COMPARING PERFORMANCE BETWEEN REACT NATIVE AND NATIVELY DEVELOPED SMARTPHONE APPLICATIONS IN SWIFT

A comparative analysis and evaluation of the React Native framework

Bachelor Degree Project in Informatics, 30ETC

Spring Term 2018

Dennis Bilberg, a15denbi

Supervisor: Henrik Gustavsson
Examiner: Yacine Atif
ABSTRACT

In today’s society, smartphones are so widely established that corporations have even changed their cooperate culture when it comes to bringing your own personal device to work. Sales graphs prove that smartphones are more established today than ever before, which creates pressure for companies big, as small to extend and provide their services from the pocket of the user in the form of a smartphone mobile application.

This paper focuses on the development of smartphone applications. Looking into the native development way for iOS and the code fragmentation that characterizes the long and costly development in order to provide the application on the big mobile operating systems by evaluating the cross-platform solution React Native that bypasses the fragmentation.

The experiment presents the collected data and its solutions, with an evaluation of the React Native framework. Finally, thoughts and future work to further extend the category is presented.

Keywords: React Native, Swift, Evaluation, Code Execution, Cross-platform, Native
# Table of Contents

1. Introduction ........................................................................................................... I

2. Background ............................................................................................................. II
   2.1 The importance and development costs for mobile applications .................... II
   2.2 Classification of a Smartphone Application ..................................................... III
   2.3 Native Applications .......................................................................................... III
      2.3.1 Swift Programming Language ................................................................ IV
   2.4 Cross-platform Applications ............................................................................ IV
      2.4.1 Web Applications ..................................................................................... V
      2.4.2 Hybrid Applications ................................................................................ VI
      2.4.3 React Native Programming Language ..................................................... VI

3. Problem ................................................................................................................ VIII
   3.1 The complexity of the native user experience .................................................... VIII
   3.2 Research Questions .......................................................................................... VIII
      3.2.1 Hypothesis ............................................................................................... IX
   3.3 Related Work .................................................................................................... IX

4. Methodology ......................................................................................................... XI
   4.1 The Method ...................................................................................................... XI
   4.2 Research Ethics ............................................................................................... XVII
   4.3 Reliability ........................................................................................................ XVII
   4.4 Measurement Environment .............................................................................. XVII

5. Implementation ...................................................................................................... XVIII
   5.1 Literature study ............................................................................................... XVIII
   5.2 Development Process ....................................................................................... XIX
      5.2.1 Native iOS Setup ................................................................................... XX
      5.2.2 React Native Setup ................................................................................ XXI
      5.2.3 Application size difference ................................................................... XXIV
   5.3 Travel Buddy Progression ................................................................................ XXV
      5.3.1 The iterations ........................................................................................... XXV
      5.3.2 Final results ............................................................................................. XXVIII
   5.4 Pilot study ......................................................................................................... XXX

6. Evaluation .............................................................................................................. XXXII
6.1 Data-graphs presentation ................................................................. XXXIII
6.2 Data-graphs analysis .................................................................... XXXVI
7. **Final Conclusion** ........................................................................... XXXIX
   7.1 Summary ...................................................................................... XXXIX
   7.2 Discussion .................................................................................... XXXIX
   7.3 Learnability .................................................................................. XLII
   7.4 Future work ................................................................................ XLIV
8. **References** .................................................................................... XLV
Abbreviations and Terminology

IDE Integrated Development Environment for software development.

JavaScript Object-oriented scripting programming language.

Swift Multiparadigm language, combines functional with object-oriented.

Codebase Describes the whole collection of the source code to build a program.

Figures

Figure 1: The web application architecture.

Figure 2: The architecture of a hybrid application.

Figure 3: Lo-fi illustration of the first view of the artifact (UITableView with text)

Figure 4: Lo-fi illustration of the second view of the artifact (UITableView with text and thumbnails)

Figure 5: Lo-fi illustration of the third view of the artifact (UITableView with text and images)

Figure 6: Hello, World! example view of the start in the iOS version developed in Swift.

Figure 7: Hello, World! implementation developed in Swift.

Figure 8: Hello, World! example view of the start in the iOS version developed in React Native.

Figure 9: Hello, World! implementation developed in React Native.

Figure 10: Flatlist implementation in React Native artifact.

Figure 11: StackNavigator implementation in React Native artifact.

Figure 12: Final version of the two different projects, React Native to the left, Swift to the right.
Figure 13: Page navigation when a destination is selected.

Figure 14: Graph charts displaying the first performance test in ms (less is better).

Figure 15: Instrument Tools overview when performing a test.

Figure 16: Code execution for 3 elements in ms.

Figure 17: Code execution for 5 elements in ms.

Figure 18: Code execution for 10 elements in ms.

Figure 19: Code execution for 100 elements in ms.

Figure 20: Overview graph of rendered elements.

Figure 21: Data and rendering visualization.

Tables

Table 1: The iOS component layer hierarchy.

Table 2: The tools used to measure the performance parameters of the service.

Table 3: Specs of the physical device performing the tests.

Table 4: "Hello, World" implementation application size of the frameworks.

Table 5: "Travel Buddy" final implementation application size of the frameworks.

Table 6: Confidence intervals of execution time.
1. Introduction

Since the release of the iPhone in 2007 and the Android platform in 2008, the market for a smartphone has grown excessively. Because of the iPhone, smartphones have gained a much wider audience compared to its predecessors like the BlackBerry which was mainly focused on business people. Today smartphones are no longer viewed as simply as phones, the smartphone today provides a range of features similar to the one of a computer system and have become a major part of people's lives, both in the working space and at home (Danny et al., 2014).

A smartphone provides a range of features, these features are powered by an application. An application is similar to a running program on a computer offering a limited set of functions. The applications are provided through a specific platform that is distributed by the creator of the OS running on the smartphone device, App Store for iOS by Apple and Google Play for the Android platform by Google.

It's often required than an smartphone application is available and distributed both in the App Store and on the Google Play Store. In order to achieve this due to the nature of the two different OS running on the smartphone device, two different applications need to be developed natively with the same design and functions but in two different programming languages with zero reuse of the codebase. Due to this demand, companies must hire two different developer teams, one for each platform in order 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 allows one implementation of the code to work on multiple platforms. Over the years different frameworks have been used but studies suggest that the end-users are not as satisfied with cross-platform applications as they are with the native applications. Studies suggest that cross-platform applications are more prone to user complaints due to the performance of the applications being worse compared to their native peer (Iván et al., 2016). However, new implementations of technologies and frameworks are constantly being created with a refined architecture, one of these new frameworks is called React Native. Research has been done on cross-platform solutions such as PhoneGap, Titanium, and Xamarin to name a few when developing smartphone applications, however, there is little research regarding the evaluation of React Native as a cross-platform solution, which this work will try to elaborate further.
2. Background

2.1 The importance and development costs for mobile applications

The market is rapidly changing in today’s industry and one of the biggest factor to take into consideration today is the user experience when it comes to using mobile smartphone device applications and their overall feel when using these applications. The personal need for using different kinds of smartphone applications has risen so radically that corporations have even changed their cooperate culture when it comes to bringing your own personal device to work (Danny et al., 2014). Smartphone subscriptions worldwide have been increasing from 2.6 to 3.4 billion in between the year 2014 to 2015 according to Ericsson. Therefore Ericsson expects the smartphone subscriptions to almost double around the year of 2021 to around 6.4 billion subscriptions (Ericsson 2015).

