Mobile App Development Using Kotlin Multiplatform

Michał Guśpiel

2024 Laurea

Laurea University of Applied Sciences

Mobile App Development Using Kotlin Multiplatform

Michał Guśpiel

Business Information Technology

Bachelor's Thesis

April, 2024

Laurea University of Applied Sciences **Abstract**

Degree Programme in Business Information Technology

Bachelor's Thesis

Michał Guśpiel

Mobile App Development Using Kotlin Multiplatform

Year 2024 Number of pages 39

The objective of this thesis was to evaluate Kotlin Multiplatform (KMP) technology in the context of mobile application development. KMP, when compared to other cross-platform technologies such as Flutter and React Native, is a relatively new and unexplored framework that enables developers to share codebase across Android, iOS, Desktop and Web platforms. This research directly benefits the client company, by enhancing their knowledge of modern mobile development. The beneficiary, Solita, is an IT consulting company that aims to create impact, by combining tech, data and human insight. This thesis could positively impact the company's future offerings, tenders and projects.

This thesis report presents an evaluation of technology through the development of a simple patient monitoring application. This specific use case not only adds a distinctive business edge to the study but also aligns with Solita's industry-specific business expertise. By applying this use case to the evaluation, this thesis aims to provide valuable insights into the effectiveness of the technology within the healthcare domain. Methods of evaluation include quantitative analysis of the codebase, and performance testing, allowing for comprehensive assessment of KMP's usability.

The development of the application encountered the challenges and obstacles. However, the overall experience with technology was positive. Choosing Compose Multiplatform as the user interface framework proved to be a critical decision that significantly influenced the outcome of the project. Kotlin Multiplatform paired with Compose Multiplatform is an excellent choice especially for Android developers familiar with Kotlin and Jetpack Compose.

During the study it was found that there is no apparent performance nor quality overhead when comparing Kotlin Multiplatform application to native Android application. Conversely, the iOS product faced significant issues, such as unnecessary recompositions resulting in hangs as well as drawbacks such as significantly larger app size and slower launch time.

The application's source code is available at https://github.com/solita/HRnD.

Keywords: Mobile Development, Kotlin Multiplatform, Native, Android, iOS, Kotlin, Compose Multiplatform

Contents

1 Introduction

Mobile applications are inseparable part of our daily lives, providing convenience, accessibility, security and efficiency of usage. From entertainment, to banking and healthcare, mobile devices and therefore mobile applications have revolutionized how people amuse themself or interact with crucial information and services.

The mobile app market continues to experience growth, with global market size value of 197.2 billion dollars in 2021(Straits Research, 2021). As businesses increasingly rely on mobile applications to reach their customers and improve operations, the demand for new apps tailored for specific use cases has increased.

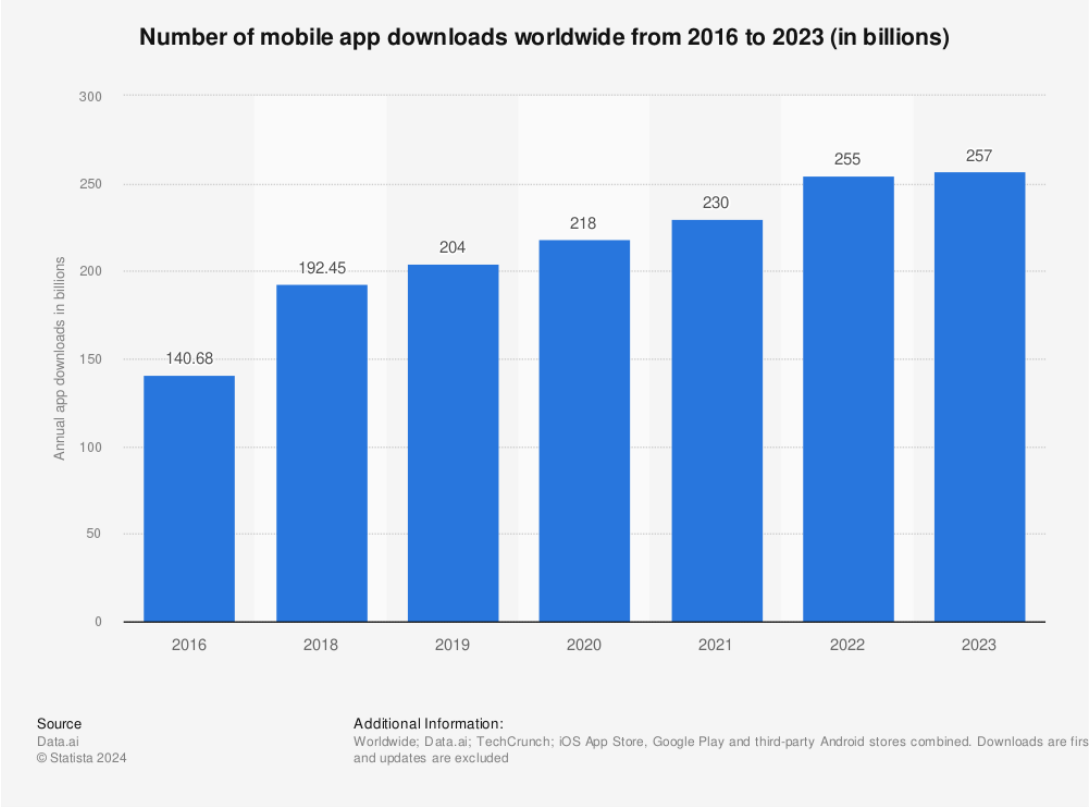

Figure 1. Number of mobile app downloads worldwide from 2016 to 2023. Statista

The market is primarily divided between two major operating systems, Android and iOS. This creates a unique challenge as each platform requires a different approach to application development.

In addition, in recent years the landscape of mobile development has evolved rapidly, driven by advancements in technology and changes in user expectations. In order to stay competitive, companies often find themselves in the need of developing two separate applications to satisfy their business requirement for all their customers regardless of their mobile device. This complicates delivery, requires more man hours and thus drives cost of the investment.

For some time cross-platform frameworks such as Flutter or React-Native have been addressing this issue by enabling developers to write code that functions on both platforms. More recently, JetBrains introduced Kotlin Multiplatform as an innovative solution to address this challenge in a slightly different approach. Kotlin Multiplatform enables cross-platform and yet supports native development on both platforms. KMP offers the flexibility to choose how much code is shared between the platforms.

This thesis aims to explore the Kotlin Multiplatform as a technology choice for mobile application development. Through the development of the Health Rundown, showcase patient monitoring application this study seeks to evaluate KMP's development experience, codebase size and performance.

2 Context

The purpose of this chapter is to supply the reader with the essential information required for understanding of this thesis. It covers basic concepts of Mobile development as well as differences between native development and cross-platform development methodologies.

2.1 Mobile App Development

Mobile app development is the process of creating software that is specifically designed to operate on mobile devices, such as smartphones, tablets, wearable devices and augmented reality smart glasses. Following sections will dive into two main methodologies of mobile app development: native development and cross-platform development

2.1.1 Native Development

Native development is the primary methodology of application development, provided by the vendors: Google for Android and Apple for iOS. Native development involves using platform specific languages, tools and Software Development Kits (SDKs).
Android development utilizes:

- Kotlin or Java as a programming languages
- Android Studio as Integrated Development Environment (IDE)

On the other hand iOS development is done using:

- Swift or Objective-C as programming languages
- XCode as IDE

Native development has its advantages and disadvantages. According to Schmitt (2023), native applications are winning in terms of superior performance, security, consistent user experience and full feature access. Kotlin Journal (2023) agrees that performance, intuitive user experience and access to full feature set are all pros of native development. On the other hand, Schmitt (2023) argues that project cost, development time and lack of code reusability are drawbacks when utilizing native technology. In addition, Kotlin Journal (2023) identifies likelihood of more errors in code due to larger codebase and risk of having different logic on Android and iOS as a disadvantages of native tech stack.

2.1.2 Cross Platform Development

Cross-platform development is the method of building software so that is interoperable across different operating systems. Instead of writing entirely different code for iOS and Android, developers can focus on building single codebase to run on multiple operating systems or leverage shared codebase between different target systems. One significant advantage of Cross-platform development is the ability to share code, leading to faster development and reduced project costs (Kotlin Journal, 2023). However, Nagy (2022) argues that cross-platform development using established frameworks such as Flutter and React-Native can ultimately result in the higher development cost than having two separate native development tracks, in his opinion it is due to the inevitable road blocks that development

teams face when using popular cross-platform frameworks.

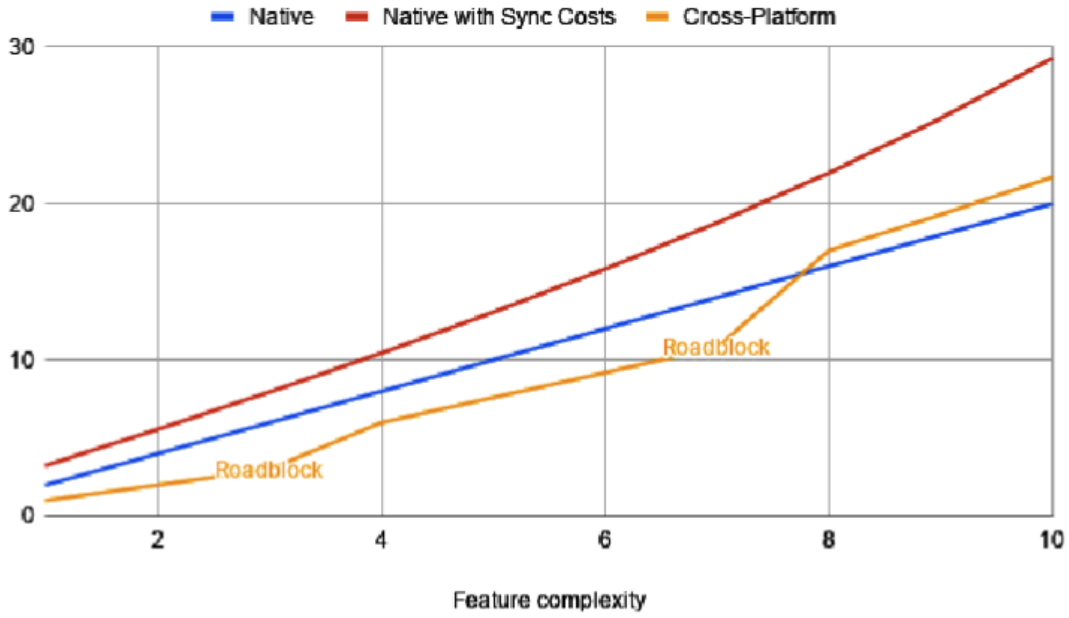

Figure 2. Actual cost of cross-platform

Developers have several options when choosing cross-platform development frameworks, including but not limited to: Flutter, React Native, Xamarin, Ionic, and Kotlin Multiplatform. As the title suggest this thesis exclusively explores Kotlin Multiplatform.

2.2 Kotlin Multiplatform

In order to fully grasp the following sections of this thesis it is essential to understand both the overall structure of Kotlin Multiplatform ecosystem and some of its specific details, such as expect-actual mechanism. This chapter provides the reader with basics of Kotlin and Kotlin Multiplatform, and also illustrates how Kotlin Multiplatform approach stands out from other cross-platform frameworks.

