Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis

Study of impact and flexibilities provided by sharing code logic in code architecture layers written using KMM for Android & iOS mobile applications

Mayur Pathak

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Eng. Mayur Pathak

Systems Engineering and Informatics
Informatics

Thesis title

Study of impact and flexibilities provided by sharing code logic in code architecture layers written using KMM for android & iOS mobile applications.

Objectives of thesis

As the mobile application development is constantly evolving, and there are always new frameworks floating around to capture the market share. But the Kotlin Muyltiplateform Mobile (KMM) has a different approach to offer when it comes to developing cross platform mobile applications. Out of this curiosity, the partial objectives of this thesis are:

1. Code an application by writing shared code at possible layers of MVVM architecture for android & iOS using KMM and also a separate native codebase for each platform.

2. Compare the complexities in native vs shared code at networking, data storage and other utilities.

3. Conclude results about which parts of the architecture and application functionalities can be cost effective and beneficial from layered code sharing & maintenance perspective.

Methodology

The idea to demonstrate the objective of this thesis will be to program an application for conscious time management. The initial direction for literature journey will be going through official documentation of android and ios applications development and then code purely native apps for both platforms. Then after coding the same app idea using the concept of KMM. On completion of the implementation part there will be analysis of the amount of code shared among both platforms (android & iOS) and also track improvements & pitfalls using various benchmarks.

The proposed extent of the thesis

60-80 stran

Keywords

Android, iOS, Kotlin Multiplateform Mobile, Cross-Plateform Mobile Development

Recommended information sources

Apple Inc. Apple Developer Documentation
Google LLC. Documentation for app developers
JetBrains s.r.o. Kotlin Multiplatform Mobile

Expected date of thesis defence

2021/22 SS – FEM

The Diploma Thesis Supervisor

Ing. Martin Pelikán, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 23. 11. 2021	Electronic approval: 25. 11. 2021
Ing. Martin Pelikán, Ph.D.	**Ing. Martin Pelikán, Ph.D.**
Head of department	Dean

Prague on 31. 03. 2022

Declaration

I declare that I have worked on my diploma thesis titled "Study of impact and flexibilities provided by sharing code logic in code architecture layers written using KMM for Android & iOS mobile applications" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the diploma thesis, I declare that the thesis does not break any copyrights.

In Prague on date of submission 31.03.2022 ____________________________

Acknowledgement

I would like to thank Ing. Martin Pelikán, Ph.D. for his support and guidance while working on this thesis. Additionally, I would like to thank my family and close friends for their encouragement, support and belief in me.

Study of impact and flexibilities provided by sharing code logic in code architecture layers written using KMM for Android & iOS mobile applications

Abstract

In the world of mobile application development there have been countless hours of discussions to justify 'Write Once Run Anywhere' taking into account many cross-platform frameworks. However, each framework had their own drawbacks along with advantages. E.g Some of them had a long learning curve or could not produce the native experience. Instead of saving time and cost, the extra work has been reported by the users. Which removes them from first class for developing cross platform mobile applications.

Kotlin Multiplatform Mobile (KMM) is considered to be most native for developing mobile applications for Android and iOS platform by sharing a single codebase. Unlike other platforms, it does not disconnect application development from native frameworks. To explore this path further, this thesis will focus on developing three applications. Two applications for Android & iOS without sharing any code. Third one will focus on writing code by using KMM at possible levels of MVVM architecture to be shared across both platforms.

After finishing the application, the thesis focuses on finding performance differences between apps and also tries to justify which are the most flexible parts of the code, where code sharing actually proves to be well fit and reduces actual work.

Keywords: Android, iOS, Kotlin Multiplatform Mobile, Cross-Platform Mobile Development

Studie dopadu a flexibility poskytnuté sdílením logiky kódu ve vrstvách architektury kódu napsaných pomocí KMM pro mobilní aplikace pro Android a iOS

Abstraktní

Ve světě vývoje mobilních aplikací se uskutečnilo nespočet hodin diskuzí, které měly ospravedlnit 'Write Once Run Anywhere' s ohledem na mnoho multiplatformních rámců. Každý rámec měl však své nevýhody spolu s výhodami. Někteří z nich měli dlouhou křivku učení nebo nedokázali vytvořit původní zkušenost. Namísto úspory času a nákladů byla uživateli hlášena práce navíc. Což je odstraňuje z první třídy pro vývoj mobilních aplikací pro různé platformy.

Kotlin Multiplatform Mobile (KMM) je považován za nejvíce nativní pro vývoj mobilních aplikací pro platformy Android a iOS díky sdílení jediné kódové základny. Na rozdíl od jiných platforem neodpojuje vývoj aplikací od nativních frameworků. Abychom tuto cestu dále prozkoumali, zaměří se tato práce na vývoj tří aplikací. Dvě aplikace pro Android a iOS bez sdílení kódu. Třetí se zaměří na psaní kódu pomocí KMM na možných úrovních architektury MVVM, které budou sdíleny na obou platformách.

Po dokončení aplikace se práce zaměřuje na nalezení výkonnostních rozdílů mezi aplikacemi a také se snaží zdůvodnit, které jsou nejflexibilnější části kódu, kde se sdílení kódu skutečně ukazuje jako vhodné a snižuje skutečnou práci.

Klíčová slova: Android, iOS, Kotlin Multiplatform Mobile, Cross-Platform Mobile Development

Table of Contents

List of figures

1. Introduction

Mobile application development industry is one of the fastest growing industries in the world. Along with that mobile apps are now a very crucial part of our day-to-day task schedule. Mobile applications can save several hours of human involvement in a task by using just a few clicks. But this rapid growth in the number of applications in various sectors is not only driven by their demands but also by new tools and technologies to develop them. These tools and technologies power development and developers by providing modern plus top notch techniques and flexibilities to improve the app performance and thus more adaptation. The remainder of this section has described information about highly adapted mobile platforms and about types of app development solutions that have helped them grow.

1.1 Mobile platforms

There are various mobile operating systems worldwide being used by different people in different geographic locations. We might have heard some popular OS names such as Android, iOS, Symbian, Windows, Samsung, BlackBerry, etc. But, whenever a topic pops up about mobile devices running on which platforms, the most heard names are Android and iOS. Android is the most widely used OS and has surpassed the spread of iOS for a long time. The market share of Android in mobile operating systems is 70.01% and of iOS is 29.24%. So, both of them jointly enjoy a market share of 99.25% worldwide (1). The context of this thesis will mostly include Android and iOS.

1.2 Development frameworks

The rapid evolution of the above platforms also led to an easier way of developing applications for those platforms. Which resulted in the existence of many cross platform frameworks to develop apps which support write once and run anywhere mechanism. The popular platforms include PhoneGap, Ionic, React native, Xamarin, etc. But there is always detachment from the native platform while developing apps using these platforms. Kotlin Multiplatform Mobile follows a different approach than all of these, which will be used in this thesis ahead.

2. Objectives and Methodology

2.1 Objectives

As the mobile application development is constantly evolving, and there are always new frameworks floating around to capture the market share. But the Kotlin Multiplatform Mobile (KMM) has a different approach to offer when it comes to developing cross platform mobile applications. Out of this curiosity, the partial objectives of this thesis are:

a. Code an application by writing shared code at possible layers of MVVM architecture for android & iOS using KMM and a separate native codebase for each platform.

b. Compare the complexities in native vs shared code at networking, data storage and other utilities.

c. Conclude results about which parts of the architecture and application functionalities can be cost effective and beneficial from layered code sharing & maintenance perspective.

2.2 Methodology

The idea to demonstrate the objective of this thesis will be to program an application for conscious time management. The initial direction for literature journey will be going through official documentation of android and iOS applications development and then code purely native apps for both platforms. Then after coding the same app idea using the concept of KMM. On completion of the implementation part there will be analysis of the amount of code shared among both platforms (android & iOS) and track improvements & pitfalls using various benchmarks.

3. Literature Review

3.1 Development Frameworks

This section will walk through types of development frameworks which can be incorporated based on the requirement of writing code once/multiple time & run code on single/multiple platforms.

3.1.1 Cross-Platform vs Multi-platform

Cross platform development is the preferred way of development when the expectation is to write code once and build for many platforms from that single codebase (2). Then the built code runs on different targeted platforms.

Multi-platform does not always mean write once and build for many platforms but instead code run on many platforms (2). From the readers' perspectives these approaches might look similar, but from a development perspective it has differences.

E.g., When the mobile apps (for Android and iOS) are developed using tools like Xamarin or Flutter, it will produce both the apps using a single code base, which can be referred to Cross Platform development. Apps written using Kotlin Multiplatform Mobile SDK will have a common module shared for both iOS and android, plus an individual module for iOS and an individual module for android.

Multiplatform essentially differs from Cross platform in the way they provide optional sharing. Most of cross platform wants developers to build the app within their technologies only. E.g., React native says build an app with JavaScript and then you are in their ecosystem and same with Xamarin and Flutter. But it’s a big decision because it does not end there, because you have to rewrite a bunch of stuffs or have to rewrite bridge logic, running in its own Virtual machines etc. It’s a big deal decision. Because the way organizations think of the product and the way development works has big gap. Multiplatform reduces this gap by optional sharing.

3.1.2 Cross-Platform frameworks

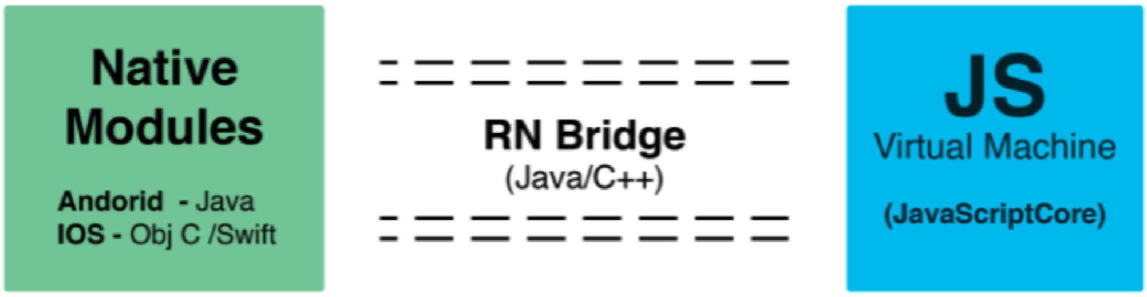
This section will consider some popular Cross Platform frameworks and describe them briefly before getting into the Multi-platform framework.

3.1.2.1 React Native

React Native is a JavaScript based framework and it takes advantage of React.js which is used on the web, but instead React Native is used for mobile application development. React Native provides components using which developers can write user interfaces in JavaScript and these components are nothing but mapped corresponding native views when applied to native mobile frameworks. When a build is produced for Android, it is bundled with the JavaScriptCore engine which powers safari in iOS. So, in case of iOS build it is not required to be bundled, and that's the reason the .ipa app size in iOS build is smaller as compared to Android (3).

In the big picture, React Native works in three parts: Native Code modules, JavaScript Virtual Machine & React Native Bridge as shown in Figure 1. When a React Native application is launched in a mobile environment, the main thread is launched (that's where application runs in Android and iOS), which in turn launches JavaScript thread where all React Native JavaScript code runs. Both threads communicate asynchronously using JavaScript bridge (3).

Figure 1 React Native architecture



Source: (3)

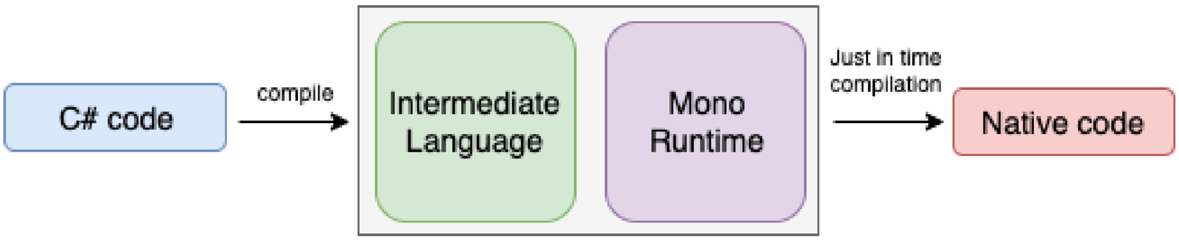
In addition to this, one more thread is spawned which is shadow thread, that's where the UI tree of React is calculated and mapped into native layout hierarchy. The real lag here is the asynchronous communication between JS thread and native UI thread, which is being improved by the new architecture rollout called Fabric(not official name yet). Also the RN bridge is being replaced with a more robust communication pattern (3).

3.1.2.2 Xamarin

Xamarin extends the .NET framework to provide cross platform solutions to develop apps for android, iOS, windows macOS etc. Xamarin uses C# to write shared code for developing applications. As Xamarin extends the .NET framework, it delegates some tasks like easy interoperability with the underlying platform, memory management, garbage collection, etc. A Xamarin app is usually built using Xamarin.Android (Previously Mono for Android) or Xamarin.iOS (Previously MonoTouch) which are built on top of Mono. Both provide access to the .NET base class libraries and to respective native api libraries. Mono is Common Language Runtime (CLR), an open-source implementation of .NET framework based on ECMA standards for C# and the CLR. In the context of android & iOS, Xamarin works a little differently (4).

When an application is built using Xamarin.Android, Xamarin C# compiler compiles the C# code into intermediate language (IL) and also emits Mono Runtime CLR with the application. When an android application is started, Mono Runtime is also loaded into the memory and compiles IL code into Native code by just in time compilation as shown in Figure 2 (4).

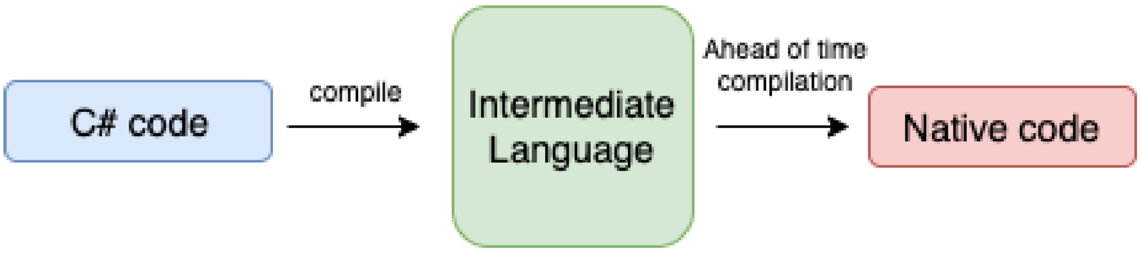
Figure 2 Xamarin.Android architecture



Source: Figure created according to (4)

In case of iOS, Xamarin C# compiler compiles the C# code into intermediate Language but does not emit MONO Runtime CLR as it does in case of android as Apple does not allow jitting. So, it uses an apple compiler (of course requires an apple machine) to transform IL code to Native code that iOS can understand. Thus, in case of Xamarin.iOS it uses ahead of time compilation instead of jitting as shown in Figure 3 (4).

Figure 3 Xamarin.iOS architecture



Source: Figure created according to (4)

3.1.2.3 Flutter

Flutter is an open-source cross platform framework consisting of an SDK to help with tools to develop and compile an application and also a rich set of reusable UI widgets library. The development in Flutter is done using Dart programming language. So, anyone starting with Flutter will face the learning path of Dart.

Flutter helps designing UI faster by providing its own widget library. Instead of wrapping native UI elements in a Flutter specific components, Flutter provides its own set of libraries of widgets which provides similar visual appearance on native platforms. That's why flutter has no overhead of converting its UI widgets to native ones (5).