When taking this factor into consideration it’s now more important than ever for the corporations to deliver high-quality mobile smartphone applications that not only run well but also deliver a good user experience for their users. This to increase revenue since the smartphone has introduced new opportunities for the companies to provide new types of services for their users (Michiel et al., 2016).

One of the unique challenges that mobile development brings is the efficiency to rapidly develop and maintain mobile smartphone applications (Danny et al., 2014). Unfortunately making the service accessible on all major mobile platforms is very costly due to the fragmentation of the smartphone market. Furthermore developing native mobile applications for each platform drastically increases the development costs of the project (Antuan et al., 2015). This due to the nature of native applications the codebase cannot be shared between the different implementations. Each platform requires dedicated tools and different types of programming languages (Michiel, et al., 2016). In order for a native application to attract a larger audience, the native application in question needs to be developed for at least one of the mainstream mobile operating systems which is the iOS platform or the Android platform. The iOS platform is mainly built on the Objective-C programming language while the Android platform is mainly built in Java (Peixin et al., 2016).
2.2 Classification of a Smartphone Application

A smartphone application is a program that gets executed and runs on a smartphone device. There is no direct scientific consensus regarding the classification of a smartphone application, although the arbitrary understanding of a smartphone application is classified in different variations such as native applications, web applications, and hybrid applications. The classification of a smartphone application depends on the technology and environment which the application gets developed in (Spyros et al., 2013).

2.3 Native Applications

Native applications are based on the specifically targeted platform language, the language then strongly relates to the programming language that supports that specific platform (Salma et al., 2014). Native framework Integrated Development Environments (IDEs) are limited to the specific tools, such as Android Studio for Android developing platform and Xcode for the iOS developing platform (Peixin et al., 2016). Due to the nature of the native developing way, it takes a great deal of work and time to make the native application compatible between different implementations due to fragmentation, which means that the developers need to create different versions of the same application in order to make it work properly between different systems (Michiel et al., 2016).

The understanding concept in general of a smartphone application includes a set of necessary hardware and software components that together form the required tools and resources for development, installation, and tests of the created application in question. The underlying technologies of the smartphone device can be illustrated and viewed as a set of hierarchical component layers as shown in Table 1, where the higher levels contain the more sophisticated type of services, and the lower component layers contain the necessary technologies that the application in question relies upon (Spyros et al., 2013).

Table 1: The iOS component layer hierarchy.

<table>
<thead>
<tr>
<th>Layer</th>
<th>iOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cocoa Touch</td>
</tr>
<tr>
<td>2</td>
<td>Media Layer</td>
</tr>
<tr>
<td>3</td>
<td>Core Services</td>
</tr>
<tr>
<td>4</td>
<td>OS Layer</td>
</tr>
</tbody>
</table>
The highest level component acts as the service in-between the user application, and also the necessary underlying components of the hardware. The Cocoa Touch-layer also handles all the pre-installed and third-party created applications that extend the functional behavior of the smartphone device.

The Media Layer component contains the information regarding the audio and video technologies of the device which together forms the multimedia experience for the user in question.

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

The last component which is the OS Layer contains the operating system interfaces that other components on the device relies on, such as memory management, files, and drivers. (Spyros et al., 2013)

### 2.3.1 Swift Programming Language

Swift is a rich and new powerful language (David et al., 2017). Apple has recently released a modern programming language to be the successor of Objective-C called Swift. Swift is a multiparadigm language that combines imperative, object-oriented and functional programming and is the natively written programming language for the iOS platform along with Objective-C (Marcel et al., 2016).

Thanks to the inherited Objective-C ecosystem the Swift programming language is already one of the most popular programming languages in the world("The fifteen most popular languages on GitHub - GitHub", n.d). This new distinctive scenario presents a unique opportunity to analyze how this new language performs (David et al., 2017). The goal of Swift is to be one of the best available programming languages when it comes to developing mobile and desktop applications, scaling up to cloud services. The primary focuses from Swift is to be safe and strict combined with being fast and expressive("Swift 4 The powerful programming language that is also easy to learn - Apple", n.d.).

### 2.4 Cross-platform Applications

Mobile Cross-Platform (CPT) provides an interesting alternative to native development due to the CPT tools aim at sharing a significant portion of the codebase of the application between different kinds of implementations. The implementations can then also be shared between different platforms such as
Android and iOS. This technique can drastically reduce the time and development costs of mobile application development projects (Michiel et al., 2016).

2.4.1 Web Applications

![Diagram of web application architecture](image)

*Figure 1: The web application architecture.*

As can be observed in Figure 1 web-applications are based upon the browser-based software which is downloaded from the web, they focus mainly on the internet technology which is built upon the programming languages HTML5 and JavaScript. A web-based application does not require an installation of the application, therefore, making the technology hardware independent, the tradeoff is, therefore, limited access to the smartphone’s hardware and data. The web application also depends on downloading and render the web page to the smartphone, which demands the smartphone to have internet access in order to use the application in question (Spyros et al., 2013).
2.4.2 Hybrid Applications

Hybrid applications combine the technologies of web-based and native developed applications as seen in Figure 2. The applications are being built with HTML5 and JavaScript and then embedded within a native container, this allows the OS on the smartphone device to handle the application in question like a native developed application and can thereby be distributed and installed through the app store provided from the specific platform of the device (Rahul et al., 2013). This also allows the application in question to get access to the underline hardware and data of the device in question. By embedding the web-based code in a native container this approach also makes the application in question hardware independent (Salma et al., 2014).

![Figure 2: The architecture of a hybrid application.](image)

2.4.3 React Native Programming Language

React Native is an open source JavaScript programming language for creating cross-platform mobile smartphone applications with a mobile native feel to it. React Native was developed by Facebook Inc. and the library is constantly evolving by the company and the community overall. React Native is based on the web-based JavaScript library called React, which is based on creating different kinds of views that get updated when a state changes within that associated component. One of React’s biggest strength lies in splitting up the codebase into different kinds of components. Therefore that specific component can then be updated and rendered when there is a need for it without updating the whole view of the page ("A JavaScript library for building user interfaces - React", n.d.). Thanks to React Native
using the same fundamental building blocks that are associated with Android and iOS applications the React Native codebase works between both implementations, also due to the nature of React Native, it is also possible to combine native components written for that specific platform such as Java, Swift or Objective-C if some type of optimization is needed ("Build native mobile apps using JavaScript and React - React Native", n.d.).
3. Problem

3.1 The complexity of the native user experience

Due to the complex nature of native mobile application development, it's simply not economically sustainable to replicate an app code, testing and debugging across two major platforms. Therefore sophisticated steps to be able to reuse the codebase across different platforms is of essential need (Danny et al., 2014).