2.2.1 Kotlin Overview

According to Isakova and Jemerov (2017) Kotlin stands out as a statically typed, versatile programming language known and appreciated for its conciseness safety and practicality. It is designed to fully integrate with Java, thus it ensures full interoperability with existing Java libraries. However, it is worth noting that the establishment of Kotlin Multiplatform has influenced the landscape of Kotlin and Java interoperability. This will be explored further in the following section – Basics of Kotlin Multiplatform.

Kotlin is especially popular in Android development. In order to understand its position and importance in Android development it is crucial to point out that at Google I/O 2019, Google has announced that Kotlin has become the first and official programming language for Android applications, effectively replacing Java (Android Developers, 2024). Java still remains a usable option for building applications. However, certain cutting-edge libraries such as Jetpack Compose lack support for Java, further deepening the shift towards Kotlin.

2.2.2 Basics of Kotlin Multiplatform

JetBrains (2024) states that Kotlin Multiplatform (KMP) enables developers to efficiently reuse code across various platforms such as Android, iOS, web, desktop, and server-side applications, while preserving the option to leverage native code when necessary. Providing option to make a decision when to utilize native code to the developer is an innovative

approach that makes Kotlin Multiplatform stand out against more established Cross-Platform frameworks. Instead of trying to facilitate the development by handling platform specific decisions, KMP recognizes both the need for implementation sharing as well as benefits of platform specific implementation. Therefore, KMP doesn't provide a wrapper layer over the native platforms but attempts to be a tool that enables code sharing between platforms (Nagy, 2022).

KMP enables developers to create platform-agnostic code in Kotlin, which is then compiled using one of three main compilers of the Kotlin ecosystem: Kotlin/JVM for Android, Server and Desktop, Kotlin/JS for web, and Kotlin/Native for iOS and macOS. Considering this, KMP becomes a handy choice for Android and Kotlin developers when project requirements go beyond a single platform (JetBrains, 2024). Because of this approach we can assume that KMP interoperability with Android ecosystem is zero, since Kotlin/JVM is fundamentally part of Android ecosystem. In other cases the overhead depends on performance of the compiler (Nagy, 2022).

One constraint of Kotlin Multiplatform is its limitation regarding the usage of Java libraries when targeting platforms that require other compilation than Kotlin/JVM. One might assume that since Kotlin was designed to be interoperable with Java, KMP would allow for having Java references in the shared platform-agnostic module. However, since other platforms do not inherently support Java, the usage of plain Java libraries becomes impossible (Nagy, 2022). Developers leveraging KMP for platforms other than Android, Desktop and Server must take that into account when designing shared codebase.

2.2.3 Development Theory

Kotlin Multiplatform follows modular design to facilitate code sharing. Each project build with Kotlin Multiplatform contains platform agnostic, Kotlin-based, shared module and number of individual platform specific modules.

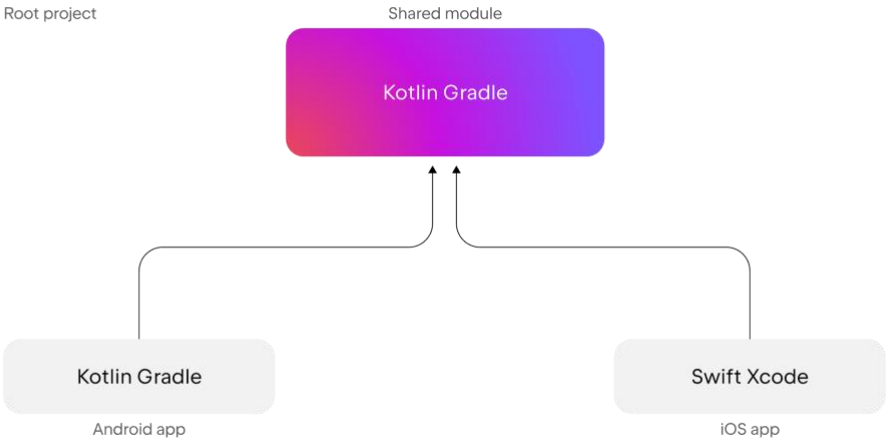

Figure 3. Basic project structure

Shared module then consist of source sets respective for all modules in the project, for instance "androidMain", "iosMain" and "commonMain".

> Source set is a Gradle concept for number of files logically grouped together where each group has its own dependencies. In Kotlin Multiplatform, different source sets in a shared module can target different platforms (JetBrains, 2024).

One of the most important features of KMP is the expect-actual mechanism. It is the main tool when one needs to write platform-specific implementations to be used in the shared module. The idea behind expect-actual mechanism is similar to the Interface from object oriented programming paradigm. The common source set of shared module requires to define expected declaration and each platform specific source set must implement it. The compiler assures that expect definition has its actual implementations in each of the platform specific source sets (JetBrains, 2024).

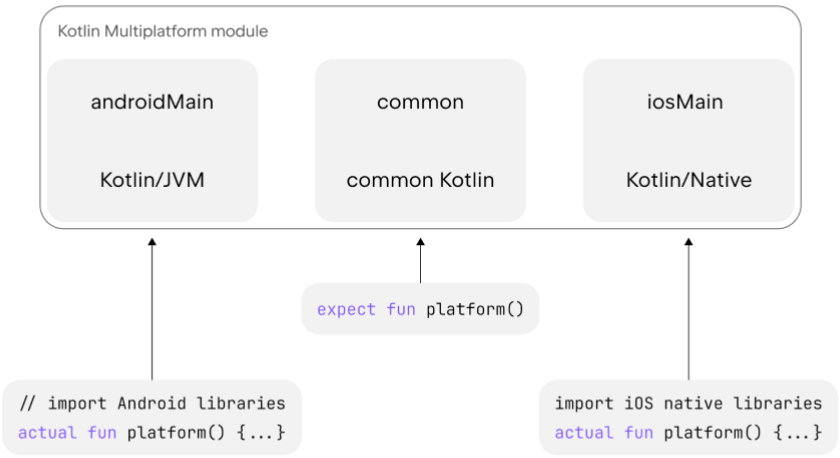

Figure 4. Expect-actual mechanism

Official Kotlin Multiplatform Development documentation (2024) advises developers to utilize existing Kotlin Multiplatform libraries before rushing into writing platform specific code themselves. These libraries have already taken care of writing expect-actual mechanism for the problems that are platform specific. Some of these libraries however, require developers to implement some parts of platform specific APIs manually. Thus, it is essential to understand expect-actual system when working with KMP.

3 Project Scope

According to Adobe Communications Team (2022), project scope is a crucial aspect of project execution. It is determined based on objectives, limitations, strategies, tasks, and deliverables. The definition of a project scope is essential for the successful project delivery. When well-defined it provides a clear roadmap, manages stakeholder expectations, makes budgeting and scheduling easier and prevents ever changing goals. While not immutable, it makes it simpler to adapt and face challenges.

In the context of this research, budgeting is not a concern as there is no allocated budget. However, establishing a properly defined project scope is still crucial to ease scheduling, ensure goal stability, and enable creation of a robust project roadmap.

3.1 Setting Project Scope

The primary aim of this thesis is to assess the feasibility and usability of Kotlin Multiplatform in case of mobile development. This evaluation will be based on various criteria:

- Effectiveness and practicality
  - This thesis aim to assess how effectively KMP facilitates mobile development in terms of code reuse, development speed, and overall productivity.
- Development speed
  - One of the most important aspects of framework is development speed. This study evaluates development speed of Kotlin Multiplatform in the scope of mobile development in comparison with traditional approaches.
- Performance benchmarking
  - According to Android Developers (2024) performance benchmarking includes measuring key common performance issues such as:
    - Scroll Jank – visual hiccup that occurs when the system isn't able to build and provide the frames to draw them on screen in time.
    - Startup Latency – amount of time it takes for an application to start. Startups can be categorized as either a cold start or a warm start. Cold start occurs when the app is not present in system memory and warm start happens when app is already launched in the background. Both start up should take less than 500 ms in order to provide seamless user experience.
    - Battery usage - application work reduces battery charge, and thus doing unnecessary work reduces battery life. Memory allocations, which come from creating new objects in code, can be the cause of significant work in the system.
    - App size – although app size is not necessarily considered performance issue it might affect be considered by the new users downloading the app for the first time.
- Maintainability – Heitlager, Kuipers and Visser (no date) divides maintainability into analyzability, changeability, stability, testability, and maintainability conformance. As suggested by Heitlager et al. The size of the source code affects how easily it can be analyzed. The complexity of individual code sections influences the system's adaptability and testability. Repetition in the code impacts both analysis and the ability to make changes. The size of code units affects their analysis and testability, which in turn affects the entire system. The extent of unit testing directly influences the system's ability to be analyzed, its stability, and its testability. These factors collectively influence the maintainability of the software system.

With comprehensive assessment of such aspects this thesis attempts to offer valuable insights essential for making informed decisions during the process of selecting the most suitable technology for mobile app projects.

Evaluation will be conducted based on both the development process, as well as result of the Health Rundown application benchmarking. Health Rundown application is described in greater detail in the subsequent section. Furthermore, this thesis seeks to identify potential areas for further research related to Kotlin Multiplatform. These areas that remain unexplored within the scope of this thesis, but appear to hold promising benefits for the client company.

3.2 Application Concept

Development of Health Rundown, a simple patient monitoring application presents a perfect opportunity to evaluate Kotlin Multiplatform within the scope of mobile development. It is a valid use case for the beneficiary of this thesis, given their expertise and established customer base in healthcare domain.

The application connects to a basic backend API developed specifically for this thesis. However, its documentation is not included as it is irrelevant to the scope of the thesis topic. Health Rundown application features include:

- List of patients
    - Users can easily access the list of patients in the application, providing them with essential information about all their patients. Each list item allows user to navigate to corresponding patient details screen.
- Patient details
    - The application provides detailed and visually informative patient profiles, encompassing critical metrics such as heart rate, blood pressure, current medications, and medical history. This feature enables healthcare professionals to access and visualize vital patient data in real-time, facilitating informed decision-making and personalized patient care.
- QR code scanning for patient data navigation
    - Application features a QR code scanning functionality that enables users to swiftly navigate and access necessary patient information without additional steps, particularly when in the presence of patient.

The showcase application intentionally excludes features such as security and authentication mechanisms to maintain a primary focus on evaluating the feasibility of Kotlin Multiplatform in context of mobile development without the added complexity of these components. This allows for more streamlined evaluation of technology capabilities in mobile environment.

3.3 Technology Evaluation Through The Process

The design and development of the application focuses on building iOS application using Kotlin Multiplatform. During this process, careful assessment is conducted to understand the additional overhead involved in simultaneously creating an Android application within the same development process. Furthermore, this thesis outlines all aspects requiring extra effort and adoption of workarounds when compared with native development.

4 Project Planning

Given the project tight timeline and its exploration of cutting-edge technology, emphasis on project planning is crucial to succeed with delivery of the application. Careful planning not only ensures timely completion but also lays the base for a professionally executed software project. Therefore, this chapter focuses on aspects of project planning. It begins by discussing utilized methodology, providing a comprehensive explanation of applied frameworks that steer the project. Then it delves into time monitoring. Finally, the chapter wraps up with software design, creating a blueprint for application implemented for the sake of this thesis.

