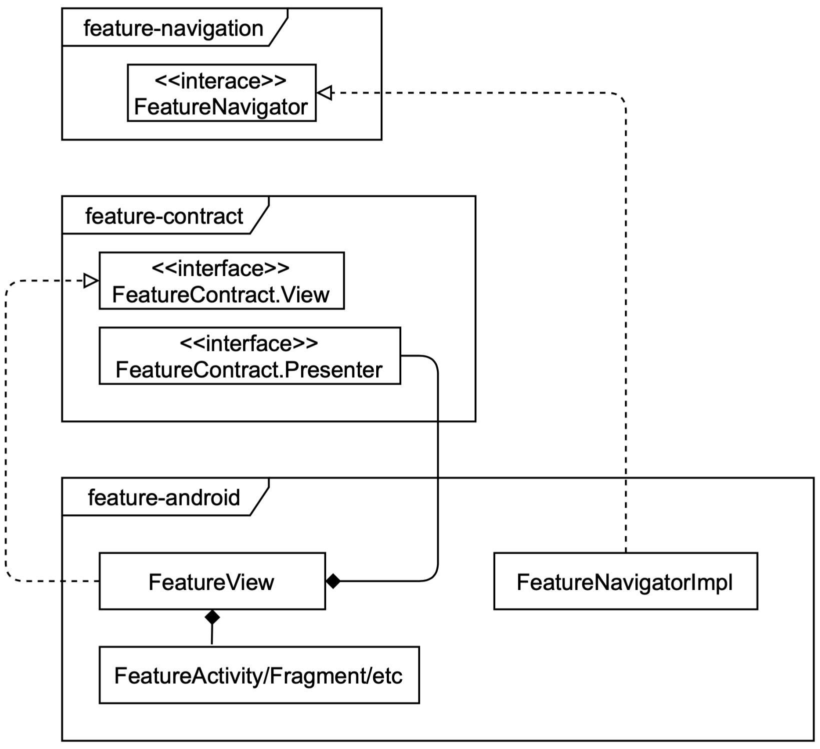


Figure 5. Schematic representation of Android implementation of an abstract feature

Feature navigator implementation class will have methods that handle Intent building, Parcelable mapping, and, possibly, work with the navigation graph from the Android Jetpack.

3.6 Android dependency injection

In order to keep communication between the different layers clean and meet the requirements of encapsulation, it is crucial to design the dependency injection approach. Given that all the features and layers are separated into individual Gradle modules, each one of them will have its own DI module, which will initialize and provide related instances to the class constructors in the feature scope. The scope will be defined by a feature component, which will contain all the required modules. All the common modules of features, such as database or network, will be composed in a base component, that will play a role of super-class for each feature component.

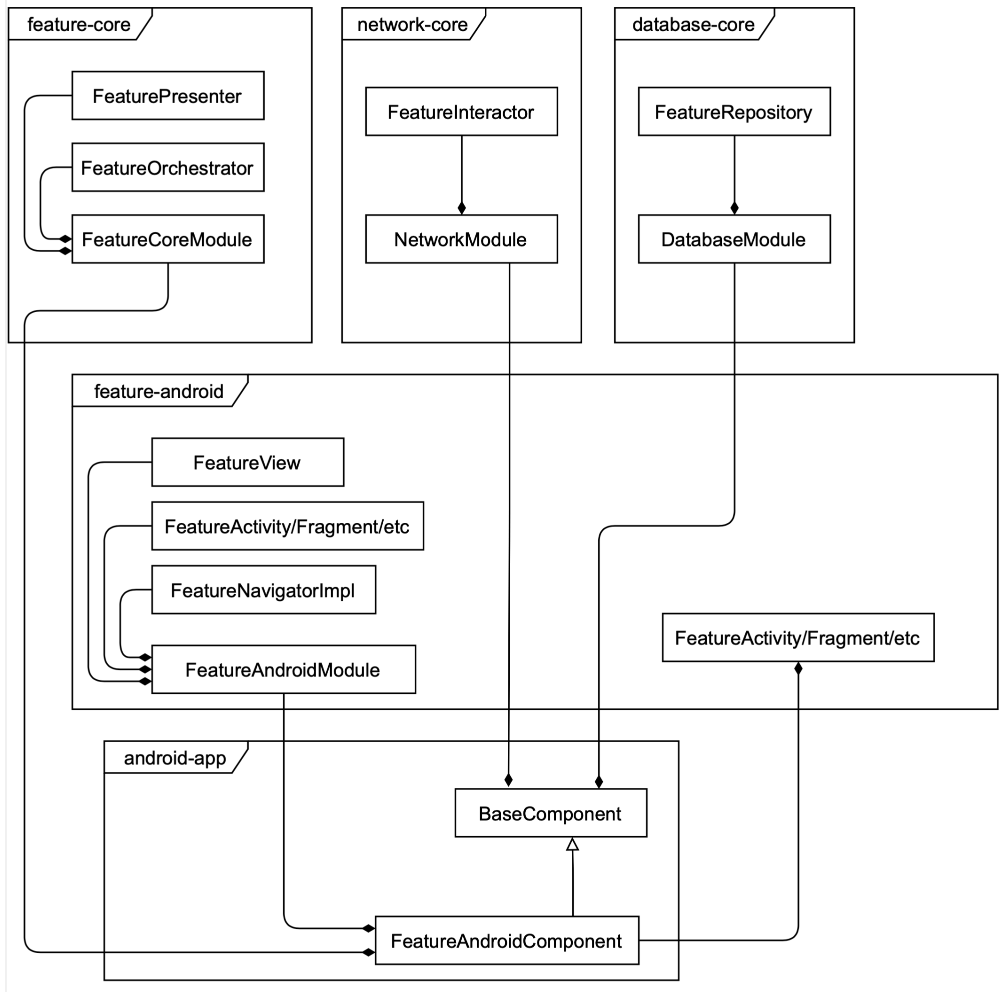


Figure 6. Schematic representation of Dependency Injection graph for an abstract feature in Android application.

As we can see in Figure 6, a feature could be easily injected into any other app with similar architecture, which will meet the requirement of **reusability**.

3.7 iOS-specific distinctions

The main challenge of implementing this approach on iOS is that Swift and Kotlin are not mutually interchangeable and it is not possible to use all the language features of Kotlin in Swift and vice versa. To solve this problem, the ios-combined Gradle module is introduced to the structure. It will mostly have wrappers and dependency injection components. iOS-specific DI modules will be declared in Swift code and then passed as arguments to the feature component. Visual representation is shown in a Figure 7.

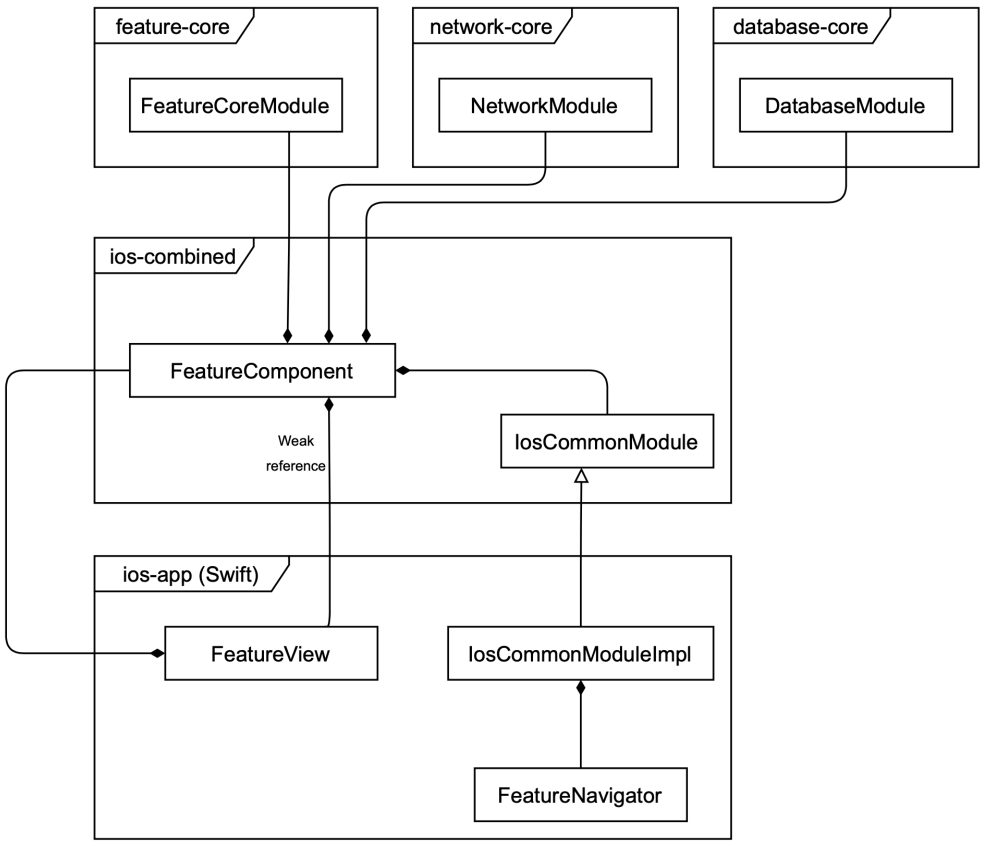


Figure 7. Dependency Injection scheme for iOS

View is basically a renamed ViewController from the default iOS project structure that will implement methods from the contract to be executed by the Presenter. Component will be instantiated in View’s viewDidLoad method. If a component requires some other Views as arguments, they will be acquired using IBOutlets and passed among others. After creating a component, developer would be able to get a Presenter from it and call its methods in accord to user and system events, and it will call View’s method updating the contents.