A promising alternative from native mobile development is mobile Cross-Platform Tools (CPTs). CPTs allow a significant part of the developer’s codebase to be shared between the implementations of multiple platforms. Also, CPTs mostly use 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.

Surveys are showing that there is some skepticism when it comes to CPTs and their performance when compared to their native counterpart, it's therefore interesting to see if the mature modern CPT tools today can overcome the skepticism and provide a viable alternative to native development today (Michiel et al., 2016).

Companies and developers today face some difficult questions regarding the evaluation of mobile application development. According to Iván et al., (2016), studies show that cross-platform developed applications are more prone to user complaints. It’s clear that there is not just only some skepticism when it comes to this type of technology, up until now cross-platform implementations simply don’t work as good quality wise from an end user perspective compared to their native developed counterpart (Iván et al., 2016). This in hand creates problems for companies who depend on the cross-platform solution to work efficiently. Therefore due to these circumstances, it’s relevant and of special interest to analyze and measure if new cross-platform technologies like React Native can be the solution to this type of problem, to achieve a well-established type of user experience that resembles the quality and feel of a native developed Swift mobile application.

3.2 Research Questions

The solution that this thesis is going to explore is two relatively new solutions for creating the modern mobile application for tomorrow. The first language that will be tested is the native mobile programming language for the iOS platform Swift. The second language that will be tested is the cross-platform solution that is React Native, which works on both Android and the iOS platform.
Will there be a significant difference in the performance and user experience of the created application between the two different languages? Will there be enough difference in performance to justify the time and costs by having to create two different codebases in order to make the service accessible on all the major platforms? Those are the main questions that this thesis and its corresponding experiment aim to explore and answer.

3.2.1 Hypothesis

The hypothesis is that there will be no significant difference in the performance of rendering data to the user when comparing the cross-platform developed React Native application to the second developed application using the native Swift programming language for the iOS system.

3.3 Related Work

There have been several tests when comparing the performance of natively developed applications compared to web-based, hybrid or cross-platform applications in the past. Lisandro et al., (2017) suggests through collected data from a performance analysis where they compare popular frameworks for developing cross-platform, hybrid and web-based approaches, that natively developed applications for the iOS-system truly outshine their hybrid and cross-platform counterparts. Similar findings are found in Isabelle et al., (2013) where the user experience is not as good as natively developed applications, but the applications can be distributed at once to several platforms due to sharing the codebase making the cross-platform approach of more value to some developers and companies when compared to their native counterpart.

There have also been past studies that mainly focus on a qualitative analysis and evaluations of cross-platform development, analyzing the available support, programming languages, and development environment. The findings in Michiel et al., (2016) suggests that because of the JavaScript runtime needed to operate in cross-development approaches, the cross-platform applications is more CPU-intensive and cost more memory compared to their natively developed counterpart to functioning properly. The JavaScript approach also shows to have the slowest launch time when measuring the different developed applications, however, when the application is actually fully loaded, navigation and response times are generally similar to native response times.
Hybrid applications have also been carefully evaluated when comparing the performance of a hybrid application and a native developed application. Peixin et al., (2016) findings are similar to the ones in Michiel et al., (2016), also showing longer launch times and heavier CPU consumption in the created hybrid artifact compared to their native peer. The results point to that hybrid developed applications are more easily produced and maintained due to less fragmentation, but the price is the performance. Suggesting that natively developed applications provide a better user experience when it comes to the performance part. The conclusion is if the end customers main priority is the performance of the application, a natively developed application is to be the recommended approach.

Effective approaches to the cross-platform developer environment and the complexity of diversity have also been discussed (Salma et al., 2014 and Manuel et al., 2012). Showing when comparing different cross-platform technologies that the ones based on JavaScript prove to be more resource demanding when compared to implementations built in C and C++. Suggesting that one must carefully evaluate what type of application that should be built, in order to choose the most appropriate approach when it comes to the performance of the application in question.

Lastly, there has also been modern analysis through surveys on a few selections of cross-platform development alternatives such as React Native when measuring technical implementation, application performance and user experience for the end-user (Tim et al., 2017). The findings point to similar results as the other articles earlier mentioned, that application performance can negatively impact the user experience when going for a cross-platform development approach. Tim also addresses the user interface (UI) to be a concern due to it being more complex to achieve the native developed look and feel of the application when developing with cross-platform implementations.
4. Methodology

4.1 The Method

The method that is going to be used to test the performance differences between Swift and React Native is a technical experiment. An experiment will be designed in a form of two mirrored applications, these applications should strive to be identical in the closest manner with the same design and functions while taking into account the distinctive architectures of both services.

Both applications will be developed in the same development environment using Xcode IDE as the main tool for analyzing the performance of the created artifacts. By measuring using the same hardware, software and developer environment for both services, a certain degree of fairness can be ensured. The Swift programming language only allows written tests in either Objective-C or Swift. This makes Swift more isolated compared to React Native that allows execution of more than just JavaScript code as mentioned in a previous section. Therefore Xcode is the primary choice when developing natively for the iOS-ecosystem. By choosing not to rely on a third party solution and also to write the same codebase for the performance tests we further enhance the fairness of the end result.

According to Michiel et al., (2016) there are four different metrics to take into consideration when evaluating the performance of a smartphone application. They argue that execution time, memory consumption and battery drain consumption are the primary factors to evaluate. The CPU usage is also an important factor to take into consideration when evaluating the performance of the service according to Isabelle et al., (2013). The four primary metrics to take into consideration when evaluating the overall term of performance is illustrated as seen in Table 2. These four categories together form the understanding of what a user experience is from a technical perspective, so when referring to an application that takes user experience into consideration at least one of these metrics needs to be tested and evaluated, in this report the code execution time of the code in question when rendered to a physical device. A more detailed explanation of the methodology behind this will be discussed further, also a more detailed look at the provided iOS measurement tools for this project.
Table 2: The tools used to measure the performance parameters of the service.

<table>
<thead>
<tr>
<th>iOS measuring tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Times (Time Profiler)</td>
</tr>
<tr>
<td>CPU Usage (CPU Activity)</td>
</tr>
<tr>
<td>Memory Usage (Allocation)</td>
</tr>
<tr>
<td>Disk Space (Visible on device)</td>
</tr>
</tbody>
</table>

The illustration in Table 2 describes in the Xcode IDE, Response Times (Time Profiler), CPU Usage (CPU Activity), Memory Usage (Allocations) and Disk Space consumption. Xcode provides and distributes all the necessary tools when honoring the performance and strategy methods of previous works when evaluating different mobile application approaches (Michie et al., 2016).

All these measuring factors should be considered when talking about the performance of the application in question and when evaluating the user experience.