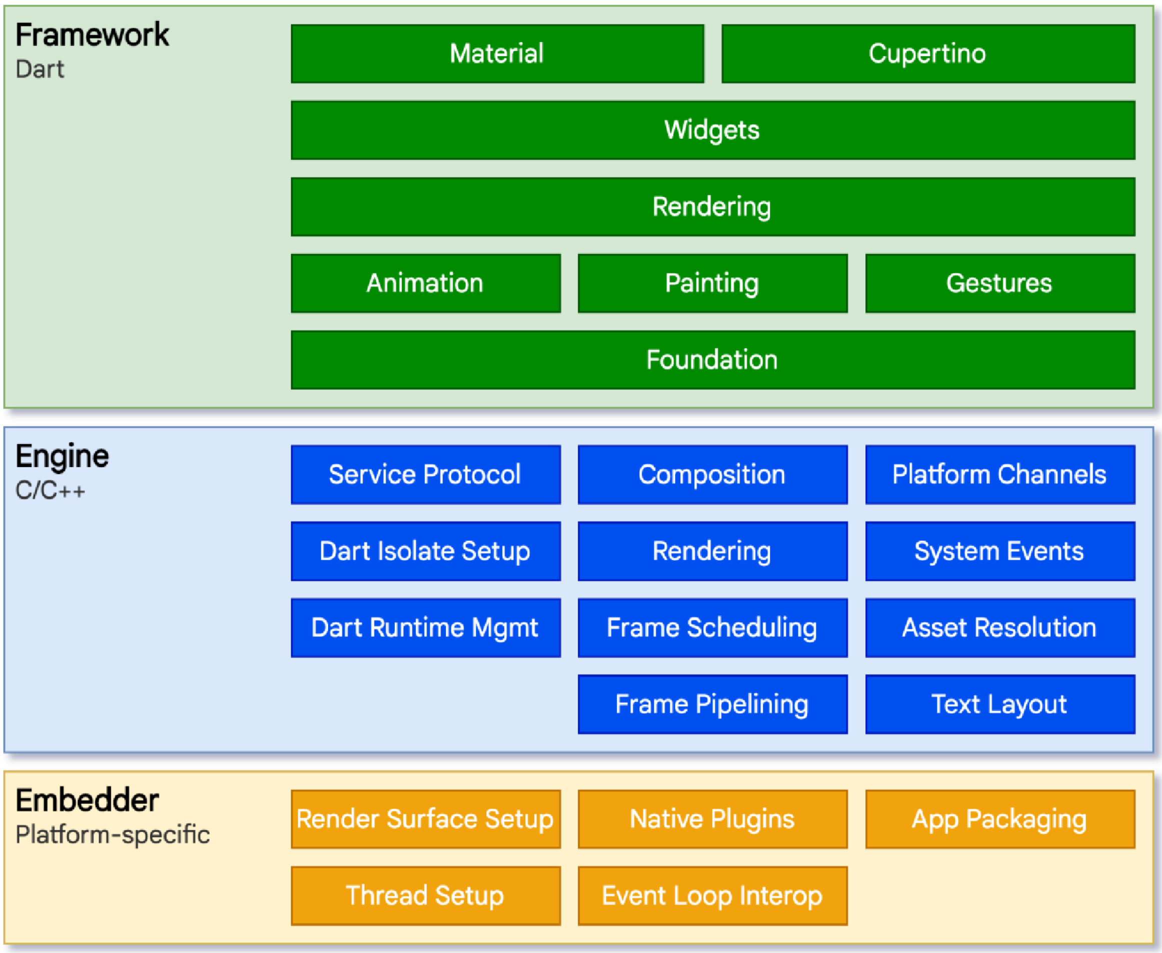
Flutter is little more straight forward compared to other cross platform frameworks when it comes to architecture. Flutter has three-layer architecture: Framework, Engine & Embedder.

a. Framework consists of platform, layout, foundational libraries. It provides a modern reactive framework. Developers interact with device capabilities through this layer.

b. Flutter Engine provides low level implementation of core API, such as graphics, layouts, file, network and Dart runtime & compile toolchain. It manages translation of this processes to embedder e.g., rendering pixels on the screen.

c. Embedder is platform specific and mostly written in a language suitable for targeted platform. e.g., Java/C++ for Android. Objective C/C++ for iOS. It provides an entry point for the app and synchronizes communication with the target OS for various services like rendering, threading, etc. (5)

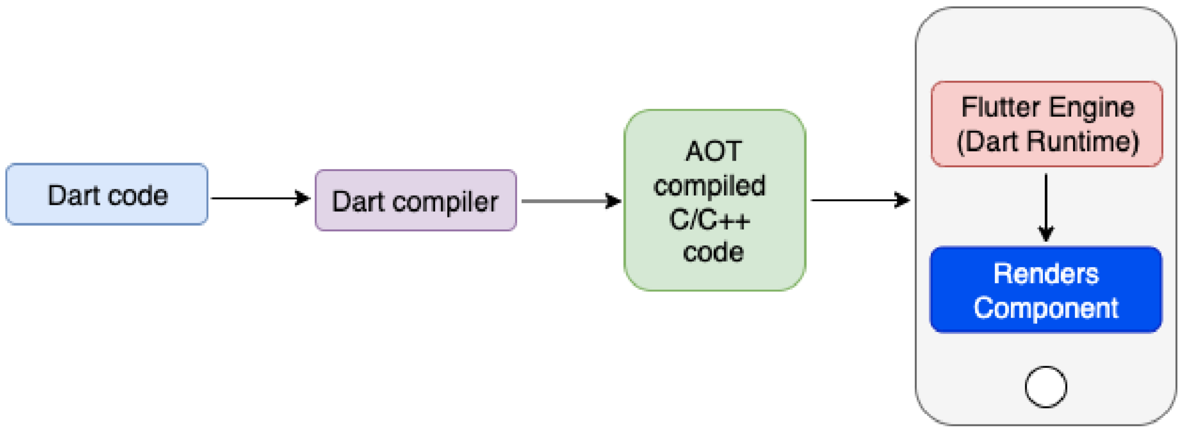
Figure 4 Flutter architecture



Source: (5; 6)

As Dart can be just-in-time compiled or Ahead-of-time compiled. So, with the feature of hot reload, it uses jitting. While at release time, it produces ahead of time compiled app code for native library code. This code is then embedded into .apk for Android &. ipa for iOS. As soon as app opens up, Flutter embedder starts interacting initializes Flutter app. After that Dart runtime is responsible for executing the native code (7). A basic representation of how flutter runs on device is as shown below.

Figure 5 Rendering Flutter app



Source: Figure created according to (7)

3.1.3 Multi-platform frameworks

In the native mobile world essentially, native mobile developers think of things as you either doing native platform and its language or you are doing everything else which is cross platform and that's not a precise conversation and its disservice to the various pieces of tech out there.

3.1.3.1 Kotlin Multiplatform

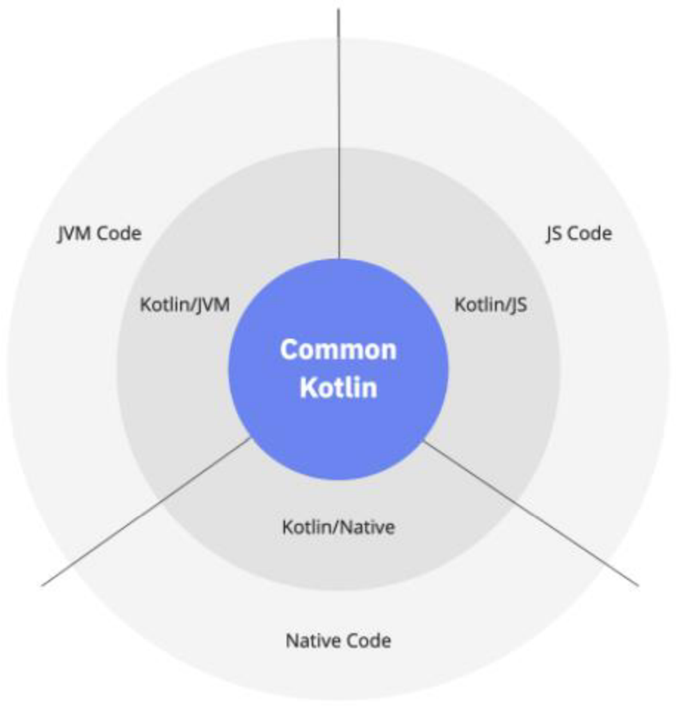
The essential definition which differentiates Kotlin Multiplatform is, it is optional natively integrated opensource code sharing platform based on the popular modern language Kotlin that facilitates non-UI logic and is available on many platforms.

Multiplatform is a compile-time construct, at runtime multi-platform does not mean anything. Multiplatform is a way to represent shared code but have it compiled down and deployed in multiple platforms specifically and the three main targets in Kotlin Multiplatform are JVM, JS, and Native (8) as shown in Figure 6.

These all have again specific targets. In the JVM again you can have multiple versions of java and Android specifics. Kotlin JS targets on browser side and can help you share code logic that runs on both server and client-side browser. Kotlin Native avoids virtual machines on target machines and instead produces native binaries by compiling Kotlin code. And that's what the target was. That means it helps compilation for platforms where virtual machines are not allowed, e.g., iOS. Kotlin Native supports macOS, iOS, Linux, windows, and many more as shown in Figure 7.

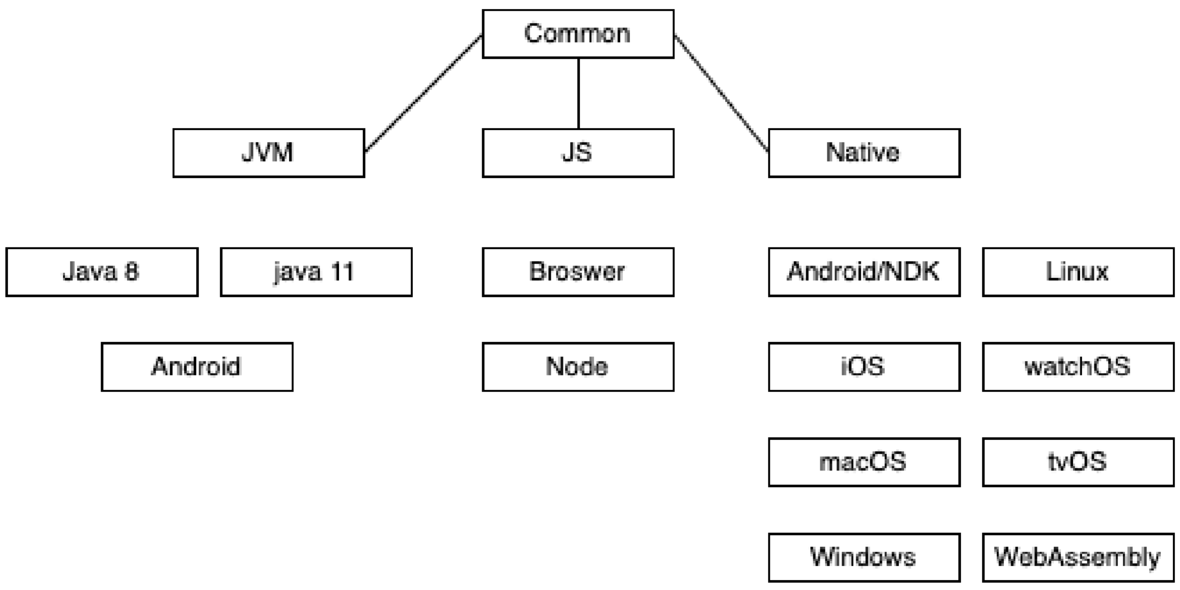
Here it shows common code that defines essentially the business logic of the application and then there is hierarchy of source sets that will provide platform specific stuffs that a developer needs to implement some of those things.

Figure 6 Kotlin Multiplatform



Source: (8)

Figure 7 Kotlin Multiplatform targets



Source: Figure created according to (9)

To be more specific it includes the language, core libraries, and basic tools. Code written in this module works everywhere on all platforms. So, if you are making a library, you could have multiple layers of code but in reality, for making an app you usually just have common and then the platform specific stuffs. Going down the hierarchy in Figure 6 developer can use platform specific version of Kotlin (Kotlin/JVM, Kotlin/JS, Kotlin/Native) which includes extensions to the Kotlin language, and platform specific tools, to interact with platform native specific facilities.

3.1.3.2 Kotlin Multiplatform Mobile

In the world of highly accepted mobile OS, that is android and iOS, Kotlin Multiplatform Mobile has brought an intelligent solution that offers different perspective on how cross platform solutions should be perceived. Kotlin Multiplatform Mobile (KMM) is an SDK for cross platform mobile development. At the core as its name suggests derives the basis from above context. It uses the multiplatform capabilities of Kotlin and includes various tools and features designed to make the end-to-end experience of building mobile platform applications efficient.

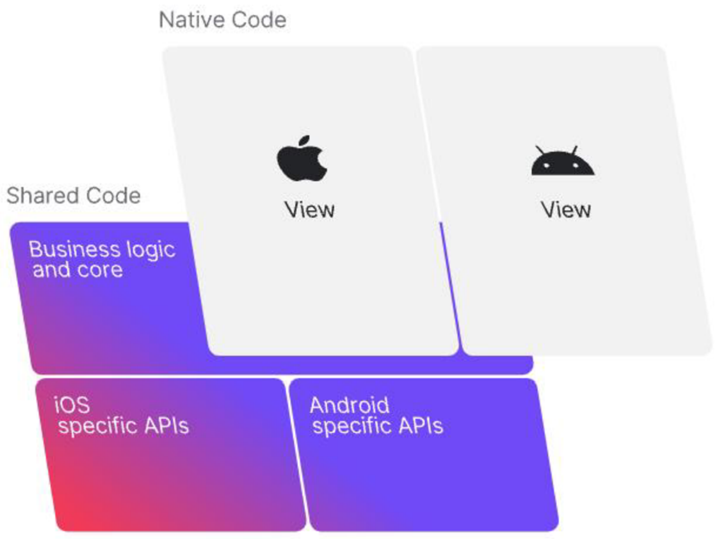
KMM allows to use a single codebase for the business logic of iOS and android apps. A developer only needs to write platform specific code where it's necessary. An example case can be to implement UI or working with platform-specific apis. Whenever it is required to use iOS or android features, the expect/actual pattern provided by the SDK can be used to implement platform specific code with shared logic in the KMM module. KMM can be seamlessly integrated into existing mobile projects as well. The Figure 8 shows how does it look when considering code layers.

Compilation & Execution of code:

- Android:
  When it comes to android, the part is pretty straight forward as android is already having native language as Kotlin & Java and has virtual machine which is capable of running JVM byte code. Thus, here Kotlin/JVM compiler frontend converts the Kotlin shared code into Intermediate Representation (IR) code, which is compiled down to JVM

bytecode with Kotlin/JVM backend (10) and then the process goes same as native code on android.

Figure 8 KMM code layers

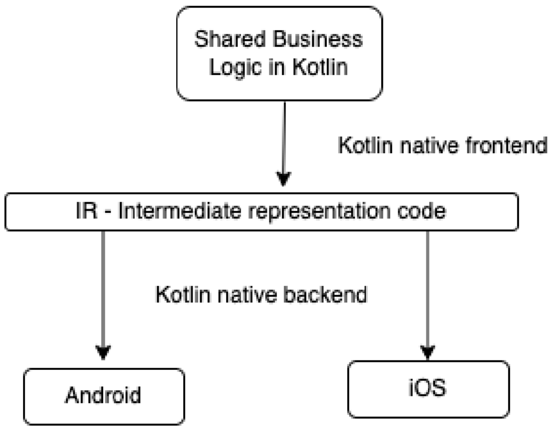


Source: (11)

- iOS:

  When it comes to iOS, it does not allow any virtual machine. Here, Kotlin code is converted into Intermediate Representation (IR) code by Kotlin/JVM compiler frontend and then it is converted into iOS native machine executable code by Kotlin/Native backend (12).

