React Native: A truly Native Experience?

Performance comparison of React Native and Swift

Dilan Zibari

Degree Thesis

Information Technology

2023

Lärdomsprov

Dilan, Zibari

React Native: En riktigt nativ upplevelse? - Prestandajämförelse mellan React Native och Swift

Yrkeshögskolan Arcada: Informationsteknik, 2023.

Identifikationsnummer:

9186

Sammandrag:

Den snabba utvecklingen av smarta enheter påverkar utvecklingen av många applikationer, som används idag. Många företag visar mer intresse för att utveckla sina egna mobila applikationer för att öka produktiviteten i sina affärsprocesser. Dessa applikationer är plattformsspecifika, vilket innebär att det behövs en separat applikation för varje plattform. Hybridapplikationer, även kallade plattformsöverskridande applikationer, ger en flexibel lösning för denna fråga. Hybridapplikationer kan användas på alla enheter oavsett vilken plattform det gäller. Tekniker som React Native lovar en lösning där utvecklare kan använda samma verktyg och teknologier på olika plattformar. Denna avhandling betonar vikten av högkvalitativa mobila applikationer med tanke på utmaningen med mobil utveckling. Denna avhandling fokuserar på att undersöka om React Native-lösningen är något för företag att överväga. Den tekniska experimentmetodiken som används för att uppnå forskningsmålen innebär att skapa två speglade applikationer, en React Native-baserad och den andra Swift-baserad. Uppsatsen diskuterar också de mått som används för att utvärdera prestandan för en iOS-applikation. Resultatet av experimentet analyseras sedan och presenteras i grafer. Den sista delen presenterar slutsatserna från denna avhandling inklusive resultaten som hittades under experimentet.

Nyckelord:

React Native, Swift, Mobil applikation, iOS, Xcode

Degree Thesis

Dilan, Zibari

React Native: A truly native experience? - Performance comparison of React Native and Swift

Arcada University of Applied Sciences: Information Technology, 2023.

Identification number:

9186

Abstract:

The rapid development of smart devices affects the development of many applications, which are used today. Many corporations are showing more interest in developing their mobile applications to increase their business process productivity. These applications are platform-specific, which means a separate application is needed for each platform. Hybrid applications, also called cross-platform applications provide a flexible solution for this matter. Hybrid applications can be used on all devices regardless of the platform in question. Technologies like React Native promise a solution where developers can use the same tools and technologies across different platforms. This thesis emphasizes the importance of high-quality mobile applications considering the challenge of mobile development. This thesis focuses on investigating whether the React Native solution is something for corporations to consider. The technical experiment methodology used to accomplish the research objectives involves creating two mirrored applications, one React Native based and the other Swift based. The paper discusses also the metrics used to evaluate the performance of an iOS application. The result of the experiment is then analyzed and displayed in graphs. The last part iterates the conclusions of this thesis including the results that were found during the experiment.

Keywords:

React Native, Swift, Mobile application, iOS, Xcode

Opinnäyte

Dilan, Zibari

React Native: Todellinen natiivi käyttökokemus? - React Nativen ja Swiftin suorituskykyvertailu

Arcada Ammattikorkeakoulu: Tietotekniikka, 2023.

Tunnistenumero:

9186

Tiivistelmä:

Älylaitteiden nopea kehitys vaikuttaa monien nykyään käytössä olevien sovellusten kehitykseen. Monet yritykset ovat kiinnostuneita kehittämään omia mobiilisovelluksiaan liiketoimintaprosessiensa tuottavuuden lisäämiseksi. Nämä sovellukset ovat alustakohtaisia, mikä tarkoittaa, että jokaiselle alustalle tarvitaan erillinen sovellus. Hybridisovellukset, joita kutsutaan myös monialustaisiksi sovelluksiksi, tarjoavat joustavan ratkaisun tähän asiaan. Hybridisovelluksia voidaan käyttää kaikilla laitteilla alustasta riippumatta. Re-act Nativen kaltaiset tekniikat lupaavat ratkaisun, jossa kehittäjät voivat käyttää samoja työkaluja ja tekniikoita eri alustoilla. Tämä opinnäytetyö korostaa laadukkaiden mobiilisovellusten merkitystä mobiilikehityksen haasteessa. Tämä opinnäytetyö keskittyy sen selvittämiseen, onko React Native -ratkaisu yritysten harkinnan arvoinen asia. Tutkimustavoitteiden saavuttamiseksi käytetty tekninen kokeilumenetelmä sisältää kahden peilatun sovelluksen luomisen, joista toinen perustuu React Native -pohjaiseen ja toinen Swift-pohjaiseen. Artikkelissa käsitellään myös iOS-sovelluksen suorituskyvyn arvioinnissa käytettyjä mittareita. Sitten kokeen tulos analysoidaan ja esitetään kaavioina. Viimeisessä osassa toistetaan tämän opinnäytetyön johtopäätökset mukaan lukien kokeen aikana löydetyt tulokset.

Avainsanat:

React Native, Swift, Mobiilisovellus, iOS, Xcode

Content

List of abbreviations and symbols **4**

Figures **5**

1 Introduction **6**

1.1 Structure 7

2 Background **8**

2.1 Motivation 8

2.2 Related popular cross-platform tools 8

2.3 What are mobile applications? 9

2.3.1 Native Applications 9

2.3.2 Web Applications 11

2.3.3 Hybrid Applications 11

2.4 Cross-Platform Tools 12

2.4.1 React Native programming language 12

2.4.2 Swift programming language 13

3 Problem **14**

3.1 The complexity of native applications 14

3.2 Research Questions 14

3.3 Related Research 15

4 Methodology **16**

4.1 Views for the experiment 18

5 Implementation **20**

5.1 Development process 20

5.1.1 React Native Development 20

5.1.2 Native iOS Development 21

5.1.3 Measurement environment 22

5.1.4 Result of the built mobile applications 22

6 Evaluation **24**

6.1 Results in a graph presentation 25

6.2 Results of application launch time 29

6.3 Application size difference 30

7 Proposal for future work **31**

8 Conclusions **32**

8.1 Summary 32

8.2 Discussion 32

9 References **34**

Appendix. Summary in Swedish **36**

List of abbreviations and symbols

ARC	Automatic Reference Counting is a way to track and manage the application's memory usage.
Android	Mobile operating system developed by Google. The name comes from the word "androids" which means a robot with a human-like appearance.
CPU	Central Processing Unit is the primary component of a computer system. Often referred to as the brain of the computer.
CPT	Cross-Platform Tools are software development tools for building applications that can run on multiple operating systems with a single codebase.
CSS	Cascading Style Sheets, a language for describing the style of a document written in HTML.
HTML	HyperText Markup Language, a markup language for creating websites and web applications.
IDE	Integrated Development Environment is a software application that provides a set of tools and features for software development.
iOS	iPhone Operating System, a mobile operating system developed by Apple.
MVC	Model-View-Controller is an architectural pattern commonly used for developing user interfaces. Divides the application into three interconnected parts.
MS	Milliseconds, a unit of time measurement.
OS	Operating System is a software program that allows the computer hardware and the software applications to communicate with each other.