3.8 Clean Architecture compliance

So far, we have designed the module structure, now we have to create rules of modules that should depend on each other. In order to make it compliant with the Clean Architecture recommendations, the Table 3 was filled with Gradle modules assigned to the corresponding layer.

Table 3. Clean Architecture layers and their counterparts in the proposed pattern scope

Clean Architecture layer	Gradle module name	Contents
Frameworks & Drivers	feature-android common-android ios-combined network-core database-core ios-app android-app	All the View implementations, networking and persistence, wrapping, navigation implementations, etc.
Interface Adapters	feature-core	Orchestrating requests, presentation logic
Use Cases	common-core	Business and domain logic
Enterprise Business Rules	common-contract feature-contract feature-navigation	Models, contracts.

As it was mentioned, the inner circle should not have access to outer circles. So, we have to keep in mind that Gradle modules can be linked to ones in the same row or to the ones below, but they can’t link to the ones above.

3.9 Gradle modules hierarchy

The Table 3 can be represented more visually. The second feature was added for demonstration of how first feature could use it.

Also, and adjustment was made: feature modules of the same level can’t be linked to each other, except for common ones. In order to interact they will have to use abstraction from the contract level. That will improve **reusability** and make the implementation encapsulated from each other.

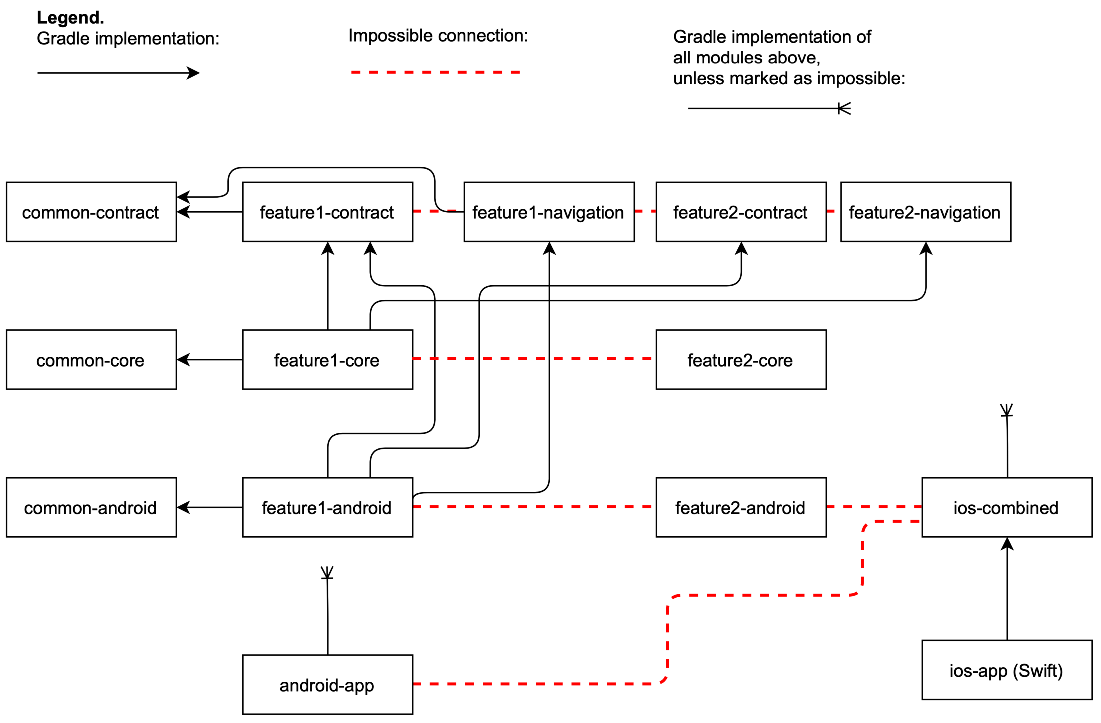


Figure 8. Gradle modules hierarchy

In this case separate Android features don’t have access to each other, and if they need to somehow interact with each other, it could be done in the presenter. In case of need to have some common elements, they could be placed in common-android module. Particularly, recourses, constant values, helper functions, reusable views.

Also, in this figure, it is notable, that one can add more applications, that will depend on different features they need.

4 Real case study

4.1 Application domain and specificity

At the very beginning of the development, application idea was to create a platform for city quests, where one segment of audience could compose interactive outdoors stories and other users could walk through them using their mobile phones. Something similar to geocaching but more complex. Application was designed to provide a set of tools to creators, where every story piece would be a separate screen or so-called Task. Application would have several task types: navigation task, where user has to physically reach to the destination; information task, which will just contain text and images; text input task, where user has to fill in the answer; multichoice, QR code scan task, put in correct order, etc.

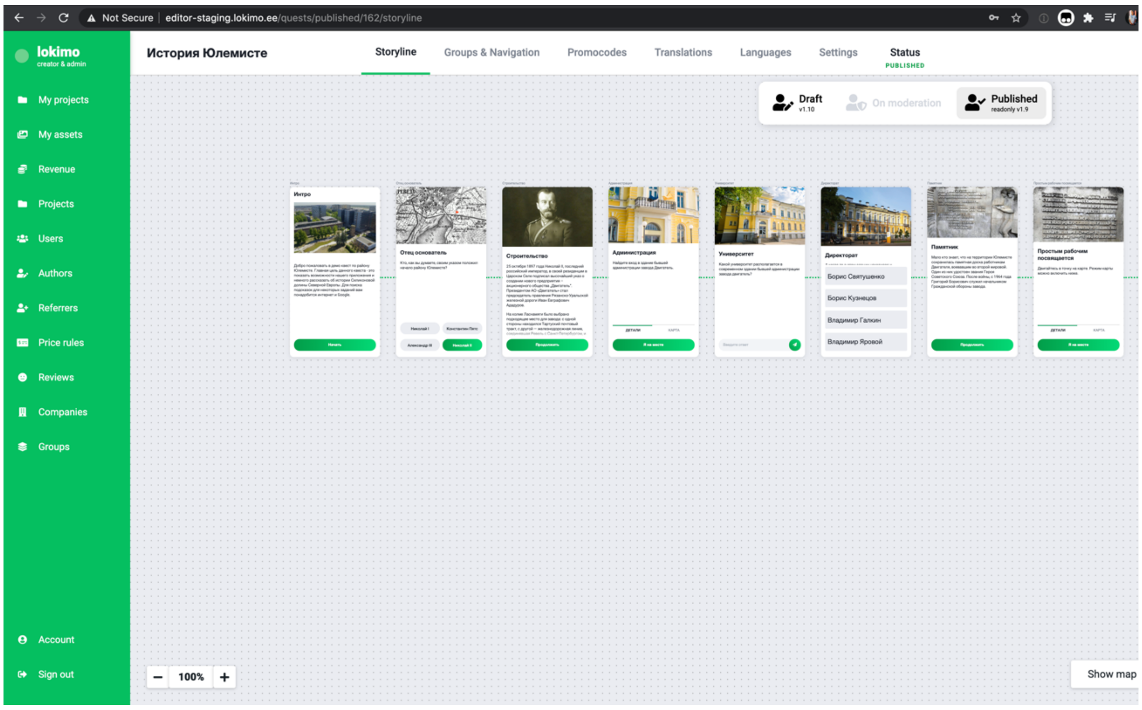


Figure 9. Screenshot of the story editor

The idea was to give the author maximum freedom of creation and allow building almost any type of game outdoors with different types of riddles and questions. After several months of the development, it became apparent that the idea is much bigger than just quests and it nicely fits the tourism market and could be used for creating interactive tours.

4.2 Choosing technologies to use

The reviewing application is an Android application with heavy offline logic and written entirely in *Kotlin*, but having a lot of *Java* dependencies. The design of the application is based on the *Clean Architecture* approach [6]. Also, the application is separated into several feature-specific modules. The architecture scheme is presented on the Figure 10.

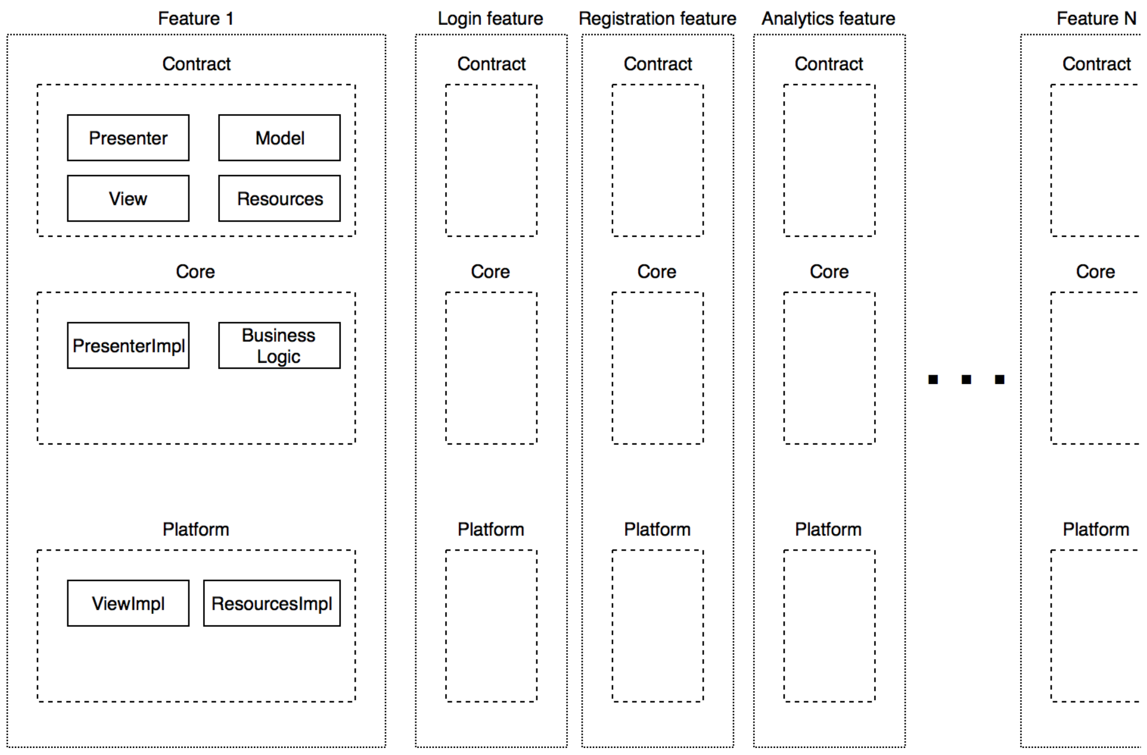


Figure 10. Simplified architecture scheme of the existing application

The platform logic is already isolated and there is no need for modifications, but the core and contract modules should get rid of *Java* dependencies, in particular, RxJava and Dagger 2.