However, for the purpose of this experiment, the execution time will be the primary focus when evaluating the performance of the different created services. This decision is made because the execution time is in direct relation to the overall user experience for the user in question. The amount of time needed for the application in question to render and perform a certain task is in direct correlation to the user experience due to, if a task takes to long to complete, it may harm the overall quality and feel of the user experience, and is thus of the highest importance when choosing the correct approach when developing the different services. The criteria for what is considered a native user experience will be explained at a later time in this section.

In order to approach the problem statement and reach valid conclusions the services will need to be tested during the start step when there are minimal function implementations, and also during the end step when all the functions have been implemented, this to analyze the scalability of the two different implementations of the services.

When executed, a service will either perform correctly, incorrectly or straight up refuse execution, therefore the first test will occur during the first implementation of the application in question. Honoring good practicing approaches for application development, a natural starting point for any application is the "Hello, World!" implementation (Lisandro et al., 2017). This first test will occur to measure and compare the size overhead of both different services.

To measure the execution time of a service, stopwatches is a natural selection tool for testing the executed code. A stopwatch iteration allows being started when the
sought execution of the code begins and then terminated when the code in question has finished execution. To encapsulate specific sections allows to avoid the inclusion of boot time of the systems OS, this is necessary to correctly measure the execution time of the service.

Since previously mentioned, the end users seek the resemblance of a native user experience when using a specific service on their personal device. Response times are of vital interest when emulating the native user experience, 100 milliseconds will feel instantaneous for the user, while delays greater than a few seconds will significantly degrade the native user experience. To evaluate this, since the "Hello, World!" service implementation is too simple due to the artifact only containing words. In order to test the service with the user experience in mind, a second more developed application will need to be developed and tested that contains more than just plain text elements.

Following on studies what the end user request in a mobile service application, the second service will need to include a scrollable list view that allows the service to render both text and images for the user. The list view will also need specific elements that are clickable for the user, allowing them to progress within the application. To correctly test the execution of these functions, the application will need to include three different views in order to test the functions separately. The different steps will be illustrated to further demonstrate the navigation-flow of the service.

The first View will include the list view with text, this view is similar to the one in the "Hello, World!" implementation due to it only rendering text as seen in Figure 3.

![Figure 3: Lo-fi illustration of the first view of the artifact (UITableView with text)](image-url)
The first view will render a list of views that only contains written text generated from an array that contains an image value and a string value, a list view that contains data is called a UITableView. By isolating rendering and execution of list-text elements the experiment will be able to test the performance of the services by its most used feature. The different elements rendered in the UITableView of the page is divided into unique categories. The different elements in the list will if clicked upon navigate the user to a specific new UITableView depending on the category of that specific element.
The second view will render a UITableView that contains a thumbnail of an image, and also a string value of text that goes around the specific images. An illustration of the second page can be seen in Figure 4.

![Lo-fi illustration of the second view of the artifact (UITableView with text and thumbnails)](image)

The second view will include anotherUITableView with text and also a thumbnail of the image. By rendering a thumbnail of an image, the performance of the Media Layer (Table1) can be tested, this will also allow the different cache algorithms of the different implementations between React Native and Swift to be evaluated. By clicking on one of the unique images, the user will further advance into the third and last view illustrated in Figure 5.
The third and last view will include the full resolution of the specific image the user clicks on, also a string value of the text. The last view allows the testing of the last step in the iOS hierarchy (Table 1) of the service.

The second service honors the commonly used features of a mobile application, browsing a list of items in forms of elements, rendering text, images, and interaction in the system. By separating the content of the service, the application in question can be tested and also evaluated if React Native performs equal to Swift in all areas or just in a specific area, allowing for better visualizations in the results after the experiments are completed.

Iterations of the experiment and its tests will be conducted for eliminating any kind of odd data such as spikes and aim to run thousands of tests. A conclusion will be made when all the necessary data is collected and when the analysis of the data is finished. The conclusion of the experiment and the provided result will be visualized and presented with charts and tables that clearly demonstrate the differences between the two different services.

Automatic UI and performance testing will also be executed when testing both services, using a service called XCTest. XCTest is the primary test environment distributed inside of the Xcode IDE and is an atomization tool that allows automatic interaction-test to evaluate the service. XCTest will allow testing the UI view of the different components if they function correctly and also how fast the code in question execute the function ("Testing with Xcode - Apple", n.d). By analyzing the
health of the specific components optimization can be achieved. The atomization part will allow human tester interaction to be emulated, by automating, thousands of tests can be achieved per day. This also allows to benchmark the battery drain of the physical device in order to also measure the effectiveness of the service in question if necessary, even though execution time is the main evaluation in this experiment.

4.2 Research Ethics

Both Swift and React Native are technologies which are open-source projects. The application that will be created during this development will not include any sensitive information or data. The data that is planned to be used for this project will not be real live data, therefore, it should not potentially cause any harm if released to the public. Also, a completely unbiased mindset has to be acquired in order for appropriate results during the experiments.

4.3 Reliability

By measuring the execution time in the different UITableViews of the services, an overview can be produced that demonstrate and shows the performance differences between the two different applications. By executing the tests in a controlled environment with the same possibilities and restrictions fairness can be ensured and therefore the data can be correctly compared (Michiel et al., 2016).

4.4 Measurement Environment

In order for fairness to be ensured all tests will be performed using the same testing device, presented in Table 3.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Device</th>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>iPhone 6 Plus</td>
<td>iOS 11.3</td>
<td>Dual-Core 1.4 GHz Typhoon</td>
<td>2gb</td>
</tr>
<tr>
<td></td>
<td>64gb</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The physical device chosen for the experiments will be an iPhone 6 Plus with a CPU of 1.4 GHz Dual-Core Typhoon and 1GB of DDR3 RAM running iOS v.11.3.
5. Implementation

5.1 Literature study

Spyros and Stelios (Spyros et al., 2013) take the user experience into account when developing a mobile application using different development approaches where the goal is to achieve native application performance. The focus is to lower fragmentation as much as possible while still achieving a user-perceived native developed performance in the developed application in question. The performance metric is the loading speed and execution time of the code. They develop an RSS client that fetch data that contains the latest Apple's press info news (http://www.apple.com/pr/feeds/pr.rss) where the data, therefore, is looped through an HTTP-client and evaluated with a JavaScript-object in runtime. The conclusion of the experiment is that there is a big gap between the execution time of a cross-platform solution when compared to a native developed application.

Peixin, Xiao and Maokun (Peixin et al., 2016) highlight in performance the viewpoint on both the user and the developer, where debugging and upgrading is discussed. The user experience is summarized in two aspects, ease of use and functionalities of the application in question. The experiment measures the installation time, boot-time and the CPU occupancy ratio of the developed application, where a vertical test is performed measuring the different amount of functionality of the application (camera, accelerometer, media player and GPS). The project concludes that there is no perfect way to evaluate hybrid and native paradigms of development, but in the performance test of the experiment, the native developed application performed better.