Figure 9 KMM compilation


Shared Business Logic in Kotlin
Kotlin native frontend
IR - Intermediate representation code
Kotlin native backend
Android
iOS

Source: Figure created according to (10)

3.2 Doubts while adopting KMM

It has always been a point to be noted stuff when shifting the usual development paradigm. The developer is mostly aware of the current architecture principles, UI stuffs, testing etc. But when it's time to change the way development currently works and adopting new technology, requires a shift from traditional thinking. The new paradigm might force people to move from the comfort zone. So, it's necessary to address some doubts.

- Usually, the first and foremost fear is sometimes from the market and sometimes from self-experience about the complexities introduced by using cross-platform solutions. One can find so many unresolved questions and deadlocks where cross-platform frameworks have restrictions for interaction with native technology features in iOS and Android. KMM provides this flexibility by expect/actual pattern where once can easily go for native solutions when not comfortable with the shared code (13).

- Next, the most crucial one for any application to succeed can be the issue of performance. Whenever there is a middle layer, which might be not much matured can cause performance headache. Not always or not in every situation, but chances are there. KMM ensures this not to happen by not only using any bridge or virtual machines, but by providing different output formats of shared compiled code. Which makes sure that there is no additional runtime overhead, and performance is like native apps (13).

- In the today's era of products, there can be many applications which have been running since long, or due to short time fast features, people might not have cared about upgrade in technologies. So moving to a new implementation of whole big codebase might be a hard task. KMM solves this issue by giving flexibility of inter operating with existing legacy codebase. It does not require to move entire code to KMM at once. But by gradually connecting and with little modification developer can start with KMM on both the platforms, iOS and android (13).

- iOS had more complexities as it does not provide cross-platform installations which modifies features or functionality of the app. But that is not the case with KMM, as it its code is compiled to native binaries and bundled as regular iOS framework into the app, ensuring no dynamic code execution (13).

- The last but not least is often developing team does not believe the worth trying concept until someone has showed the successful results. There are giant organizations who have already implemented KMM in their large projects successfully, even though its in alpha stage. There is a curated list of case studies on KotlinLang.org portal (14). It does not end there, but someone has to come up and try various facilities for iOS and android to get more insights on it (13).

3.3 Android Platform

Android is an open software platform for mobile development. It was intended to be a complete stack that includes everything from operating system through middleware & up through applications. This section will walk through overview of architecture of android platform and some key principles underlying its design.

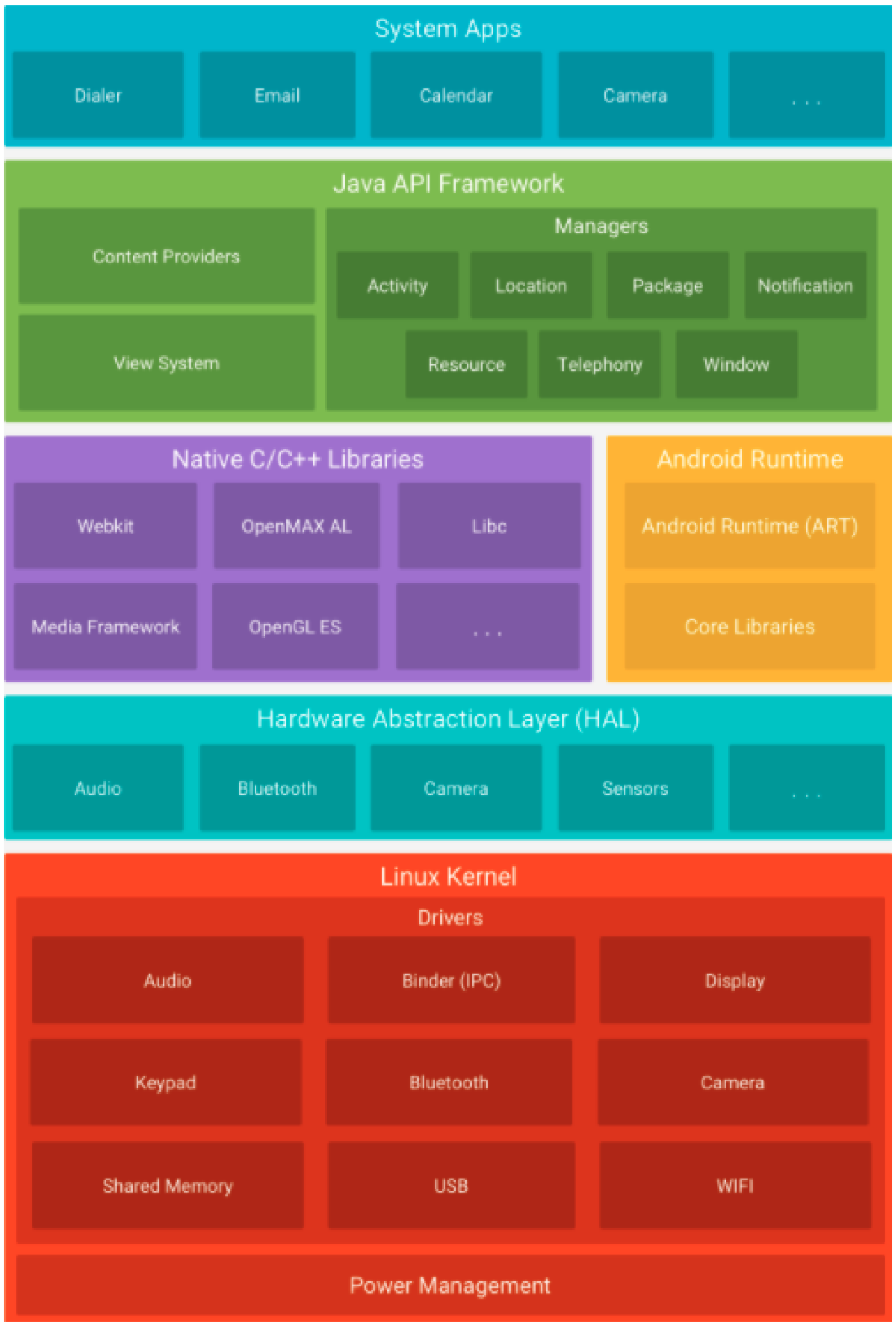
3.3.1 Platform architecture

- Visualizing from Figure 10 Android architecture stack the bottom layer states that, it is based on Linux kernel, which is in continuous updates as per new android OS releases, which can be found here (15). It allows the manufacturers to develop hardware drivers based on their own flexibilities to interact with threading and memory management (16).

- Linux kernel has a Hardware abstraction layer (HAL). It provides a proven driver model & in a lot of cases existing drivers. It also provides memory management, process management, a security model, networking a lot of core operating system infrastructures that are robust and have been proven over time. It provides high level access to hardware features (16).

- The next level up is native libraries, where everything is written in C/C++. It's at this level where lot of the core power of the android platform comes from. E.g., Surface Manager, which is responsible for composing different drawings surfaces onto the screen. So, it's the surface manager that's responsible for making different windows that are owned by different applications running in different processes and all drawing at different times and making sure the pixels end up on the screen when they are supposed to (16).

Figure 10 Android architecture stack



Source: (16)

- The next is Android Runtime. The Android Runtime was designed specifically for android to meet the needs of running in an embedded environment, where there is limited battery, limited memory, limited CPU. It runs DEX files, which are byte codes that are the results of converting at build time, .class and .jar files. These DEX files are much more efficient byte codes that can run very well on small processors. They use memory very efficiently (16).

- The next is core libraries titles as Java API Framework which are written in Java Programming Language. The core library contains all of the collection classes, utilities, IO, etc., covering all the building blocks to ensure creating a modular android app (16).

- Moving up there is application framework, which are all written in Java or now a days Kotlin. These includes system apps and also developer written apps which provides from basic to advance functionalities for the user by encapsulating all the complexities discussed above (16).

3.3.2 Understanding Android Applications

Each mobile application development ecosystem has their main languages prioritized for the development. Similarly, android applications used to be developed using Java. But since the introduction of Kotlin in android ecosystem, it’s recommended to follow Kotlin first approach. Other than that, the development can also be done using Java, and C++. Below are displayed building blocks of android application development (6).

3.3.2.1 Android Application Package (apk) & Android App Bundle (aab)

These are publishing formats of an android applications. Once an android application is coded and ready to be published, the resources and code are compiled by android SDK tools into files with suffix .apk or .aab. An android package contains all the files required to install an application at runtime. Android App Bundle file contains same thing but also additional metadata. It can not directly be installed on device like apk file. AAB files are submitted to google play store and it generates optimized apk file for different devices and released apk contains only files required by that device. AAB formats are more better when the application is for targeted device facilities ore have resources integrated for region specific targeted audience (6).

3.3.2.2 Application life on device

Android operating system is structured to be multi-user, in the sense that each application is a different user and has its own isolated environment or virtual machine. Each application runs in its own Linux process and has protected & limited access to files on the device based on the permissions. As a consequence of limited memory, any process can turn to death if required by OS to recover memory. Every application can obtain the permission for accessing device capabilities, by requesting it explicitly from the user (6).

3.3.2.3 Components and fundamentals of Android application

There are different types of components in the application which serves as an entry point to the application. They all differ in their purpose for providing service, in terms of their construction and destruction and lifespan.

- Activities

  An activity in android application provides a visual interface to interact with the application. It is the first thing to be visible when an application starts. An activity is created by extending the Activity class from android SDK, which provides basic functionalities needed or to make a developer written class to be acted as android activity. By combining multiple activities, a flow of a particular functionalities can be developed. But usually, these activities consume more resources than it actually requires. And that's why android framework introduced something called fragment which acts as part of an activity and thus providing efficient memory usage by using multiple fragments instead of multiple activities. That's how single activity architecture works in android (6).

- Services

  A service is something which does not usually have a UI and is used mostly for long running background operations which cannot directly be performed on the main visible screen context. But a service can be bound to any component which shows UI updates of what is going on inside service e.g., by using notification or by binding service to the activity. Similar to activity, Service is a base class from android SDK, by extending which any class can be declared as service class (6).

- Broadcast receivers

  A broadcast receiver is a way by using which any part of the app which is registered as receiver can listen for events, it is registered in the system for. Android system can wake that listener up if its not already running in the application process when sending a broadcast. E.g., a listener who is registered for doing something when battery is low. Android system will automatically notify this listener when battery will be considered low after falling below certain level. Same as other component a broadcast receiver can be created by extending BroadcastReceiver class from android SDK (6) (17).

- Content Providers

  As its name suggests, it provides managed access to the shared content or content which is private to app, which is stored on any persistent location that an app can access. Through the content provider, other apps can modify or query data if content provider allows it. A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions. An extraordinary part of the Android framework configuration is that any application can begin another application's part. For instance, assuming you need the client to catch a photograph with the camera, there's most likely another application that does that and your application can utilize it as opposed to fostering an action to catch a photograph yourself. You don't have to consolidate or even connection to the code from the camera application. All things being equal, you can basically begin the movement in the camera application that catches a photograph. At the point when complete, the photograph is even gotten back to your application so you can utilize it. To the client, maybe the camera is really a piece of your application (6).

- Intents

  Intent helps interacting with individual components to another one at runtime. You can consider them the couriers that carries information with them about the action you want to perform and delivers it to another component which might be out of your application. Intent can be used to start another activity from one activity. It can be used to start a service. Or by providing a system specific function announcement action or custom action a broadcast can be delivered to a listener with optional data (6).

There are two types of intents: Explicit & Implicit. Implicit intents are used when it is unknown that which component is going to handle the request. So, only an action Uri is specified in it. E.g., Intent(Intent.ACTION_VIEW). This action is handled by the OS and based on extra data provided, it will invoke appropriate component if present. Explicit intent are used, when the target component which will handle the action to perform is already known. For example while opening another activity already known: Intent(context, KnownActivity.class) (6).

- Manifest File

  This file contains list of components that an Android system must know before it allows an application to start. The app must declare all the components in this file. In addition to this, a manifest file is used to declare permissions which are required at runtime in order to access restricted features which might require explicit user permissions. It also contains information about minimum and maximum android API level versions an app is going to support. In addition to this, manifest file can provide place to declare extra metadata, which are required in order to access features provided by an external library. For instance, the google map SDK used to show map in an android app requires to declare it configuration along with privileged API key which authenticates your app while accessing map SDK features (6).

3.3.3 Android project structure

As every framework project requires a project structure to generalize logical separation between various types of files. Similarly, an android application project is divided into various folders with dedicated file types to store inside. Below are the description of files and folders which are mostly edited in order to build the app by developer.

3.3.3.1 Modules

Android projects are at high level nothing but a set of various types of modules and their configuration. Each module can be independent or might contain some dependency on other modules. These modules contains source files and settings required to build those modules. A project can have many modules serving different purpose. These modules can be independently build, test and debug. Mostly a new module is created when it is required to develop it as an independent library or to logically isolate a feature or when the functionality required is for specific purpose like only for wearables, tablets etc. There are various module types (6).

- An App module aggregates app's source code, files and resources. It contains all the module level settings in a build.gradle file. Whenever a new project is created the default name of the first module is app. Though you can change it to any other name. An app module can be a phone and tablet module, wear OS module, Android TV module and glass module. Each of this type differs in a way, they provide default files require to start the project for the mentioned device type (6).

- Feature module is the one that can be used to provide on-demand dynamic feature delivery using google play feature (6).

- Library module provides a separation of the reusable code which can independently be used at other places in the same project or other project. There are two types of library modules: Android Library & Java Library.

  Android Library module is the one which contains all file types supported in an android project. The build result format for Android Library module is Android Archive (AAR) which can independently be added in any android project.

  Java Library module is the one which contains all java file types. The build result format for Java Library module is Java Archive (JAR) which can independently be embedded in any android or java project (6).

3.3.3.2 Important files & folders based on grouping

Android studio, an IDE provides many views to display grouping and separation of files. E.g., Android, Project, Project files, Packages etc. By not going into the details of each view, general grouping where each file is separated based on file type is discussed below:

- Manifest

  This contains manifest.xml file. According to its actual location in file system this file resides in the third level folder from the root level project folder. E.g., module, src, main. But in other views, it might be seen residing in a manifest folder (6).

- Java

  This folder resides in module, src, main hierarchy. It contains all the source code including Java and Kotlin files. It divides the code in the sub packages (6).

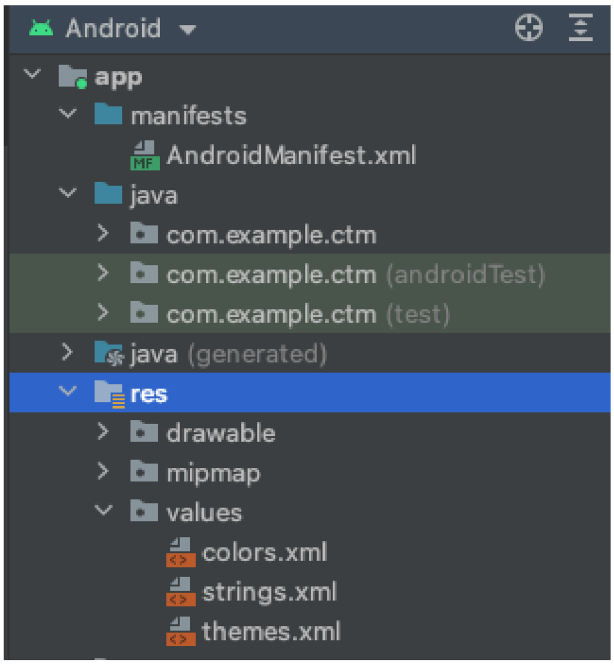
- res

  As its name suggests, it contains all the file which can be considered as no-Java or no-Kotlin resources. It has sub folders based on types of resource. There are various drawable folders which contains .XML files that provide custom graphics stuffs based on device orientation, device density, etc. It also contains folders for app wide used colors, strings and app theme. Also, some raw files like animation files which are supported at runtime on device and unknown to Android Studio IDE by default are present inside subfolder here (6).