Also, it is worth considering to use some *Kotlin/Native* libraries instead of those, used in the platform and move them to the core. For example, Retrofit was used to handle HTTP requests. It could be replaced with *Ktor*, and there will be no need for writing a separate platform implementation for iOS.

4.3 Libraries replacement

4.3.1 Dependency injection

Initially Dagger 2 was used in the Android application. While using this library, developer has to declare modules with bindings, which combine these modules into components that have equivalent scope with Android components (Activity, Fragment, Service). Also, it is possible to declare subcomponents for smaller objects: for example, for an item in list that should have its own dependency graph developer can declare subcomponent, and import needed modules. There are 2 popular Kotlin replacements available: Koin and Kodein. And only Kodein is compatible with Kotlin/Native. During the replacement the same logic was applied: components and subcomponents separation with modules importing. For iOS application components were declared in Kotlin code, because the library API is based on infix functions which are neither supported in Swift nor Objective-C. In overall, the replacement process took a week and made development process much easier: library is feasibly simpler. The only disadvantage is that Dagger 2 creates dependency graph during the compilation, and Kodein does that during the runtime, so sometimes it takes some time to find an error in dependency injection and there is an additional overhead while running the application to provide dependencies.

After refactoring was done, it was suddenly discovered, that both iOS and Android application have memory leaks because of some references being stored too long and not released after UI element was destroyed. The solution was to use weak references for all UI components managed by the platform, such as Activities/Fragments on Android and ViewControllers on iOS. More about that in Memory Leaks section.

4.3.2 Asynchrony and concurrency

The initial solution was RxJava 2: powerful library for reactive and asynchronous programming. The library itself contains approximately 10000 methods and significantly increases application installer file size even when ProGuard tool is used. it could be replaced with Kotlin coroutines, which moreover have better computational performance. It was found that coroutines are significantly faster when to comes to thousands of operations [12]. The migration was not very trivial: we had to rewrite a lot because of completely different approaches: RxJava has the logic defined in chained method calls in declarative programming style with a lot of callbacks. Coroutines code is written in synchronous style wrapped in coroutine context closures (e.g., launch function). In most cases it makes code simpler, developer can read the asynchronous blocks without learning coroutines, while it is almost impossible to understand RxJava chains without knowing its operator functions and scheduling logic.

For example, two methods, show in Figure 11, do the same. First one is written in RxJava, second in coroutines. While reading the first one, it is not obvious, what Single, Single.concat or fromCallable do if you haven't worked with Rx before.

```
override fun loadAssets(filePath: String): Single<SectionContainer> {
    val asset: Single<String> = Single.fromCallable { con-
text.readTextAsset("$filePath.json") }
    val remote: Single<String> = Single.fromCallable {
        remoteConfigInteractor.getRemoteConfig(filePath)
    }

    return Single.concat(remote, asset)
            .filter { it.isNotEmpty() }
            .firstOrError()
            .map { it ->
                Gson().fromJson<SectionContainer>(it,
                        SectionContainer::class.java)
            }
}
```

(a)

```
override suspend fun loadAssets(filePath: String): SectionContainer {
    val asset: String = context.readTextAsset("$filePath.json") ?: ""
    val remote: String = remoteConfigInteractor.getRemoteConfig(filePath)

    return listOf(asset, remote)
            .first { it.isNotEmpty() }
            .let {
                Gson().fromJson<SectionContainer>(it,
SectionContainer::class.java)
            }

}
```

(b)

Figure 11. Same method written using RxJava (a) and Coroutines (b). Kotlin

Second one written in coroutines is easy to read if you know Kotlin syntax. Since core codebase is also intended to be read by iOS team, this migration really helped to make logic more readable for them.

4.3.3 Network

Initial network implementation on Android was done with OkHTTP and Retrofit libraries. It was possible to left it untouched and implement separate networking service for iOS that will fulfil the protocols defined in the core, but in order to make all networking consistent it was decided to rewrite it from the scratch. Used technology is Ktor-client library. It is based on Coroutines and provides Kotlin-idiomatic code style.

4.4 Database migration

Originally, persistence was implemented using Room library, which is a part of Android Jetpack developed by Google. It was not obligatory to change it to some Kotlin alternatives, because it’s not used in core, only in platform, but this could force us to use CoreData or alternatives on iOS side, so we investigated if there are some cross-platform database solutions compatible with Kotlin/Native. And the selected library is SQLDelight. Both Android and iOS have SQLite as their database management system. SQLDelight is a library that is based on SQLite and generates type safe Kotlin API for SQL statements defined by developers during the compile time.

As we were doing the transition to the new library in the moment, we already had active users, it was crucial to migrate user data seamlessly, so that nobody would lose their progress. In order to carry that out, first thing to investigate before making a transition was to check migration functionality. Otherwise, in that moment, iOS application was not yet released and it was not important to keep progress. To summarize, these were the steps:

1. Analyse generated database on Android and iOS
2. Create a new database
3. Write migrations

In the project all entity models were declared as standard annotated data classes and data access objects were defined as annotated interfaces with SQL requests in annotation parameters as shown below in the Figure 12:

```
@Entity(tableName="profile_entity")
data class ProfileEntity (
        @PrimaryKey(autoGenerate = true)
        var pid: Long = 1,
        var id: Long,
       <…>
)
```

(a)

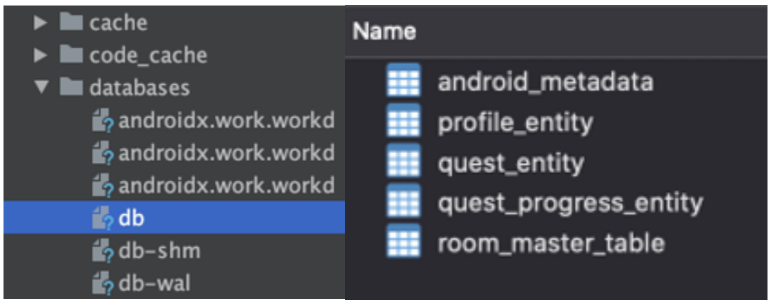
```
@Dao
interface ProfileDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insert(profileEntity: ProfileEntity)


    @Query("SELECT * FROM profile_entity WHERE pid=1")
    fun getProfile(): ProfileEntity?


    @Query("DELETE FROM profile_entity")
    fun deleteProfile()


    @Language("RoomSql")
    @Query("UPDATE profile_entity SET accessToken=:accessToken,
refreshTo-ken=:refreshToken WHERE pid=1")
    fun updateProfileTokens(accessToken: String, refreshToken: String)
}
```

(b)

Figure 12. Simplified example of entity (a) and data access object (b) declaration using SQLDelight. Kotlin

Room generates the database in the runtime, and it is possible to analyze the contents using the Device File Explorer in order to mitigate the unexpected contingency, as shown in Figure.

The name of selected class equals the database name, which is given in the databaseBuilder method of Room. It is possible to open the file using any third-party software, such as DB Browser for SQLite.



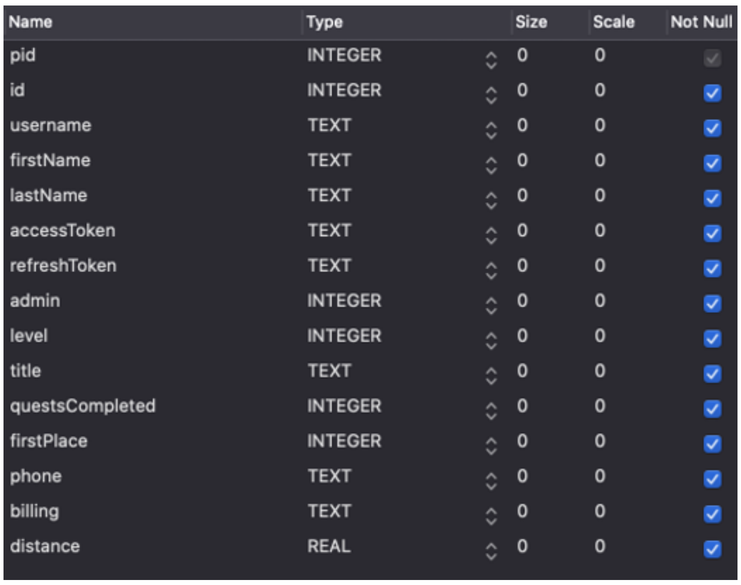
```
▶ cache
▶ code_cache
▼ databases
    androidx.work.workd
    androidx.work.workd
    androidx.work.workd
    db
    db-shm
    db-wal
```

```
Name
android_metadata
profile_entity
quest_entity
quest_progress_entity
room_master_table
```

(a) (b)

Figure 13. Database files (a) and tables (b) on Android device

Thereby, we can see what was generated besides the annotated tables. Also, we can check if the real table structure is different from the one in the code.

