Cross-platform development of smartphone applications
An evaluation of React Native

Martin Furuskog
Stuart Wemyss
Abstract

Cross-platform development of smartphone applications

Martin Furuskog & Stuart Wemyss

During the last ten years the market for smartphones has grown drastically. Because of the current state of the market with different operating systems many smartphone applications need to be developed for several platforms. With this thesis, the goal was ultimately to contribute to the understanding of cross-platform development as a way of developing smartphone applications. React Native has been evaluated as a framework with which development is done for both Android and iOS using the same code. To study React Native as a framework, a smartphone application for Android and iOS was developed at an Uppsala based IT-company with expertise in web services, smartphone applications, and online gaming. Furthermore, performance tests and user tests were conducted in which React Native was compared to native applications and applications developed using Xamarin (similar cross-platform development framework owned by Microsoft). It was evident that using the same code for both Android and iOS was time saving. However, the performance tests results showed that applications developed with React Native did not perform as well as the native and Xamarin versions. Leading to the conclusion that choice of framework when developing cross-platform applications need to take into consideration performance, development time, and programming language preference.
Populärvetenskaplig sammanfattning


Denna rapport syftar till att utvärdera React Native som verktyg för utveckling av smartphone-applikationer för Android och iOS. Förutom under utvecklingen av applikationen på Interactive Solutions har detta även gjorts i samband med mindre applikationer som utvecklats med React Native men också separat till Android och iOS samt med Xamarin (ett ramverk liknande React Native). Med dessa applikationer har prestanda- och användartester utförts tillsammans med ytterligare jämförelser.

Table of Contents

1. **Introduction** .............................................................................................................. 6
   1.1. Problem statement and purpose .......................................................................... 6
   1.2. Scope and delimitations ...................................................................................... 7
   1.3. Disposition ............................................................................................................. 7
2. **Background** ............................................................................................................... 8
   2.1. Smartphone applications ...................................................................................... 8
   2.2. Cross-platform approaches .................................................................................. 8
       2.2.1. Web approach ............................................................................................... 8
       2.2.2. Hybrid approach (web and native) ................................................................. 9
       2.2.3. Cross-compile approach .............................................................................. 10
       2.2.4. Interpreted approach .................................................................................... 11
   2.3. Cross-platform tools ............................................................................................ 13
       2.3.1. React Native ................................................................................................ 13
       2.3.2. Xamarin Platform ....................................................................................... 14
   2.4. Related research .................................................................................................. 14
3. **Method** ..................................................................................................................... 18
   3.1. “Hello, World!” .................................................................................................... 18
   3.2. Performance evaluation ....................................................................................... 18
       3.2.1. Performance measures ............................................................................... 18
       3.2.2. Measurement environment .......................................................................... 19
       3.2.3. Bubble sort .................................................................................................. 19
   3.3. Native user experience ........................................................................................ 20
       3.3.1. Application design ....................................................................................... 21
       3.3.2. User testing with interviews ....................................................................... 21
   3.4. Real-life case ......................................................................................................... 22
4. **Results and analysis** ............................................................................................... 24
   4.1. “Hello, World!” .................................................................................................... 24
       4.1.1. Native Android setup .................................................................................. 24
       4.1.2. Native Android implementation ................................................................. 25
       4.1.3. Native iOS setup ......................................................................................... 26
       4.1.4. Native iOS implementation .......................................................................... 26
       4.1.5. React Native setup ...................................................................................... 27
       4.1.6. React Native implementation ..................................................................... 29
       4.1.7. Xamarin setup ............................................................................................. 30
       4.1.8. Xamarin.Forms implementation ................................................................. 31
       4.1.9. Setup and development comparison ............................................................ 32
4.1.10. Application size comparison ................................................................. 32
4.2. Performance evaluation .................................................................................. 33
  4.2.1. Android .................................................................................................... 34
  4.2.2. iOS .......................................................................................................... 35
4.3. Native user experience .................................................................................... 37
  4.3.1. The application ......................................................................................... 37
  4.3.2. Test results ............................................................................................... 38
4.4. Real-life case .................................................................................................. 38
  4.4.1. Documentation .......................................................................................... 39
  4.4.2. Community ............................................................................................... 39
  4.4.3. Maintainability ......................................................................................... 39
  4.4.4. Testing ...................................................................................................... 41
  4.4.5. Learnability ............................................................................................... 42
  4.4.6. Cross-platform potential .......................................................................... 42
5. Discussion and conclusions .............................................................................. 44
6. References .......................................................................................................... 46
Abbreviations and Terminology

C# A general-purpose, object-oriented programming language.

DOM manipulation Allows for the navigation and manipulation of documents through a programming language.

DMG Container disk image file commonly used to distribute applications for Mac OS X.

IDE Integrated Development Environment, software for development.

JavaScript Cross-platform, object-oriented scripting language.

PHP Open source server-side scripting language designed mainly for web development.