- test & androidTest

  The test folder contains unit test which are concentrated on testing a smaller part of code mostly. The androidTest folder contains longer test like UI and instrumentation tests.

Figure 11 Android application project structure



Source: Created according to (6)

3.3.4 Android studio – Development IDE

Choosing the right tools for the android application development is more important than ever. As the android platform got matured, so have the developer's needs. When the community has a feature rich and complete development environment, making the app is much more easier. This was the driving decision behind the creation of Android Studio though eclipse was there. Definitely, eclipse was an innovation closer to better performance. But Android Studio is far more suitable when it comes to fast paced android application development.

- Android studio is officially recommended IDE for android application development by Google. It is developed based on IntelliJ Idea. Android Studio provides feature which were hard to expect or use in eclipse (18).
- It provides state of the art dependency management by using Gradle build system with more android specific capabilities such as producing different flavors of same application based on build flavors and variants (18).
- The in-built emulator provides an easy way to run android application without using a physical device (18). Though when it comes to low memory configuration pc/laptops, using physical device is far better. In some cases even pushing live changes on the preview device is easier without reinstalling entire application.
- Android studio provides version control integration in a project, thus interacting with any version control system is relatively easy for a newbie (18).
- Easy division between actual code and test cases, makes it clean to view at project structure and understand clearly (18).
- Also some of the google API functionalities have built-in support for easy integration into the application code (18).
- It has more advanced support for smart code completion than eclipse and code suggestions are even more closer (18).
- It provides performance profilers which gives you visually understandable view of how the app memory usage is increase or decreased, how CPU usage is fluctuated, and most important memory leaks in various scenarios which are crucial to take care of while making an application (18).

3.3.5 Android design framework evolution to Jetpack compose

Whether a normal person realizes or not but design can be beautiful and simultaneously functional. But most of the time we don't realize it until it goes wrong. E.g. in an app to manage the smart home, if a user can't find appropriate action which provides feedback of the action, the technology can reduce the actual value of the product.

Until recent times, the design in android was used to be done in XML files and other than that everything which used to link this design to business logic used to be written in Java/Kotlin. The issue with this imperative approach was that every simple component declared in XML e.g., checkbox had internally managed their own state of active/inactive or data they visually used to reflect on the screen. And the same data was replicated in the business logic. This is like managing two truths and not only memory overhead but also violating single source of truth (19).

To co-operate with the demanding user experience and the speed with which current products are evolving the Google introduced a declarative UI toolkit jetpack compose by collaborating efforts with JetBrains. Which basically focuses on stateless UI component. And if something needs to be displayed by the UI component in compose, one must pass data and state to it, externally. Thus, maintaining single source of truth. This is the major advantage over XML based approach. And if we look at the other major technologies, like iOS development or web development, they all use the same approach as Jetpack compose uses. No doubt that XML is not going away in the near future, because a vast majority of android design codebase is still written in XML and compose has recently entered in the stable state in Jan 2022. As the organizations always have the trader that whether it is spending finance for changing code and improving it or spending finance to implement more functionality in the product that actually users have something from that and bring direct money (19).

Less code or clean code has always been a priority when it comes to develop any software. There is 10x less lines of code in jetpack compose than it required to write in XML based imperative approach. E.g., a simple list in imperative approach would cost you ideally a total of 4 snippets distributed in Java/Kotlin and XML (19).

- Create a list component in XML.
- Create a child component which will reflect in every row of a list.
- Third - connect data to each child structure using adapter.
- Forth combine this adapter to main list component created in first step.

While the same code in compose can be ideally re-written as:

- Just create composable function which combines supplied data to child structure inside it.

Which will barely make you write 15-20 lines of code in the single language. This might be little biased but here we are considering an ideal situation and again the code to show this scenario is not in the scope of this topic for comparison.

3.4 iOS Platform

Apple iOS is a closed source mobile operating system developed by Apple inc. that runs on Apple production mobile devices like iPhone, iPad, and iPod. It is the second most operating system adapted market after android. iOS is based on the Mac OS X for desktop and laptop computers. The iOS developer kit provides tools that allow for iOS app development (20).

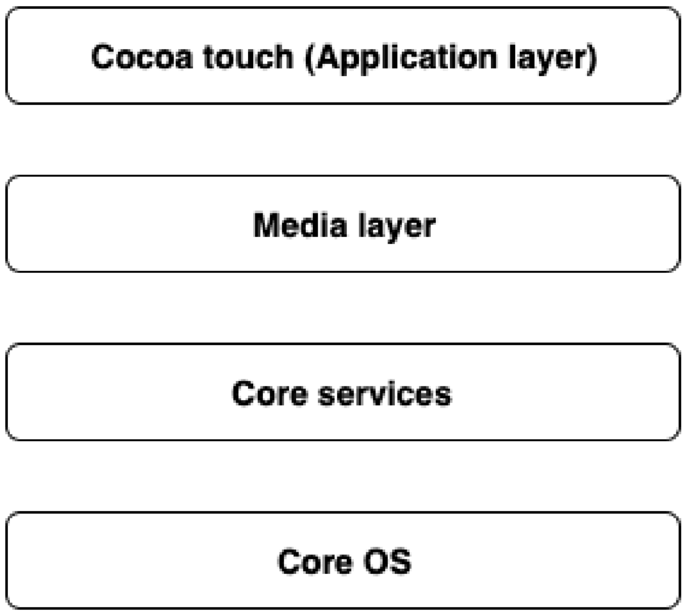
3.4.1 Platform architecture

Like any other system architecture where most of the complexities are hidden at bottom and on top of that usually the simpler interfaces are exposed to work with, iOS follows a layered architecture. It contains intermediate layers between the applications and the hardware. So, they do not communicate directly. The lower layer in iOS provides the basic device services and the higher layer provides the interface and sophisticated graphics. These interfaces are called as frameworks in iOS. Frameworks are nothing but a set of code and/or files that intend to expose a feature and are like jar or modules or libraries.

There are four layers in the architecture of iOS. At the uppermost level, the iOS works an intermediary between the underlying hardware and the apps developers create. Applications do not communicate directly to the hardware, instead they communicate with each other by using the system interfaces. These interfaces allow the developers to write the applications that works constantly on capable devices. Below are discussed these four layers shown in Figure 12 iOS architecture with brief details about them.

- The bottom most Core layer is the most critical layer on which other layers relies upon. This layer is responsible for interacting with the hardware features and core device capabilities. This layer consists of frameworks like Bluetooth, Security services, External accessories, Accelerate and Local authorization which can be used by the external application by using exposed interfaces. It manages security by kernel framework and low-grade UNIX interfaces with which applications can't interact directly (21).

Figure 12 iOS architecture



Source: Created according to (22)

- The Core Services layer is responsible core device services that are not directly related to the UI but are used by native iOS users. It provides variety of frameworks which helps create features needed to make a fully features smartphone. The Address Book, Cloud Kit, Core data, Core Foundation, Core location, Core motion, Health Kit, Home Kit, and many more are essentials of this layer (22).

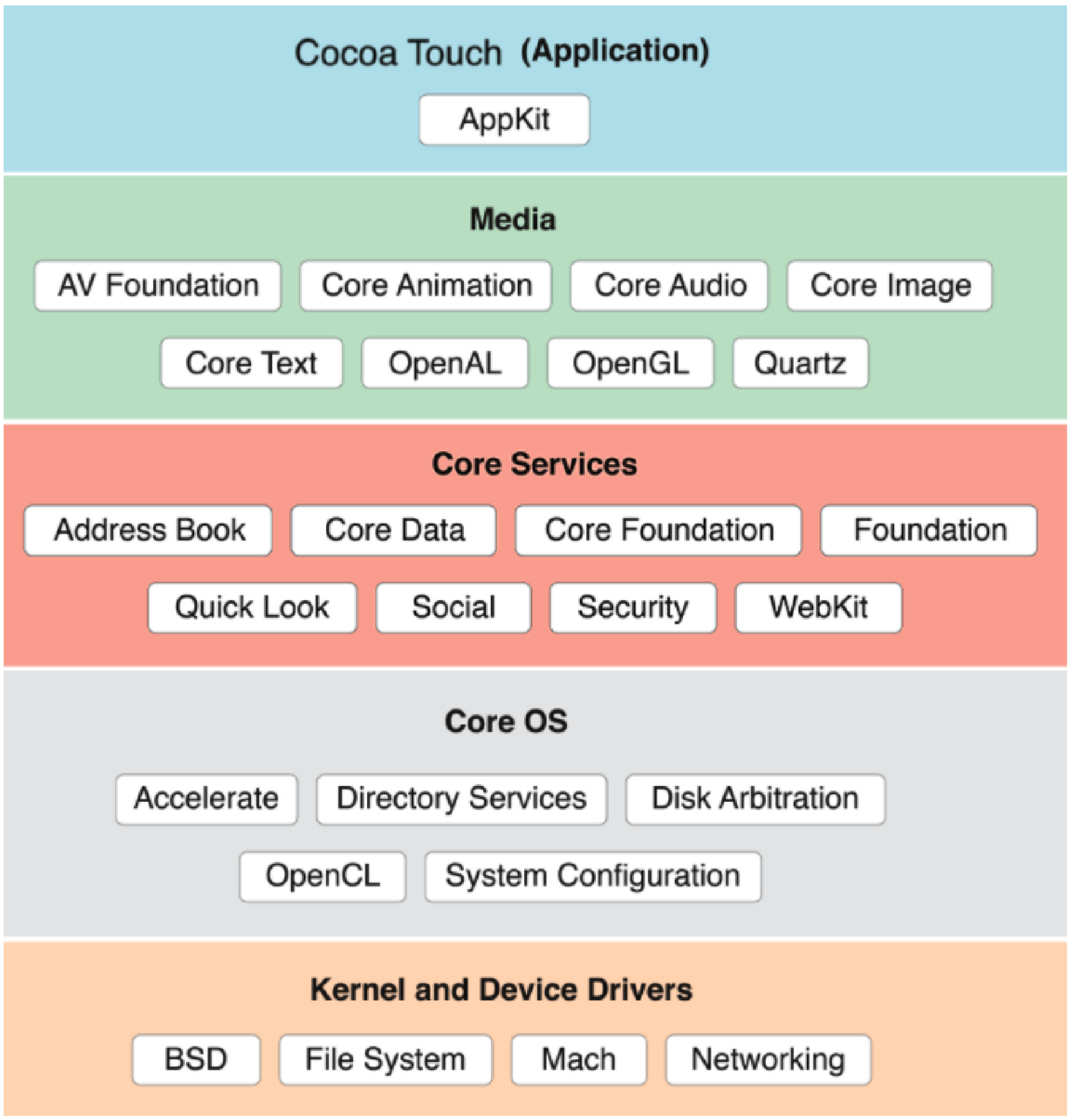
- The Media layer is responsible for creating a dynamic immersive experience for the user by providing audio, video and graphics components. There is a range of frameworks like Core Animations, OpenGL ES, GLKit, AVKit, ULKit, etc., which drives the growth of immerging technology implementation in the iOS device (21).

- The Cocoa touch layer is actually responsible of how we interact with the device. The notifications, multi-tasking, touch inputs, motion events are all optimized by this layer.

It supports all these functionalities by providing frameworks like EventKit, GameKit, MapKit, PushKit etc. (21).

An expanded iOS stack is as shown below in Figure 13 iOS detailed architecture

Figure 13 iOS detailed architecture



Source: (23)

3.4.2 Understanding iOS Applications

Earlier Objective-C used to be the dominant language in the iOS ecosystem as a proffered mean to develop applications. But it remained largely unchanged since 1980s and was lacking features of modern language. To deal with this, Apple introduced Swift in 2014 and open sourced it in 2015, which gave it momentum for the growth. Swift is powerful, intuitive, and highly English like readable programming language. Swift is now the recommended language to develop applications in Apple ecosystem (24) (25).

3.4.2.1 iOS App Store Package (.ipa)

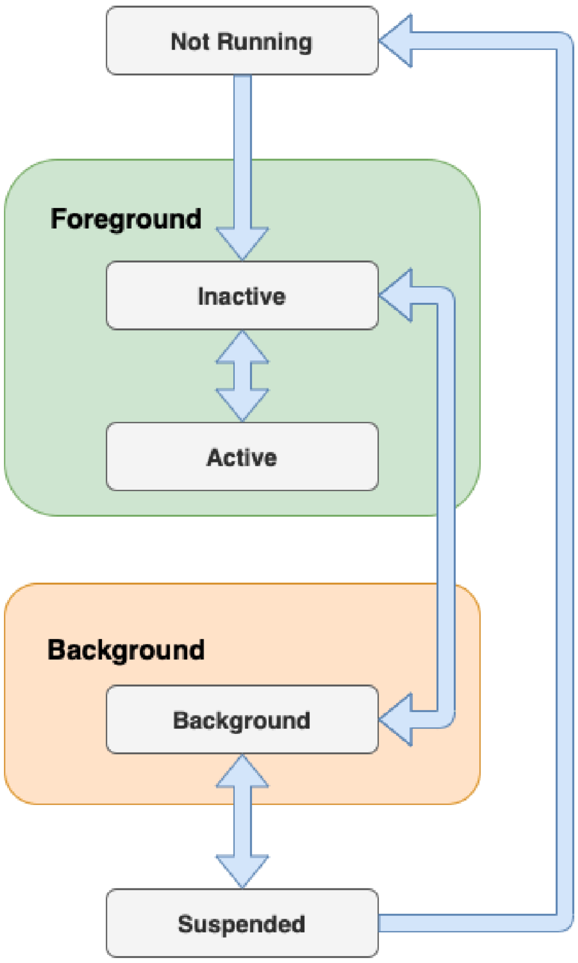
An .ipa is the format used to package iOS applications and then distributed to end user by Apple App store. These files are like containers like a zip file that holds different pieces of data that make up an iPhone app. Each .ipa file includes a binary and can only be installed on an iOS or ARM-based MacOS device. .ipa extension can be uncompressed by changing the extension to .zip and they contain compiled code.

3.4.2.2 Application life on device

iOS provides a sandbox to each application thus a restricted place from where it can not access files stored by other application. This mechanism is designed to prevent applications from modifying and using data stored by another applications. Each application will have a unique home directory to store its file which is randomly assigned when the pp is installed on device. If an application needs to use information stored by other application, it can do so by only using services provides by operating system. To perform a privileged operation a UNIX system might require root access from the entity who wants to do so. iOS solves this problem to avoid providing root access by entitlements which are key value pairs and signed in to an application, which allow the application authentication to perform privilege operation at runtime. These entitlements are assigned at application signing time, so they are unmodifiable (26).

When an application is opened up, it can process user events and user can interact with it, it is considered in an active state. Before this state there is inactive state, which means application is entering the foreground state but cannot process any event. The application goes into background state when e.g. user presses minimize button. In background state, application will execute the ongoing operation, but if any operation is not running, application will go into suspended state. In suspended state application is still in memory, but if operating system runs out of memory, suspended apps more prone to get killed first. Before application is launched or an application is terminated by user or system, it goes into Not Running state. All these states can be visually interpreted as below (27).

Figure 14 iOS application lifecycle



Source: (27)

3.4.2.3 Components and Fundamentals of iOS application

The components and fundamentals are little different than the android components and requires a shift from android thinking to understand it.

- App

  In iOS, protocol is a set of methods and properties that encapsulates a unit of functionality. This is similar to an interface in Java/Kotlin. The protocol doesn’t contain any of the implementation for these things; it merely defines the required elements. Any class that declares itself to conform to this protocol must implement the methods and properties dictated in the protocol. Similarly, App in iOS is a protocol that represents the structure and behavior of an app. The App protocol provides a default implementation of the main() method that the system calls to launch your app. You can have exactly one entry point among all of your app's files. A developer usually conforms the App protocol and provide entry point @main into the app (28).