Salma, Zakaria, and El Habib (Salma et al., 2014) focus on different performance measurement approaches. One of which is a runtime approach. A runtime approach allows a developer to write a scripting-language to test the developed application in question, the suggested programming-languages is either JavaScript, Lua or Ruby. The other approach that is suggested is the source code translator approach which focuses on a source code translator written in byte code in either C or JavaScript or directly assembled into machine code. The main priority is to lower the platform fragmentation while still considering the performance of the application. The conclusion of the result is that third-party APIs is more limited and has a lower performance rate when compared to their native counterpart.

Lisandro, Nicolás, and Leonardo (Lisandro et al., 2017) present a performance study of different approaches used to develop software for mobile devices. They present a total of 42 test cases that consist of intensive mathematical calculations in order to evaluate the execution time of the code. They compare the results of a native
developed application in both Android running Java, iOS running Objective-C and also cross-platform solutions like Cordova, Xamarin, and Titanium. The results point to the cross-platform solutions to perform just as good as their native counterpart. The mathematical experiment points to being a solid solution when developing measurement scripts when comparing cross-platform to a native developed application.

Michiel, Jan, and Vincent (Michiel et al., 2016) measure the response times for three different locations, the launch time of the application, the time to load a new page and the time to return to the previous page of the application. In their report, they state that the selected page to test is the "favorite" page of the application, this due to being the only page of the application that doesn’t require an internet connection. This is important when comparing two different frameworks since relying on an internet connection in order to load a component introduces additional communication overhead. The selected measurement tool in their report is the Instruments Tool provided trough Xcode for the iOS-system. The Instruments tool makes it possible to display the total execution time of each component of the application. It is also possible to implement and measure components startup times of the application in order to log the effectiveness of the launch time. A similar approach was taken for the page navigation response times as well in their report. Michiel provides the most detail high-level overview of every step in their progression and also why every step is taken in that direction. They make it clear that in order to fully test an IOS-system application the Instruments tool is the most logical approach due to third-party not being able to launch as fast therefore introducing a small delay when measured. A small delay at the start will make it more difficult to measure the execution time of the code when the application first is launched. Therefore the major progression of the development process will honor the development workflow of what was established in the report of Michiel.

5.2 Development Process

The steps in the development progression will be presented in this section. The native development in Swift and the React Native framework will be presented in each separate category in order to more clearly separate the two different approaches.
5.2.1 Native iOS Setup

The main recommended IDE for developing native applications for the Apple ecosystem is the Xcode IDE. For this thesis, the IDE Xcode version 9.3 distributed from the App Store available in macOS High Sierra version 10.13.4 was used. No additional software or hardware is necessary to create and run an iOS-project. Also, no previous settings were imported and all the options during installation were set to their default value.

In order to test on a physical device, the developer trusted mode needs to be activated, this can be achieved by connecting the physical device by USB-cable to the Mac, after that navigating to settings on the physical device > developer > trust computer. This step is also necessary to if the developer wants to debug their application.

When booting up the Xcode IDE the user will be greeted by a welcome screen. The taken steps after is as follows, create a new Xcode project > single view app > select product name > select language swift > create.

![Figure 6: Hello, World! example view of the start in the iOS version developed in Swift.](image)
import UIKit

class ViewController: UIViewController {

override func viewDidLoad() {
    super.viewDidLoad()

    _ = UIScreen.main.bounds

    let label = UILabel(frame: CGRectMake(0, 0, screenSize.width, screenSize.height))
    label.text = "Hello, World"
    label.textAlignment = NSTextAlignment.center
    self.view.addSubview(label)
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
}
}

Figure 7: Hello, World! implementation developed in Swift.

Figure 6 shows the native iOS version of the "Hello, World" example. Figure 7 presents the implementation, written in Swift. Constrains is used to style and dynamically position elements on the screen, similar to how CSS operates and functions in web development. The viewport of the device is declared with UIScreen and screenSize (line 9, 11). This allows the program to retrieve information about the device's screen size. The label declaration is used to display the text "Hello, World" (line 12) to the screen while textAlignment (line 13) is used to position the label element to the center of the screen of the device. These lines of code are declared inside of the viewDidLoad function (line 6) which is executed once the view has loaded to the device's screen.

5.2.2 React Native Setup

In order to set up a React Native project, a few different pieces of software needs to be installed on the computer. First of all, in order to access the iOS library, Xcode needs to be installed. The second program that needs to be installed is Homebrew, which can be located at brew.sh. Homebrew is a package manager that makes it possible to download and install different dependencies from the internet via the built-in terminal distributed inside of macOS. Node.js and Watchman are two additional dependencies that need to be utilized in order to run a React Native project. Node.js allows the written JavaScript to be utilized on the local machine, this is necessary to emulate the iOS environment. Watchman is a file watching system that rebuild the assets when writing the codebase in question. Installing
node.js and Watchman can be achieved by typing "brew install node watchman" inside of the terminal. The last step is to download the React Native CLI by running inside of the terminal "npm install -g react-native-cli".

When all these software dependencies are installed a React Native project can be created by running inside of the terminal "react-native init". This command creates all the needed files for developing an iOS application.

![Hello, World! example view of the start in the iOS version developed in React Native.](image1.png)

**Figure 8:** Hello, World! example view of the start in the iOS version developed in React Native.

```javascript
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello, World!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create(
  { container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  } }
)
```

**Figure 9:** Hello, World! implementation developed in React Native.
Figure 8 shows the "Hello, World" version of React Native. Figure 9 shows the implementation developed in React Native using JavaScript ES6. Different types of libraries need to be imported in order to run and execute the application. The first library is called "React" (line 1). The React library contains by its name all of the React logic which can also be used for web development. React utilizes a programming syntax called JSX. JSX is ES6 logic written with an HTML5 like syntax. In order to develop for native, a second library needs to be imported which can be seen in line 2 which imports Stylesheet, View, and Text from the React Native library. These are native components that utilize the native part of both iOS and also Android.

The React JSX is called by its extends statement (line 4), after that, a render function is defined (line 5). Every written code defined inside of the render function is content that will be rendered to the screen for the user to see and interact with. Since React Native utilizes JavaScript as its core programming language a value must be returned since it can not be defined as null (line 6). The first native component that is called is the View component (line 7). A View component only concern is to display elements to the screen. Inside of the View component lies the Text component (line 8). The Text components primary concern is to render text, similar how an h1 or p-tag element works in web development. The last component is the Stylesheet component (line 14). Stylesheet allows CSS to be written and exported as a JavaScript function, this makes it possible to style and position element to the screen of the physical device. Inside of the Stylesheet component, FlexBox is utilized in order to position the elements. AlignItems (line 18) targets the horizontal line of the screen while justifyContent (line 19) targets the vertical line of the screen.
5.2.3 Application size difference

After the "Hello, World" implementation of both the applications on its respective framework, the size of the different projects was documented. All applications were ejected and deployed as release versions, which means that extra functionality like debugging was excluded from the builds, therefore making the size more reliable. The comparison can be viewed in Table 4.