Figures

Figure 1. The layer hierarchy in the iOS architecture 10
Figure 2. The architecture in web applications.. 11
Figure 3. The architecture in hybrid applications... 12
Figure 4. Measurement tools for the performance of an iOS application 16
Figure 5. Instrument tools in XCode 17
Figure 6. Profiling templates in XCode.. 18
Figure 7. The first displaying view of the experiment 19
Figure 8. The second view of the experiment 19
Figure 9. The last view 20
Figure 10. The application in React Native.. 23
Figure 11. The application in Swift... 23
Figure 12. Graph presentation of the results 2 elements 25
Figure 13. Graph presentation of the results with 5 elements 26
Figure 14. Graph presentation of the results with 10 elements 26
Figure 15. Graph presentation of the results with 100 elements 27
Figure 16. Graph presentation of the results with 500 elements 28
Figure 17. Presenting all result data in one final table 29
Figure 18. Application launch time results.. 29

1 Introduction

Mobile applications, or as we say "apps”, have become a ubiquitous part of our daily lives. Allowing us to perform a wide range of tasks from anywhere at any time. Whether we want to check the news, order food, or monitor our fitness goals, mobile apps have transformed the way we interact with technology. Irrespective of what you want or need to do; everything is simply at your fingertips.

It’s usually required that a mobile application is available and distributed in both the AppStore and the Google Play Store. To achieve this, two different applications need to be developed natively with the same design and functions but in two different programming languages. Due to this demand, companies must hire two different developer teams, one for each platform to reach the required audiences.

As a response to the complexity of native development, cross-platform development has emerged. Cross-platform makes it possible to reuse the same codebase which means that only one implementation of the code is required. Over the years, different frameworks have been used but studies indicate that the end-users are not as satisfied with cross-platform applications as they are with the native application. Studies indicate that cross-platform applications are more prone to complaints due to the performance being worse compared to their native peer. (Nitze, Rösler & Schmietendorf, 2014)

However, new technologies and frameworks are constantly being created. One of these frameworks is called React Native. In the thesis a technical experiment will be conducted, aiming to elaborate further on the performance of React Native. Based on the results of the experiment we will try to assess if a React Native application could be recommended as a viable option for developers who consider using it.

1.1 Structure

The rest of the thesis is structured as follows: Some background information is presented in chapter 2, including the motivation for the study and the popular cross-platform frameworks. The chapter also discusses the different types of mobile applications in the market today and introduces the React Native and Swift programming languages.

In Chapter 3 the problem is explained, including the complexity of native applications, the research question, and some related work that has previously been done. Chapter 4 focuses on the methodology used in the study, including the functionalities to be tested and the environment. In chapter 5 the implementation of the research is presented. As well as discussing the development process of both applications, conducting the experiment, and presenting the results. Chapter 6 evaluates the research results with a graph presentation.

In chapter 7 proposals for future work are iterated and explained. Chapter 8 presents the conclusion of the thesis by summarizing the key findings and further discussing the subject.

2 Background

2.1 Motivation

The global app market is rapidly changing in today's industry and one of the biggest factors to consider is the user experience when it comes to using mobile applications. Around 92% of mobile users spend time with apps and social media, and only 8% use web browsers (Fireart Studio, 2022). Looking back at only 3 years ago the global mobile app market was valued at $154 billion, and it is expected to continually grow in the coming years (Linn, 2023).

Taking this into consideration it's now more important than ever for companies to deliver high-quality mobile applications. Increasing their revenues requires being available on all platforms so that they won't miss the chance of attracting clients via all the various channels.

However, one of the unique challenges that mobile development brings to the market is the efficiency of rapidly developing and maintaining them. Unfortunately making the service accessible on all platforms is very costly due to differences in syntax, language, test suites, and packages in the different platforms. Furthermore, developing native mobile applications for each platform drastically increases the development costs of the project.

2.2 Related popular cross-platform tools

In January 2022 the Stack Overflow community conducted its yearly survey of over 100,000 of their professional developers. The survey consists of many questions regarding their everyday job. Looking at the most interesting matter, namely the most popular frameworks. In that list, we can observe that Flutter places 6th, right after React Native. (Stack Overflow, 2022)

Flutter is an open-source framework by Google that uses the Dart programming language. It is today the second most popular framework which has been gaining

popularity since 2021, mainly known for its fast development cycle. It is said to be overtaking React Native and is today used by already 39% of developers globally. (Fire-art Studio, 2022)

It is also interesting to note that Cordova and Xamarin were popular enough to be included in the list. Xamarin is one of the oldest cross-platform frameworks available, founded in 2011. Allowing developers to create native applications for Android, iOS, and Windows platforms, with one single codebase. Cordova on the other hand, is a hybrid open-source framework that enables web developers to use their HTML, CSS, and JavaScript knowledge to build applications. However, Cordova and Xamarin are among the most dreaded frameworks that were included in the survey meaning that developers who have used the frameworks, do not wish to continue using them in the future. (Madeshvaran, 2019)

2.3 What are mobile applications?

There are three types of architectures used in mobile application development: web, native and hybrid applications.

2.3.1 Native Applications

Native Application development is one of the traditional ways to develop an application. They are based on a specifically targeted platform language, which makes them bound to the platform they're designed for. This means that if you write Android apps it will only function on that operating system. If you then eventually decide to target Apple devices as well, your team will have to write completely new code from scratch. The development IDEs are limited to specific tools as well, Android Studio for the Android developing platform and XCode for the iOS developing platform.

The iOS architecture is made up of four layers that, from bottom to top, provide increasingly important services to help the application communicate with the device hardware. Each layer has its own set of responsibilities, where the higher levels contain more sophisticated types of services, while the lower layers contain the necessary technologies.

1	Cocoa Touch Layer
2	Media Layer
3	Core Services
4	OS Layer

Figure 1. The layer hierarchy in the iOS architecture

The Cocoa Touch layer, which is the highest-level component in the iOS component hierarchy, acts as the service in-between the user application. This layer provides all the essential infrastructure that is needed for iOS development. It controls how the applications look and how they respond when we interact with them.

The Media Layer component contains the information for multimedia features such as audio, video, and other graphics of the device. It is the layer that makes the picture on the iOS device crisp and the song we're listening to sound clear. (Besant Technologies)

The Core Service layer includes all the fundamental system tools and services that an application use directly or indirectly. It could be such as accessing or storing data on a smartphone device.

The last component, the OS Layer, provides the underlying system services for iOS, that the other component on the device relies upon. It handles for instance the CPU, memory management, files, and drivers. (GeeksforGeeks 2023)

2.3.2 Web Applications

Web applications mainly focus on Internet technology which is built using web technologies like HTML, CSS, and JavaScript. Businesses running on a budget often stick to web applications, due to the minimal cost of production.

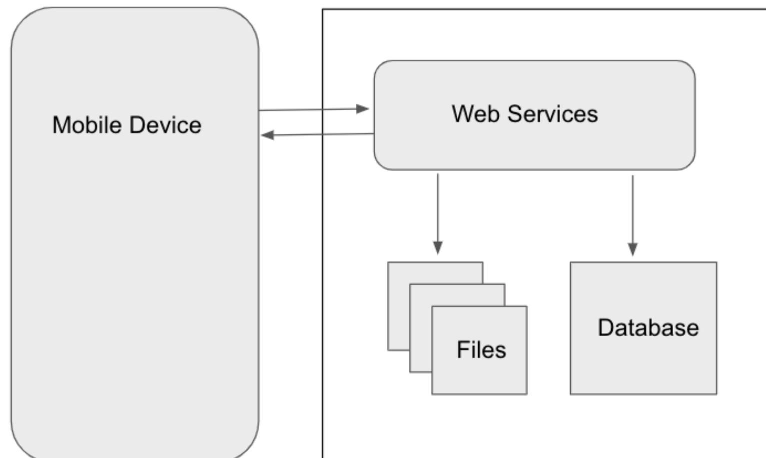

Figure 2. The architecture in web applications

Web applications can be processed in a web browser, making them accessible on most devices with a consistent look and feel. The major drawback of these applications is the intense graphics and the problems that occur when the applications need to be adjusted to that graphics. These applications cannot be used without an Internet connection. (TechTarget, 2023)