Figures

Figure 1. Web mobile application architecture, inspired by Raj and Toley (2012).
Figure 2. Hybrid mobile application architecture, inspired by Raj and Toley (2012).
Figure 3. Cross-compiled mobile application architecture, inspired by Raj and Toley (2012).
Figure 4. Interpreted mobile application architecture, inspired by Raj and Toley (2012).
Figure 5. Pseudocode for Bubble sort.
Figure 6. Screenshot of Android version of Hello, World! run on the Android test device.
Figure 7. “Hello, World!” code for Native Android.
Figure 8. Screenshot of iOS version of Hello, World! run on emulator through Xcode.
Figure 9. “Hello, World!” code for Native iOS using Swift.
Figure 10. React Native version of Hello, World! run on iOS emulator through Xcode (left) and Android test device (right).
Figure 11. “Hello, World!” code using React Native.
Figure 12. Xamarin version of Hello, World! run on iOS emulator through Xcode (left) and Android test device (right).
Figure 13. “Hello, World!” code using Xamarin.Forms with C#.
Figure 14. Mean values of execution times in linear scale for Bubble sort on Android test device.
Figure 15. Mean values of execution times in logarithmic scale for Bubble sort on Android test device.
Figure 16. Mean values of execution times in linear scale for Bubble sort on iOS test device.
Figure 17. Mean values of execution times in logarithmic scale for Bubble sort on iOS test device.
Figure 18. Screenshots from the native version of the Android user test application.
Figure 19. Screenshots from the native version of the iOS user test application.
Figure 20. One of the problems encountered during development was the right button of a view’s navigation bar (highlighted in red) being inherited by the following view with no such button.
Figure 21. The application’s chat view, displaying a conversation between two nightclub enthusiasts.
Figure 22. Views of the application adapt to the keyboard, as seen when comparing the left and right image.

Tables

Table 1: Characteristics of web, hybrid, cross-compilation, and interpreted approaches.
Table 2: Devices used for testing.
Table 3. Metrics used in the evaluation of React Native.
Table 4. “Hello, World!” application size for the respective platform and framework.
Table 5. Confidence intervals of execution times for Bubble sort on Android test device.
Table 6. Confidence intervals of execution times for Bubble sort on iOS test device.
1. Introduction

Since the release of the iPhone in 2007 and Android in 2008, the market for smartphones has grown drastically. From being a type of device targeted mainly at business people, with early smartphones such as Palm Treo in the late 90s and BlackBerry at the turn of the century, the smartphone has gained a much wider audience and innovated the mobile market much because of the iPhone (Allen et al., 2010). Phones are no longer viewed simply as phones. They provide a range of features and have become a major part of people’s lives. One of the smartphone’s key features is the ability to install applications. These applications are similar to the ones on for example computers, offering a limited set of functions, and often either come pre-installed on the phones or are available for download from the platforms’ different application stores, such as Google Play for Android and App Store for iOS. With the current market having several widespread platforms it is often required applications are developed for more than one platform.

Consequently, companies that develop mobile applications must hire developers with skills in a large set of languages to reach the wide audience. As a response to this, cross-platform development has emerged which essentially allows one implementation that works on multiple platforms. Using a cross-platform approach, developers can thus avoid repetition and increase productivity. There are a number of cross-platform tools and frameworks that has been around for years, such as PhoneGap and Xamarin. However, new tools and frameworks are constantly being developed with new features and architecture. One of these is React Native.

In 2015, Facebook released React Native, a framework which among other things enables the development of smartphone applications simultaneously for Android and iOS by using the same JavaScript code for both platforms. This type of development is of great interest, as it is both expensive and time-consuming to develop native applications separately for each platform. Research has been done on cross-platform development of mobile applications as well as other mediums such as games. However up until this point none of it has covered development with React Native.

1.1. Problem statement and purpose

The market for mobile applications is huge and growing. Understanding cross-platform development is therefore important. The overall purpose of this thesis is to provide knowledge as we begin to understand the potential of React Native as a tool to develop cross-platform applications. The goal is also to evaluate React Native in contrast to more traditional parallel development. Furthermore React Native is compared to Xamarin, a cross-platform development framework owned by Microsoft (as of February 2016).

According to an analysis of current application development conducted by Vision Mobile (2015), Xamarin was the second most used tool for cross-platform development in 2015.
Xamarin differs in architecture and programming language from React Native, but also allows for development for Android and iOS as well as Windows Phone.

No claim is made to determine which approach to cross-platform application development should be used. The goal of this thesis is rather to contribute to understanding the approaches and provide a basis for the choice of tools when developing for multiple platforms.

To fulfill its purpose, the thesis attempts to answer the following questions:

- What is the potential of React Native as a tool for cross-platform development of applications?
- How do cross-platform applications developed with React Native compare performance-wise to cross-platform applications developed natively and with Xamarin?

To clarify, the first question refers to the degree which applications can be developed cross-platform with full functionality for both Android and iOS, without having to develop separately for the platforms.