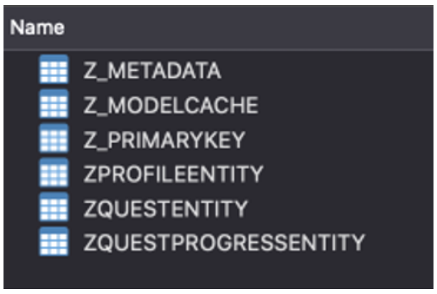


Name	Type	Size	Scale	Not Null
pid	INTEGER	0	0	✓
id	INTEGER	0	0	✓
username	TEXT	0	0	✓
firstName	TEXT	0	0	✓
lastName	TEXT	0	0	✓
accessToken	TEXT	0	0	✓
refreshToken	TEXT	0	0	✓
admin	INTEGER	0	0	✓
level	INTEGER	0	0	✓
title	TEXT	0	0	✓
questsCompleted	INTEGER	0	0	✓
firstPlace	INTEGER	0	0	✓
phone	TEXT	0	0	✓
billing	TEXT	0	0	✓
distance	REAL	0	0	✓

Figure 14. Contents of quest_entity table

As can be noted in a Figure, there is no significant difference between entities in the code and in the database. Next, we checked if database version is also consistent using PRAGMA user_version; command.

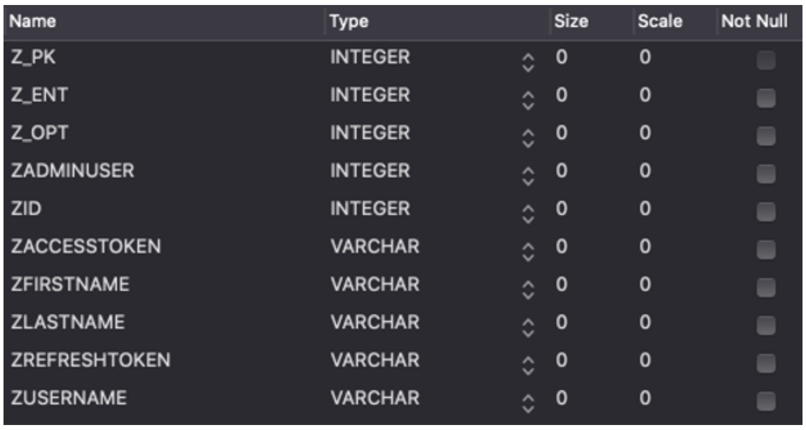
On iOS the database structure description was made in .xcdatamodeld file and had the a little bit less fields, because not all the features were implemented yet. By default, XCode generates database to /Library/Application Support/. It is also possible to open them and check the contents:



Name
Z_METADATA
Z_MODELCACHE
Z_PRIMARYKEY
ZPROFILEENTITY
ZQUESTENTITY
ZQUESTPROGRESSENTITY

Figure 15. Tables of database generated by CoreData

As we can see on the Figure, table names are different from the ones in the project. The table structure also is not the same:



Name	Type	Size	Scale	Not Null
Z_PK	INTEGER	0	0	
Z_ENT	INTEGER	0	0	
Z_OPT	INTEGER	0	0	
ZADMINUSER	INTEGER	0	0	
ZID	INTEGER	0	0	
ZACCESSTOKEN	VARCHAR	0	0	
ZFIRSTNAME	VARCHAR	0	0	
ZLASTNAME	VARCHAR	0	0	
ZREFRESHTOKEN	VARCHAR	0	0	
ZUSERNAME	VARCHAR	0	0	

Figure 16. Contents of ZQUESTENTITY table

Since CoreData stores its tables in different folder compared to SQLDelight, and it has different table naming rules, it was necessary to spend some time to investigate how to make this work seamless.

4.4.1 Creating a new database

Before anything else, we need to create a new module, that will have Android, Common and iOS source sets. The final structure after running migrations is illustrated below:

```
}
```

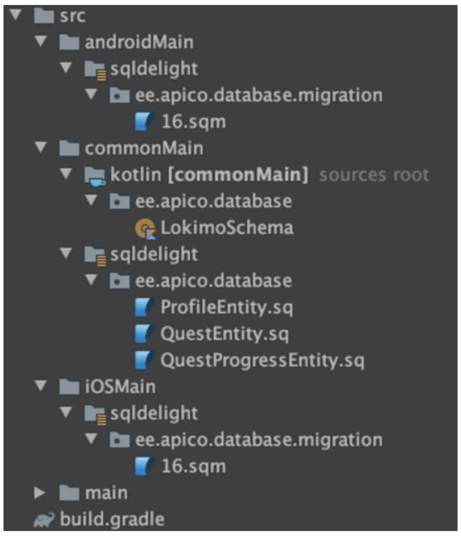


Figure 17. SQLDelight module structure

Next step is to add SQLDelight dependency to the project, what can be done with adding a classpath to buildscript of the root Gradle file and adding database metadata to the build.gradle file, as shown in Figure 18 below:

```
buildscript {
  repositories {
    google()
    mavenCentral()
  }
  dependencies {
    classpath "com.squareup.sqldelight:gradle-
plugin:$sqldelight_version"
  }
}
```

(a)

```
apply plugin: 'com.squareup.sqldelight'
apply plugin: 'com.android.library'
apply plugin: 'kotlin-multiplatform'
apply plugin: "com.squareup.sqldelight"

sqldelight {
    LokimoDb {
        packageName = "ee.apico.database"
    }
}
```

(b)

Figure 18. Linking database in root Gradle file (a) and in the database module (b). Groovy

Where “LokimoDb” is the project name and “ee.apico.database” is the package name. By default, sources are located in <sourceSet>/sqldelight

In this structure, androidMain and iOSMain will only contain files for the migration from native tools to the cross-platform one. All the consequent migrations will be stored in commonMain only. In order to create an entity it is enough to write an .sq file that will contain plain SQL query. Simplified example of scheme declaration could be observed in Figure 19 below. As an outcome, SQLDelight generates a separate class for each entity and statement.

```
CREATE TABLE profileEntity (
    pid INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL DEFAULT 1,
    id INTEGER NOT NULL,
    <…>
);

selectAll:
SELECT * FROM profileEntity;

insert:
INSERT OR REPLACE INTO profileEntity(pid, id, <…>)
VALUES ?;

getProfile:
SELECT * FROM profileEntity WHERE pid = 1;

deleteProfile:
DELETE FROM profileEntity;

updateProfileTokens:
UPDATE profileEntity SET accessToken=?, refreshToken=? WHERE pid = 1;
```

Figure 19. Creating an entity and queries for data access object. SQL

As it was stated above, there will be two separate migrations, one for Android, and one for iOS. Migrations are also described in format of plain SQL query. For Android, migration is quite simple, just renaming the tables:

```
DROP TABLE room_master_table;
ALTER TABLE profile_entity RENAME TO profileEntity;
ALTER TABLE quest_entity RENAME TO questEntity;
ALTER TABLE quest_progress_entity RENAME TO questProgressEntity;
```

Figure 20. Database migration query for Android. SQL

Also, it is important to consider, how user data will be migrated, if Android application will be updated from the version, that has an older scheme of database. In this case, intermediate migrations also have to work. For this case, it is necessary to leave all the previous migrations that were written for Room in the project and write code that will execute them.

In case of iOS, writing a migration will be a bit more complex, because all the tables and columns have different names. Simplified version of the migration of Profile entity and two fields is shown below in Figure 21:

```
DROP TABLE IF EXISTS profileEntity;
CREATE TABLE profileEntity (
    pid INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL DEFAULT 1,
    <…>
);

INSERT INTO profileEntity(pid, <…>)
SELECT 1, ZPID, <…>
FROM ZPROFILEENTITY
LIMIT 1;
<…>

DROP TABLE IF EXISTS Z_METADATA;
DROP TABLE IF EXISTS Z_MODELCACHE;
DROP TABLE IF EXISTS Z_PRIMARYKEY;
DROP TABLE IF EXISTS ZPROFILEENTITY;
<…>
```

Figure 21. Simplified query for iOS database migration. SQL

Next step is to rewrite data access objects, which is quiet straight-forward: just remove Room annotations, and implement the interface using the generated Query class:

```
class QuestDaoImpl(
        private val queries: QuestEntityQueries
) : QuestDao {

    override fun insert(qp: QuestEntity) {
        queries.insert(qp)
    }

    override fun getQuests(): List<QuestEntity> {
        return queries.getQuests().executeAsList()
    }

    <…>

}
```

Figure 22. Simplified example of data access object written using SQLDelight. Kotlin

4.4.2 Integration of the shared module

All we have to do for the Android integration is to add Gradle dependencies to library and shared module and launch the AndroidSQLiteDriver, which is a class bundled in the library. In case of iOS in is required to write custom behaviour for NativeSqliteDriver, because by default it creates a new database, if current version is 0.

4.5 How multiplatform works?

One of key features of Kotlin is native support of cross-platform projects. Kotlin/Native is not the only one target platform. It is also possible to compile to JVM and JS.