2.3.3 Hybrid Applications

The hybrid applications combine the technologies of both a web and native-developed application. The applications are built with web-based technologies but embedded within a native container. The native container allows the OS on the mobile phone to handle the application and can thereby be distributed and installed through the app store. (TechTarget 2023)

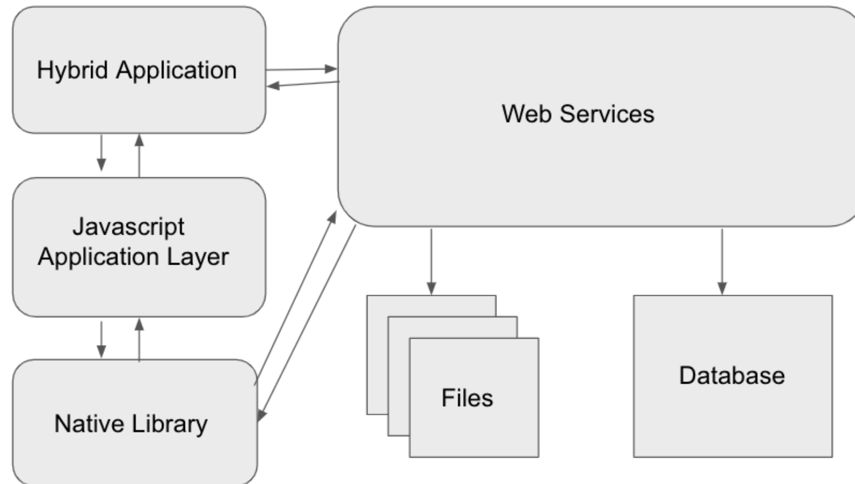

Figure 3. The architecture in hybrid applications

2.4 Cross-Platform Tools

Mobile Cross-Platform tools aim to share a significant part of the codebase between different platforms. This kind of technique can drastically reduce both the time and development costs. The first cross-platform framework for mobile application development was PhoneGap, released in early 2008. Since then, numerous frameworks have emerged, with React Native being the most popular one today. (Manchanda, 2023)

2.4.1 React Native programming language

React Native is an open-source JavaScript programming language for creating cross-platform applications with a mobile native feel to it. It is based on React, which is Facebook's JavaScript library for building user interfaces, but instead of targeting the browser, it targets mobile platforms. One of React's biggest strengths lies in splitting up the codebase into different kinds of components. Each specific component can be updated and rendered whenever there is a need for it, without having to update the whole view of the page. React Native builds on the same concepts as React but does not render HTML elements. Instead, React Native uses the fundamental UI building blocks of the

native platform. The result of this is that the React Native codebase can work between several platforms. (Budziński, 2022)

2.4.2 Swift programming language

The Swift programming language has quickly become one of the fastest-growing languages. The language was introduced by Apple Inc in 2014 and is intended to be used for developing software for iOS, macOS, watchOS, and tvOS. (SlashData, 2022)

One of the main objectives of Swift is to simplify programming, and it does so by incorporating modern programming concepts and syntaxes to make it easy to learn, read and write. Swift being designed to be easy to learn and easy to use, has attracted a lot of new coders. (Reshetnikov, 2021)

3 Problem

3.1 The complexity of native applications

The complex nature of native mobile application development makes it simply not economically sustainable to replicate an app code, testing, and debugging across two major platforms. Therefore, there is an essential need to sophisticate the steps and be able to reuse the codebase across different platforms.

A promising alternative for native mobile development is mobile Cross-Platform Tools. Cross-platform tools allow a significant part of the codebase to be shared between different platforms. Cross-Platform Tools use mostly web-based programming languages to implement the logic of the application, therefore allowing developers with a background in web development to start developing mobile applications as well. (Lagerberg, 2017)

Surveys today are showing that there is some skepticism when it comes to cross-platform tools' performance compared to their native counterpart. It is therefore relevant and of special interest to analyze if the most popular cross-platform tool today can be the solution. The solution is to achieve a well-established type of user experience that resembles the quality and feel of a native-developed iOS mobile application. (Kozielecki, 2022)

3.2 Research Questions

The main goal of this thesis is to explore two solutions for developing mobile applications:

Swift, the native solution that works only on iOS.
React Native, a cross-platform solution that works on both iOS and Android.

Aiming to investigate whether there is any significant difference in the performance and user experience between the two different approaches. This thesis will mainly aim to explore if there will be enough difference in the performance of the two applications, for companies to justify the costs of having to create two different applications.

3.3 Related Research

There has been research comparing the performance of natively developed applications to web-based, hybrid, or cross-platform applications. In a 2016 study, Willocx, Vossaert, and Naessens compared a hybrid application to a native-developed application. Findings like longer launch times and heavier CPU consumption in the hybrid application were found. In the study, it is stated that the hybrid application was more easily produced and maintained, but the price to pay was the performance. However, this study was conducted in 2016 and may not reflect the status of React Native today. (Willocx, Vossaert & Naessens, 2016)

There have also been studies that have investigated how users perceive cross-platform applications. In 2015 there was a study conducted by Andrade and Albuquerque, where they asked a group of users to provide feedback on their experience using either their native or hybrid version of the application for two weeks. After the two-week period ended, there was a second two-week period, but in this period half of the users had their applications switched to a different version. Only 8 out of 60 testers noticed a performance difference between the two versions, suggesting that the performance differences between the hybrid and native versions are not too noticeable in everyday usage. (Andrade & Albuquerque, 2015)

4 Methodology

The method used to accomplish the objectives of this thesis is a technical experiment. The experiment will be in the design of creating two mirrored applications, one React Native-based and the other Swift based. The applications will strive to be as identical as possible with the same design and functionality.

The applications will be built in the same development environment, using XCode as the IDE tool. To further ensure the fairness of the result, I will not rely on any third-party solution and try to write the same codebase.

According to Apple, there are several metrics to consider when it comes to evaluating the performance and user experience of an iOS application. These metrics are listed in the table below. However, CPU usage is also an important factor to take into consideration when evaluating performance. The CPU usage indicates the total percentage of the CPU capacity that we're using at any given time. (Apple Inc, 2023)

iOS measuring tools

- App Launch Time (*execution time*)
- Responsiveness
- Memory Usage
- Battery Consumption
- Network Performance
- Crashes

Figure 4. Measurement tools for the performance of an iOS application

However, for this experiment, I will have a primary focus on the execution time, whether it's about launching the application or performing any other task within the application. I have decided to focus on the execution time mainly because it is in direct relation to the overall user experience. If a task takes too long to complete, it will harm

the overall user experience. It is thus also one of the most important factors when choosing the correct approach and programming language.

To reach a valid conclusion and to analyze the scalability of the two different implementations, the initial phase of the implementation will also be crucial. Therefore, the applications will be tested from the very start when there is minimal functionality until the very end when all the functionalities have been implemented.

To measure the execution time, a stopwatch will be used as they allow for a precise measurement. The stopwatch iteration can be started when the code execution begins and stopped when the code execution ends. A stopwatch in this context is an UI automated test found inside of the instrument tools in XCode (Figure 16).