Table 4: "Hello, World" implementation application size of the frameworks.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Swift</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>16 MB</td>
<td>8 MB</td>
</tr>
</tbody>
</table>

As seen in Table 4 the natively developed application in Swift takes up 16 MB in size while the React Native developed application takes up 8 MB in size. A reason for this may be the model view controller in the Swift project that handles the memory allocation by default, a comparison needs to be made between the "Hello, World!" solution and the "Travel Buddy" solution.

The result above implies that React Native for iOS will outperform the contender Swift with respect to the application size. It's however noted that the size of the application does to reveal the true performance of an application. The "Hello, World" example also doesn't take scalability into account since there are just a few elements that need to be loaded into the device. Additional tests need to be conducted by developing different applications with additional functionality and complexity.
5.3 Travel Buddy Progression

5.3.1 The iterations

This section will focus on presenting the progression and the necessary steps, design and programmatic questions regarding the end results of the Travel Buddy artifacts, also the GitHub-projects hash ID on commit will be presented in order to clarify the steps taken during the development of the artifacts.

Looking back on the design choices made for the artifact presented in 4.1 "The Method"-section a few iterations were made, as seen in Figure 3 and Figure 4. The lo-fidelity drawings presented a three-page layout where the user first had to choose a category and after the category-selection, select a destination to arrive on to the details page. This was changed in an early development stage due to the focus to test code execution when evaluating the performance of the created artifact, the first two pages didn’t present any technical difference to distinct on another, therefore the first page of the artifact had to be removed.

During beginning of development the artifacts had the same file size of all the images Git ID#686938f, there was no difference between the first and second page in file size of the images, this was later changed in Git ID#a4802b0 to smaller file sizes of the images in order to more optimize the application, a performance comparison between the higher resolution images compared to the more optimized images will be presented in the measurement section.

Also presented in Git ID#a4802b0 was the first major overhaul of the Swift-project where the project adapted to an MVC-structure allowing the project to handle more elements with ease.

Git ID#901cd36 presents the final iteration of the Swift-project were final touchups were implemented such as constraints in order for the artifact to function properly and adapt dynamically to the users' viewport. This is also important for the model view controller which handles memory consumption to only load and render elements that are visible to the users' viewport. The last important feature is the data load from a data-scheme where the artifact reads and loads data from a schema and then dynamically renders and updates accordingly the main files, this makes the artifact just as effective to handle 3 elements as it were to handle 1 000+ elements.

Looking at Git ID#3f633c6 the first iteration of the React Native artifact was created.

A couple of Components were created, mainly an overall import Component, and also a Deck-Component, where the Deck-Component were to hold all the data for the listView-items, honoring the MVC-structure established in the Swift-project.

The next update can be viewed at Git ID#cd2e338 where props and state were implemented in order to have reusable Components, following best practice
development principles with the React framework, also the overall design was established.

The most important mid-update can be seen at Git ID#0b70c3b where a "FlatList" Component was implemented which can be seen in Figure 10. A FlatList main purpose is to render a list of items to the screen which the user can perform actions against, however a FlatList only renders in memory the visible list presented on the users screen, which means that those elements that are above or below the viewport are not in memory, this is of vital interest when loading a large amount of data, and also makes the behavior similar to the Swift-artifact experience, however the FlatList had to be removed. Looking at Git ID#4327819 the FlatList Component had to be changed to a navigator Component called "StackNavigator". The reason for the change is that during this time of development the FlatList Component does not support offline reading of a data-schema-list, therefore the only way to make a fully functional FlatList Component was to fetch the data from an external source such as an API. To correctly honor the established tests and measurement practices in this report, and evaluate the artifacts in a controlled environment the FlatList function had to be cut.

However, the StackNavigators main purpose is to handle dynamic state-updates that respond to a users action, which makes it online independent. When a user performs an action against one of the elements the StackNavigator handles the request as a function that is called and then transfers the user to the page that triggered the corresponding action.

```javascript
01. // the lifecycle
02. export default class App extends React.Component {
03. render () {
04. return {
05. <View
06. <FlatList
07. data=[
08. require('./assets/greenc1.jpeg'),
09. require('./assets/london1.jpeg'),
10. require('./assets/paris1.jpg'),
11. ]
12. renderItem={({ item }) => {
13. return <ListItem image={item} />
14. }}
15. keyExtractor={({ index }) => {
16. return index
17. }}
18. }
19. />
20. <View
21. }
22. }
23. }
24. }
25. }
26. }
```

Figure 10: Flatlist implementation in React Native artifact.
The StackNavigator solution that replaced the FlatList Component can be seen in Figure 11. The `static` function handles the navigation-bar behavior and styling while the different arrow-functions awaits to be called upon by the user.

The final design and functionality will be presented in the next section, giving a high-level overview of both the projects while also proving that the native use and feel can be achieved from a user experience perspective from a cross-platform solution.

Figure 11: StackNavigator implementation in React Native artifact.
5.3.2 Final results

Two different services were developed during this project, as can be seen in Figure 12 above. The main goal was to create a service that provides the core functionalities that a user demands in today's standards, as discussed in a previous section of this report. The graphical interface represents a travel app where the user may select a travel destination of their choice.

Figure 13: Page navigation when a destination is selected.
When an element is selected the user is navigated deeper into the application where a more detail picture and description of that travel area is presented as can be seen in Figure 13.

At a first glance, the services look similar to one another but the implementation of the two is vastly different. The Swift service is developed with the MVC model in mind. The data is provided in a "DataService.swift" file while the views are broken into two different sections "CategoryCell.swift" and "ProductCell.swift". The model layers are divided into "Category.swift" and "Product.swift", this in hand corresponds to the Controller in Xcode with the files "CategoriesVC.swift" and "ProductsVC.swift". The native project is divided into different story sections via the "Main.storyboard" that combines declarative swift files with the CocoaTouch Framework for the iOS-system that gets interconnected with @IBOutlets.

The React Native project is structured with a main "App.js" file that imports the rest of the files in order to correctly initialize them all. The structuring process gets vastly different because React Native doesn’t have a Controller that controls the navigation and memory flow of the project. Therefore everything in React Native needs to be divided into different kinds of functions. An example of this is the navigation part of the React Native service where each file is divided into different kinds of screens. When a screen is clicked a "this.props.navigation.navigate" function is invoked, the function then search for the location of that specific props in order to know which props should be triggered. The different location selections, Santorini, London or Paris is divided into one specific JavaScript file each, while in Swift they are stored into a data-table that later push the stored information into the different swift files previously mentioned.
5.4 Pilot study

A few measurements were made to test and evaluate the method previously established. The measurement test was to measure the cold boot time of the application, as well to measure the rendering time when navigating to a new screen in order to evaluate the execution time of the code. In total 10 performance tests were made that measures the code execution in milliseconds (ms).