- View
  View in iOS SwiftUI is definition of piece of UI. To describe what's displayed onscreen, a graph of views is created. SwiftUI uses this definition to create an appropriate rendering. View is a function of a state. To update UI, it does not require the view graph to update directly. Instead, the state is modified, and a new view graph is calculated from the state. Then SwiftUI performs rendering to reflect the changes. The identity and lifetime of SwiftUI views are separate from the lifetime of structs that define them. To create any view, View protocol has to be conform from the view definition struct (29).

- NavigationLink & Segue
  NavigationLink is used in conjunction with NavigationView which provides an easy redirection to-and-from various screen destinations. Previously, Segue was a mechanism for the changing destination in UIKit and NavigationLink provides this functionality while using SwiftUI. And both Segue and NavigationLink won't work without NavigationController and NavigationView respectively.

- info.plist
  This file contains all the meta data about the application. It is a resource containing key-value pairs that identify and configure a bundle. Operating system of the device after installing the application uses this file to derive information like name of the application, bundle identifier, supported interface (portrait/landscape), launch screen images, multi window support attribute, device support capability information etc (30).

- Like in android, there are not much more basic pilers that are necessary to be there, but they are all included in the form of Kit. Each Kit is nothing but framework with dedicated feature to provide and are already mentioned in architecture of the iOS operating system.

3.4.3 iOS project structure

An Xcode project is the source for an app; it's the entire collection of files and settings needed to construct the app. To create, develop, and maintain an app, one must know how to manipulate and navigate an Xcode project. Below are the description of files and folders which are mostly edited in order to build the app by the developer.

3.4.3.1 Frameworks

Similar to modules in android, A framework is a structured directory that encapsulates shared resources, such as a dynamic shared library, image files, localized strings, header files, and reference documentation in a single package. Multiple applications can use all of these resources simultaneously. The system loads them into memory as needed and shares the one copy of the resource among all applications whenever possible. In the latest releases, Apple recently started providing Swift package manager to support easy adding of external frameworks and also creation of new frameworks made easier (31).

3.4.3.2 Important folders based on grouping

An Xcode project is a repository for all the files, resources, and information required to build one or more software products. A project contains all the elements used to build your products and maintains the relationships between those elements. The Project navigator displays your project's files and lets you open, add, delete, and rearrange those files. Xcode shows project files and folders using tree structure. Below are some folders discussed as shown in Figure 15 iOS application project structure.

- Project root folder

  The root folder is usually an Xcode project executable file, which in accordance with the file system not a root folder but in logical structure Xcode shows it like that. The actual root folder is outside executable folder in file system. Both of them are named as project name. The root project folder is called as main bundle (32).

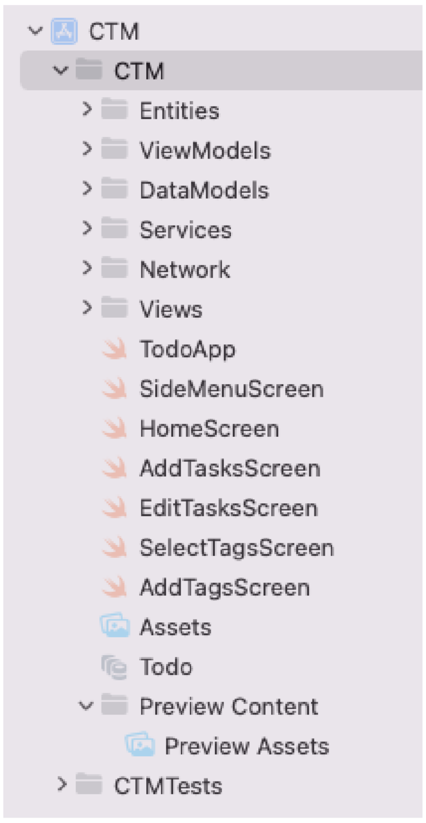
- Test folder

  Test folders are also called as bundles and lives in parallel to main folder in file system. In the Xcode there is a dedicated view where the division is shown based on types of tests. There can be two types of test folders provided by Xcode. One is for unit test or also called as functional test and another is called UI testing bundle. These test bundles have their own info.plist to store their configuration (32).

- Assets

  Asset is a single folder in Xcode that is used to organize application images, icons, colors, custom artworks and more. A single Assets catalog makes it easy to add assets in a neatly organized manner instead of adding individual images to Xcode's file organizer. When it comes to image or icon, asset catalog contains a list of image sets. Each set contains all the versions of image that are necessary to support various devices and scale factors (33).

Figure 15 iOS application project structure



Source: Created according to (32)

3.4.4 Xcode – Development IDE

When the transition comes for choosing options for right IDE, in Android ecosystem might have few choices, but in iOS ecosystem as Apple does not allow publishing app other than from Xcode, the Xcode is the most preferred IDE. Though there are usage of another IDE's like AppCode from JetBrains, or while developing cross-platform apps VS Code etc.

- Xcode is an IDE created by Apple for developing software for macOS, iOS, watchOS, and tvOS. It is the only officially supported tool for creating and publishing apps to Apple's app store and is designed for use by beginners and experienced developers (34).
- With the introduction to swift Xcode also provides built-in control support for source control accounts and makes it easy to leverage available Swift packages. It Integrate package dependencies to share code between projects, or leverage code from other developers (35).
- Xcode can build, install, run, and debug Cocoa Touch apps in a Mac-based Simulator with iOS SDK.
- Xcode supports git and subversion as its version control integration directly in IDE with its availability in command line.
- XCTest framework makes it easy with Xcode to write unit tests for Xcode projects that integrate seamlessly with Xcode's testing workflow.
- Xcode makes it easy to write code using with advanced code completion, code folding, syntax highlighting and bubbles that display warning, errors, and other context-sensitive information inline with the code.
- Within Xcode, Instruments records information about all the processes on the system, revealing performance bottlenecks caused as processes interact. Developer can choose any of the bundled instruments in the library from low-level CPU, network, or file activity to advanced graphics and user-event instruments.
- Xcode makes it easy to inspect data spikes on the graph to see what code is executing at the time, then easily jump into Xcode to fix the problem.
- Reporting error is a common features in IDE but with a coding mistake while writing code, Xcode immediately alerts, and a keyboard shortcut instantly fixes the issue.

3.4.5 iOS design framework evolution to SwiftUI

When it comes to designing iOS application Apple wants its platform to be extra ordinary for the user and for that it has a dedicated section in its documentation called Human interface guidelines where it mentions the design guideline with considering various factors like clarity, deference and depth (36).

To deal with the above-mentioned design standard Apple provided UIKit framework to build user interfaces that can handle touch events and inputs while managing interactions between the user, the system, and your app. UIKit was developed and released based on Objective-C language and with a mental model of building user interfaces in code or using storyboard or with the xib file (37).

With Xib, it was only possible to represent one view element (View, controller or table cell, etc.) So, for every view there needed an individual xib file. This approach was easier to develop, test and reuse. But they were limited to simple views. With the increase in complexity, it was hard to create dynamic views and debug them. Also they were lazily loaded which was advantageous according to memory usage but also reflects latency in loading (38).

The storyboard is a part of interface builder which has been the most popular way to create UI for iOS applications after its release. The main advantage associated with storyboard is its visualization and rapid prototyping and easy learn for a newbie. Creating mockups, navigation flow and transitions can be faster without writing code. But they are limited when it comes to reusability as all of its dependencies also need to be moved. Since the storyboard generates XML files which are hard to read, creates a havoc with changes when it comes to merge conflicts while working in a bigger team (39).

To deal with these disadvantages Apple introduced SwiftUI based on Swift and interoperable with existing Objective-C. SwiftUI is relatively new and has many areas to improve because the area covered by Objective-C and storyboard in development, is much larger. Still SwiftUI solves many problems. It reduces double state maintenance of the view like compose solves in android. The single framework can be used for multiple platform development. The reusability of developed view is more better. The most important is its declarative approach to provide reactive programming. Though it requires again a shift in the way developer used to think. Because it shifts from what it should be rather than how it should be approach (39).

4. Practical part

The practical part is going to describe about app conceptualization and its implementation. It will also go through some major parts of code which is necessary to satisfy objective of this thesis.

4.1 Application concept

The concept to implement this idea came from the time management article written by Ex-Meta employee Rahul Pandey. While scrolling across, this article was found and it inspired me to work on the idea. In this article, he talks about effective time management techniques by aiming for no more than one bigger tasks a day and as many smaller tasks as you like (40). And that's why I have kept the name conscious time management (CTM). I have added a little more to that by adding tags which displays an event about what happened with this task. In short, this app is about applying a task management framework in a day-to-day life, which lets you enter the title of the task, description about the tasks and tags to see the task at a glance. Although, there are many matured applications out in the market when it comes to time and task management, but here I am using the concept to dig more into the implementation technology rather than actual concept. To make the application interact with network I have also provided quotes feature, which shows quotes from remote server api.

4.2 Requirements, Project creation & Project structure

It is necessary to get some insight on setting the environment by having proper installations for actual coding. The application I am targeting to implement is multiplatform application. It will work on Android and iOS both. When the implementation target is to only write shared code for Android platform, Android studio is enough running on any platform. When the implementation target is to only write iOS specific code, and run on iOS simulators or physical Apple device it requires Xcode running on Mac. Apple does not allow it for doing the same on any other platform or operating system (41).

It is necessary to create a application project on Android Studio to start developing for Multiple platform using Kotlin Multiplatform Mobile. And requires minimum certain version for that which is described ahead. Similarly, to build and run iOS project it requires certain minimum version for Xcode. Also the structure of the KMM project is little bit different and requires choosing project view to see multiple platform files. The project structure in more details is described in child section below.

4.2.1 Requirements for starting KMM project

Next are the steps I have followed to prepare environment for the development. Missing any of the steps will surely get into the error that is hard to recall, as the technology is still in alpha and unable to show actual cause properly.

- It requires minimum Android studio version 4.2 or 2020.3.1 canary 8 or higher. I have installed Bumblebee 2020.1.1 Patch 2 which was released on February 17 2022 and is stable release version (41).

- To write iOS specific code and run iOS applications on simulator it requires minimum Xcode version 11.3 or higher. I have installed Xcode version 13.2 beta 2 (41).

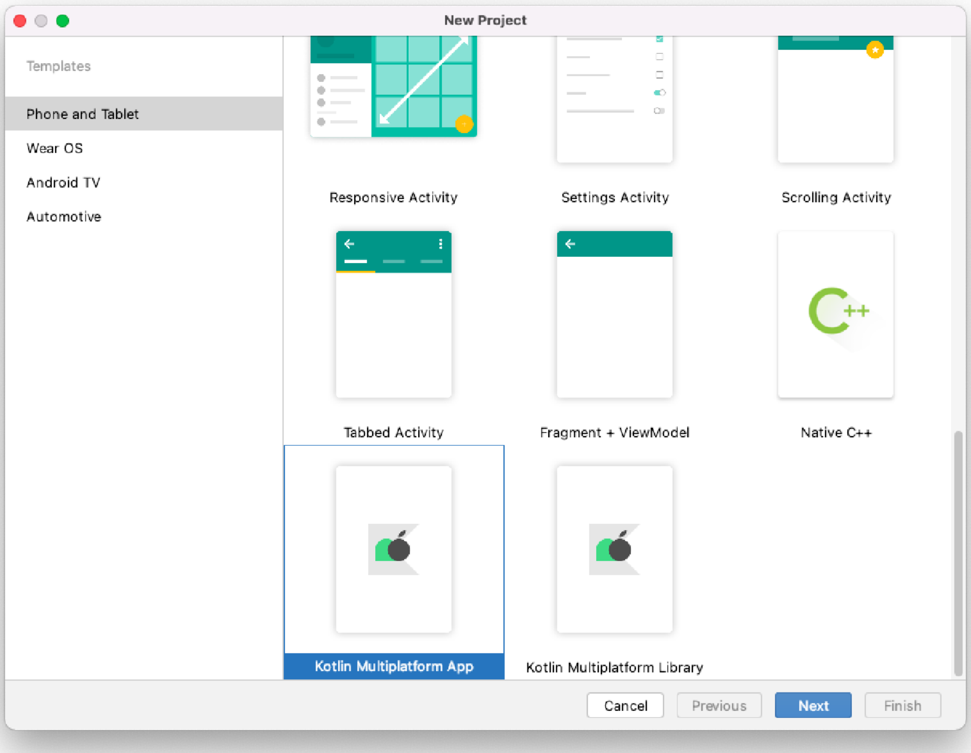
- As discussed above, it is required to create a project in Android studio for Kotlin Multiplatform application. It needs a plugin named Kotlin Multiplatform Mobile to install in Android Studio to create the project. I have installed the plugin by following path Preferences | Plugin in android studio. The plugin versions while developing the application I have used are 0.3.0 and then upgraded to 0.3.1 (41).

- It is also recommended to update the Kotlin language plugin version before creating the project. As the Multiplatform facility is still in alpha, it gets many language compatible bug fixes. JDK is also required to install in order to build and run project successfully in Xcode. Because Android Studio provides it in built support for using JDK. Without JDK you won’t be able to build project successfully and might get the error which initially hard to understand (41).

4.2.2 Creating cross platform mobile app

In android studio, after installing plugin, the steps are somewhat similar to create the project. File | New | Choosing the right template and then enter the name of both the applications. The difference to note is at the place while selecting the template as shown in Figure 16 Selecting KMM project template (42). Rest part is natural human guidance for anyone who is in the mobile application ecosystem. After creating the application one can directly see the preview of hello platform in Android and iOS device where hello platform actually come from shared code.

Figure 16 Selecting KMM project template



Source: Author

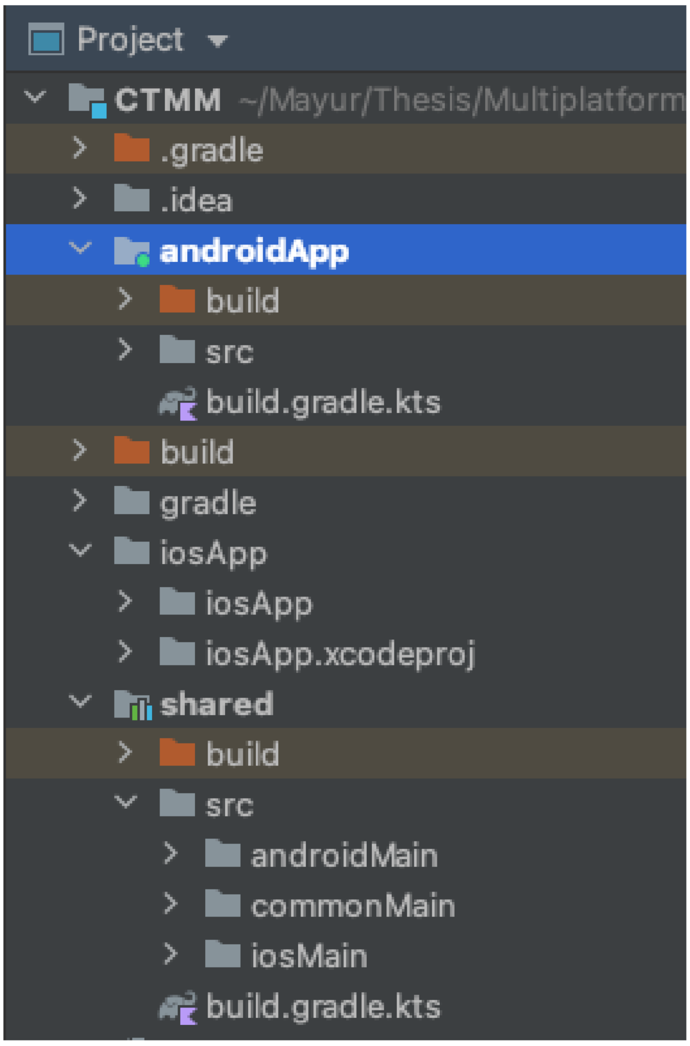
4.2.3 Structure of KMM Project

When a basic Kotlin Multiplatform project is created, it has mainly three components viz. Android application, iOS application, and Shared module. The sample structure which is used for creating application discussed above is shown in Figure 17 KMM project structure.