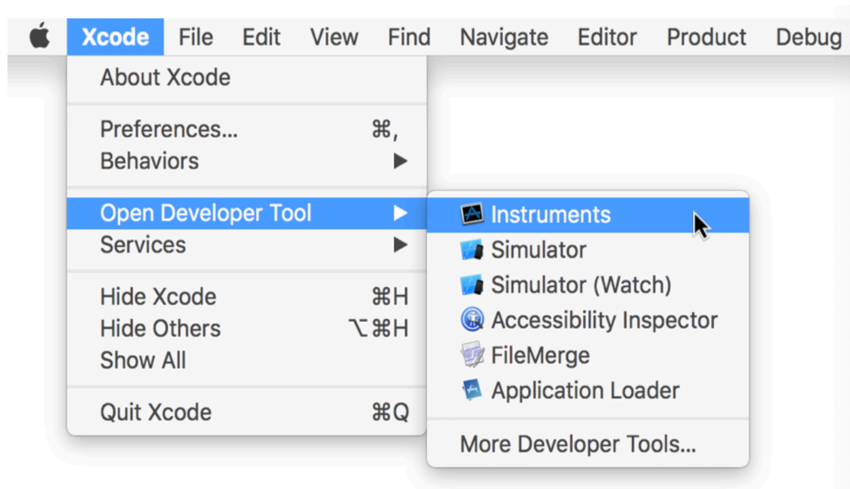

Figure 5. Instrument tools in XCode

After that, you'll choose the profiling template you wish to use, in this case for measuring the execution time I went ahead with the Time Profiler. The Time Profiler instrument tool allows for collecting data on the execution time and displays the results in graph format.

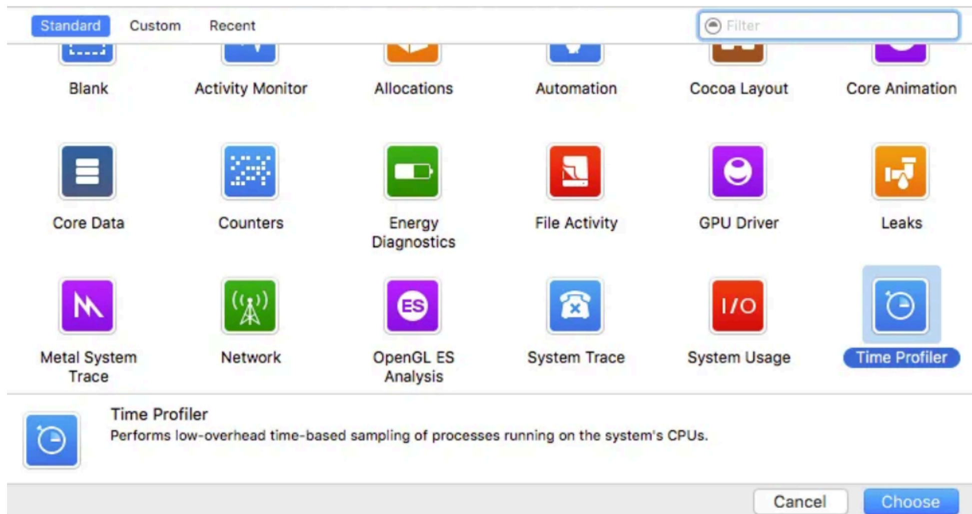

Figure 6. Profiling templates in XCode

This makes it possible to get an accurate measurement of the time it takes for that specific code to be executed, without being affected by any external factors like the boot time of the operating system.

4.1 Views for the experiment

The first view to be tested will display a list of items that only contain written text. The text will be generated from an array that consists of both images and string values. When the user clicks on a specific element, they will be directed to a new view that will display more information about the selected element.

Figure 7. The first displaying view of the experiment

The second view will display a list of views that contain both a thumbnail of the image and text values. By rendering the thumbnail of the image, the performance of the Media Layer can be tested. By clicking on one of the images, the user will further advance to the last view.

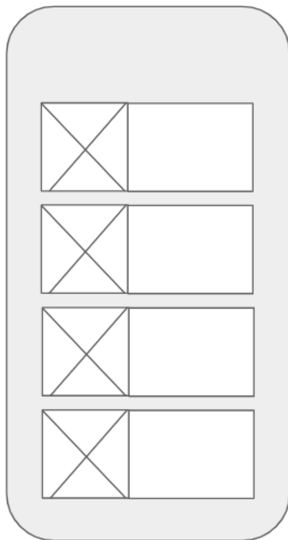

Figure 8. The second view of the experiment

The last view will include the full resolution of the image with a string value of the text. This view will allow the testing of the OS layer, which is the last layer in the iOS hierarchy.

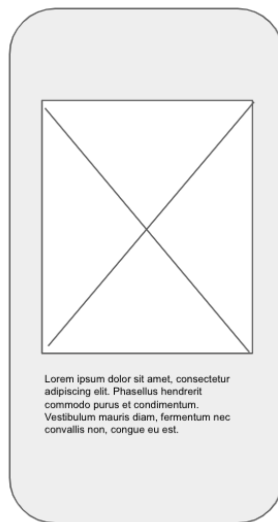

Figure 9. The last view

5 Implementation

5.1 Development process

In this section, we will delve deeper into the development process. The development of the applications in Swift and React Native will be presented in each separate category. These applications built during the process will be available on my GitHub account for easy access (Zibari, 2023).

5.1.1 React Native Development

React Native offers two methods for building apps. The first is Expo Go, which is recommended for beginners. The second is with React CLI, which is intended for those who are more familiar with mobile development. In this case, I went ahead with the React CLI, mainly because Expo adds some additional overhead to the app's size and performance.

A requirement for iOS development in general is a Mac computer with XCode installed. XCode is an IDE for developing iOS applications, which can be easily installed via the Mac App Store. Installing XCode will also install the iOS Simulator and all the necessary tools to build the iOS application.

To begin setting up a React Native project, the first step is to install Homebrew. Homebrew is a package manager that makes it possible to download and install dependencies from the internet via the terminal of macOS. Once Homebrew is installed, you can continue to install the following dependencies: Node and Watchman. Although Watchman is not strictly necessary for React Native development, you may experience slower reload times without it. Node, on the other hand, is a critical component in React Native development. It serves as the JavaScript runtime, making it possible to run JavaScript code outside of a web browser.

The following step is to install the React Native CLI which is a command-line interface that allows creating and managing React Native projects.

- npm install -g react-native-cli

Once the React Native CLI is installed, the last step is to finally create the React Native project with the *react-native init* command followed by the name of the app. This command will automatically set up the basic file structure and the remaining dependencies required for a React Native project. (React Native, 2023)

- npx react native init RNapp

5.1.2 Native iOS Development

Swift offers a relatively easy setup. The IDE used here is XCode as well, where you can easily click to create a new project and then continue to select the App option under the iOS heading. There you can choose the type of application that you have in mind to create, in this case, I created the “Single View App” which is the basic iOS application.

After that, you'll select Swift as the programming language of the project, and you can proceed to click the Create option. Once the project has been created, you're ready to start building the application. (Apple Inc, 2023)

5.1.3 Measurement environment

To ensure fairness, all tests will be performed using the same testing device. The physical device for the experiment is an iPhone 11 running on iOS version 16.4.1.

To test the application on my physical device, the developer trusted mode needs to be enabled. This can be achieved by connecting the device via a USB cable to the computer and then enabling the "Developer mode" found in the settings of the device. This step is also necessary for debugging the application.

5.1.4 Result of the built mobile applications

The result of the two different projects, React Native and Swift can be seen in the figures below (Figures 8 & 9). The main goal was to create a service that provides the core functionalities that a user demands by today's standards. The graphical interface represents an application where the user can scroll through the destinations and choose one of their choices.