4.1 Methodology

Development of the application is carried out complying with mobile app development best practices, taking into account design guidelines as well as architectural patterns. Since application is developed using Kotlin Multiplatform and targeted for iOS and Android it needs to obey guidelines of both platforms. According to Apple Designing for iOS guidelines (2024), it is important to limit number of screen controls, adapt seamlessly appearance changes, and support interactions that accommodate the way people hold their device, for example placing primary buttons in the bottom area. Android Design & Plan guideline (2024) outlines importance of accessibility and proper usage of system bars. In addition it is necessary to remember that some of the Android devices have curved edges therefore, it is also critical to include safe margins in the UI design.

4.1.1 Kanban

Although implementation is carried out by a single developer, it is essential to leverage project management framework to keep track of current tasks and those that follow. For this purpose, this project employs Kanban. Kanban is a widely adopted framework for managing work across different industries initially developed by Toyota. It is based on the idea of just-in-time production, visualizing workflows and limiting work in progress. The approach utilizes boards divided into columns representing stages of work, for instance planning, doing and testing (Atlassian, 2024). Kanban board management is provided by such a tools as Jira or Trello. This project relies on Trello due to its simplicity.

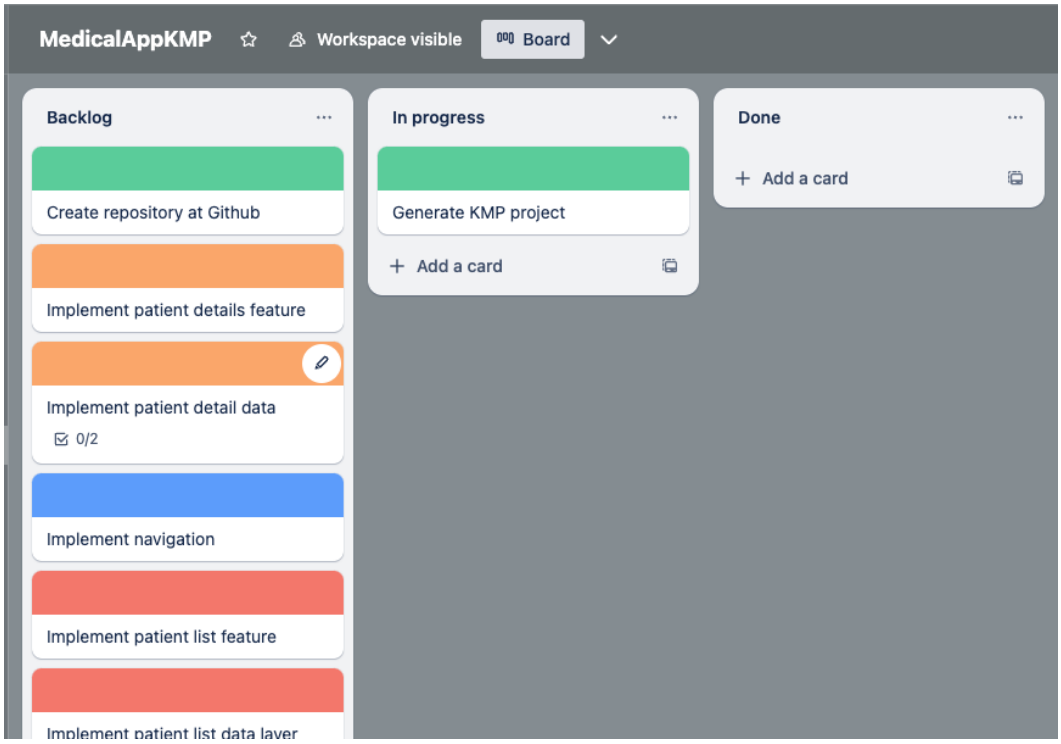

Figure 5. Project Kanban board in Trello

4.1.2 Lean Software Development (LSD)

Lean Software Development is a methodology that emphasizes delivering value by optimizing the entire software development process and eliminating waste. Originating form Lean manufacturing process LSD aims to create more efficient and effective development environment.

LSD advocates practices such as eliminating unnecessary work, building quality into processes, and fostering culture of continuous learning and adaptation. In addition, it stresses the importance of deferring commitment until necessary. Such principles aim to optimize the entire value stream, ensuring efficient and high-quality software development (Agile Velocity, 2010).

Figure 6. Lean – Krusche K&C

4.1.3 Feature Driven Development & Vertical Slicing

In addition, project will be guided by a methodology that prioritizes feature-driven development and vertical slicing. This approach involves breaking down the development tasks into smaller manageable chunks allowing rapid delivery of a minimum viable product (MVP). In a real world project, vertical slicing benefits include reduced feedback cycle, increased stakeholder visibility, and most importantly agility in meeting changing requirements (Agile Data, 2022).

4.2 Time Monitoring

In tight schedule projects time monitoring is one of the most crucial elements that need to be handled properly in order for the project to succeed. Kanban board does not offer time management functionality. Therefore a Gantt chart was chosen as a primary tool ensuring progress of the project.

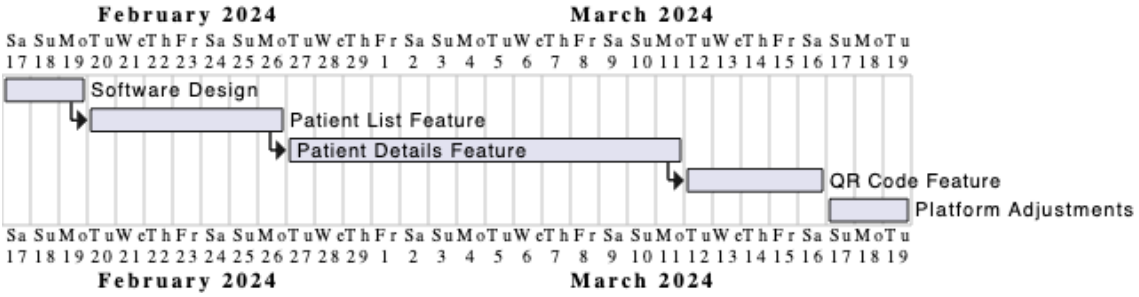

Figure 7. Gantt Chart

Presented Gantt chart outlines initial project delivery plan, encompassing timelines for software design, feature implementation, and allowing for flexibility in platform adjustments.

4.3 Software design

This chapter breaks down the design plan used by the application. It explains how different parts of the app work together and why they are set up the way they are. It attempts to help a reader understand big picture of the application structure. Additionally, it provides an insight into the libraries integrated into the project.

4.3.1 Architecture Pattern

Application follows MVI (Model-View-Intent) design pattern. MVI separates model that represents the application state and data, view layer – responsible for user interface and intent which represents user action. Most important principle of MVI is that it follows unidirectional data flow.

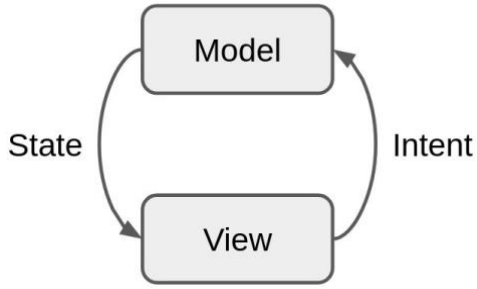

Figure 8. MVI - MVI Kotlin. https://github.com/arkivanov/MVIKotlin

Utilizing MVI in a Kotlin Multiplatform application involves leveraging Voyager's “ScreenModel” to effectively store and manage UI-related data in a lifecycle-aware manner. The lifecycle of an Android application involves creation, activity startup, pausing, stopping, and termination, responding to user interaction and system events to ensure efficient resource management and a seamless user experience, thus requiring careful lifecycle management. Voyager's “ScreenModel” offers fundamental “ViewModel” capabilities tailored for Android while maintaining compatibility with iOS.

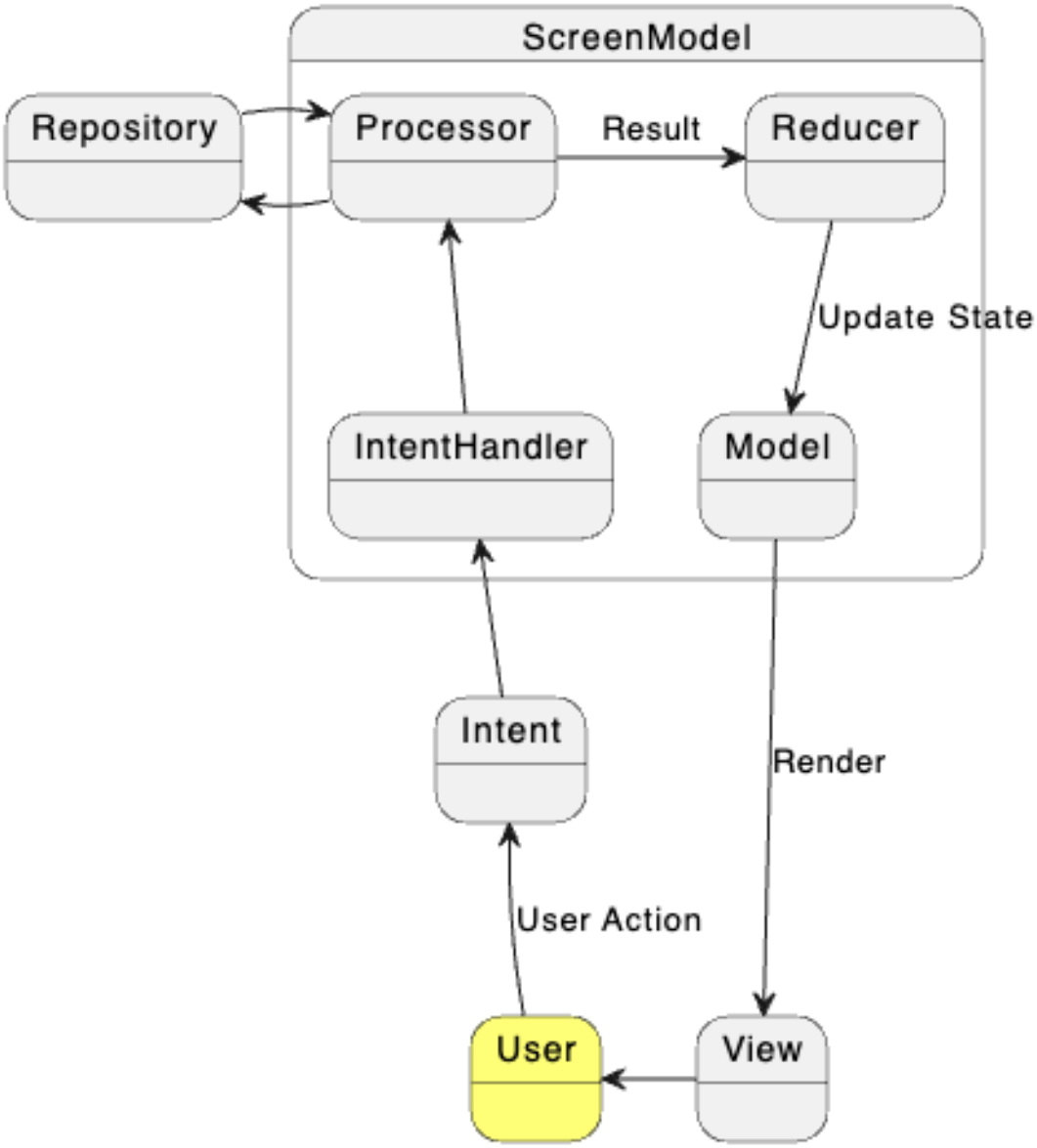

Figure 9. MVI Architecture of patient monitoring app

4.3.2 Utilized Libraries

Application leverages robust set of cutting-edge technologies to speed up the development and enhance user experience.