- Android application

  This component is same as regular module in android project. It uses Gradle as build system. This point is to be noted, because iOS project has other structure for build system. Gradle in android is used as build system and dependency management as well (43).

Figure 17 KMM project structure



```
Project
CTMM ~/Mayur/Thesis/Multiplatform
  .gradle
  .idea
  androidApp
    build
    src
    build.gradle.kts
  build
  gradle
  iosApp
    iosApp
    iosApp.xcodeproj
  shared
    build
    src
      androidMain
      commonMain
      iosMain
    build.gradle.kts
```

Source: Author

- iOS application

  It is the Xcode project that builds into the iOS application and usually the code is written in Xcode by opening the same iOS project which is shown in above Figure 17. While creating project, android studio asks to choose the method for shared Kotlin module framework distribution. We have to choose between cocoa pods dependency manage or regular framework distribution (43).

- shared module

  This is pure Kotlin module that contains common logic for both android and iOS applications. It also uses Gradle as build system and for android builds into android library and for iOS into framework. Shared module has another three important packages which separates the common and platform specific logic. commonMain is used to declare/define functionality with expect keywork which needs platform specific implementation. androidMain & iosMain contains the actual platform specific logic and expose using actual keyword. Definitely the name definition of keywords used revels the usage clearly (43). Each of these source sets have their own dependencies defined in the sourceSets section in Gradle file for shared Module. Along with these three source sets packages there are additional test folders as well, which lets you define test cases for each platform and for common logic separately (43).

4.3 Source code and architecture

The architecture is one of the most important things which makes the code not only performance effective but makes it clean, readable and maintainable. After the era of Model-View-Control (MVC), Model-View-Presenter (MVP), the most loved architecture with current stack of the technologies used is Model-View-ViewModel (MVVM).

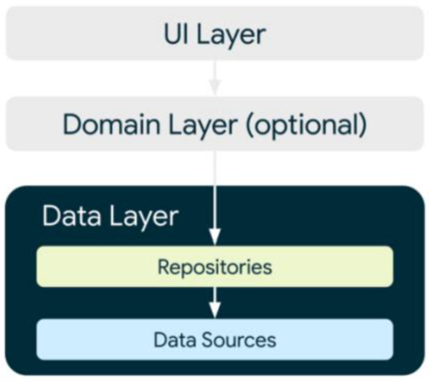
4.3.1 Insights on architecture used

I have implemented MVVM architecture in these projects which best fits in the reactive paradigm. The concept behind this architecture is based on clean code unidirectional data flow.

- In MVM, there is usually data layer at the bottom part which is made of two components. Depending on the requirement in this layer there can be various types of data sources like remote or network data source, local data source, etc. The repository layer usually exposes the mix of these data sources combined to expose data related to a particular entity. E.g., Task repository exposes only task related data and quote repository exposes quote related data, which will be discussed in more detail ahead (44).

- The domain layer applies the business operations that might need before the data reaches to the UI. This layer is optional, and you will most probably find it in the form of use-cases (44).

Figure 18 Application code architecture



Source: (44)

- The topmost layer which is UI layer consists of two parts, one that actually has code for showing and manipulating the visual components shown to the user and other which is mostly called as presenter or in mvvm ViewModel, is responsible for holding & manipulating the state of the UI (44).

- This architecture is implemented as unidirectional architecture where data flows from bottom most data layer to topmost UI and event flows from top to bottom, which changes the state of the data.

- Kotlin shared code fits well in the data layer as this is the layer which hides the complexities of communication with external sources. Going above this layer increase the cohesion with platform specific code.

4.3.2 Insights on source code

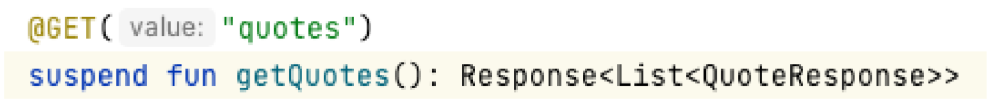
The source code insights are explained by using MVVM layer discussed above by comparing source code for each layer.

4.3.2.1 Data layer

Data layer in CTM, has two data sources, Local and Remote. Local data source is used to persist tasks in local database. While remote sources is fetching a list of quotes from an api. Local data source used to demonstrate how it interacts with native platform in each scenario considered and remote data source for demonstrating interaction with network on each platform and scenario.

- Network pure android

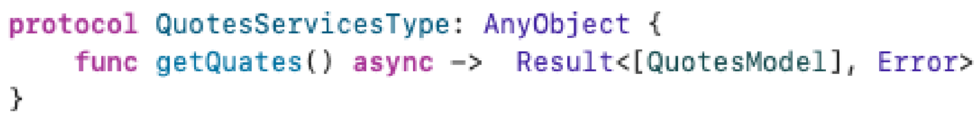
The network/remote layer in a pure android app is written using retrofit which is also recommended by Google for type safe network interaction. It generates type safe network call for the request.



```
@GET( value: "quotes")
suspend fun getQuotes(): Response<List<QuoteResponse>>
```

- Network pure iOS

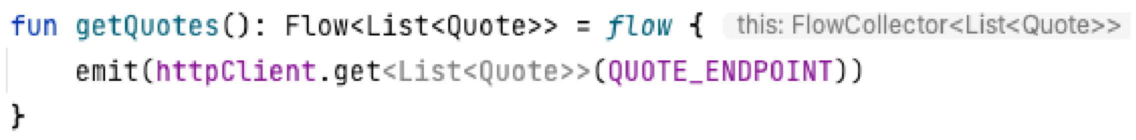
  The network layer in pure iOS app is written using URLSession, which is native and recommended way from apple and mostly used by developers.



```
protocol QuotesServicesType: AnyObject {
    func getQuates() async ->  Result<[QuotesModel], Error>
}
```

- Network Kotlin Multiplatform

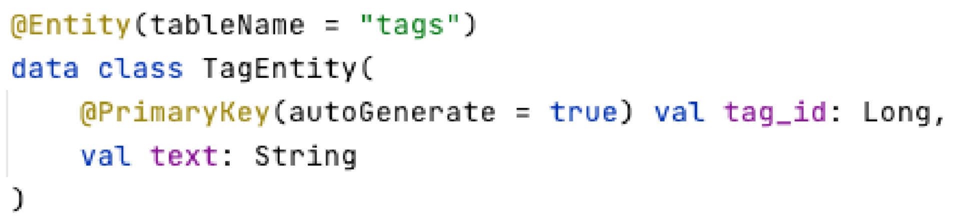
  The network layer in Kotlin Multiplatform is written using Ktor client, which is multiplatform asynchronous HTTP client, which allows you to make requests and handle responses, extend its functionality with plugins. It provides its platform specific implementation using it client specific dependencies.



```
fun getQuotes(): Flow<List<Quote>> = flow {  this: FlowCollector<List<Quote>>
    emit(httpClient.get<List<Quote>>(QUOTE_ENDPOINT))
}
```

- Database pure android

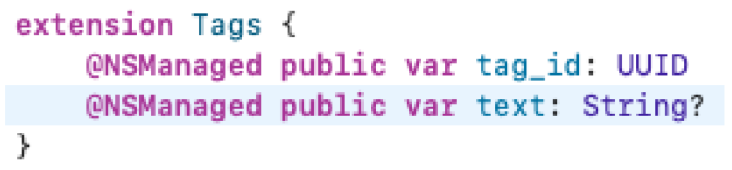
  Database in android is handled using Room which provides abstraction layer over SQLite to allow more robust way to access database locally. Here, we will consider a simple Kotlin data class which generates table without writing any query.



```kotlin
@Entity(tableName = "tags")
data class TagEntity(
    @PrimaryKey(autoGenerate = true) val tag_id: Long,
    val text: String
)
```

- Database pure iOS

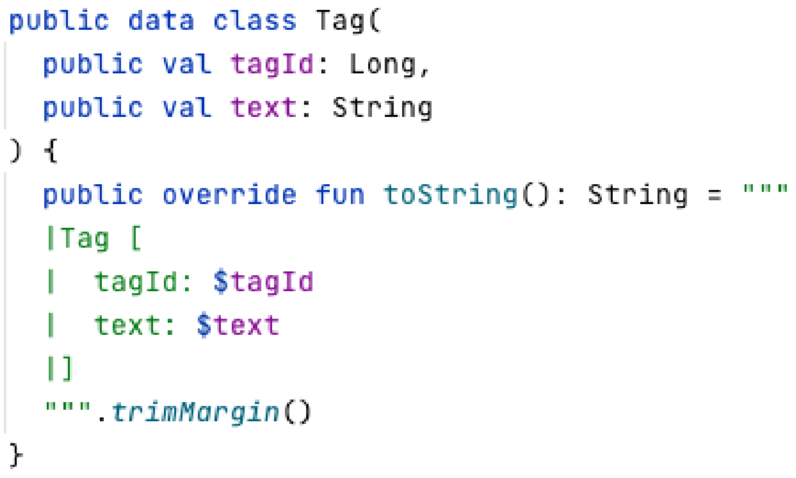
  In iOS Core data provides visual interaction with data modeling, which makes it quite easy to deal with entity creation. Core data abstracts the mapping of objects to a store easy and manually create and edit them.



```swift
extension Tags {
    @NSManaged public var tag_id: UUID
    @NSManaged public var text: String?
}
```

- Database Kotlin Multiplatform

  In multiplatform project, I have used SQLDelight. SQLDelight generates type safe Kotlin APIs from your SQL statements. It verifies the schema defined in .sq files, statements, and migrations at compile-time and provides IDE features like autocomplete and refactoring by using its plugin which make writing and maintaining SQL simple.

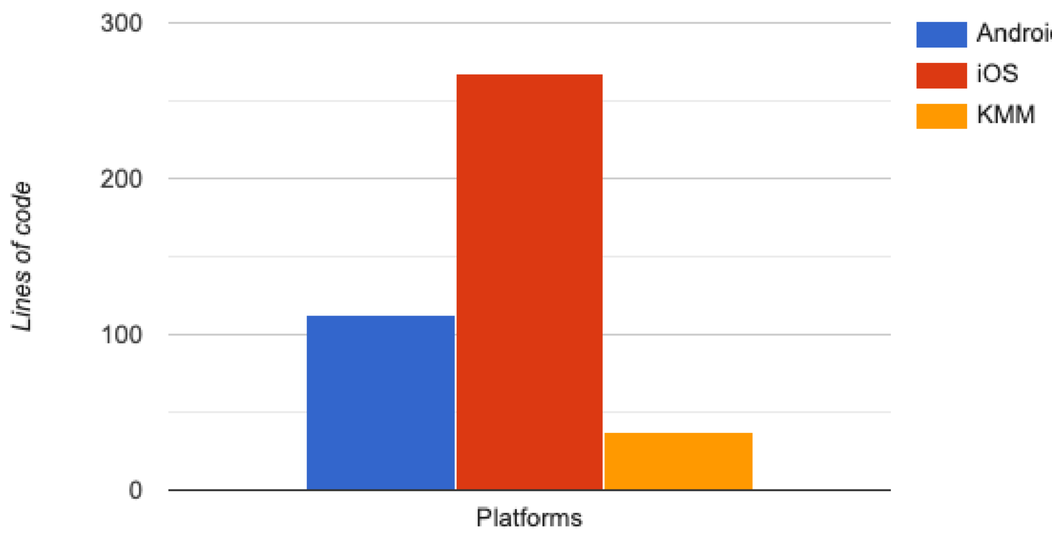


```kotlin
public data class Tag(
  public val tagId: Long,
  public val text: String
) {
  public override fun toString(): String = """
  |Tag [
  |  tagId: $tagId
  |  text: $text
  |]
  """.trimMargin()
}
```

- As we saw above, each native platform have their own mechanism to deal with local or remote data sources. And similarly, shared code in KMM code shows that it can deal quiet easily with both the platform simultaneously.

- In the data layer, the code which is responsible for interacting is quiet less compared to the native platform. It fetches the list of quotes from a remote api. Below is the graph to show number of lines of hand written code without considering auto generated code.

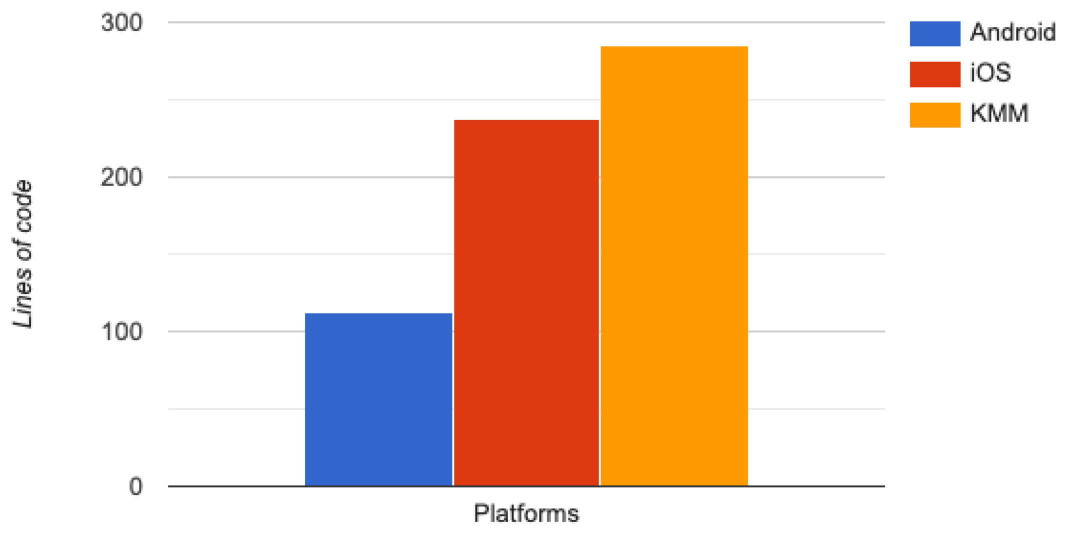
Figure 19 Network interaction code comparison



Source: Author

- The analysis of amount of code required to interact with local database in data layer is shown in the below chart. It stores and retrieves task related data from the database.

Figure 20 Database interaction code comparison



Source: Author

4.3.2.2 UI and Presentation and domain layer

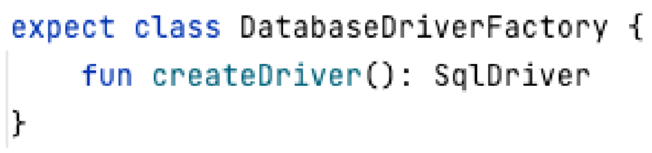
As discussed, above the UI layer is responsible for showing data based on the state received from a state holder like ViewModel. In android, the UI layer is written using the Jetpack compose framework, which is also a hot topic in town in android ecosystem. In iOS, the presentation layer is written using the SwiftUI, which is also declarative in nature like compose. These same components which are written for native frameworks are used in the KMM project, as it allows to choose the UI layer to be written in native technology. So, there will not be any point in comparing those parts of the code. Also, this is comparatively simple project and does not require to manipulate data from data layer to presentation layer. So, there is no need of use cases in optional domain layer.

4.4 Focusing on shared code

As the, UI and presentation layers remain almost same, I will focus here in terms of lines of code, how much code is shared and how much code is removed / added in shared code module.