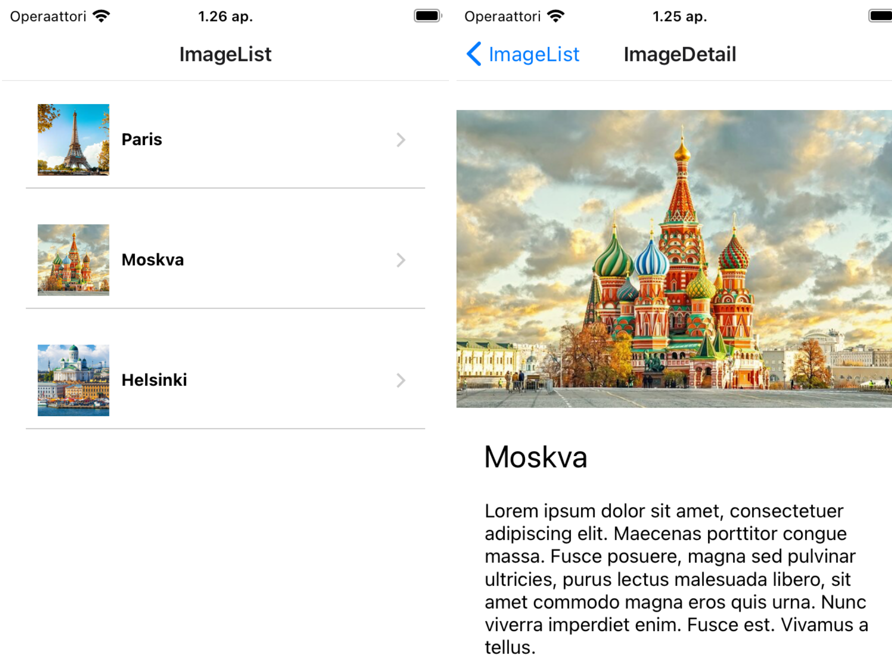

Figure 10. The application in React Native

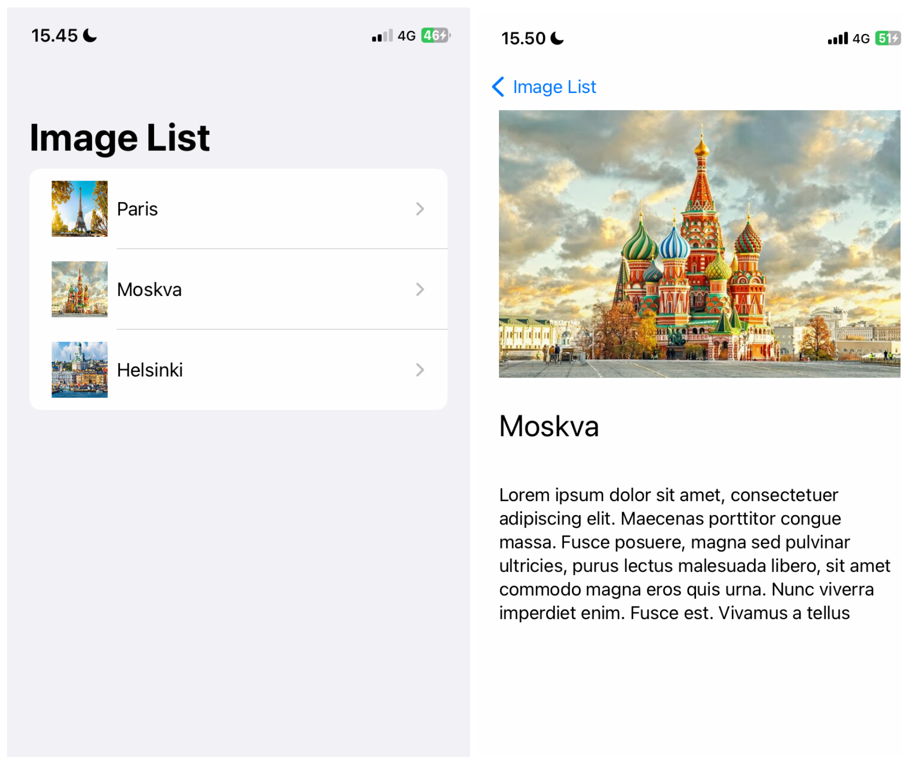

Figure 11. The application in Swift

At first glance, the services may appear like each other but the implementation of the two is vastly different. The two applications have different approaches to structuring and implementing their projects. Swift uses the MVC model and has the data, views, and models organized into separate files and folders. React Native on the other hand uses a function-based approach, it divides everything into separate screens and components.

The React Native project is structured with a main file that imports all the necessary files to initialize them all. React Native does not use a Controller that controls the navigation and memory flow as Swift does. This requires on the other hand that everything is divided into different kinds of functions, and one example is the navigation part. Whenever a screen is clicked, the function "this.props.navigation.navigate" is invoked, it then continues to search for the location of that specific prop, to know which props should be triggered.

6 Evaluation

Two artifacts have been created for this experiment, one of them being the natively built for the iOS system using Swift while the other artifact is created using the cross-platform solution React Native. An artifact in this context is the output created during the development process.

The main purpose of the experiment was to investigate if React Native gives a native experience, by comparing the two mirrored applications. The experiment was done by testing the code execution for both services and comparing the results to each other, to evaluate their performance.

Every test was performed through the XCode instrument tools, using the UI automated tests. The workflow involved cold booting the application, followed by selecting each item one by one. Upon entering an item, I would back out again to the main list view before continuing to select the next item. This process was repeated until all the elements were selected twice.

Each test consisted of 10 runs that were performed on the cold-booted application. An average of these 10 runs was then calculated. The first test would load 2 elements, the second 5 elements, the third 10 elements, the fourth 50, the fifth 100 elements, and the last 500 elements.

6.1 Results in a graph presentation

In this section, we'll delve into results and display them in graphs. The blue bar will represent React Native while the green bar will represent Swift.

In the first test run where we only had 2 elements, we can observe that the end user won't notice any performance difference. In this case, React Native performs slightly faster than the Swift application.

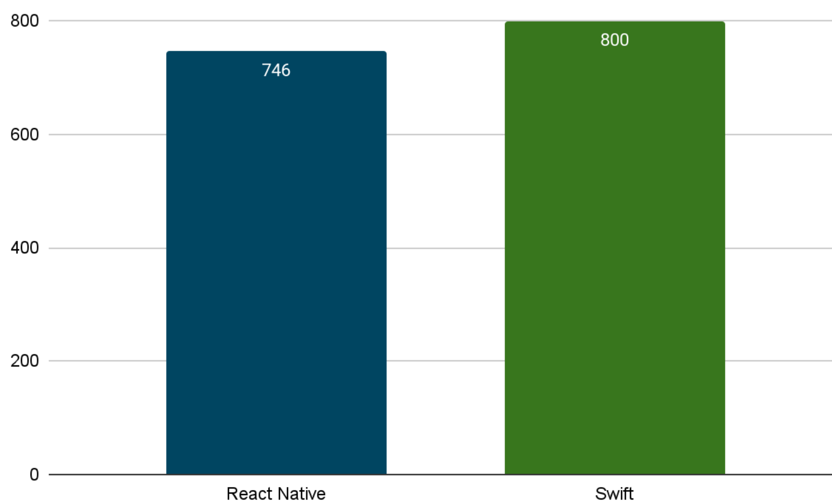

Figure 12. Graph presentation of the results 2 elements

In the next run where we have 5 elements, we can see the same result as we saw above. React Native still wins the execution time compared to Swift.