![Graph charts displaying the first performance test in ms and MiB (less is better).](image)

The data display the first performance runtime in ms (milliseconds) as well as the memory usage in MiB (mebibyte). As can be seen in Figure 14 the results of rendering the 3 elements to the users’ viewport points to React Native performing slightly more effective compared to the native developed Swift-project. The memory consumption also as seen in Figure 14 seems to be slightly less efficient on the Swift side compared to the cross-platform solution React Native.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Swift</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>27.2 MB</td>
<td>30.1 MB</td>
</tr>
</tbody>
</table>

Table 5: "Travel Buddy" final implementation application size of the frameworks.

As can be analyzed in Table 5 the application size comparison shows that the React Native project has surpassed the Swift project in file size when compared to Table 4 in a previous section of the report.

As of right now when analyzing Figure 14 and Table 5, the data points to Swift being more adapted for larger scale projects while React Native may be more suitable for smaller projects.
Overall the method works to measure and evaluate the performance of the app, although further research needs to be implemented in order to analyze the artifacts and get more conclusive data to prove the hypothesis right or wrong.
6. Evaluation

Two artifacts were created, one natively build for the iOS system which is in Swift, the other presented artifact created in the cross-platform solution that is React Native for both the iOS and Android system. The reason is to prove the hypothesis true by performing an experiment to test the code execution for both services and to compare to results to one another in order to evaluate the performance.

![Instrument Tools overview when performing a test.](image)

Every test was performed by doing a UI automated test inside of the instrument tools, which can be observed in Figure 15. The workflow was to cold boot the application in question. After that start the navigation the same time that a user can interact with the application, then select every element selectable one by one, when entering an element, back out again back to the list-view, when all elements visible in the users viewport have been selected two times and backed out again, then conclude the test. When a test is performed instrument tools allow to record the different processes that the application run in order to perform the implemented functions. This, in turn, allows to separate the different functions and further analyze them one by one or all together, it’s also possible to analyze how the different cores inside of the physical device perform together, which can be further optimized to more efficiently split the resources between the cores manually.

Every test that will be run is all from cold-boot, where 10 runs will be made for each test, then an average of the 10 runs will present the result in a graph. The first test will load 3 elements, the second 5 elements, the third 10 elements and the last 100
elements. This experiment is performed in an isolated environment as mentioned in an earlier part of the report which will further improve the quality of the data since no external environmental factor can be presented such as ethernet-spikes. If the data can prove that the artifacts can perform under a rendering time of 2 000ms then the hypothesis can be proven true if the react native artifact proves to perform over 2 000ms, then the hypothesis is proven false.

6.1 Data-graphs presentation

Figure 16: Code execution for 3 elements in ms.

Figure 16 illustrates the rendering of 3 elements to the users' viewport. The blue bar represents the execution time from React Native, while the green bar represents the performance from Swift. The data from this graph suggest that the end user will not take notice of any performance difference between the artifacts. Overall React Native perform 7.7% better when compared to Swift in this test.
Figure 17: Code execution for 5 elements in ms.

Figure 17 illustrates the rendering of 5 elements to the users’ viewport. The blue bar represents the execution time from React Native, while the green bar represents the performance from Swift. This graph also points to that the end user will not notice any performance difference between the artifacts. It’s however noted that 5.2% is not as big compared to the Figure 16 graph.

Figure 18: Code execution for 10 elements in ms.

Figure 18 illustrates the rendering of 10 elements to the users' viewport. The blue bar represents the execution time from React Native, while the green bar represents the performance from Swift. In this graph, there is clearly a difference in performance between the artifacts. A performance difference at 24%, a difference
large enough for the end user to notice a difference. However, the end user will still find the performance good enough without feeling frustrated.

Figure 19 illustrates the rendering of 100 elements to the users' viewport. The blue bar represents the execution time from React Native, while the green bar represents the performance from Swift. In this graph, the performance of the React Native artifact is far from a discretionary user experience for the end user. The Swift artifact still holds up providing a good user experience for the end user. The comparison here holds up to 114% in performance difference between the two artifacts.
6.2 Data-graphs analysis

There is much to say about the presented data from the different graphs. React Native proved to perform in some instances better when compared to Swift. When rendering 3 elements to the viewport, looking at Figure 16, React Native performed 7.7% better. When that number was 5 as in Figure 17, React Native still performed better with the full extent of the users' viewport, a winner of 5.2% when compared to the performance of the Swift project. Extending the elements to 10 is where we first noticed a severe difference in performance between the two artifacts. When studying Figure 18 we can observe a performance difference at 24% to the favor of the Swift project. The last test which served 100 elements to the artifacts proved to be the biggest difference between the two, showing a whopping difference with 114% in performance difference in the favor of the Swift project.
Table 6: Confidence intervals of execution time.

<table>
<thead>
<tr>
<th>Element Size</th>
<th>Swift</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 elements</td>
<td>0 MS</td>
<td>0 MS</td>
</tr>
<tr>
<td>3 elements</td>
<td>1201 MS</td>
<td>1115 MS</td>
</tr>
<tr>
<td>5 elements</td>
<td>1394 MS</td>
<td>1325 MS</td>
</tr>
<tr>
<td>10 elements</td>
<td>1550 MS</td>
<td>1923 MS</td>
</tr>
<tr>
<td>100 elements</td>
<td>1761 MS</td>
<td>3723 MS</td>
</tr>
</tbody>
</table>

As can be observed in Table 6 and more clearly illustrated in Figure 20 when rendering data of 10 elements and more, the difference between the artifacts becomes more noticeable. When observing the rendering of 100 elements to the viewport, it’s clearly evident that React Native scales a lot worse. Table 6 reinforces statistically the confidence intervals since there is no overlap between the different artifacts. However, the confidence can't be stated to be accurate in all cases outside of these provided artifacts due to limitations in the Xcode testing environment.
because of the restrictiveness of the Apple ecosystem. Pure data-dumps can’t be accessed and therefore cannot be analyzed to confirm the true performance and scalability of the provided frameworks. This experiment can therefore only claim to an extent that the collected data presented in this work and in these artifacts is true. This work can therefore only give a suggestion when choosing the right tools for smartphone application development, but cannot confirm which of the frameworks performs better when compared to the other when developing different applications compared to the created artifacts in this experiment.

A conclusion with a discussion regarding the statistics of the performed tests is presented in the following section, addressing the strengths and concerns regarding the evaluation of React Native as a cross-platform solution.
7. Final Conclusion

7.1 Summary

The goal of this work has been to prove that React Native can perform equal to that extent that the end user won’t notice a difference in performance against a native developed application. By collecting and achieving more data to further advance the discussion regarding solution alternatives for companies when developing mobile smartphone applications to further extend their services. Two different artifacts with the same design and functionality were created, one with the cross-platform solution that is React Native, the second artifact developed as the natively developed solution that is Swift. Both artifacts were recorded and evaluated from an experiment using Instrument Tools provided inside of the Xcode IDE and tested in the iOS-system on a physical iPhone 6 Plus device.

The hypothesis is proven true that React Native can be a solution for some companies looking for a cross-platform solution in order to extend their services through a smartphone application. However, the hypothesis is also proven false when looking at rendering a large number of elements at the same time onto the device viewport. Therefore suggesting that there is no perfect way to fully test cross-platform solutions against native solutions applications.