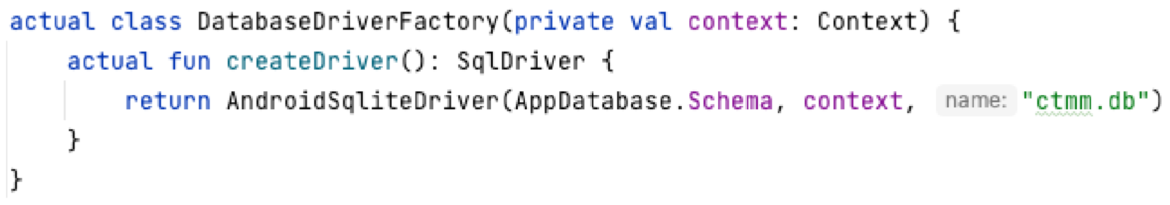
- As discussed in 4.2.3, The shared modules contain three packages androidMain, iosMain and common main. The commonMain module contains the logic for network and database creation. Networking with Ktor did not require any platform specific implementation. It manages with its dependencies provided in Gradle already.
- But the database provide by each mobile platform has their own configuration of creation and interaction specific to it. SQLDelight which handles in this project database interaction provides auto generation of entities which can be used on both the platforms to interact with database. But the database drivers which are platform specific needs to be initialized separately in different way. It is handled by expect-actual mechanism provided by Kotlin Multiplatform. commonMain contains the declaration for getting database drivers from androidMain and iosMain. androidMain and iosMain provides it by using actual implementation of that expect declaration (45).

  This snippet shows expect driver declaration from commonMain.



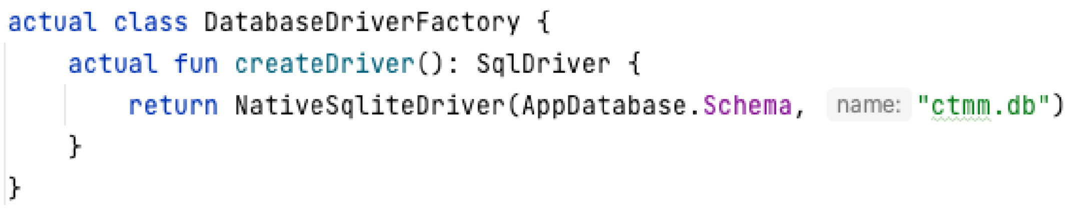
```
expect class DatabaseDriverFactory {
    fun createDriver(): SqlDriver
}
```

This snippet shows actual driver provider from androidMain.



```
actual class DatabaseDriverFactory(private val context: Context) {
    actual fun createDriver(): SqlDriver {
        return AndroidSqliteDriver(AppDatabase.Schema, context,  name: "ctmm.db")
    }
}
```

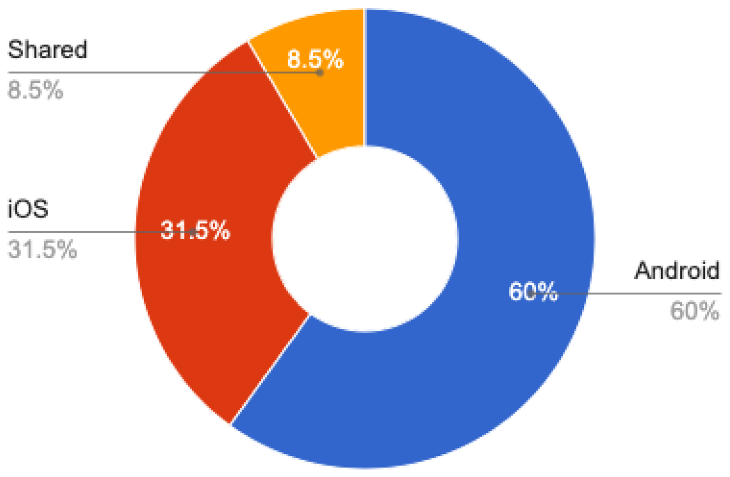
This snippet shows actual driver provider from iosMain.



```
actual class DatabaseDriverFactory {
    actual fun createDriver(): SqlDriver {
        return NativeSqliteDriver(AppDatabase.Schema,  name: "ctmm.db")
    }
}
```

- This is the only thing in this sample project I had to provide platform specific implementation for. Though in bigger projects might need some more amount of code to provide.
- The total amount of lines of code considering network and database in the shared module is 435 including the type safe queries written in .sq file common for both the platforms. The total lines of code required for networking and database interaction natively on android was 225 and that on iOS was 505. In android the code is little less because the Room Jetpack library handles relations well with less code. Even though shared module code is little more than android code for same thing, but shared code actually replaces code from both the platforms.
- When we consider a part of code which will be same in KMM and native applications. It will mostly comprise data classes, presentation, and view code. Considering the same, the total lines of code from iOS is 1610 and from 3063. The android compose part is relatively new and requires some view to be created custom than native. That’s why it might seem more. But with improvements and increasing reusability it can be as less as near to iOS. Combining this with database and networking using KMM the total overall occupied share is shown below

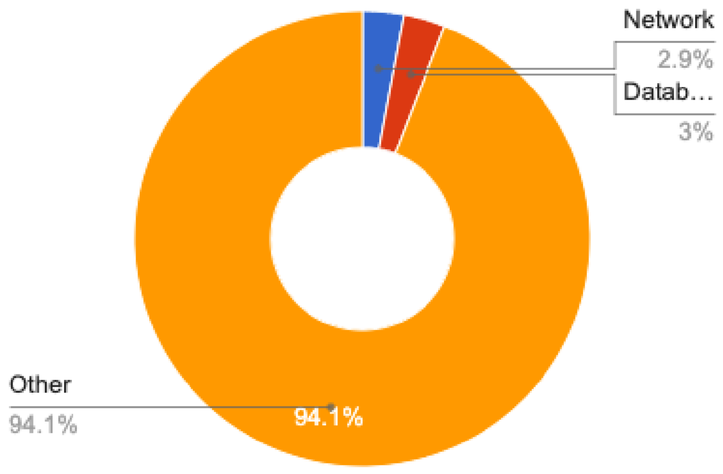
Figure 21 Code distribution in KMM project



Source: Author

- Below figure shows network and database share from native android project which can be replaced with KMM. From that we can see we can replace almost near to 6% of the code using KMM. This might look smaller but in bigger projects it is quiet huge and also from maintenance perspective it can be huge benefit to update same code once rather than updating on two platforms.

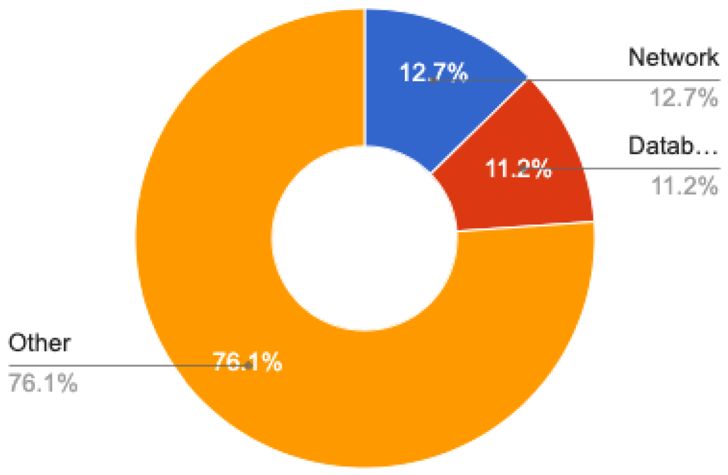
Figure 22 Android native project code distribution



Source: Author

- The next figure shows network and database share from native iOS project which can be replace with KMM.

Figure 23 iOS native project code distribution



Source: Author

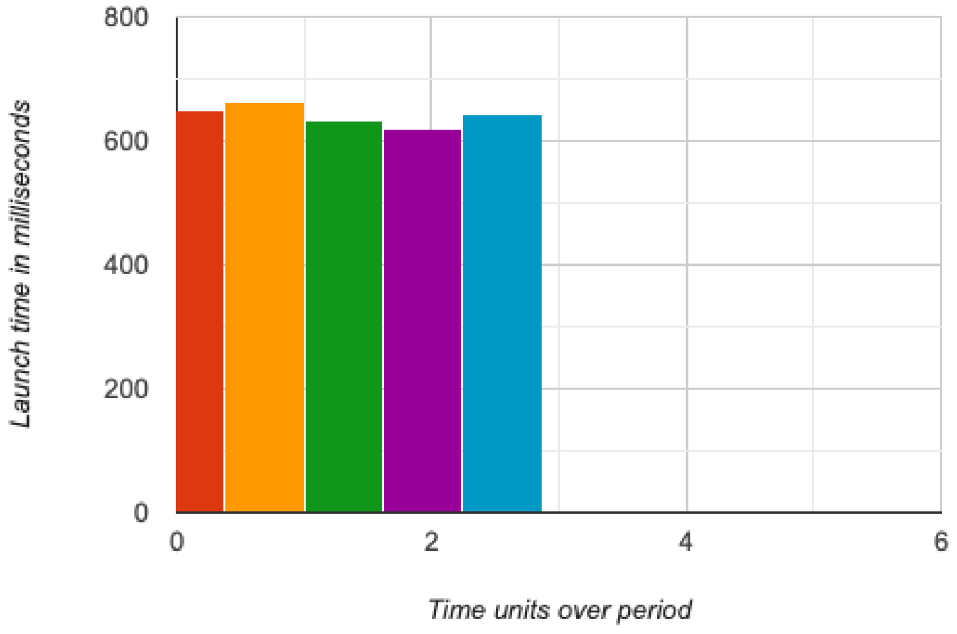
4.5 Application launch time

Definitely, the user experience better and at more higher level when the application launch and response time is faster. KMM does not seem to be affecting much this parameter and on an average it stays similar to native application.

The launch time can vary depending on whether is going to be cold start, warm start or hot start. Definitely warm and hot start are more safer place when it comes to application launching as the process is mostly running and system needs to recreate the UI or display already existing UI (46). That's why I have chosen cold start parameter to check whether KMM affects launching time of the application.

- Next Figure 24 Native android application launch times shows launching time of android native application, after killing the application entirely from memory over the period of time.
- Figure 25 KMM android application launch times shows launching time of android application with KMM shared code, after killing the application entirely from memory over the period of time.
- Figure 26 Native iOS application launch times shows launching time of iOS native application, after killing the application entirely from memory over the period of time.
- Figure 27 KMM iOS application launch times shows launching time of iOS application with KMM shared code, after killing the application entirely from memory over the period of time.

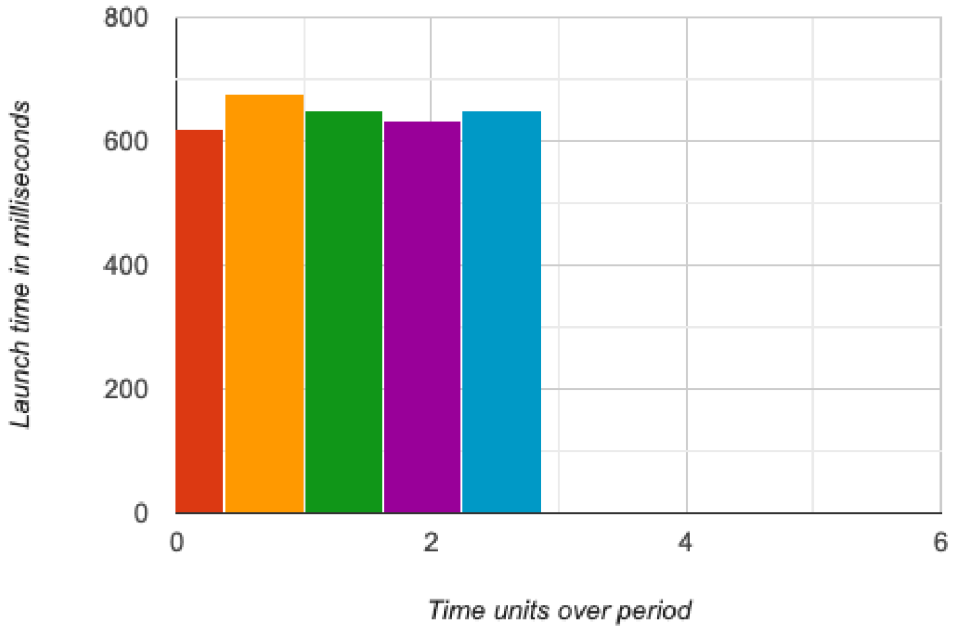
Figure 24 Native android application launch times



Source: Author

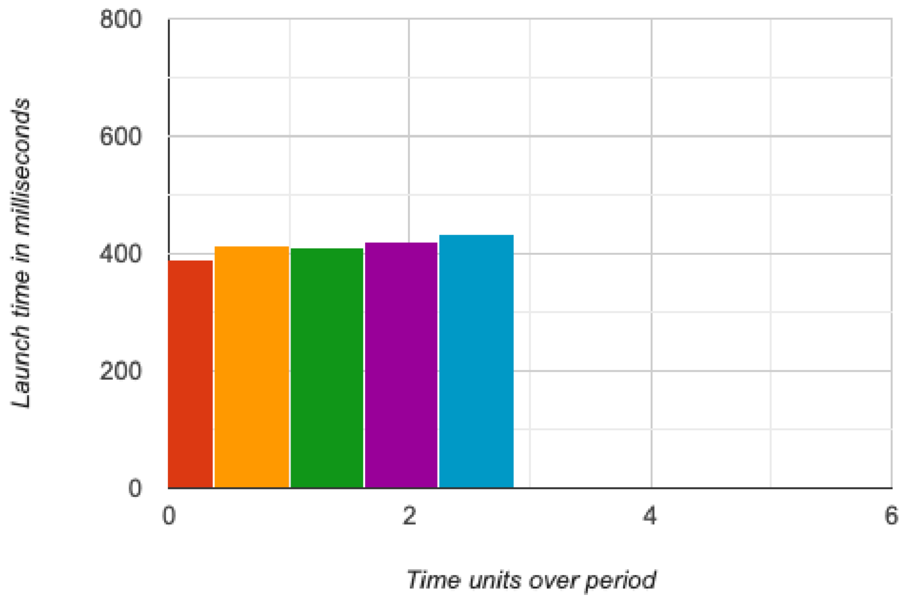
- An Android simulator Pixel XL running on Android 12 (API 31) is used in both the scenarios.

Figure 25 KMM android application launch times



Source: Author

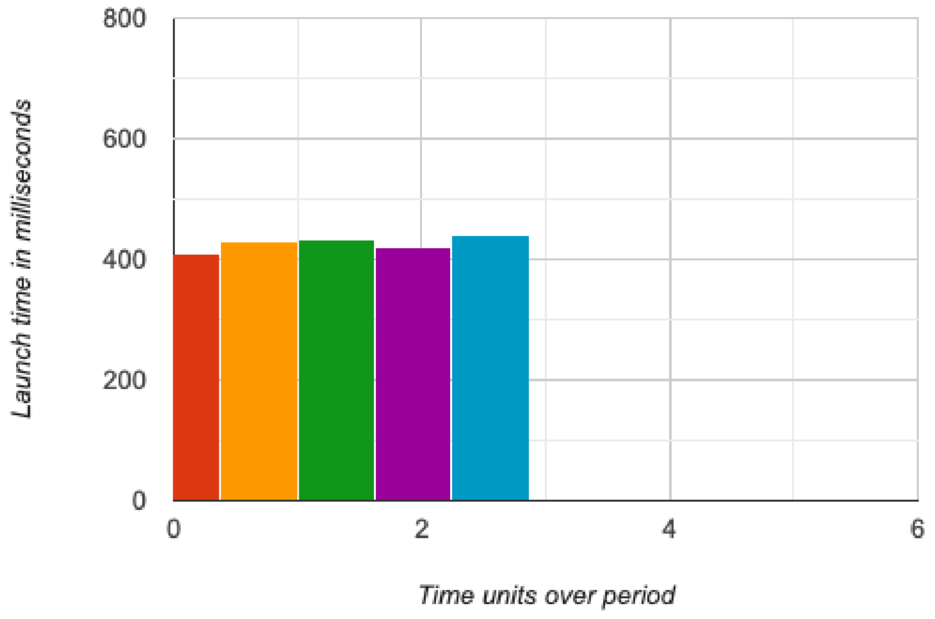
Figure 26 Native iOS application launch times



Source: Author

- An iOS simulator iPhone 13 Pro Max with iOS 15.2 is used for testing in both the scenarios.