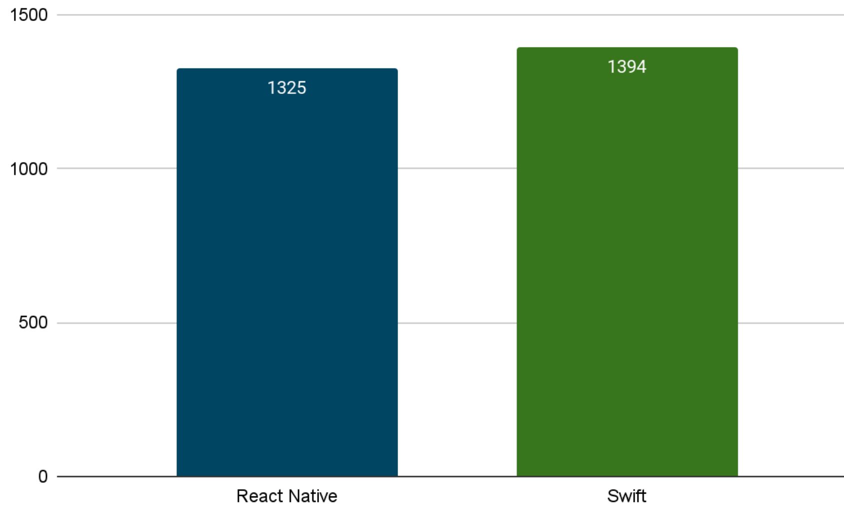

Figure 13. Graph presentation of the results with 5 elements

In the next case where we have 10 elements, we start to see a bigger difference in the results. There is a 374 MS difference, with React Native being the slower application now. However, the difference is still not that big for the user to explicitly notice a difference using the applications.

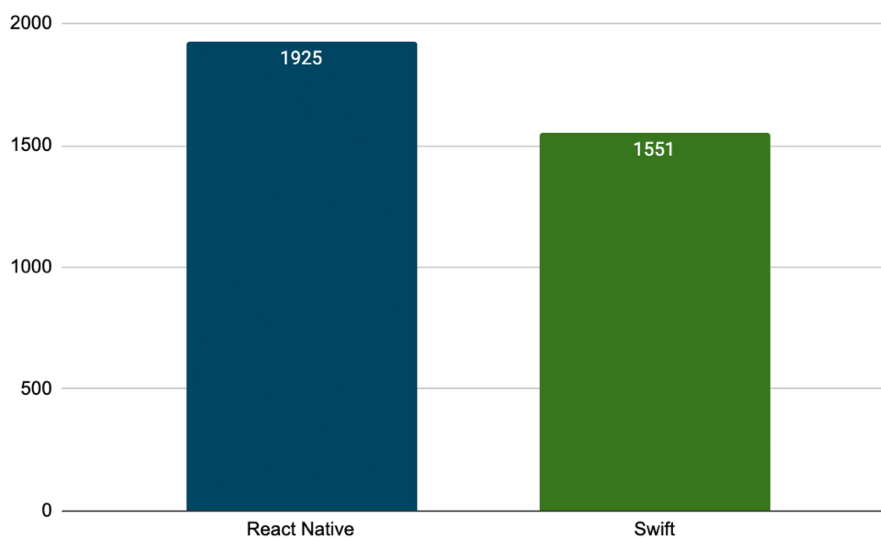

Figure 14. Graph presentation of the results with 10 elements

However, in the next test run with 100 elements, we start to see a difference that the user is most probably going to notice. Swift performs 1764 MS and React Native 3723

MS, a total difference of 1959 MS. This does suggest that Swift's performance advantage over React Native becomes more clear. We can see that the bigger and more complex the application becomes, the more Swift is winning in this competition.

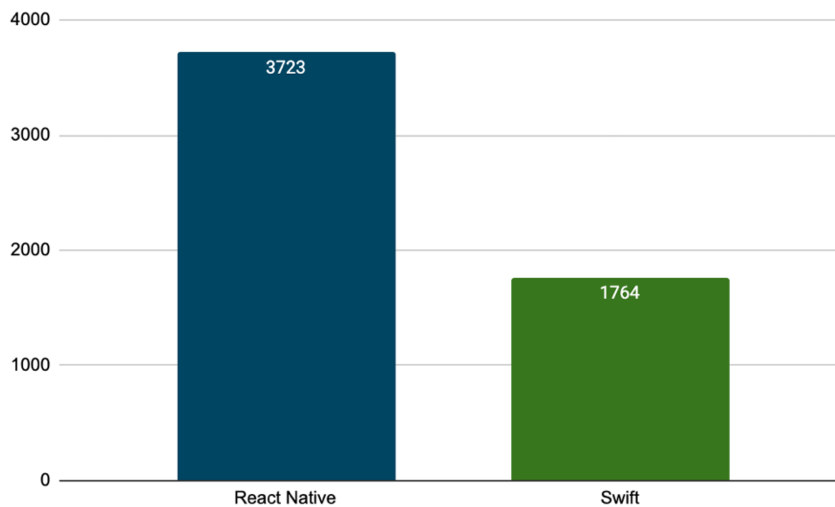

Figure 15. Graph presentation of the results with 100 elements

In the next test run with 500 elements, we see an even bigger difference. The application with React Native had an execution time of 18615 MS while the Swift application only took 8820 MS, a total difference of 9795. To get a better grasp of these execution times, I went ahead and calculated the percentage difference. In percentage React Native took up to 52.64% longer to execute compared to the Swift application.

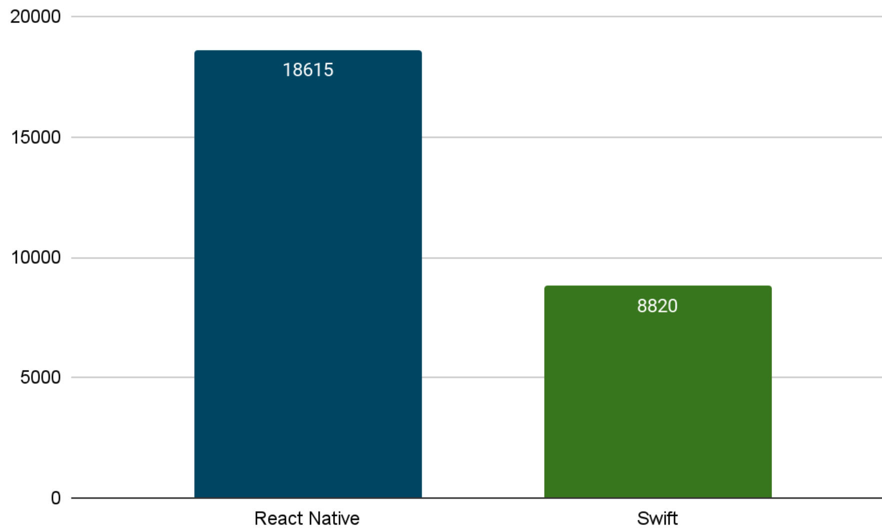

Figure 16. Graph presentation of the results with 500 elements

The data presented in the graphs above, reveals that React Native outperforms Swift in certain cases. In the first graph (Figure 9) where we only used 2 elements, we can see that there is not a big difference that the end user would notice. We can observe that this continues also in the next step, with 5 elements, where the difference is only around 70 MS (Figure 10). However, in the graph that follows, where we have 10 elements, we can see a difference of 373 MS with Swift performing better. This continues in the next graph as well where we have 100 elements, and the difference is 1962 MS with Swift performing a lot better. In the last view with 500 elements, we can see an enormous difference. These results mean in other words that the cross-platform solution loses to the native application when it comes to performance and speed. (Osadchuk, 2023)