Project adopts Compose Multiplatform for unification of user interface. Built for Kotlin Multiplatform and based on the established Jetpack Compose, Compose Multiplatform empowers developers to rapidly create functional and beautiful user interfaces for both iOS and Android. Compose Multiplatform lacks navigation support therefore, Voyager is used to ensure smooth navigation within the application.

To maintain a structure of the codebase and ensure separation of concerns, project embraces MVI pattern utilizing Orbit MVI library.

In order to simplify development and management of application dependencies project employs Koin, a pragmatic Kotlin Dependency Injection framework.

5 Project Implementation

This chapter documents the implementation of Health Rundown project. It carefully describes the whole process starting with the design of the application, through development, which itself consists of several sub sections dedicated to different development phases and findings, finally ending with writeup about testability of KMP.

5.1 Design

Starting development without solid design infrastructure highly inefficient. This approach forces developers to struggle with determining the layout of components on the screen and defining the user experience, encompassing factors such as navigation mechanics, button positioning, and presentation of essential information.

Therefore, to mitigate these challenges a simple design was implemented in order to streamline development. It is important to note that the UI and UX design aspects are out of scope of this thesis and are not executed to the professional standard.

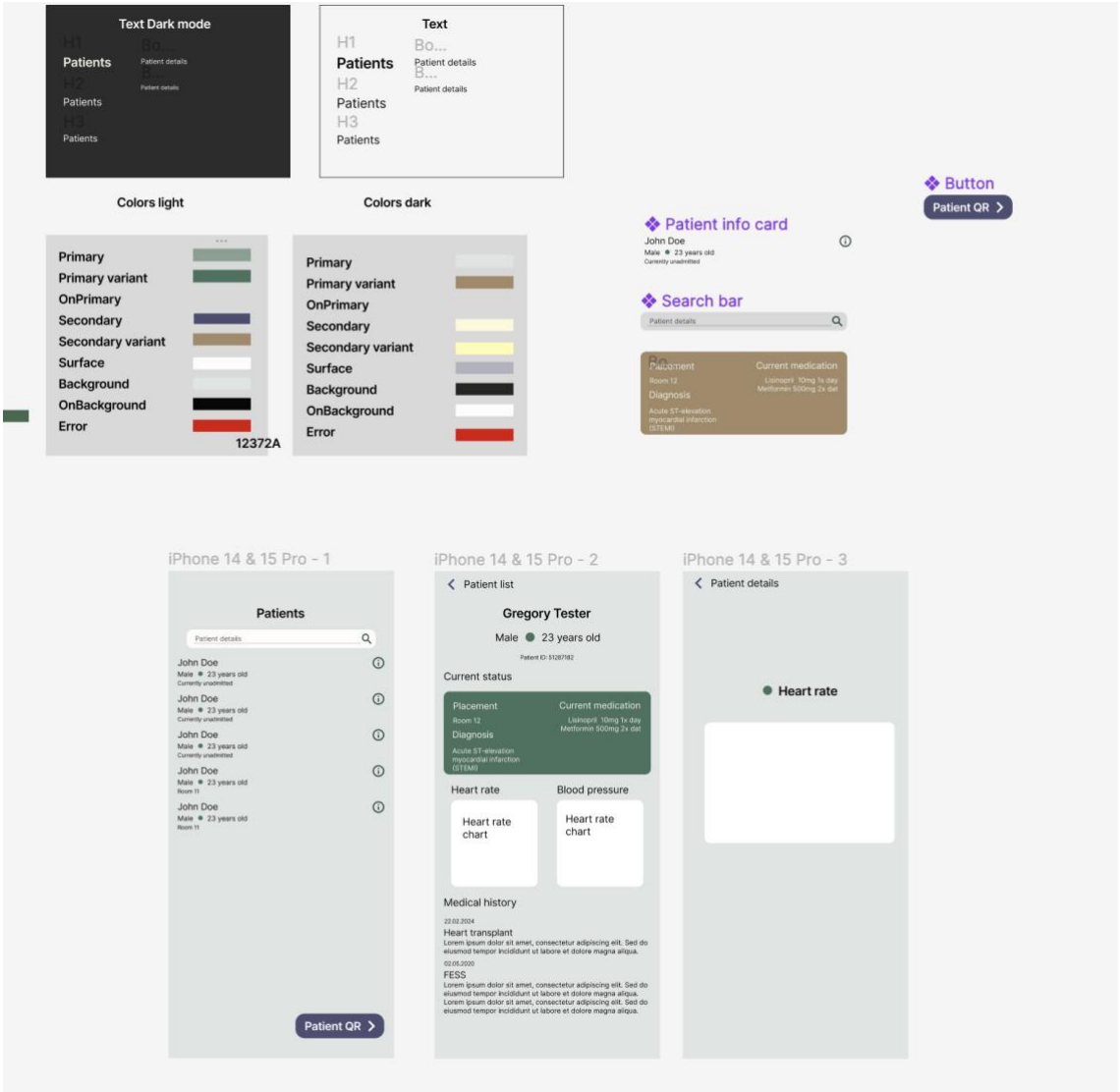

Figure 10. Health Rundown design in Figma.

5.2 Project Initialization

Project was generated with Kotlin Multiplatform Wizard, utilizing newest templates. Shared UI Multiplatform App was selected as a starting point for the project, since it comes with ready configuration for iOS and Android, Compose Multiplatform, Koin, Voyager and Moko

Resources dependencies. In addition it includes ready codebase showcasing how to structure Compose Multiplatform project. Project templated turned out to accelerate the early stage of development with a great deal. However, it lacked basic theme configuration, common for Android and iOS projects.

5.3 Development

After succesfully migrating the Figma designed theme, to KMP project, development has started. This chapter carefully describes the development process, outlining faced issues as well as highlighting achievements and areas for improvement.

5.3.1 Early Issues

At the time of starting the project first striking issue regarding KMP with Compose Multiplatform development was the usage of Previews. Compose Multiplatform could not display preview in the common module, even though the UI elements, called Composables are platform independent. This makes sense when it comes to platform specific details, however Compose Multiplatform could still display different set of previews based on project targets from the common module. Not being able to display preview from the common forced developer to ultimately copy paste preview to Android or Desktop module, since iOS module did not support preview. This also meant that each time developer wanted to see the UI they needed to launch the application, while doable on small scale apps this approach was totally inconvinient for larger projects.

In addition working with string resources was not as easy as in native development ,there was no official solution. Usage of moko resources has been satisfactionary however, whenever resource was added it required a rebuild of a project

5.3.2 Improvements

Luckily, at the time during the time of writing this thesis, significant update was released to Compose Multiplatform -version 1.6.0. It included features such as:

- Highly anticipated improvement of common resources API
- UI testing API
- iOS accessibility support
- Common Preview in Fleet

During the time of writing this thesis, on 1st of March 2024, showcase Health Rundown app was migrated to use newest 1.6.0 Compose Multiplatform, it followed with migration from Moko Resources to official Common resources API. This enabled smoother work with things such as usage of string resources.

Previews were also migrated from Android source set to common. This change eases usage of preview since it can now be used from the same file at the time of development. While sort of working, feature shows its own set of problems; long rendering time and failure to render when combined with new Common resources API. Issue with preview was reported in Compose Multiplatform repository https://github.com/JetBrains/compose-multiplatform/issues/4338.

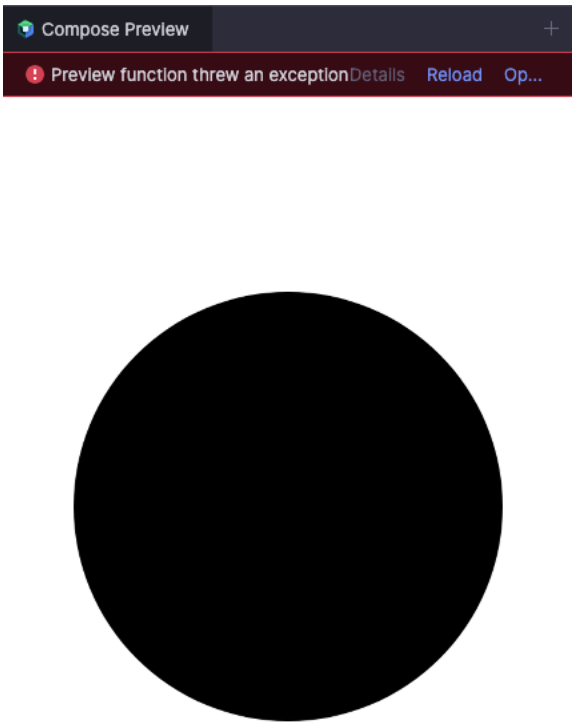

Figure 11. Issues with Preview

5.3.3 Disadvantages of New

Another thing worth noting is the inconvinience of working with datetime data. It is common in software development that datetime data is formated differently across different API and backends systems. When developing native Android application we can utilize well established Java libraries in order to handle the task of parsing or formatting the data. In KMP project targetting both iOS and Android we are forced to use kotlinx-datetime library developed by JetBrains or write an expect Interface that has different implementations for Android and iOS. While well documented, usage of pure kotlin library is not as common as it's Java equivelent thus limiting community support and making it more difficult.

When developing with Compose Multiplatform we must take into account the fact that Compose Multiplatform does not equal Jetpack Compose. There are less productivity boosting libraries for Compose Multiplatform, each Jetpack Compose library must be adjusted to be compatible with Compose Multiplatform. Thus in case of Health Rundown app charts showing the core patient data had to be developed from scratch using Compose Canvas. It is not trivial task, however there was no extra overhead, since canvas usage was almost 1:1 considering Jetpack Compose Canvas.

It was disappointing to realize that it is impossible to use "String.format()" function in KMP common code. This issue has been reported over 5 years ago in Kotlin Multiplatform YouTrack board (2024, YouTrack). It posed a setback for the Health Rundown app's requirements, particularly in rounding decimal numbers within Double variables. Fortunately, a straightforward workaround was readily available through the "roundToInt()" function. Setback would have been more serious if Double number would need to be formatted to one decimal number, requiring custom solution or separate implementations on iOS and Android.