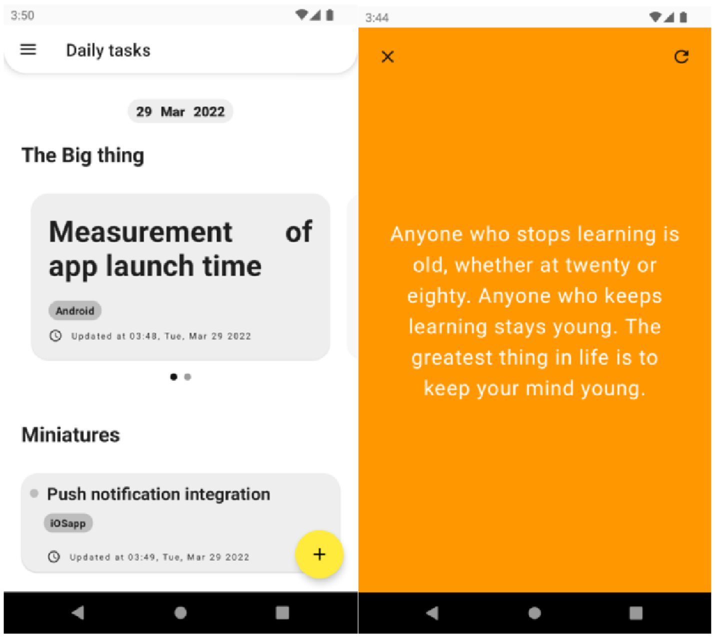
Figure 27 KMM iOS application launch times



Source: Author

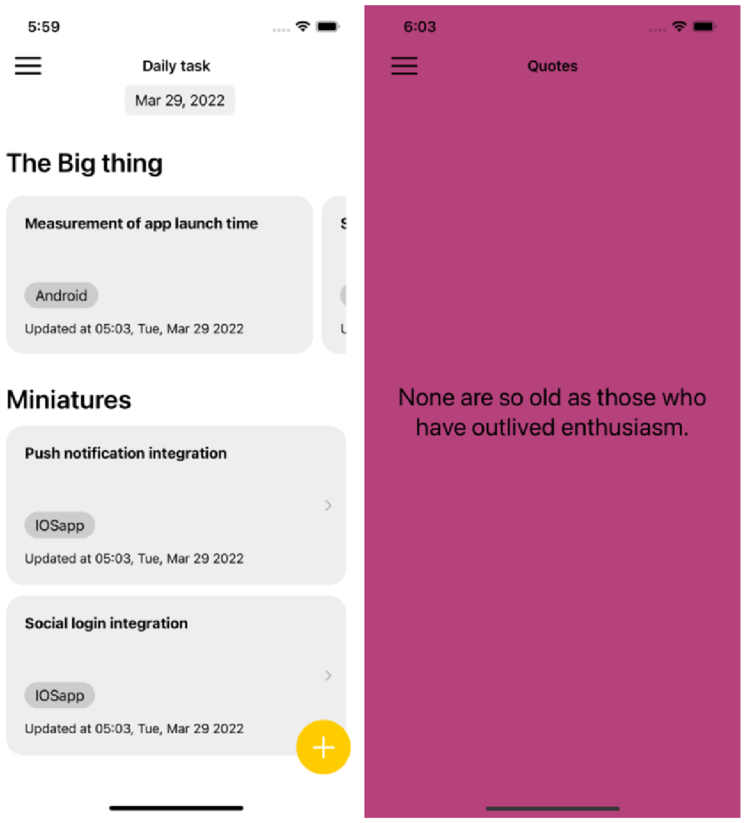
4.6 Application visuals

Figure 28 CTM android application visuals


3:50
Daily tasks
29 Mar 2022
The Big thing
Measurement of app launch time
Android
Updated at 03:48, Tue, Mar 29 2022
Miniatures
Push notification integration
iOSapp
Updated at 03:49, Tue, Mar 29 2022
3:44
Anyone who stops learning is old, whether at twenty or eighty. Anyone who keeps learning stays young. The greatest thing in life is to keep your mind young.

Source: Author

Figure 29 CTM iOS application visuals


5:59
Daily task
Mar 29, 2022
The Big thing
Measurement of app launch time
Android
Updated at 05:03, Tue, Mar 29 2022
Miniatures
Push notification integration
IOSapp
Updated at 05:03, Tue, Mar 29 2022
Social login integration
IOSapp
Updated at 05:03, Tue, Mar 29 2022
6:03
Quotes
None are so old as those who have outlived enthusiasm.

Source: Author

5. Results & discussions

The developed application for time and task management and the code analysis done above are the result of this thesis. The main use case of Kotlin multiplatform Mobile is applied for Android/iOS pair apps. Since its introduction and regular releases it has reached alpha stage. Even though being in alpha stage, many developers have started to use it in production. Sharing code among multiple platforms does not only save time but can save a significant cost when it comes to find and fix bugs while maintaining shared codebase. It provides a structured way to associate different parts of data layer of you codebase.

The other point through which it attracts is when there is another library or API which is native to platform and provides better alternative, KMM does not force or restrict the usage. It can be told to it to use the native implementation. Which is one of the best advantage and relief after using other cross platform frameworks.

The KMM is around since short time and used as experimental feature. For this new technology certain flows may not be stable or partially supported. As any new community with the new technology grow over time, so finding support for a problem might be difficult for now. For android developers, KMM is little easier than for iOS developer to adopt. As the Kotlin is now getting its root in the android ecosystem, KMM works well with android. The developer coming from iOS background my have to struggle a bit more due to the difference between Kotlin and Swift. Cocoapods is quite still quiet new in KMM. Sometimes the errors are hard to understand due to unclear messages. Also, it happens like there is an error but it does not show up. Plus for android developers making a small fix in Swift is also quiet struggling as it will require dealing with Xcode. But after all of these things starts getting solved as framework evolves, it brings a lot of benefits for both iOS and Android developer by providing a chance to grow into other technology.

When it comes to accessing some Kotlin code from KMM into iOS, it might require a bridge. E.g., Swift can't directly access Flow. Though there less code as of now which can be shared, but whatever is being shared in data layer is what drives the application. Even though presenters/ViewModel/State holders can become platform independent, but it might take some time to properly standardize this pattern.

- Future outlook

  As future roadmap of Kotlin talks about rewriting its compiler for speed optimizations and unify its difference for all other multiplatform frameworks, it might be promising to expect improved performance of KMM as well.

6. Conclusion

At the beginning of the thesis, I did not have much coding experience with KMM. My knowledge with Android helped me to get into it. My goal was the study of impact of applying this Kotlin Multiplatform Mobile in the cross-platform world. And also get insights on the flexibility provided by KMM. During the tenure while I was doing this thesis, I had the opportunity to apply my learnings into implementing this framework and also get familiar with iOS application development. I studied how KMM can resolve the inconsistencies that used to occur as a result of different perceiving things differently at the core data layer of the application code architecture.

During writing this thesis, I have gone through setting up a Multiplatform project and what are the complexities that may arise while creating it. I got to know that there can be many pitfalls which are hard to understand even for experience developers. But again got most of the answers from the community.

The result of this thesis expose the experience of code sharing between different a pair of platforms and how it can create a single source of truth for the data requirement of the application. The main idea behind the practical part was to evaluate the possibility of adopting a technology which is in alpha stage but growing faster and giving a considerable competition to the peer technologies in the market.

My final thought on this would also share that KMM is a matter of team work for both platform developers. There will still arrive the problems until it reaches the stable state. But its promising to believe that it will not keep questions unanswered like hybrid platforms. The evaluation on KMM adoption should be done on case by case basis. Because with the flexibility sometimes it might be overkilling. Overall, it is one of the trusted technology that will grow with stable versions.

7. Bibliography

1. **StatCounter.** Mobile Operating System Market Share Worldwide. [Online] [Cited: November 19, 2021.] https://gs.statcounter.com/os-market-share/mobile/worldwide.

2. **Top Digital Agency.** Cross Platform and Multi-Platform: Differences to Know. [Online] [Cited: November 21, 2021.] https://topdigital.agency/cross-platform-and-multi-platform-differences-to-know/.

3. **Gaba, Rahul and Ramachandran, Atul.** React Made Native Easy. [Online] [Cited: November 21, 2021.] https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html.

4. **Microsoft Corporation.** Xamarin | Microsoft Docs. [Online] [Cited: November 24, 2021.] https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin#how-xamarin-works.

5. **Google Inc.** Flutter architectural overview. [Online] [Cited: November 24, 2021.] https://docs.flutter.dev/resources/architectural-overview.

6. —. Application Fundamentals | Android Developers. [Online] [Cited: November 25, 2021.] https://developer.android.com/guide/components/fundamentals.

7. **Srivastava, Naveen.** Under the hood rendering in Flutter. [Online] FlutterDevs. [Cited: November 25, 2021.] https://medium.flutterdevs.com/under-the-hood-rendering-in-flutter-ddc5aadd65ba.

8. **Kotlin Foundation.** Kotlin Multiplatform | Kotlin. [Online] [Cited: November 25, 2021.] https://kotlinlang.org/docs/multiplatform.html#how-kotlin-multiplatform-works.

9. —. Multiplatform Gradle DSL reference. [Online] [Cited: November 26, 2021.] https://kotlinlang.org/docs/multiplatform-dsl-reference.html.

10. **Shekhar, Amit.** How does the Kotlin Multiplatform works? [Online] Mindorks. [Cited: November 26, 2021.] https://blog.mindorks.com/how-does-the-kotlin-multiplatform-work.

11. **Kotlin Foundation.** Kotlin Multiplatform for Cross Platform mobile development. [Online] [Cited: November 28, 2021.] https://kotlinlang.org/lp/mobile/.

12. **Grebenkina, Alina.** The new JVM IR backend is stable | The Kotlin Blog. [Online] JetBrains s.r.o. [Cited: November 29, 2021.] https://blog.jetbrains.com/kotlin/2021/02/the-jvm-backend-is-in-beta-let-s-make-it-stable-together/.

13. **Kotlin foundation.** Introduce cross-platform mobile development to your team. [Online] [Cited: December 5, 2021.] https://kotlinlang.org/docs/multiplatform-mobile-introduce-your-team.html.

14. **Kotlin Foundation.** Case Studies | Kotlin Multiplatform Mobile. [Online] [Cited: December 8, 2021.] https://kotlinlang.org/lp/mobile/case-studies/.

15. **Google Inc.** Android Common Kernals | Android OpenSource Project. [Online] [Cited: December 11, 2021.] https://source.android.com/devices/architecture/kernel/android-common.

16. —. Platform Architecture | Android Developers. [Online] [Cited: December 11, 2021.] https://developer.android.com/guide/platform.

17. **MIT.** Application Fundamentals | Android Developers. [Online] [Cited: December 13, 2021.] https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/fundamentals.html.

18. **Google Inc.** Meet Android Studio | Android Developers. [Online] [Cited: December 13, 2021.] https://developer.android.com/studio/intro.

19. —. Why Compose | Jetpack Compose. [Online] [Cited: December 14, 2021.] https://developer.android.com/jetpack/compose/why-adopt.

20. **Posey, Brien.** Apple iOS. [Online] [Cited: December 16, 2021.] https://www.techtarget.com/searchmobilecomputing/definition/iOS.

21. **Sierra Software Ltd.** Explain architecture of iOS. [Online] [Cited: December 16, 2021.] https://www.ssla.co.uk/architecture-of-ios/.

22. **Singh, Jagroop.** Architecture of iOS operating system. [Online] geeksforgeeks.org. [Cited: December 16, 2021.] https://www.geeksforgeeks.org/architecture-of-ios-operating-system/.

23. **Dubey, Suyash.** Android and iOS: Basics and Comparision. [Online] [Cited: December 17, 2021.] https://www.pcloudy.com/android-and-ios-basics-and-comparison/.

24. **Nix United.** Why swift programming is better for app development. [Online] [Cited: December 19, 2021.] https://nix-united.com/blog/pros-and-cons-of-swift-for-ios-development/.

25. **Apple Inc.** Swift-Apple Developer. [Online] [Cited: December 19, 2021.] https://developer.apple.com/swift/.

26. —. Security of runtime process in iOS. [Online] [Cited: December 19, 2021.] https://support.apple.com/en-in/guide/security/sec15bfe098e/web.

27. **JavaTpoint.** iOS app lifecycle. [Online] [Cited: December 20, 2021.] https://www.javatpoint.com/ios-app-lifecycle.
28. **Apple Inc.** App | Apple Developer. [Online] [Cited: December 20, 2021.] https://developer.apple.com/documentation/swiftui/app.
29. —. View. [Online] [Cited: December 20, 2021.] https://developer.apple.com/documentation/swiftui/view.
30. —. Managing your app's information property list. [Online] [Cited: December 21, 2021.] https://developer.apple.com/documentation/bundleresources/information_property_list/managing_your_app_s_information_property_list.
31. —. What are frameworks? [Online] [Cited: December 22, 2021.] https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html.
32. **Apple inc.** Managing files and folders in your Xcode project. [Online] [Cited: December 22, 2021.] https://developer.apple.com/documentation/xcode/managing-files-and-folders-in-your-xcode-project.
33. **Apple Inc.** Asset Management. [Online] [Cited: December 23, 2021.] https://developer.apple.com/documentation/xcode/asset-management.
34. **Chung, Eddy.** What is xcode and why do I need it? [Online] Zero To App Store. [Cited: December 23, 2021.] https://www.zerotoappstore.com/what-is-xcode-and-why-do-i-need-it.html.
35. **Apple Inc.** Adding package dependencies to your app. [Online] [Cited: December 23, 2021.] https://developer.apple.com/documentation/swift_packages/adding_package_dependencies_to_your_app.
36. —. Themes-iOS-Human Interface Guidelines. [Online] [Cited: January 4, 2022.] https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/.
37. **L, Jeroen.** UIKit vs. SwiftUI: Choosing the Right Framework! [Online] Stream.io. [Cited: January 4, 2022.] https://getstream.io/blog/uikit-vs-swiftui/.
38. **Creitive.** Storyboards versus XIBs versus custom code. [Online] [Cited: January 6, 2022.] https://www.creitive.com/global/blog/storyboards-versus-xibs-versus-custom-code.
39. **Arora, Smriti.** Swift UI or StoryBoard? [Online] Technology at Nineleaps. [Cited: January 6, 2022.] https://medium.com/technology-nineleaps/swift-ui-or-storyboard-675ff2b40829.

40. **Pandey, Rahul.** Achieve more with concious time managemnt. [Online] [Cited: January 17, 2022.] https://www.linkedin.com/pulse/achieve-more-conscious-time-management-rahul-pandey/.
41. **Kotlin Foundation.** Setup an environment | Kotlin. [Online] [Cited: January 19, 2022.] https://kotlinlang.org/docs/multiplatform-mobile-setup.html.
42. —. Create your first cross-platform mobile app. [Online] [Cited: February 8, 2022.] https://kotlinlang.org/docs/multiplatform-mobile-create-first-app.html.
43. —. Understand mobile project structure. [Online] [Cited: February 13, 2022.] https://kotlinlang.org/docs/multiplatform-mobile-understand-project-structure.html.
44. **Google Inc.** Guide to app architecture. [Online] [Cited: February 27, 2022.] https://developer.android.com/jetpack/guide.
45. **Kotlin Foundation.** Connect to platform specific APIs. [Online] [Cited: March 3, 2022.] https://kotlinlang.org/docs/multiplatform-connect-to-apis.html.
46. **Google Inc.** App startup time. [Online] [Cited: March 15, 2022.] https://developer.android.com/topic/performance/vitals/launch-time.