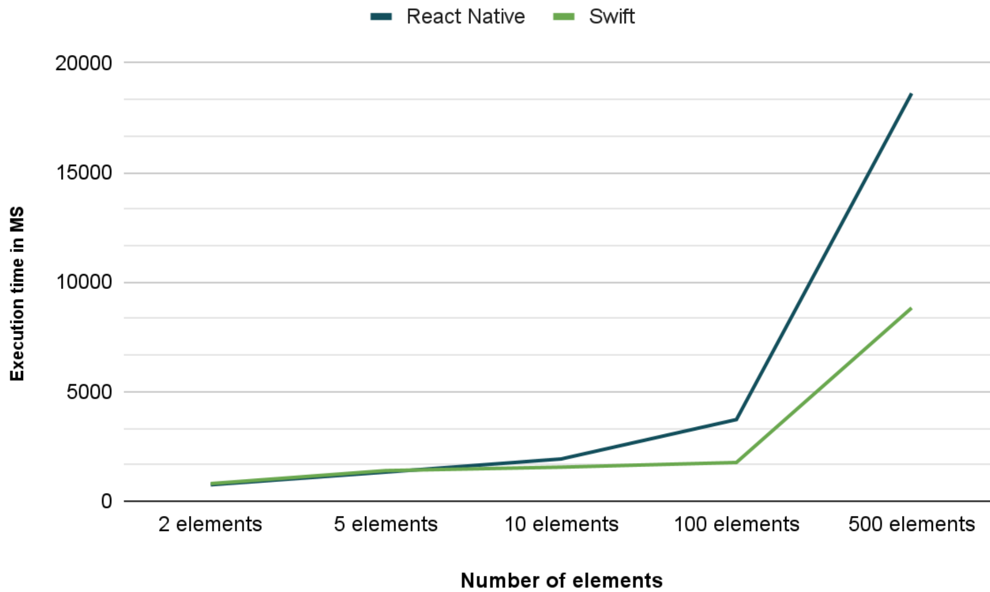

Figure 17. Presenting all result data in one final table

React Native proved to perform better in some instances when compared to Swift. However, we can see from the results that when extending the elements, the Swift application performs better. The most significant difference can be observed in the last testing step with 500 elements, where Swift outperformed React Native.

6.2 Results of application launch time

The application launch time tests showed us that the React Native application did load significantly slower than the native iOS application. The initial render time can be up to three times slower. The React Native application had load times up to 700 MS while the Swift application went only up to 220 MS.

Launch Time	Application
700-800 MS	React Native
150-220 MS	Swift

Figure 18. Application launch time results

While the native application only needs to load the native runtime, the React Native application must start a JavaScript engine and run a JavaScript application within it to know what to initially render. We can see the impact of this in the results presented above. However, there are some ways to alleviate a longer application launch, for instance, to show a splash screen in the beginning while performing the initializations. This can give reassurance to the user that the application is working as intended and is only loading the content. However, these applications are also quite easy in functionality. The difference in the launch times between the two applications becomes less significant the more complex the applications. More modern devices with better performance also improve the launch times, further reducing the impact React Native has in this category.

6.3 Application size difference

When comparing the two applications, I noticed that there was a significant difference in the size of the frameworks as well in the very initial phase. The Swift application took up 16 MB, while the React Native only required 8 MB.

Swift	16 MB
React Native	8 MB

The reason for this discrepancy is not entirely clear. One possibility could be because of the model view controller in the Swift project is handled by the memory allocation by default (Swift, 2023). React Native on the other hand uses a garbage collector which is highly effective (Ahmed, 2020). Memory allocation affects the size of the application because it determines how much memory will be used while running the application. However, it’s noted that the size of the application does not necessarily reveal the performance of the application.

7 Proposal for future work

There are quite a few additional Cross-Platform tools that are not covered in this thesis but would be interesting for future work to investigate. Such tools are Flutter, Cordova, Xamarin, Ionic, and PhoneGap. Mainly to investigate whether any of these tools would perform any better than React Native. Would they in other words give a more native feel than what React Native can provide?

It would also be very beneficial to improve the functionalities of the prototypes, this would allow the prototype to be more comprehensive. Two more comprehensive applications would further help evaluate React Native as a solution to native development. It is also important to mention that there is a lack of focus on the Android side of the development. This thesis does mention that React Native works as a cross-platform solution where the once-written codebase can function on both the iOS and Android systems. However, there’s not any further investigation on the Android side. Any further work on extending the testing into the Android environment would be very beneficial to thoroughly evaluate React Native. This would require some research into testing on Android devices since XCode is only accessible for iOS applications.

There are several potential options for further work on this subject. To further evaluate React Native, one could consider this work and further develop it. It could be in the form of implementing a more advanced architecture in the applications, shifting the focus to Android development, or comparing React Native to one of the other cross-platform tools.

8 Conclusions

8.1 Summary

The main goal of the thesis was to answer the question of whether React Native truly gives a native experience. A native experience where the end user won't notice a difference in performance in comparison to a real natively developed application. The experiment is conducted by creating two applications, one developed with a cross-platform solution React Native, and the other developed natively with Swift. These two different applications had the same design and functionality. They were then evaluated from an experiment using XCode's Instrument Tools. In the experiment, the focus was the execution time, which is a critical factor in evaluating the performance of applications.

Throughout this study we found that React Native has some limitations in providing a truly native experience, considering the longer execution times when rendering many elements compared to Swift. However, it's noted that there's no perfect way to compare cross-platform solutions to native solutions. The "native experience" varies depending on the context and it's therefore important to consider all the factors when choosing between cross-platform and native solutions. React Native has on the other hand proven to be a viable solution when developing smaller applications, for its fast development. While Swift would be a better solution when developing larger and more complex applications.

8.2 Discussion

I think that it is essential to evaluate different approaches for mobile application development, that could reduce the time and costs for companies. On the other hand, if cross-platform applications result in reducing the development team from two to one, it could translate to fewer job opportunities for mobile developers, a potential downside from another perspective. However, I do think that if two development teams are replaced by one, then the team is going to be larger than the previous ones. Having a bigger team with React Native developers specializing in different platforms, such as iOS and Android, may not necessarily then lead to cost savings. While cross-platform applications

do allow around 70-90% of the codebase to be shared between the platforms, certain areas of the code may require a significant modification. This could require React Native developers that specialize in different platforms.

The choice of Swift and React Native depends heavily on the specific need of the company. In my opinion, if a company requires a quick and affordable solution to production as soon as possible, then React Native would be the approach to take. On the other hand, if the application is more complex, I would suggest a Swift approach to it, taking into consideration the speed and performance tests conducted above.

Comparing the programming languages, both were quite easy to take grasp of. React Native does remind a lot of React and JavaScript, while Swift has similarities to the C programming languages. Coming from a background with no previous experience in Swift, it did require a learning curve.

9 References

Apple Inc. (2023). Improving Your App's Performance. https://developer.apple.com/documentation/metrickit/improving_your_app_s_performance

Apple Inc. (2023). Creating and Combining Views. https://developer.apple.com/tutorials/swiftui/creating-and-combining-views

Ahmed. A., (9 November 2020). Keeping Memory Leaks in Mind to Program Better. https://medium.com/swlh/keeping-memory-leaks-in-mind-to-program-better-25f3acf4ba90

Besant Technologies. (n.d.). What is iOS? https://www.besanttechnologies.com/what-is-ios

Budziński. M., (2022). What Is React Native? Complex Guide for 2022. https://www.netguru.com/glossary/react-native

Fireart Studio. (26 April 2022). Flutter vs React Native: Which one is better for 2023?. https://fireart.studio/blog/flutter-vs-react-native-what-app-developers-should-know-about-cross-platform-mobile-development/

GeeksforGeeks - Jagroopofficial. (23 January 2023). Architecture of iOS Operating System. https://www.geeksforgeeks.org/architecture-of-ios-operating-system/

Kozielecki. P., (11 May 2022). Cross-Platform vs Native App Development: What's the Difference? https://www.netguru.com/blog/cross-platform-vs-native-app-development