Compose Multiplatform does not provide straightforward access to current device orientation. On one hand it makes sense because it is platform specific problem and platforms such as Desktop or Web will not experience orientation changes, however, Compose Multiplatform is an UI framework which should be able to figure out current orientation. Luckily, KMP provides

very simple expect-actual mechanism that enables developers to solve such problem with ease.

```
/**Returns true when screen is in portrait mode.*/
expect fun isPortrait() : Boolean
```

Figure 12. expect ”isPortrait” function

```
actual fun isPortrait(): Boolean {
    val context = HrndApp.appContext
    val orientation: Int = context.resources.configuration.orientation
    return orientation == Configuration.ORIENTATION_PORTRAIT
}
```

Figure 13. actual “isPortrait” Android implementation

```
actual fun isPortrait(): Boolean {
    val window = UIApplication.sharedApplication.windows.first() as? UIWindow
    return when (window?.windowScene?.interfaceOrientation) {
        UIInterfaceOrientationLandscapeLeft, UIInterfaceOrientationLandscapeRight -> false // landscape
        else -> true // portrait
    }
}
```

Figure 14. actual “isPortrait” iOS implementation

5.3.4 Inconvenient Crash Reporting

When working with KMP and Compose Multiplatform, program runtime crashes pose a bigger issue than it is in native development. It is due to the fact that runtime crashes do not point to the direct line when the crash occurred like it is in for instance Android development. Kotlin Multiplatform crashes provide developer with much less information making it more difficult to solve the underlying issue. Therefore it is advised to incorporate Kotlin Multiplatform and Compose Multiplatform code with log statements.

5.3.5 Platform Differences

Although developers can write same UI code for Android and iOS using Compose Multiplatform, navigation doesn’t work seamlessly out of the box. At least using Voyager as a solution. In Android navigation might be solved using default back button, swipe gesture or toolbar back button. However, iOS Devices do not have a default back button nor swipe gesture by default. When developing Compose Multiplatform utilizing Voyager as navigation tool, developer must provide user option to navigate back explicitly.

Health Rundown app solved this problem using custom “NavigationElement”. The drawbacks of such workaround were the need to declare component in every screen that needed to provide back navigation to the user. Additionally, component need to be provided with the title of the previous screen. In SwiftUI such information is provided automatically by the standard navigation library, omitting boilerplate code. Although this solution works smoothly it is noticeable to users that it differs from the native out of the box solution.

```
@OptIn(ExperimentalResourceApi::class)
@Composable
fun NavigationElement(
    previousScreenTitle: String,
    modifier: Modifier = Modifier,
    onClick: () -> Unit
) {
    Row(
        modifier.height(22.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.Start
    ) { this: RowScope
        if (getPlatform() == Platform.iOS) {
            IconButton(onClick = { onClick() }) {
                Icon(
                    painterResource(Res.drawable.chevron_ios),
                    contentDescription = stringResource(Res.string.back_button_desc)
                )
            }

            Text(text = previousScreenTitle, style = MaterialTheme.typography.h5)
        }
    }
}
```

Figure 15. “NavigationElement”

In addition, iOS version of Health Rundown does not enable edge to edge background by default like it’s Android equivalent. This results in a major difference in user interface representations between platforms and must be addressed.

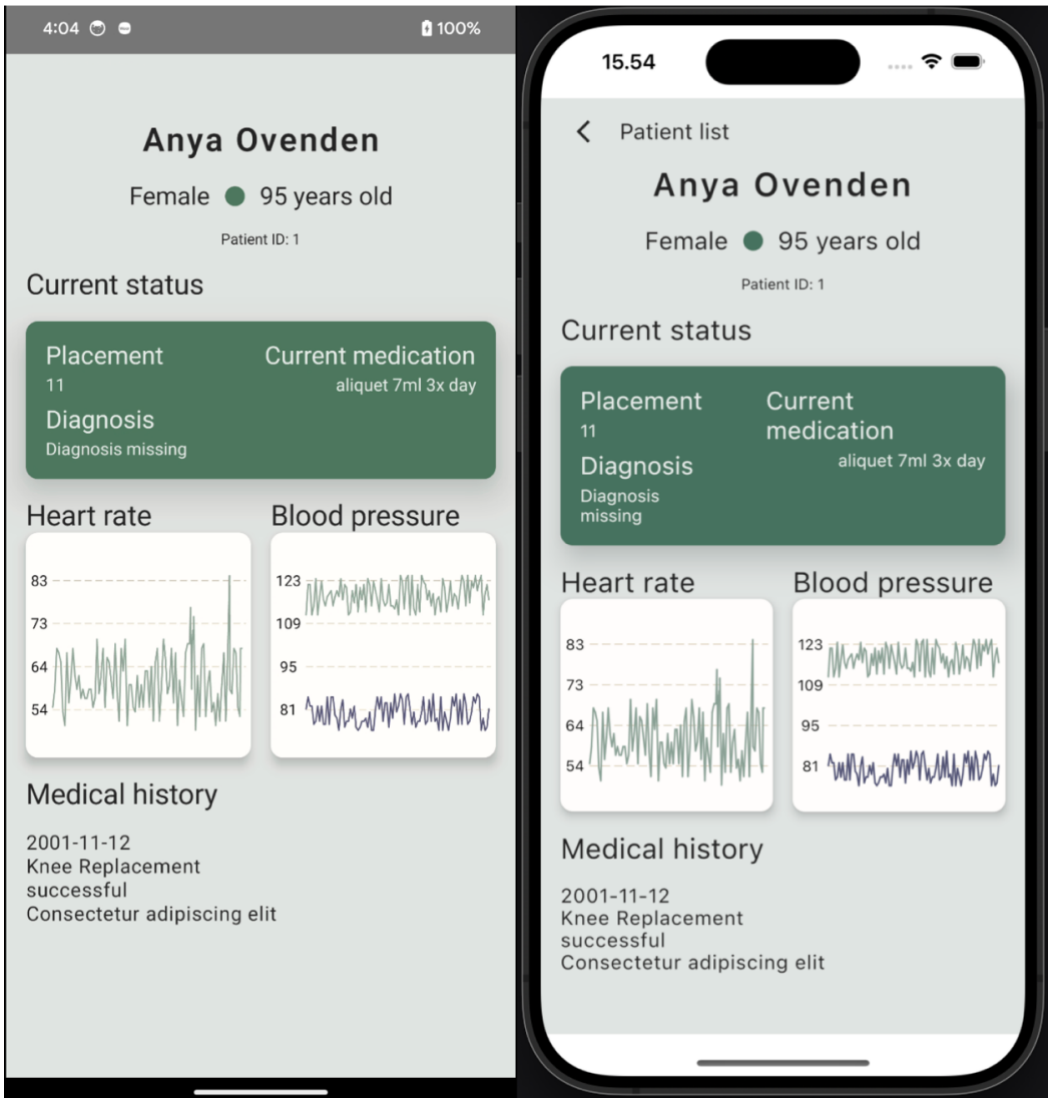

Figure 16 Android(on the left) and iOS applications

White space issue in iOS application can be easily fix by ignoring safe area explicitly on the native side with “ignoreSafeArea()” modifier, and making sure that safe area padding is taken into account by compose with “windowInsetsPadding(WindowInsets.safeDrawing)” modifier. Although fixing this takes only two lines of code, it is necessary to coordinate fix in two places which introduces another level of complexity to the problem.

5.3.6 Platform Specific Implementation

As described earlier, Kotlin Multiplatform was not designed to abstract platform specific code by providing common solution to platform specific problem. Instead it facilitates development by providing option to seamlessly integrate platform specific implementation in multiplatform project. This is also the case for Compose Multiplatform.

The whole reason for QR code scanning requirement was to evaluate simplicity of integrating platform specific functionality into KMP and Compose Multiplatform. Compose Multiplatform is not as mature framework as Flutter or React Native, therefore it lacks ready-made solutions provided by community. This created perfect opportunity to test out how easy it is to deal with the mobile specific problems using chosen technologies.

JetBrains (2024) provide documentation on how to handle such scenario. Development of QR code feature, was guided by the official documentation, however, it is not up to date and it lead to failure. This issue was raised by several users in Compose Multiplatform Github.

To be precise and fair, it is necessary to mention that integration of Android specific code and Jetpack Compose implementation is so simple and seamless that it does not need its own chapter in documentation.

With some trial and errors implementation of iOS specific implementation was ultimately a success. Since this thesis is not about iOS development specifically, let us skip SwiftUI implementation of QR code scanner and focus on integrating the feature in shared code.

In order to integrate platform specific view into shared UI code it is necessary to declare expected function that represents it. The function takes two parameters, modifier, so that it is possible to influence view when using it, and a callback, that provide shared codebase with scanned code. Expected function then needs to be actualized in platform specific source sets.

```
@OptIn(ExperimentalForeignApi::class)
@Composable
actual fun Scanner(modifier: Modifier, passResult: (String) -> Unit) {
    UIKitViewController(
        modifier = Modifier,
        factory = { ScannerFactory.shared!!.makeController(passValue = { it: String
            Napier.i {
                "Reading in Compose Multiplatform Passed value $it"
            }
            passResult(it)
        }) },
    )
}
```

Figure 17. Actual implementation of scanner in iOS codeset.

The SwiftUI integration trick is enabled using “UIKItViewController” which expects a Factory responsible for creating iOS view. Since, it is impossible to access swift code from shared code, factory needs to have its abstraction in Kotlin code. It is achieved by declaration of interface in iOS source set. That Interface is then implemented inside iOS platform specific code.

```
interface ScannerFactory {
    companion object {
        var shared: ScannerFactory? = null
    }

    fun makeController(passValue : (String) -> Unit): UIViewController
}
```

Figure 18. Interface of “ScannerFactory”

```
import UIKit
import ComposeApp

class TheFactory: ScannerFactory {
    func makeController(passValue : @escaping (String) -> () ) ->
        UIViewController {
        QRCodeScannerController(passValue: passValue)
    }
}
```

Figure 19.”ScannerFactory” implementation using iOS native code.

Finally, the factory needs to call “QRCodeScannerController”, class that implements “UIHostingController” which wraps SwiftUI “ScannerView”.

```
import SwiftUI

class QRCodeScannerController: UIHostingController<ScannerView> {
    // Optionally, you can add custom initialization or configuration here

    init(passValue : @escaping (String) -> () ) {
        super.init(rootView: ScannerView(qrCodeAction: passValue))
    }

    // Required initializer when subclassing UIHostingController
    @objc required dynamic init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder, rootView: ScannerView(qrCodeAction: {_ in }))
    }
}
```

Figure 20. “QRCodeScannerController” native wrapper.

Not accurate documentation, and juggling between two different development environments, and programming languages significantly complicates process of integrating native UI code into Compose Multiplatform. Understanding how all the pieces fit together becomes a challenging puzzle. This complexity makes KMP and Compose Multiplatform less beginner-friendly.

Integration of Compose Multiplatform with native UI frameworks, enables it to be utilized as a component library for the development of two native applications.

5.3.7 Positives

Despite encountering some challenges and minor issues with Kotlin Multiplatform and Compose Multiplatform, the overall experience with this technology stack has been rather smooth, efficient and effective. KMP serves as a great facilitator of cross-platform development, consistently enabling problem-solving without a significant roadblocks, although occasional slowdowns may be encountered.

For developers familiar with declarative user interface frameworks like SwiftUI or React, transitioning to Compose Multiplatform is generally seamless. Particularly for those experienced with Jetpack Compose and native Android development, the shift to KMP + Compose Multiplatform tech stack is nearly effortless.

It is noteworthy that Compose Multiplatform currently includes experimental features from Jetpack Compose such as “PullRefresh”, which adds an element of positivity and excitement to the development landscape. Additionally, the ongoing updates and improvements to the

technology further contribute to its promising outlook, with significant advancements observed at the time of writing this thesis.

5.3.8 Development Summary

Although during the implementation of the project minor unexpected challenges were encountered including technical difficulties such as not working previews or lack of common tools in common code, for instance missing “String.format()” function. Additionally, there were issues with missing or inaccurate documentation. Despite these obstacles, project implementation was executed exactly as planned, with extra two days ahead of schedule.