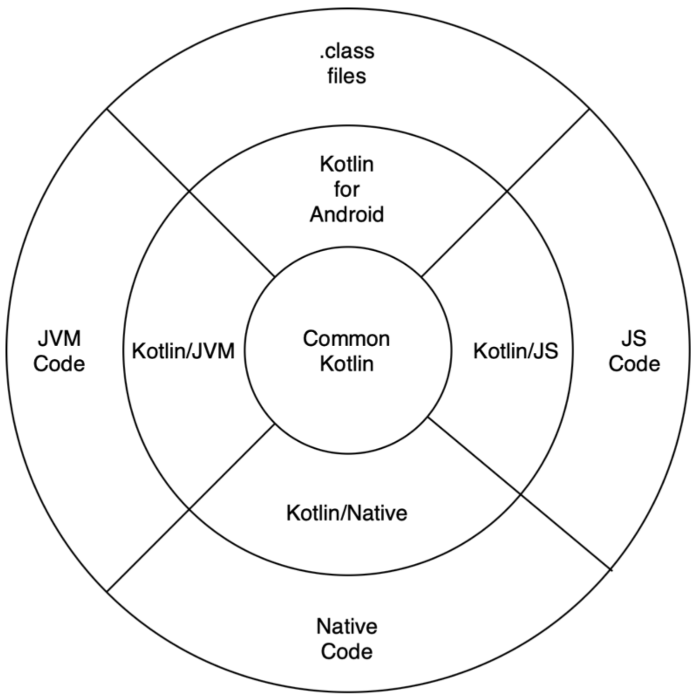
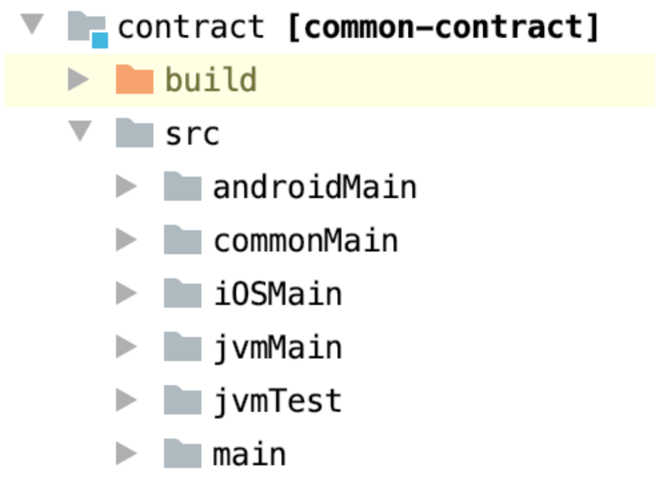


Figure 23. Compile targets of Kotlin

As shown in Figure 23, it is also possible to compile the shared codebase to JS and use in browser if needed. Kotlin for Android is usual Android module written in Kotlin. It will always have Android Manifest and access to the context and executed with Dalvik VM. Kotlin/JVM will be executed in regular JVM, instead of DVM. Could be used, for example, for unit tests. The folder structure of the module is illustrated below on the Figure 24.



```
▼ contract [common-contract]
  ▶ build
  ▼ src
    ▶ androidMain
    ▶ commonMain
    ▶ iOSMain
    ▶ jvmMain
    ▶ jvmTest
    ▶ main
```

Figure 24. Example on module structure

Illustrated module it the one that mostly consists of the interfaces (*contract* part in terminology of MVP). Where *main* contains all the *expected* classes, androidMain is usual Android module, commonMain is what will be compiled into a shared codebase. jvmMain has only helper classes for unit-tests and *actual* implementation of some classes, that have to be different from Android and iOS ones. iOSMain is also a Kotlin module that has access to Foundation namespace. In this case it contains utility classes for concurrency and memory management and some wrappers to use Kotlin Coroutines classes from Swift code. Unfortunately, IDE navigation in iOSMain modules is not working yet, due to beta status of technology.

```
class ChannelWrapper<E> {
    private val channel: Channel<E> = Channel(RENDEZVOUS)

    fun get() = channel

    @InternalCoroutinesApi
    fun send(item: E) {
        CancelableCoroutineScope(MainLoopDispatcher()).launch {
            channel.send(item)
        }
    }
}
```

Figure 25. Example of wrapper written in Kotlin to be accessed in Swift code

Here is an example of Channel wrapper class. The problem is that Kotlin global functions are inaccessible from Swift code, since then it is not possible to instantiate a new channel directly, because it does not have public constructors, only a function with this signature.

One interesting part of code fragment in Figure 25 is that we create MainLoopDispatcher which is a CoroutineDispatcher implementation.

```
@ExperimentalCoroutinesApi
@InternalCoroutinesApi
override fun scheduleResumeAfterDelay(timeMillis: Long, continuation:
CancellableContinuation<Unit>) {
    dispatch_after(dispatch_time(DISPATCH_TIME_NOW, timeMillis *
NANOSECONDS_IN_MILLISECOND), dispatch_get_main_queue()) {
        try {
            with(continuation) {
                resumeUndispatched(Unit)
            }
        } catch (err: Throwable) {
            NSLog(err.message.toString())
            throw err
        }
    }
}
```

Figure 26. Code that calls native iOS functions. Kotlin

Here is an example of one function of coroutine dispatcher implementation. It calls some iOS native functions, such as dispatch_after() and dispatch_get_main_queue(). Although, now there is no need to write custom coroutine dispatchers for iOS module, because developers had added their implementation to the library.

4.6 Expect and Actual

Kotlin has a powerful language feature to work with the platform-specific code. *Expect* class works like an interface, that could be declared in the main module, and used both in the shared code library and in the platform code. The difference with the usual interface declaration is that those *expect* classes would be replaced by *actual* platform implementations during the compilation, depending on the target platform. Below there is a simple example with a big decimal number declared in contract module.

```
expect class BigDecimal(
        value: String
) {
    fun getValue(): String

    fun toDouble(): Double
}
```

Figure 27. Expect declaration of BigDecimal class. Kotlin

In order to use it the same way on both compile targets, there are two implementations, that are wrapping native classes, NSDecimalNumber for iOS and BigDecimal on Android.

```
actual class BigDecimal actual constructor(value: String) {
    private val value: NSDecimalNumber = NSDecimalNumber(value)

    actual fun getValue(): String = value.stringValue

    actual fun toDouble(): Double {
        return value.doubleValue
    }

    fun getNSDecimalNumber() = value

    override fun toString(): String {
        return getValue()
    }
}
```

(a)

```
actual class BigDecimal actual constructor(value: String) {
    private val value: BigDecimal = BigDecimal(value)

    actual fun getValue(): String = value.toPlainString()

    actual fun toDouble(): Double {
        return value.toDouble()
    }

    override fun toString(): String {
        return getValue()
    }
}
```

(b)

Figure 28. Actual implementation of BigDecimal class on iOS (a) and on Android (b). Kotlin

Expect/Actual pattern could be a good choice in cases when developer needs to wrap such native pairs of libraries as AVFoundation/CameraX, Core ML/MLKit, LocalAuthentification/Biometric, Accounts/Room, etc, and wrap them with the same expected signature in order to use in the shared codebase.

4.7 Linking of Gradle modules

Next step is to review is how to link modules with each other. In order to make code look cleaner, all the boilerplate was moved to a separate Gradle file, which is included to all build files. Below, there is an example of linking one feature on contract and core levels.

```
apply plugin: 'com.android.library'
apply plugin: 'kotlin-multiplatform'
apply from: rootProject.file('gradle/common_android-setup.gradle')

depends([
        ':common-contract',
        ':network-contract'
])
```

(a)

```
apply plugin: 'com.android.library'
apply plugin: 'kotlin-multiplatform'
apply from: rootProject.file('gradle/common_android-setup.gradle')

depends([
        ':common-core',
        ':review-contract',
        ':home-navigation',
        ':analytics-contract',
        ':profile-contract'
])
```

(b)

Figure 29. Gradle files examples on contract level (a) and core level (b)

What is important to mention is that all modules, even those, that are platform-independent, should have Android plugin applied, because Kotlin Multiplatform requires all modules, that will be used in Android, to have a manifest file, that will have a package name.

4.8 Localization synchronization

One of important challenges was to make the resources (e.g., strings) easily synchronized between Android and iOS apps. They should be updatable, localizable and easily manageable. The application supports 8 languages and it is obvious that developers cannot update all localizations when they want to add or adjust something. Thus, it is crucial to develop an approach or tool to handle the problem.

In Android strings are represented with XML-files, where the name attribute is a key, accessible from the R file, when application context is available. Android strings support string arrays, C-style templates. Also, there is quantity strings support, which is needed for templates with plural forms, because different languages have different grammatic rules for that. For example, the plural string, shown in Figure 30, would have similar rules in English, Spanish and many other languages: But in Russian there will be an additional case for numbers that end with 2, 3 or 4.

```
<plurals name="quest_start__creations_mask">
    <item quantity="one">%1$d publication</item>
    <item quantity="other">%1$d publications</item>
</plurals>
```

(a)

```
<plurals name="quest_start__creations_mask">
    <item quantity="one">%1$d publicación</item>
    <item quantity="other">%1$d publicaciones</item>
</plurals>
```

(b)

```
<plurals name="quest_start__creations_mask">
    <item quantity="one">%1$d публикация</item>
    <item quantity="few">%1$d публикации</item>
    <item quantity="many">%1$d публикаций</item>
</plurals>
```

(c)

Figure 30. Plurals example for English (a), Spanish (b) and Russian (c) languages. XML

To make application look more natural and grammatically correct it is important to take care of these small details. In iOS strings also support templates, but arrays and plurals support are limited. But the most critical difference is that resources are accessed by a string key. And if there is no value available, they **key** would be returned. This can possibly lead to unpleasant consequences: if there is no localized value, user will see the key, while it will be better to show English value.