Lagerberg. M., (4 October 2017). Why we are not cross-platform developers. https://medium.com/pixplicity/why-we-are-not-cross-platform-developers-fd7ef70e976d

Linn. S., (20 March 2023) The 10 Largest Mobile App Companies In The World, And What They Do. https://history-computer.com/the-largest-mobile-app-companies-in-the-world-and-what-they-do/

Manchanda, A., (7 April 2023). The Ultimate Guide to Cross Platform App Development Frameworks in 2023. https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/

Madeshvaran. S., (2 December 2019). Everything You Need To Know Before Starting Xamarin Development [2020 Edition].https://medium.com/a-developer-in-making/everything-you-need-to-know-before-starting-xamarin-development-2019-edition-497446161196e

Nitze. A., Rösler. F., Schmietendorf . A., (March 2014). Performance Evaluation of Cross-Platform Mobile Applications. https://www.researchgate.net/publication/296700470_Performance_Evaluation_of_Cross-Platform_Mobile_Applications

Osadchuk. S., (4 February 2023). React Native vs Swift in 2023: Which One is Better for Your Project?.https://doit.software/blog/react-native-vs-swift#screen2

Paulo R. M. de Andrade, Adriano B. Albuquerque. (February 2015). Cross Platform App – a comparative study. https://arxiv.org/pdf/1503.03511.pdf

React Native. (2023). Environment Setup. https://reactnative.dev/docs/environment-setup?guide=native

Reshetnikov. D., (13 September 2021). The Good and the Bad of Swift Programming Language. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-swift-programming-language/

Swift, (2023). Automatic Reference Counting. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/

SlashData. (July 2022). Most used programming languages among developers worldwide as of 2022. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

Stack Overflow. (2022). Developer Survey.https://survey.stackoverflow.co/2022/#most-popular-technologies-misc-tech-prof

TechTarget Contributor. (January 2023) Hybrid application. https://www.techtarget.com/searchsoftwarequality/definition/hybrid-application-hybrid-app

TechTarget. (2023). Web Application. https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app

Willocx. M., Vossaert. J., Naessens. V., (2016). Comparing performance parameters of mobile app development strategies. https://msec.be/crossmos/onderzoeksresultaten/performancePaper2.pdf

Zibari. D., (5 May 2023) Swift app.https://github.com/dilanzib/swift-app

Zibari. D., (5 May 2023). React Native app. https://github.com/dilanzib/reactnative-app

Appendix. Summary in Swedish

Introduktion

Mobilapplikationer har blivit en vanlig del av våra dagliga liv och har förändrat sättet vi integrerar med teknologi. För att nå de största publikgrupperna krävs däremot två nativa applikationer, en för iOS och en för Android. Dessa måste utvecklas med olika programmeringsspråk vilket kräver en högre komplexitet och kostnad som företagen måste stå för. Cross-plattformutvecklingen har kommit som en lösning till det, men det har visat sig att användarna inte är lika nöjda med prestandan som de är med de nativa (Nitze, Rösler & Schmietendorf, 2014).

Det här arbetet behandlar en teknisk undersökning som utförs för att utvärdera prestandan hos React Native. Syftet är att bedöma om React Native kan rekommenderas som ett alternativ mot dess nativa applikation i Swift.

Bakgrund

Bakgrunden i arbetet diskuterar betydelsen av mobilapplikationer i dagens bransch. Mobilapplikationernas betydelse medför samtidigt ett krav för företagen att leverera högkvalitativa mobilapplikationer på alla plattformar, för att helt enkelt kunna öka sina intäkter. Men för att utveckla mobilapplikationer för varje plattform är kostsamt.

Artikeln belyser också de andra populära plattformsoberoende verktygen, som till exempel Flutter, Xamarin och Cordova. Dessutom förklaras även de tre olika arkitekturen som används i mobilutveckling, vilka är nativa, webb och hybridapplikationer. Var och en av dem har sina egna fördelar men därpå också begränsningar. Nativa applikationer är begränsade till specifika verktyg och webbapplikationer kräver internetanslutning för att kunna nås. Hybrid applikationer kombinerar både webb och nativa applikationer.

Problem

I problem avsnittet diskuteras komplexiteten i att utveckla nativa applikationer för diverse plattformar. Vi ser ett behov av plattformsoberoendeverktyg för att kunna återanvända samma kod över olika plattformar.

Denna studie syftar på att utforska skillnaderna i prestandan mellan Swift och React Native, den nativa lösningen som endast fungerar på iOS och den plattformsoberoendelösningen som fungerar på både iOS och Android.

I stycket nämns också relaterad forskning som visat att de plattformsoberoende applikationerna har haft en grad sämre prestanda. Det kom bland annat fram att de plattformsoberoende verktygen kan ha längre lanseringstid och tyngre CPU-förbrukning. Men i den dagliga användningen är dessa prestandaskillnader inte alltför märkbara. (Andrade & Albuquerque, 2015)

Metod

Avhandlingens metod innebär att skapa två identiska applikationer med olika programmeringsspråk, React Native och Swift. Därefter testa deras prestanda med fokus på exekveringstiden. Exekveringstiden kommer att mätas med hjälp av ett Time Profiler verktyg som finns i XCode programmet. Experimentet kommer att testa tre vyer med olika typer av funktion och olika mängd innehåll.

Genomförande

Detta avsnitt diskuterar utvecklingsprocessen för de två mobilapplikationerna, varav den ena är utvecklad med React Native och den andra med Swift. Nödvändiga verktyg för React Native utvecklingen är Homebrew, Node, Watchman och React Native CLI. Swift kräver endast XCode programmet installerat.

Avsnittet presenterar också resultaten från de två olika mobilapplikationerna och belyser skillnaderna i deras inställning när det gäller att strukturera och koda. React Native

använder ett funktionsbaserat tillvägagångssätt och organiserar allt i separata skärmar och komponenter. Swift applikationen använder MVC-modellen och separerar data, vyer och modeller i olika filer och mappar.

Utvärdering

Avsnittet utvärderar resultaten som experimentet gav. Experimentet syftade på att avgöra om React Native ger en nativ upplevelse och hur den presterar jämfört med Swift applikationen.

Testerna genomfördes med XCode instrumentverktyget. Resultaten presenterades sedan i grafer som visade att React Native presterade bättre än Swift i visa fall, men då man utökade mängden innehåll presterade Swift applikationen bättre. Dessutom nämndes även applikationernas lanseringstid, och där visade sig att React Native applikationen tog betydligt längre tid att lanseras.

Förslag till framtida arbeten

Avsnittet föreslår framtida arbeten som att undersöka ytterligare plattformsoberoende-verktyg, till exempel Flutter, Cordova och Xamarin. För att närmare se om de presterar bättre än React Native när det gäller att erbjuda en mera nativ känsla.

Ytterligare arbete för studien kan innebära att implementera en mer avancerad arkitektur i applikationerna, flytta all fokus till Android-utveckling eller jämföra React Native med andra plattformsoberoende verktyg.

Slutledning

Avhandlingen syftade på att svara frågan om React Native ger en verklig nativ upplevelse. Detta åstadkoms genom att utföra ett experiment. Experimentet jämförde två applikationer med samma design och funktion som utvecklats med hjälp av React Native och Swift. React Native visade sig ha vissa begränsningar, en längre exekveringstid, och anses därför som en gångbar lösning för mindre applikationer på grund av dess snabba

utveckling. Studien anser Swift som en lösning för större och mer komplexa applikationer. Studien kom fram till att valet mellan de två språken beror i slutändan på företagets specifika behov.