Given that the application included a platform-specific QR code scanning feature, four different screens, API integration, and custom charts, it's reasonable to consider the delivery of two mobile applications within the project timeframe as success.

In conclusion, despite unexpected issues, the successful execution of the project demonstrates the resilience and possibilities of Kotlin Multiplatform paired with Compose Multiplatform.

5.3.9 Development Experience Summary

Working with Kotlin Multiplatform and Compose Multiplatform offers rather a smooth developer experience for developers that have been working with Kotlin, Android and specifically Jetpack Compose. Let us rate each individual aspect of Kotlin Multiplatform in scale from 1 to 10 to provide a good overview of development experience. Please keep in mind this rating is concluded by person having experience with each of technologies mentioned earlier.

Code reuse: 8/10.
One of the key aspects of cross-platform technology is its ability to facilitate code reusability. With the emergence of the recently developed UI framework Compose-Multiplatform, Kotlin Multiplatform now offers extensive opportunities for code reuse. It is no longer a framework that only allowed sharing of data and business logic layers; it has evolved into a comprehensive cross-platform framework.

Pure Kotlin is capable of handling almost every aspect of a common code project, including data management, networking, and both business and presentation logic layers. In cases where Kotlin falls short such as platform necessary implementations, the expect/actual mechanism provides a straightforward simple to implement solution.

Regarding user interface, Compose Multiplatform seamlessly integrates with Jetpack Compose and is also compatible with UIKit and SwiftUI. However, achieving interoperability with iOS frameworks can initially be challenging due to the lack of direct guidance and comprehensive documentation.

Development speed 8/10

The development speed rating is significantly boosted by the presence of Compose Multiplatform, which builds upon the foundation of the well-established Jetpack Compose. Other than that, expect/actual mechanism is working for the advantage of the result. As mentioned before, development speed using same tech stack choices as in this project will be significantly faster for developers experienced with Kotlin and Jetpack Compose. Through efficient code sharing and platform-specific abstractions, Kotlin Multiplatform indeed empowers developers to deliver features rapidly.

On the other hand, integrations of iOS platform specific features will pose a significant setback for the developers which lack proficiency in iOS development. In addition, at first it is challenging to integrate iOS UI elements into Compose Multiplatform.

Overall productivity 7/10

Issues with previews, relatively small community resulting in not so rich assortment of libraries, missing documentation are all reducing the rating of KMP in terms of overall productivity.

5.4 Testing

Developers who strive to achieve great reusability of the shared code are guided by Kotlin Multiplatform to ensure proper separation of concerns. This is due to the fact that in order to be able to share big part of code across platforms, codebase cannot be entangled with platform specific classes which usually results in testing difficulties.

In addition, following strictly an architecture pattern that separates UI code with presentation logic helps to write testable code. Summing up these two KMP can enable developers to easily achieve great deal of testability.

The assurance of proper app behaviour is crucial in software development, particlarly concerning the user interface. 29th February 2024 and release of Compose Multiplatform 1.6.0 brings a significant improvement in terms of user interface testing. In the most recent version of Compose Multiplatform the UI testing allows for the verificatioon of expected behavor of components. Utilizing familiar tools such as finders, actions and matchers akin to those in Jetpack Compose for Android, developers can now validate behaviour of their UI seamleslly (Aigner and Zamulla, 2024).

However, teams launching KMP projects still have to take into account the scarcity of proven testing tools. For instance KMP lacks trusted mocking library such as Mockk in Android or Mockingbird for iOS.

6 Possible Future Improvements

Health Rundown certainly is not industry level mobile application offering rich set of features. It serves primarily as a demonstration app created for the purpose of evaluating technology. Application surely could welcome plenty of features to further present the possibilities of technology and expertise of beneficiary of this thesis. However, it is essential to note that the current state of the app suffices for the objectives outlined in this thesis.

Some of the possible improvements include:

- Although out of scope of mobile application development, server improvements and thus data enrichment is necessary to further develop the project.
- Pagination – currently the mobile application is not supplied with heavy data sets. However, if patient data would be closer to real world scenario application with current implementation would inevitably choke. It is because this implementation all Patient Details are fetched at once. Pagination would include fetching of detailed data only when it’s essential, such as heart rate and blood pressure by date. Additionally, list of all patient should be paginated. This feature was not considered for this thesis project since it would require additional work on the server side, which was out of scope of this thesis.
- Accounts and security – In real world scenario perhaps doctors would have access only to their own patients. Data access rights and filtering implementation would depend on accounts.
- Real time updates of data – patient information could be updated in real time whenever patient is being inspected in the application. It would bring application closer to being a health monitoring app.

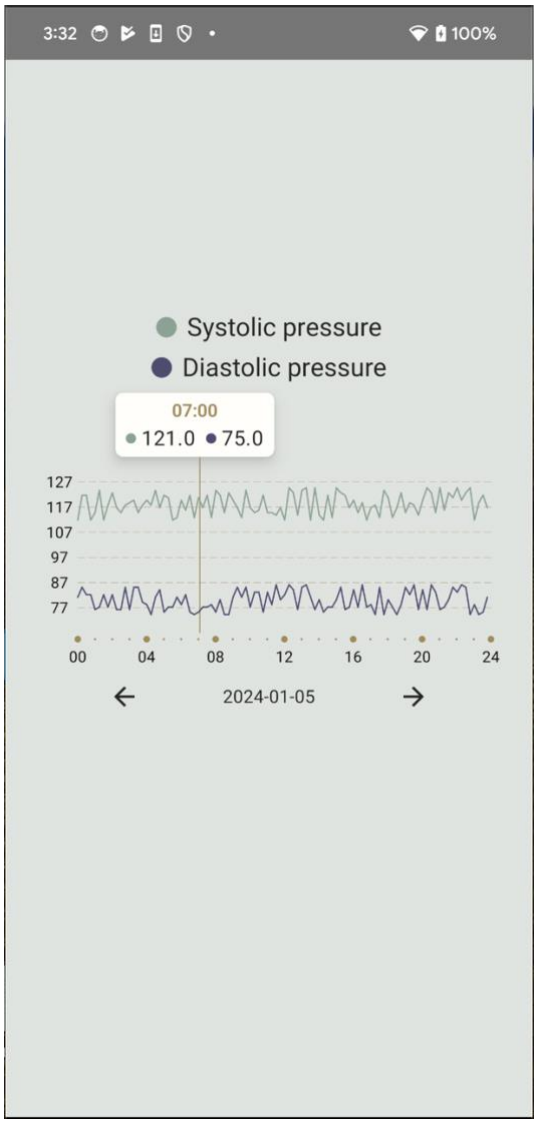

Figure 21. Charts without pagination.

7 Data Collection and Analysis

Code sharing capability is one of the most important aspect of cross-platform framework. It decreases needed development time and reduces required maintenance efforts. It serves as a prime motivation for both developers building such frameworks and those who use them. It would be unreasonable to not include an analysis of code that is written in Kotlin and extra code needed to build iOS application. Thus, ”cloc” tool was utilized to measure lines of code in the project.

Language	files	blank	comment	code
Kotlin	73	473	73	3364
Swift	11	81	76	319
SUM:	84	554	149	3683

Figure 22. Sum of lines of code written in Kotlin and Swift.

As it turns out under 9% of the whole codebase of Health Rundown is written in Swift programming language. Since, QR code feature is the only feature that requires platform specific code, without measuring details of Kotlin code we can safely assume that there is a similar percentage of Kotlin code that is written specifically for Android. This means that over 80% of code is completely shared between platforms. This finding shows efficiency of the framework, confirming its robust cross-platform compatibility that increases code maintainability and analyzability.

In addition, to establish performance and efficiency of Kotlin Multiplatform and Compose Multiplatform, Health Rundown was benchmarked across various metrics including Scroll Performance, Startup Latency, and App size.

Benchmarking involves examining and monitoring application performance. Regularly conducting benchmarks makes it possible to asses and troubleshoot performance issues, ensuring application quality. In case of this thesis benchmarking is used to evaluate if KMP and Compose Multiplatform introduce additional overhead when producing iOS and Android applications.

Battery usage was excluded from the benchmarking since application does not perform any heavy long process tasks that could significantly drain the battery.

7.1 Android Application

Android application benchmarking is performed by writing automated benchmarking tests. As carefully documented in Android Developers (2024), macro benchmarking covers testing of app startup, power consumption and complex UI operations such as scrolling. App size benchmarking is performed manually.

Android version of Health Rundown was compared with Compose-ShoppingList (Github, 2024), a native Android application developed using Jetpack Compose.

Data collection was performed on Google Pixel 6 Pro.

7.1.1 Startup Latency

Android OS categorizes application startup into three states: cold, warm and hot. Cold startup takes the longest since application starts from scratch, warm and hot startups are faster because they are just relaunching app from the background to foreground (Android Developers, 2024).

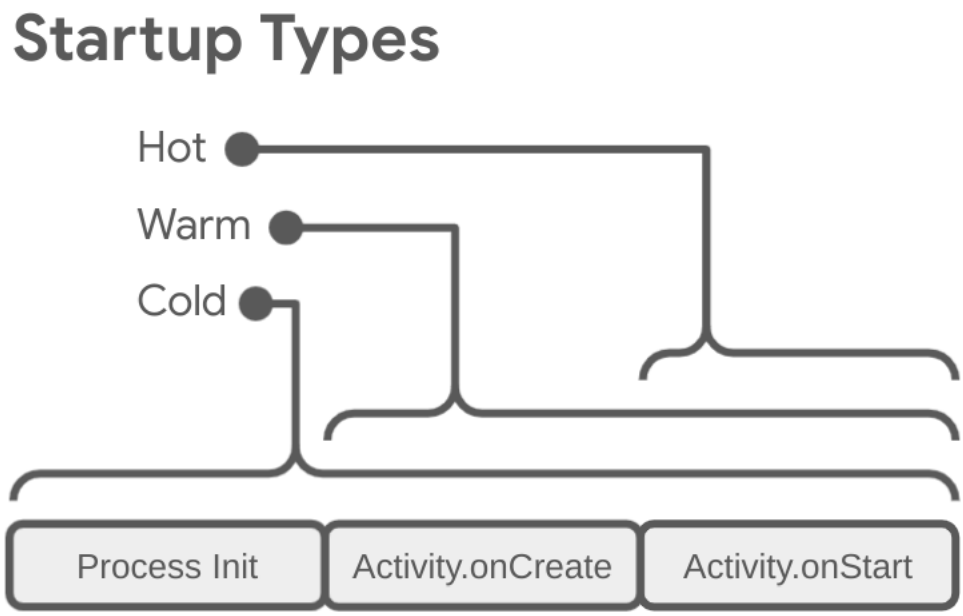

Figure 23. Diagram with startup types.

The following tables present startup times of the Health Rundown Android application, and Compose-ShoppingList app with all measurements expressed in milliseconds. The data was collected over five iterations using an automated tests.

	Minimum time	Median time	Max time
Hot	19.0	26.4	29.7
Warm	54.0	58.6	65.9
Cold	185.9	192.8	206.4

Table 1. Android Health Rundown startup times.

	Minimum time	Median time	Max time
Hot	18.5	25.2	32.2
Warm	48.7	56.8	60.8
Cold	204.6	216.2	244.4

Table 2. Android Compose-ShoppingList startup times.