The solution was a custom Gradle task that takes all the strings.xml files from Android resources, converts them to iOS Localizable format and adds them to the project. If translation is missed in some of the languages, it adds the English version to the localization file to avoid the problem with returning the key as a default value.

4.9 Development process

Initially regular git-flow approach was used. But we encountered several problems with that. When something is changed on Android side, affecting the core, those changes will also affect iOS: sometimes it will fail to compile until contracts are fulfilled on iOS side, or, it can lead to unexpected behaviour. And when iOS developers see these problems, they have to solve them out-of-context.

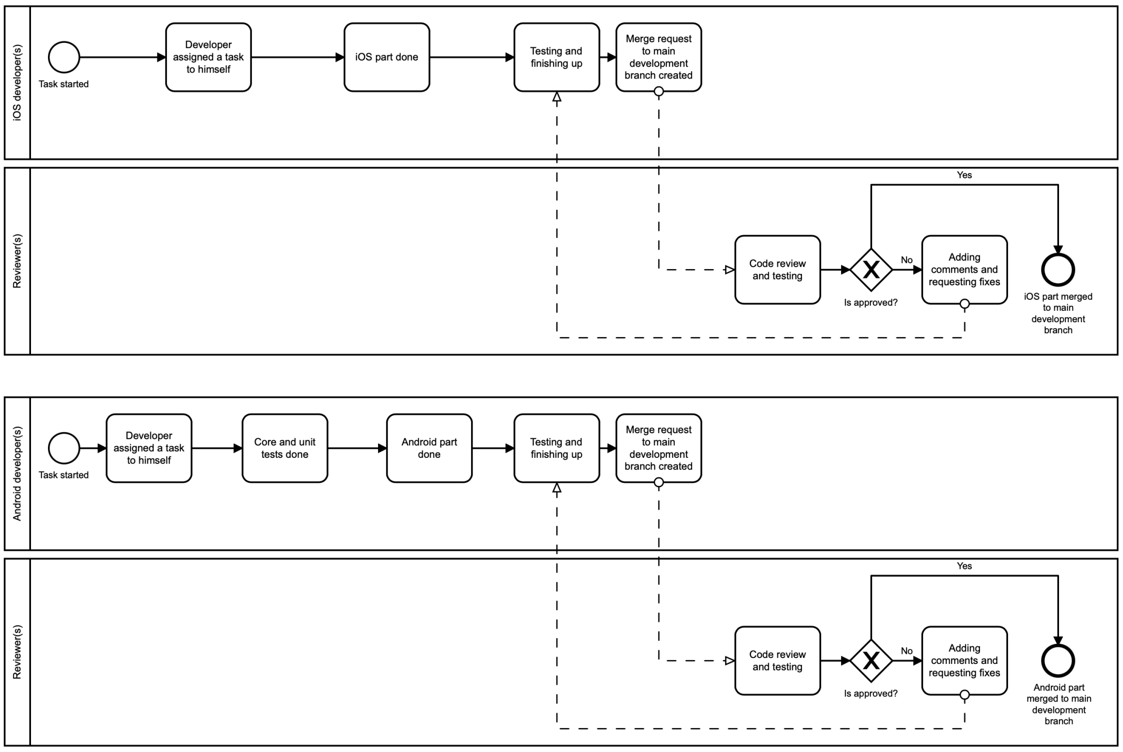

iOS developer(s)
Task started
Developer assigned a task to himself
iOS part done
Testing and finishing up
Merge request to main development branch created
Reviewer(s)
Yes
Code review and testing
No
Is approved?
Adding comments and requesting fixes
iOS part merged to main development branch
Android developer(s)
Task started
Developer assigned a task to himself
Core and unit tests done
Android part done
Testing and finishing up
Merge request to main development branch created
Reviewer(s)
Yes
Code review and testing
No
Is approved?
Adding comments and requesting fixes
Android part merged to main development branch

Figure 31. Initial task development process in BPMN

It was causing a lot of problems, so next step was to use approach with 2 separate development branches, that work the same as in git-flow: when a new task for iOS is started, developer branches off the iOS development branch, and the same process for Android. Though, to keep codebase up to date, we synced those branches each month, having full attention on that process. That should have helped to avoid having unexpected bugs and not to waste time solving problems out of context.

After several months of using this approach, we found out, that regular branch syncing is still very nervous process and we tried another way to handle it: single main development branch, and pull requests containing both iOS and Android code. This one was the most time effective and convenient, but required developers to know a lot about both platforms and Kotlin/Native specificity, so, the final approach is not recommended to be acquired if team members are not yet experienced for it.

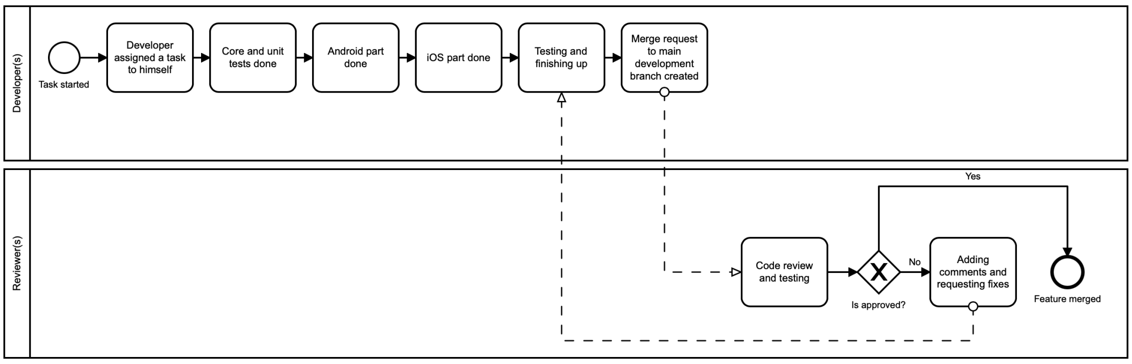


Figure 32. Final task development process in BPMN

4.10 Debugging core code

Since we are recompiling the core each time into a library, it begs the question: how should we debug the application during the runtime? Is it possible to put breakpoints in Kotlin code while running Swift application from XCode? Or the only solution is to print strings to the log? Fortunately, if core code is compiled using LLVM, it is possible to debug it with LLDB. But it is not very convenient to use LLDB as is. There is a plugin for XCode that integrates this tool to IDE's debugging GUI. And it becomes possible to carry out the debugging in a familiar way: with breakpoints, stepping into or out, reading variable values. It requires adding Kotlin files as sources, and for that there is additional plugin available: *Kotlin XCode Sync.* It is not very hard to add all this to project, but the plugins are not very advanced, they are still in active development.

4.11 Memory leaks

Memory leaks are the huge problem that happens when program prevents deallocation of objects in memory, which are not used anymore. Mostly, it happens in static typed languages that rely on garbage collectors [19]. Thus, the next important thing to consider is the way iOS and Android manage memory allocations. There is a huge difference in approaches, and what works perfectly in Android could easily leak in iOS. In Android's garbage collector mechanism, it is enough that unused object is inaccessible from root object even if it has cross references from other unused objects to be removed, while in ARC object that have strong references on itself will be kept in memory. In other words, retain cycles that are not a problem in Android, could make iOS application very memory-consuming, and that problem needs special attention. As it was mentioned in study [19], the initial step to detect memory leaks is to make a heap dump.