The data points to React Native being a viable solution when developing smaller applications, while Swift seems to be the better solution when developing larger applications such as enterprise levels.

7.2 Discussion

There is a lot to win when getting more data regarding different approaches in mobile smartphone application development. Companies today face some difficult questions regarding being able to provide their services through an application on the end costumers smartphone device. Developing smartphone applications is a time consuming and costly option for the companies, as previously stated in section 3.1. The development also becomes far more complex and time-consuming when taking fragmentation into considering, having to develop two applications with basically the same look and functionality just to having their services accessible through the two major mobile smartphone operating system that is Android and iOS. Therefore it’s clearly essential to evaluate different approaches for mobile application development that can reduce the time and complexity, also in some instances even reduce the development team of two to one.
There were a ton of new grounds to learn and cover during the development of this project. Coming from a background of a web developers experience and going into the development of modern smartphone applications was a daunting and difficult journey, especially developing in two new programming languages with zero experience in advance. Development of the artifacts took a great deal of time due to no previous experience with the technologies, it also required a ton of theory and research to fully understand both the different technologies. The technologies also presented a big contrast between them due to them being extremely different in every way from one another, from a developers perspective that is.
During the development of the React Native project, it was clear that in order to keep up performance wise to the Swift project when scaling the application in question was to implement the FlatList component. A visualization of how the data and rendering process works with the model view controller can be observed in Figure 21. The colored elements represent the users' viewport of his/hers physical device, the colored elements is also the data that get loaded into the device. The uncolored/grey elements outside of the viewport represent elements in a queue to be loaded into the device. The model view controller, therefore, handles all of this data-handling according to the users' viewport. This technique is also how the FlatList component in React Native operates, which would significantly influence the data during the experiment if it had been implemented. Unfortunately, the FlatList component needs to have a key operator which handles every processed element with a unique ID-tag, in turn, an initialization of the ID-tag needs to collect and fetched from a list of items, that is a JSON-based array. Fetching this kind of array has yet to be implemented when fetching or requiring a locally stored list of elements, therefore only making this type of technique possible with an online fetch, which in turn would corrupt the experiment anyway because of the isolated requirements. This is an extremely important point to address since React Native literally have to fetch and render every element required since there is no queue of data.
A requirement for a natively developed application is that the user should be able to interact with the application, even in offline mode. The data from this report is therefore still relevant for companies, proving that it’s possible to develop an application with cross-platform technologies that allows the processing of an offline-friendly application, which makes React Native a more viable option when compared to some online-only cross-platform solutions previously available.

This report has also proved that it’s possible to create and emulate the native feel and design with a cross-platform solution that is React Native since the presentation of both artifacts is equally presented.

The performance in 3x10 out of 4x10 tests showed that React Native can handle code execution operations under 2 000ms, which is the limit before an end user will feel the application in question less responsive. Which proves that React Native can be a viable option when rapidly developing a certain extent in size of an application.

One of the cons with this work is the lack of online-testing, one can argue that online-work is a priority due to the norm in today's society, one can also argue that to only focus on online activities is considered bad practice.

This work also doesn’t take the others factors of the user experience into consideration such as system maintenance from a developers perspective. The report only highlights the subject to the extent when discussing the problem about code fragmentation and splitting the codebase between the different systems. Notes like system updates and micromanagement are not taken into consideration in this report and experiment.

It’s also worth mentioning that no test subjects were used in this experiment which would be beneficial with providing a wider extent of user data, the report only takes the parameters between 0 ms to 2 000 ms which is collected from previous experimental reports that analyze and evaluates in the same technical field such as this one.

### 7.3 Learnability

React Native allows for the development of smartphone applications with the JavaScript programming language. JavaScript is traditionally used for web development and will be unfamiliar to most in the context of developing smartphone applications for native. This is therefore important for organizations wanting to utilize React Native to know how learnable the framework is, either for web developers with a JavaScript experience wishing to develop for smartphone applications or native developers looking to develop for cross-platform.

Prior to this work, the author had experience with both web development, and also the React framework for the web, but zero experience with Swift or how to develop
a smartphone application, to begin with. It’s therefore important to clarify that the learnability-section can be somewhat biased. So, during the early time of development, the artifacts were mainly about understanding the iOS environment and to further understand the Swift & Cocoa Touch syntax. Understanding the core concepts of Swift was manageable fairly quickly, but understanding how to incorporate the Cocoa Touch framework in a smartphone application setting for the iOS proved to be a real challenge. Understanding JavaScript with React Native in a smartphone application setting was fairly easy and development took off very rapidly. One of React Native’s strengths lies in its refresh ability, where the project auto-rebuilds the project each time the developer saves the codebase. This makes it possible to debug rapidly and also develop fast with a try and error approach, compared to native that doesn’t allow fast auto-refreshes.

This to the extent of this context, suggests that web developers will not have an issue adjusting to a smartphone application setting when developing for React Native, but suggests that a pure web developer with zero experience with Swift or native development will have a harder time adjusting to the environment of native.
7.4 Future work

One can, by all means, take this work into consideration and develop it further, and further help evaluate React Native as a solution to native smartphone application development. One direction, as previously touched upon is the lack of online implementation and testing. This work proves that code execution can be on par with a natively developed application in the Swift programming language. The work still doesn’t do a good job presenting the true scalability of the React Native framework. The failure to implement FlatList as a viable solution is also lackluster and leave much to be desired since the final results would most likely be very different.

It would also be interesting to implement a more advanced architecture, especially in the React Native solution to further test the available functionality provided in the framework compared to what can be done in functionality from the natively developed side.

It’s also important to mention the lack of focus when it comes to the Android side of development. This report mentions that React Native works as a cross-platform solution and that the codebase written once can function on both the iOS system and also in Android. Further work to extend the testing into the Android-environment would be beneficial in the evaluation process for React Native as a framework. This would require more research into the testing phase since Xcode with Instrument Tools is only accessible for iOS applications. So an alternative testing solution would be required, which also comes to further evaluate that testing solution in order to confirm the viability of the results provided from that experiment when compared between the different solutions and systems.

As a last note, performing the same tests on different physical devices with different variations in its hardware would also lead to a wider and more solid amount of data, which would lead to further evaluate the React Native framework when it comes to operations on low-end devices and also high-end devices.

This will conclude the report.
Thank you for sticking with me this far!

/ Dennis
8. References


The fifteen most popular languages on GitHub - GitHub [WWW Document], n.d URL https://octoverse.github.com [accessed 06.04.18]

Swift 4 The powerful programming language that is also easy to learn - Apple [WWW Document], n.d URL https://developer.apple.com/swift/ [accessed 08.12.17].

Rahul, R., and Seshu, T. (2013) A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In India Conference (Indicon) 2012 Annual IEEE.


Appendix A - Swift project (external file)

Appendix B - React Native project (external file)