Upon analyzing results above, it is apparent that Health Rundown does not present additional delay when launching in comparison to native Android application. According to Android Developers (2024), Android vitals determine app startup as an excessive if:

- Cold startup takes longer than 5 seconds
- Warm startup takes longer than 2 seconds
- Hot startup takes longer than 1.5 seconds

Both KMP and native application stay withing these limits. Notably, the compact sizes of these applications significantly contribute to this outcome, keeping them well below the suggested thresholds.

Therefore, based on these finding it is safe to conclude that KMP and Compose Multiplatform do not introduce any overhead in terms of application startup.

7.1.2 Scroll Performance

Scrolling performance was measured using “FrameTimeMetric” which captures timing information from frames. It produces “frameOverrunMs”, which is the amount of time a frame misses its deadline by. Positive number indicate dropped frame or lag, and negative frame indicates faster performance than expected. It also produces “frameDurationCpuMs” – the amount of time frame takes to be produced on the CPU.

Measurements are collected in a distribution of 50th, 90th, 95th, and 99th percentile.

Scroll performance of Health Rundown was measured in Patient List screen.

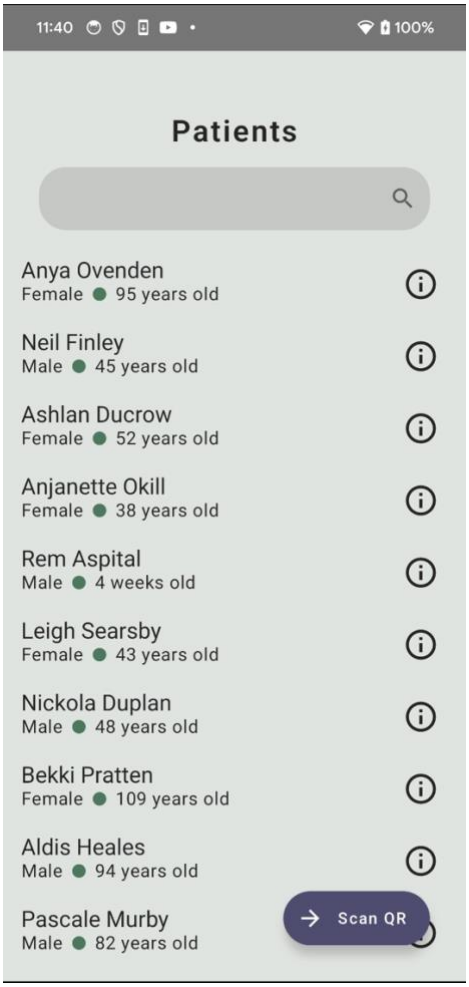

Figure 24. Patient list screen

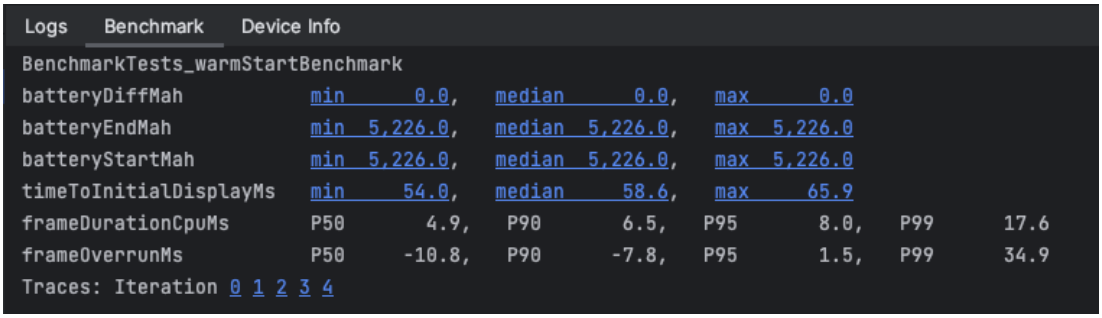

Figure 25. Health Rundown benchmark.

The performance analysis of KMP application reveals that its “frameDurationCpuMs” median duration stands at 4.9 milliseconds with 90% of the frames processing within 6.5 milliseconds

and 95% within 8 milliseconds. The 99th percentile indicates potential lags with framing taking 17.6 milliseconds. On the other hand, frame overrun “frameOverrunMs” shows efficient resource usage, with negative values until 90th percentile assuring frames competition ahead of schedule. The median overrun is -10.8 milliseconds, with 90% of frames finishing 7.8 milliseconds before the deadline. 95th percentile shows insignificant lateness of 1.5 milliseconds, and the 99th percentile reveals late frames processing of 34.9 milliseconds. Overall the analysis shows efficient processing for most of the frames, while also identifying potential for improvement in higher percentiles.

The native application, shows a median frame duration of 5.3 milliseconds, with 90% of frames completing within 6.6 milliseconds and 95% within 7.3 milliseconds. However there are also notable glitches particularly in 99th percentile where frame processing takes 25.7 milliseconds. The frame overrun indicates efficient resource usage, with 95% of frames being completed before the deadline.

```
ExampleStartupBenchmark_scrollTest
frameDurationCpuMs   P50      5.3,   P90      6.6,   P95      7.3,   P99     25.7
frameOverrunMs       P50    -10.2,   P90     -8.7,   P95     -6.4,   P99     36.5
Traces: Iteration 0 1 2 3 4
```

Figure 26. Native application scroll performance

In conclusion scroll performance analysis reveals efficient frame processing for both the Health Rundown and native Compose-ShoppingList apps, with minor discrepancies observed in the 99th percentile. Despite these differences, overall performance remains satisfactory in both cases, suggesting that Android application built with KMP and Compose Multiplatform are as efficient as native Android applications

7.1.3 App Size

The Health Rundown application's file size is 2.8MB, which is relatively small. This comes as no surprise, given that the application comprises only three features and four screens, and does not include any heavy resources.

7.2 iOS Application

Development phase benchmarking is not as easy as when it comes to iOS as it is in Android. Apple enables great deal of metrics out of the box for the applications that are already in App Store and have active users. Reading from Apple Developer (2024), vendor suggest usage of XCode Organizer to read metrics. It is a problem for proof of concept applications that are in pre-store development phase, in such scenario there is no data to analyze.

Health Rundown iOS benchmarking data collection was performed manually, using XCode Instruments. Instruments is a tool that is built into XCode, its purpose is to profile and analyze app performance along with investigating system resource usage. It is a perfect workaround given lack of proper automated tooling.

To benchmark Health Rundown, open-source project InfiniteListSwiftUI (Github, 2024) was selected as a reference. It was chosen because of its similarity with Health Rundown in terms of simplicity and scale. This ensures relatively accurate and relevant comparison.

Data was collected on iPhone 12.

7.2.1 Startup Latency

An application running on iPhone is launched in a warm or cold manner. Similarly to Android, cold launch occurs when system can’t resume the application but rather starts it anew. Warm startup happens when user re-enters the application (Apple Developer, 2024).

At WWDC 2019, Apple recommended a startup limit of 400ms for optimal performance of a top-notch iOS application.

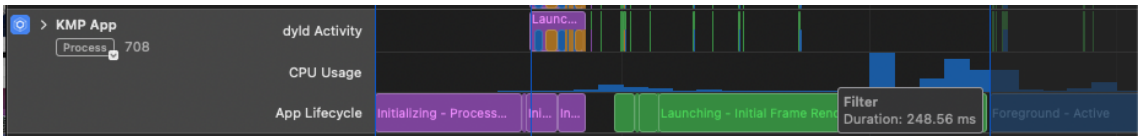

Figure 27. Profiling iOS Cold Startup using XCode Instruments

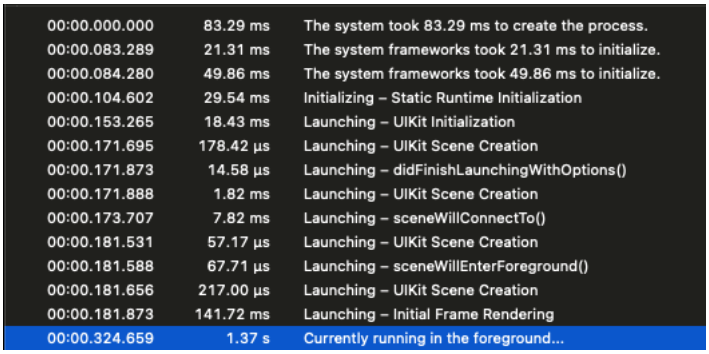

Figure 28. Instruments App Lifecycle timeline

Cold startups - Health Rundown measurements: 248.56ms, 253.20ms, 251.23ms, 217.92ms,324.66ms

The table below presents the cold startup times of the Health Rundown application. Warm startup data is not included due to limitations in the information provided by the Instruments tool. All values are measured in milliseconds.

	Minimum time	Median time	Max time
Cold startup - Instruments	217.92	251.23	324.66

In order to understand this result, let us measure time to initial launch of InfiniteListSwiftUI app. Cold startups, InfiniteListSwiftUI: 167.34ms, 143.96ms, 79.89ms, 118.42ms, 113.31ms.

The table below represents cold startup times of InfiniteListSwiftUI. All values are measured in milliseconds

	Minimum time	Median time	Max time
Cold startup - Instruments	79.89	118.42	167.34

In conclusion, with a target of launching the Health Rundown app within 400 milliseconds on an iPhone, the small proof-of-concept application comfortably meets this requirement. However, it is noteworthy that the maximum cold startup time is rather close to the upper limit suggested by Apple. Furthermore, comparison with native application suggests that application developed with KMP and Compose-Multiplatform require more time for a startup when compared to native applications.

7.2.2 Scroll Performance

The scrolling measurement was conducted manually utilizing Instruments. The test involved launching the application, followed by a single downward swipe gesture and concluded with an upward swipe gesture.

Below are performance measurement from Health Rundown and InfiniteListSwiftUI. Due to limitations within Instruments, a direct comparative assessment of the results is challenging. Particularly, the absence of a standardized method for result comparison complicates the evaluation. Furthermore, the lack of documentation regarding values provided by Advanced Graphics Statistics, such as Device Utilization % or Render Utilization %, adds to the complexity of the analysis. Therefore, the comparison of these performance metrics relies largely on intuitive interpretation rather than a structured methodology.