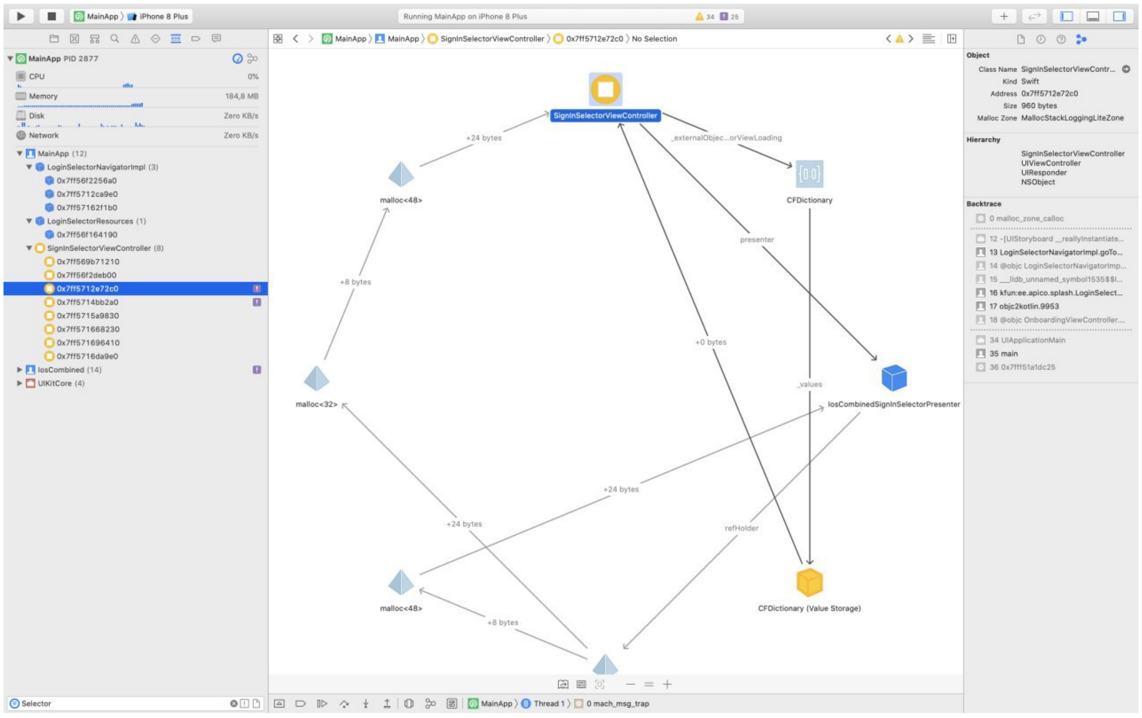

MainApp 〉 iPhone 8 Plus
Running MainApp on iPhone 8 Plus
MainApp 〉 MainApp 〉 SignInSelectorViewController 〉 0x7ff5712e72c0 〉 No Selection
MainApp PID 2877
CPU 0%
Memory 184,8 MB
Disk Zero KB/s
Network Zero KB/s
MainApp (12)
LoginSelectorNavigatorImpl (3)
0x7ff56f2256a0
0x7ff5712ca9e0
0x7ff57162f1b0
LoginSelectorResources (1)
0x7ff56f164190
SignInSelectorViewController (8)
0x7ff569b71210
0x7ff56f2deb00
0x7ff5712e72c0
0x7ff5714bb2a0
0x7ff5715a9830
0x7ff571668230
0x7ff571696410
0x7ff5716da9e0
IosCombined (14)
UIKitCore (4)
Selector
SignInSelectorViewController
+24 bytes
_externalObjec...orViewLoading
malloc<48>
CFDictionary
presenter
+8 bytes
+0 bytes
_values
malloc<32>
IosCombinedSignInSelectorPresenter
+24 bytes
+24 bytes
refHolder
malloc<48>
CFDictionary (Value Storage)
+8 bytes
MainApp 〉 Thread 1 〉 0 mach_msg_trap
Object
Class Name SignInSelectorViewContr...
Kind Swift
Address 0x7ff5712e72c0
Size 960 bytes
Malloc Zone MallocStackLoggingLiteZone
Hierarchy
SignInSelectorViewController
UIViewController
UIResponder
NSObject
Backtrace
0 malloc_zone_calloc
12 -[UIStoryboard __reallyInstantiate...
13 LoginSelectorNavigatorImpl.goTo...
14 @objc LoginSelectorNavigatorImp...
15 ___lldb_unnamed_symbol1535$$l...
16 kfun:ee.apico.splash.LoginSelect...
17 objc2kotlin.9953
18 @objc OnboardingViewController....
34 UIApplicationMain
35 main
36 0x7fff51a1dc25

Figure 33. Heap dump of application made in XCode

Screenshot in the Figure 33 was made during investigation the memory leaks problem. In illustrated situation, the view controller is being leaked 8 times, the exact number of times the screen was launched during the use session. Due to visual representation, it is possible to easily notice that view controller and presenter are holding cross-references to each other.

As it was mentioned before, Android and iOS have different ways to manage memory, and using code where object are perfectly deallocated in Android could cause huge problems on iOS. For example, this is what was happening with memory consumption on iOS application:

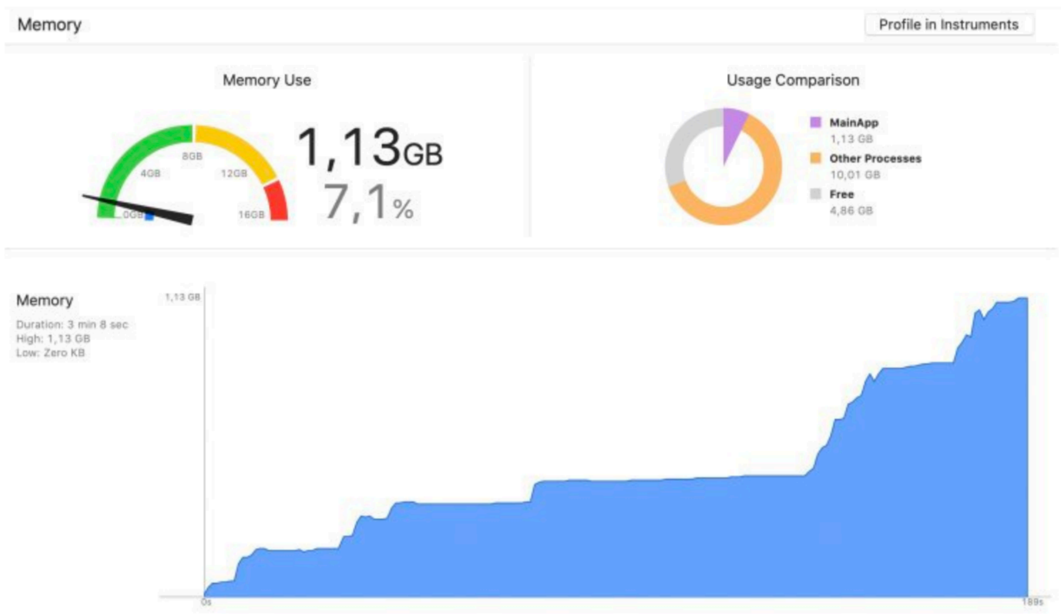


Figure 34. Memory consumption of iOS application before using weak references

Automatic reference counting will always think that cross-referring objects are needed by application, and thus, all objects that are referenced inside of them, would also leak. For example, if a view controller with a map view was leaked, it would cause a loss of 150-200 megabytes of RAM. When it happens several times, memory consumption can easily grow up to 1 Gb in just several minutes, which is unacceptable. The solution is to add weak references. In case of this architecture, it will look like this:

There will be *expect* class called Reference, which will have *actual* implementations on both platforms. Android version will just simply return wrapped object, but iOS version will return weak referenced version. This class will be used to wrap all the view controllers, so that they will be released as soon as their presenters will be not used.

```
class DiscountPresenter(
        override val viewReference: Reference<DiscountContract.View>,
        private val commonDbRepository: CommonContract.CommonDbRepository,
        private val dialogPresenter: DialogContract.Presenter,
        <…>
        coroutineScope: CoroutineScope
) : DiscountContract.Presenter, CoroutineScope by coroutineScope
```

Figure 35. Shortened example of presenter class dependencies. Kotlin

```
bind() from singleton { Reference(view) }
```

Figure 36. Kodein component binding. Kotlin

After having done this refactoring to all the screens, memory consumption started to look like this:

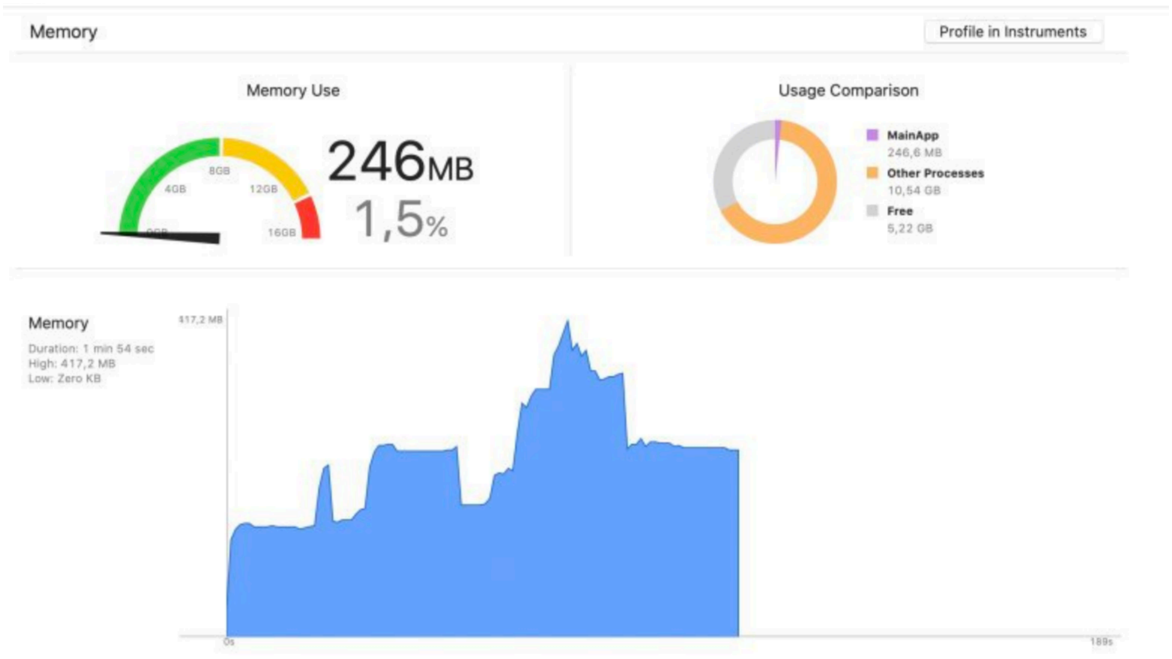


Figure 37. Memory consumption after adding weak references

As we can see in Figure 37, the amount of used memory grows up when screen is opened, and goes down when closed.

4.12 Risk and safety analysis

The framework should provide the same level of code protection as the native tools in order to prevent reverse engineering and leakage of private keys. In case of Android the shared library would be compiled the same way, the regular libraries would do. It means that regular obfuscation tools, such as ProGuard would work. Yet it means, that leaving private keys in core code as string constants would make them accessible during decomplication. Thereby, the secret keys should be stored in Gradle configuration files on Android level, and if they are needed in the core modules, developer has to provide an abstraction with getters, that would invoke context-aware calls on Android side. And on iOS the same values should be stored in .plist files. Expect/Actual feature fits the requirement the best.