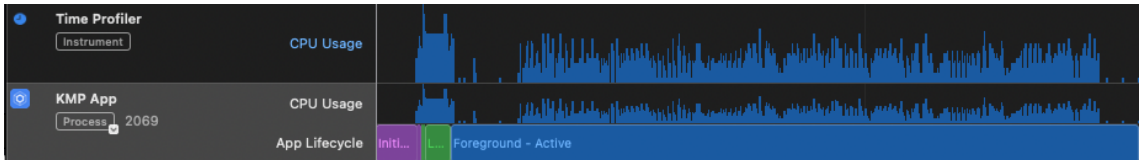

Figure 29. Health Rundown iOS profiling

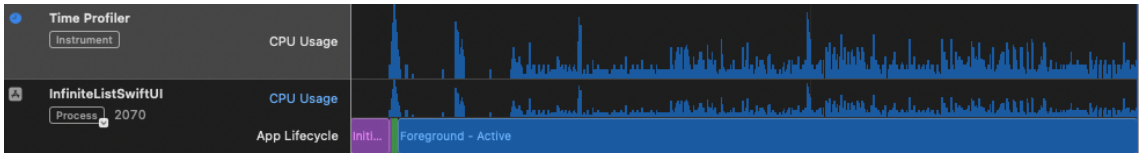

Figure 30. InfiniteListSwiftUI profiling

Graphics Driver / Statistic	Sum	Min Value	Avg Value	Std Dev Value	Max Value
* All *	2 070 348 197	0	34 505 803	70 946 027	232 112 128
Built-In	2 070 348 197	0	34 505 803	70 946 027	232 112 128
Alloc system memory	1 259 765 760	203 784 192	209 960 960	11 387 707	232 112 128
Allocated PB Size	51 904 512	6 029 312	8 650 752	1 284 238	9 175 040
CoreAnimationFramesPerSecond	169	0	28	14	40
Device Utilization %	85	6	14	12	39
In use system memory	755 466 240	99 745 792	125 911 040	39 180 140	180 699 136
recoveryCount	0	0	0	0	0
Renderer Utilization %	83	6	14	11	38
SplitSceneCount	0	0	0	0	0
TiledSceneBytes	3 211 264	491 520	535 211	107 019	753 664
Tiler Utilization %	84	6	14	12	39

Figure 31. Health Rundown iOS graphics statistics

Graphics Driver / Statistic	Sum	Min Value	Avg Value	Std Dev Value	Max Value
* All *	1 907 474 849	0	27 249 641	58 833 953	211 058 688
Built-In	1 907 474 849	0	27 249 641	58 833 953	211 058 688
Alloc system memory	1 261 240 320	155 992 064	180 177 189	16 175 404	211 058 688
Allocated PB Size	33 030 144	4 718 592	4 718 592	0	4 718 592
CoreAnimationFramesPerSecond	214	0	31	24	60
Device Utilization %	68	0	10	10	33
In use system memory	609 501 184	50 036 736	87 071 598	38 236 632	170 475 520
recoveryCount	0	0	0	0	0
Renderer Utilization %	67	0	10	10	33
SplitSceneCount	0	0	0	0	0
TiledSceneBytes	3 702 784	491 520	528 969	99 081	753 664
Tiler Utilization %	68	0	10	10	33

Figure 32. InifiniteListSwiftUI graphics statistics

However, it is a fundamental knowledge of computer science to understand that the less system memory program allocates, the easier it is for the device to run it. Furthermore, it is reasonable to assume that the lower device, render and tiler utilization indicate a more optimized application. Hence, it is safe to assume that natively developed InfiniteListSwiftUI performs better than KMP based Health Rundown.

7.2.3 Extra Performance Discovery

Delays in app interactions, known as hangs, can disrupt user experience and lead to frustration. Delays shorter than 100ms feel as instant to the user, above 250ms are clearly noticeable and therefore reported by Instruments as hangs (Lin Tun, 2023).

During the performance evaluation of Health Rundown, a notable discovery was made. I It was found that the text field used for filtering the list of patients not only introduces significant hangs but also causes unnecessary recompositions of the user interface even after its usage. While the application can be easily fixed by disabling animations, such workaround will likely be noticeable to users.

Figure 33. Hangs when using text field.

7.2.4 App Size

KMP and Compose Multiplatform introduce a major overhead in terms of app size. Comparatively, the InfiniteListSwiftUI application occupies less than half a megabyte of disk space, while Health Rundown requires a substantial 58.2 megabytes.

The issue of an app size has a significant impact when it comes to publishing app to end users on the app store.

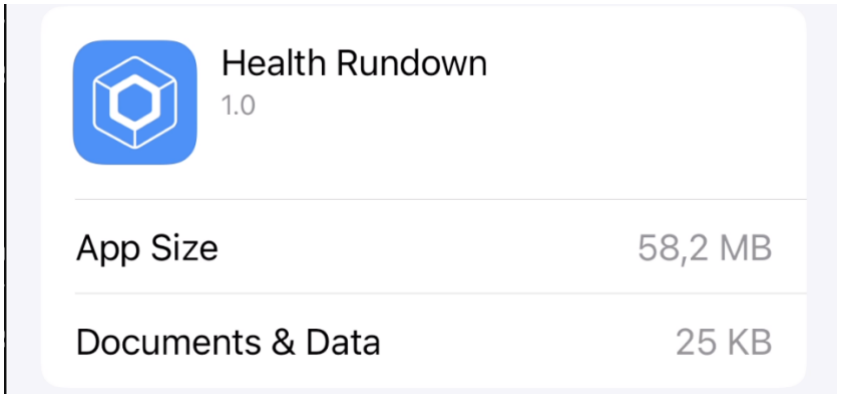

Figure 34. Health Rundown disk space

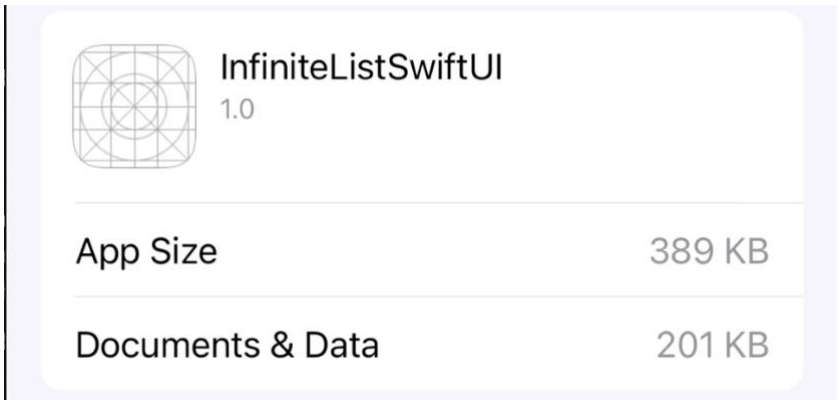

Figure 35. InfiniteListSwiftUI disk space.

8 Conclusion

Kotlin Multiplatform is undeniably interesting technology that can be paired with several different technologies. In case of this study it was linked with Compose Multiplatform for the sake of increasing proportion of shared codebase.

KMP with Compose Multiplatform is almost identical to native Android development, making it exceptionally easy for Android Developers. In addition, Android application does not seem to include any drawbacks when implemented with technology in question. A really good positive of this technology it is to being able to build iOS application with very little additional effort – that of course depending on the amount of platform specific features. Hence, Kotlin Multiplatform might be a very tempting choice for Android Developers who start new project, or who want to migrate their existing codebase in order to receive an iOS as a by-product.

When it comes to iOS application, Kotlin Multiplatform introduces drawbacks that teams deciding for usage of this technology must agree to. Furthermore, there is no other choice for cross-platform user interface framework compatible with KMP other than Compose Multiplatform which considering its performance overhead, lack of documentation and unexpected issues does seem dangerous for serious projects. Certainly it is necessary to mention that project can be structured differently so that Kotlin Multiplatform would take care of facilitating shared codebase apart of presentation layer, leaving it for the native implementations. In such scenario there is definitely less disadvantages when it comes to performance and quality of an application.

Given the simplicity of integrating Compose Multiplatform code into SwiftUI and vice versa, it would be beneficial to conduct another study to develop an application based on Kotlin Multiplatform. This application would leverage all the benefits of two separate natively written UIs using Jetpack Compose for Android and SwiftUI for iOS. The study could explore enhancing percentage of shared codebase by implementing a shared library of components consumed by native UIs using Compose Multiplatform. Such approach could effectively leverage Compose Multiplatform while mitigating its drawbacks, such as the absence of native navigation, challenges with certain components, and performance overhead on iOS platforms.

References

Printed

Heitlager, I., Kuipers, T., Visser, J. A Practical Model for Measuring Maintainability – a preliminary report -. No Date.

Isakova, S., Jemerov, D. Kotlin in Action. 2017. Manning Publications.

Nagy, R. 2022. Simplifying Application Development with Kotlin Multiplatform Mobile. Packt Publishing.

Electronic

Adobe Communications Team, 2022. Project scope – definition, best practices, examples and more. Accessed 16 April 2024. https://business.adobe.com/blog/basics/project-scope-definition-best-practices-examples-and-more.

Agile Data. 2022. Vertical Slicing: The Key to Agile Data Warehousing. Accessed 17 February 2024. https://agiledata.org/essays/verticalslicing.html.

Agile Velocity. 2010. 7 principles of Lean Software Development. Accessed 21. February 2024. https://agilevelocity.com/7-principles-of-lean-software-development/.

Aigner, S., Zamulla, A. 2024. Compose Multiplatform 1.6.0 – Resources, UI Testing, iOS Accessibility, and Preview Annotation. https://blog.jetbrains.com/kotlin/2024/02/compose-multiplatform-1-6-0-release/. Accessed 1 March 2024

Android Developers. 2024. Android's Kotlin-first approach. Accessed 15 February 2024. https://developer.android.com/kotlin/first.

Android Developers. 2024. Design & Plan. Accessed 17 February 2024. https://developer.android.com/design.

Android Developers. 2024. Overview of measuring app performance. Accessed 23 February 2024. https://developer.android.com/topic/performance/measuring-performance.

Android Developers. 2024. App startup time. Accessed 25 March 2024. https://developer.android.com/topic/performance/vitals/launch-time.

Android Developers. 2024. Write a Macrobenchmark. Accessed 23 March 2024. https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview.

Apple Developer. 2024. Designing for iOS. Accessed 17 February 2024. https://developer.apple.com/design/human-interface-guidelines/.

Apple Developer. 2024. Reducing your app's launch time. Accessed 25 March 2024. https://developer.apple.com/documentation/xcode/reducing-your-app-s-launch-time.

Atlassian. 2024 Kanban. Accessed 17 February 2024. https://www.atlassian.com/agile/kanban.

Github, 2024. InfiniteListSwiftUI. Accessed 26 March 2024. https://github.com/V8tr/InfiniteListSwiftUI.

Github, 2024. Compose-ShoppingList. Accessed 28 March 2024. https://github.com/QArtur99/Compose-ShoppingList.

Gulya, I. 2022. Preview annotation in commonMain sourceSet. Accessed 29 February 2024. https://github.com/JetBrains/compose-multiplatform/issues/2045.

JetBrains, 2024. Kotlin Multiplatform. Accessed 14 February 2024. https://www.jetbrains.com/kotlin-multiplatform/.

JetBrains, 2024. Integration with the UIKit and SwiftUI frameworks. Accessed 16 March 2024. https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-ios-ui-integration.html.

Kotlin Journal. 2023. Native and cross-platform app development: how to choose?. Accessed 13 February 2024. https://kotlinlang.org/docs/native-and-cross-platform.html.

Lin Tun, K. 2023. Analyze hangs with Instruments. https://www.wwdcnotes.com/notes/wwdc23/10248/. Accessed 28 March 2024.

Schmitt, J. 2023. Native vs cross-platform mobile app development. Accessed 13 February 2024. https://circleci.com/blog/native-vs-cross-platform-mobile-dev/.

Straits research. 2021. Mobile App Development Market. Accessed 13 February 2024. https://straitsresearch.com/report/mobile-app-development-market.

YouTrack, 2024. JetBrains, Stdlib: String.format in common. Accessed 11 March 2024. https://youtrack.jetbrains.com/issue/KT-25506

Figures

Tables