4.13 Computational performance overhead

One of main problems with cross platform solutions is losing performance. Fortunately, when it comes to Android, we don't have any computational overhead: it works as a native multi module application. In order to test it on iOS a small sample application was created with basic architecture (dependency injection + network + database + MVP) and measured it with and without Kotlin/Native shared library and compared to build time of the project being reviewed in this study.

One of feasible problems with Kotlin/Native is slow build time on iOS: the mean clean build time of native layer of the sample project is 10 seconds, yet it takes 250 seconds of additional time on average only to build a shared library. Although, the average build time of the reviewed application, which has 50+ Gradle modules, is close to the sample application: 31 seconds for native layer and 280 seconds for shared codebase. Build machine: MacBook Pro 2018, Core i7, 16 Gb RAM.

However, everything is fine, when it comes to incremental builds. If nothing was changed in the core, it just checks that there were no changes and skips the build phase. If some modules are changed since last build, the changed modules will rebuild, plus the ones, that depend on them. So, if common-contract module is changed, then almost all other modules will be rebuilt, because they have a dependency on it.

In our experience, in most of the cases when a developer is working on iOS part in this framework, core is already done and there is no need to edit it, so the long build problem was not that feasible.

Another important thing is RAM consumption overhead. While the clean iOS version of the sample app was consuming 11 Mb of RAM, it took 17 Mb at peak with shared library connected.

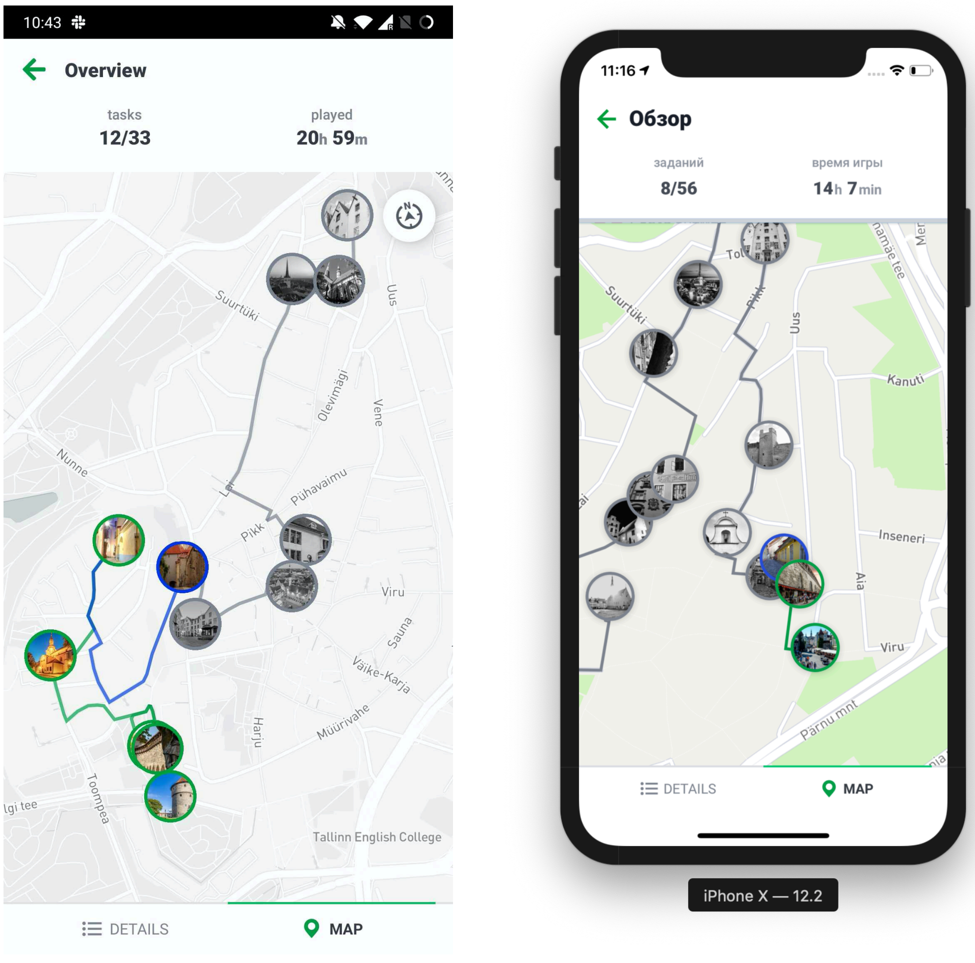
5 Summary

5.1 Organizational problems

The main problem that appeared during the process was that it is not very easy for iOS team to work with core code written in Kotlin. When, in theory, they do not need to work with it a lot, in practice each time they were working on a new feature platform implementation, it was necessary to check through the business logic.

5.2 Development performance boost

Even when the approach is quite complex and could seem tangled, it gives a real boost when it comes to logic-intensive parts of application. For example, it took 2 weeks of development to make a screen with map of whole quest for Android, shown in the Figure 38, but having all the core logic implemented it took only **4 days** to make the same screen on iOS, and could have been done even faster, if it was done be someone with a greater iOS development experience.



(a) (b)

Figure 38. Screenshot of Quest map screen on Android (a) and iOS (b)

5.3 When to use this approach?

Since the moment of making the research, choosing a technology to use for the migration until now, frameworks had developed a lot, and it is necessary to give an updated opinion. Flutter had become a mature technology and augmented a huge variety of third-party libraries, also the Dart language had evolved a lot: now it has null-safety and greater static code analysis tools. On the other hand, Kotlin/Native stopped being an experimental technology, and now it is being adopted by many development teams. Other technologies, such as Xamarin, Adobe Air are gradually losing the popularity. Personally, I have developed several real-world applications in Flutter, and I can say that the technology is really great and promising, and I would choose it in case of limited resources, while Kotlin/Native can give much smoother user experience with native UI, but it would fit to the teams, that have great experience in Android and iOS development and have more recourses. Also, it is important to notice, that there had been done a huge step towards using MVVM: now there are some libraries written to use with Kotlin/Native, that encapsulate a lot of native logic and provide possibility to use the advantages of the pattern without having to write wrappers. Also, there is an ongoing development of unifying SwiftUI and Jetpack Compose technologies using Kotlin/Native, that are native UI frameworks being developed by Apple and Google to replace current ones.

References

[1] M. E. Joorabchi, A. M. and P. K. , “Real challenges in mobile app development.,” in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013.

[2] V. Brossier, Developing Android Application with Adobe Air, 2011.

[3] W. Wu, “React Native vs Flutter, cross-platform mobile application frameworks,” Metropolia University of Applied Sciences, 2018.

[4] J. Fayzullaev, “Native-like Cross-Platform Mobile Development Multi-OS Engine & Kotlin Native vs Flutter,” South-Eastern Finland University of Applied Sciences, 2018.

[5] S. Borisenkova, Developing Sopima Cross-platform Mobile Application With Xamarin, Haaga-Helia University of Applied Sciences, 2015.

[6] J. B. Lorenzo, “Fast Prototypes with Flutter + Kotlin/Native,” 2018. [Online]. Available: https://tech.olx.com/fast-prototypes-with-flutter-kotlin-native-d7ce5cfeb5f1. [Accessed 11 06 2020].

[7] A. Sullivan, “Examining performance differences between Native, Flutter, and React Native mobile development.,” [Online]. Available: https://robots.thoughtbot.com/examining-performance-differences-between-native-flutter-and-react-native-mobile-development. [Accessed 12 04 2020].

[8] V. B. Andreas Lelli, “Evaluating Application Scenarios with React Native,” Uppsala University, 2016.

[9] A. Bizzotto, “How fast is Flutter? I built a stopwatch app to find out.,” [Online]. Available: https://medium.freecodecamp.org/how-fast-is-flutter-i-built-a-stopwatch-app-to-find-out-9956fa0e40bd .

[10] S. Jiang, “Comparison of Native, Cross-Platform and Hyper Mobile Development Tools Approaches for iOS and Android Mobile Applications,” University of Goethernburg, 2016.

[11] S. Xanthopoulos, “A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications,,” in *Proceedings of the 6th Balkan Conference in Informatics*, 2013.

[12] T. Lou, “A comparison of Android Native App Architecture MVC, MVP and MVVM,” Aalto University.

[13] R. Nunkesser, “Choosing a Global Architecture for Mobile Applications,” Hamm-Lippstadt University of Applied Sciences.

[14] M. Potel, “MVP: Model-View-Presenter the taligent programming model for C++ and Java,” Taligent Inc., 1996.

[15] M. L. Karina Sokolova, “Android Passive MVC: a Novel Architecture Model for Android Application Development,” University of Technology of Troyes.

[16] M. A. Mariam Aljamea, “MMVMi: A Validation Model for MVC and MVVM Design Patterns in iOS Applications,” IAENG International Journal of Computer Science.

[17] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017.

[18] A. Bresolin, “Kotlin coroutines vs RxJava: an initial performance test,” [Online]. Available: https://proandroiddev.com/kotlin-coroutines-vs-rxjava-an-initial-performance-test-68160cfc6723.

[19] G. Novark, “Efficiently and precisely locating memory leaks and bloat,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] Microsoft, “Xamarin.Android / Concepts and Internals / Limitations,” [Online]. Available: https://docs.microsoft.com/en-us/xamarin/android/internals/limitations.

[21] Microsoft, “Understanding the Xamarin Mobile Platform,” [Online]. Available: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform. [Accessed 20 04 2020].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I, Roman Ismagilov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Migrating an existing Android application to a cross-platform”, supervised by Juhan-Peep Ernits and Oleg Petšjonkin.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

10.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.