1.2. Scope and delimitations

The scope of this thesis has been delimited according to the problem statement, defined earlier in the introduction, and the time limit. The application for the real-life case will exclusively be developed using React Native since this is the primary framework to be evaluated, and not with any other framework. Instead, a selection of core functionalities and components of the application will be implemented natively and using Xamarin for the purpose of the comparison. Since the application is supposed to be released for iOS and Android, the study has been limited to these platforms only.

1.3. Disposition

The remainder of the thesis is divided into four sections. First a background is presented, covering smartphone applications, cross-platform development, and related research. The goal is to provide the reader with necessary information as well as definitions which are important to the thesis. Second, the different methods used to answer the questions are presented. They are described individually, which is carried on into the third section where each set of results are presented and analysed. Lastly the results are summarized and discussed in the final section, in an attempt to answer the thesis’ questions.
2. Background

2.1. Smartphone applications

A smartphone refers to a high-end cell phone with certain advanced capabilities in comparison to a low-end feature phone. In essence smartphones possess powerful processors, advanced device capabilities such as cameras and connectivity, and high-resolution touch screens (Allen et al., 2010). Additionally, a smartphone must use an operating system that allows advanced, third-party applications that are acquired from application repositories such as App Store and Google Play (Theoharidou et al., 2012).

The use of smartphones is rapidly increasing. According to Ericsson (2015) smartphone subscriptions worldwide have increased from 2.6 to 3.4 billion between 2014 and 2015. Ericsson forecast that this will almost double by 2021, to 6.4 billion subscriptions. In mid-2015 Android was the dominating operating system in the smartphone market with a market share of 82.8% while iOS was second with 13.9% (IDC, 2015). The other OS competitors mainly include Windows Phone and BlackBerry OS. The amount of applications available at the leading application markets, Google Play and App Store, have both passed 1.5 million (Statista, 2015). Tim Cook, CEO of Apple, announced in 2016 that the App Store have had more than 100 billion application downloads (Taves and Tibken, 2015).

Each platform differs in some way and the operating systems traditionally required development in a certain native development language. For example, for Android the native language is Java and for iOS it is either Objective-C or Swift (Apple, 2016). This means that the application developers have had to develop platform-specific applications (henceforth referred to as native applications) separately in order to reach larger audiences. Consequently, companies that develop mobile applications must hire developers with skills in a large set of languages (Allen et al., 2010). As a response to this problem, cross-platform development has emerged which essentially allows one implementation that works on multiple platforms.

2.2. Cross-platform approaches

Today there are several tools for building cross-platform applications which utilise various types of architectural approaches. Four commonly used approaches are web, hybrid, cross-compilation, and interpreted. They are all described in this section, along with their respective advantages and disadvantages.

2.2.1. Web approach

One approach for cross-platform mobile development is building the application as mobile web application that is executed in the mobile’s web browser as seen in Figure 1. This approach utilise the standard web technologies such as HTML, CSS and JavaScript to build an application that resemble a native application. HTML 5 and CSS uses
advanced capabilities in doing this, such as embedded databases, local storage and animations (Raj and Tolety, 2012). This approach is utilised by tools such as Sencha Touch (Sencha, 2016) and jQuery Mobile (jQuery Mobile, 2016). IKEA Sweden’s mobile site (IKEA, 2016) is one example that is built using jQuery Mobile.

![Figure 1. Web mobile application architecture, inspired by Raj and Tolety (2012).](image)

The web approach suits certain kinds of applications since it is cheap and has the potential of building applications that work on multiple platforms with only minor changes. Another advantage is the instantaneous availability of the application since it only requires a browser to be executed. Be that as it may, this approach also has its drawbacks. One major shortcoming is the difficulty to emulate a native user interface since the developer is limited to the standard web elements. The use of the device capabilities is also restricted in comparison to other approaches (Hartmann et al., 2011). Another challenge is that developers have little control of how the application is rendered in different browsers. To overcome this and still achieve a native look and feel, much time must be put in testing. Lastly, the application cannot be released through application stores which can have a negative impact on its distribution (Raj and Tolety, 2012).

### 2.2.2. Hybrid approach (web and native)

There is also a hybrid approach to cross-platform development (Figure 2) which lies between web and native methodology. Following this approach, the application is built with web technologies but is executed in a native container which provides access to the device capabilities (Hartmann et al., 2011). The container commonly runs a WebView, a lightweight web browser where the code gets executed. With the help of frameworks, native user interface elements may be recreated with HTML (Railslove, 2015). The communication between the web- and hybrid application is usually through a JavaScript API (application programming interface). A popular hybrid cross-platform tool is PhoneGap (Phonegap, 2016). PhoneGap has been used to build for example the mobile application for TripCase (TripCase, 2016).
An advantage with hybrid applications is that these are locally stored on the device to run more efficient in comparison to a standard web application. This also allows using device capabilities such as the camera and GPS (Hartmann et al., 2011). Another advantage is that hybrid applications are released through official application stores. The disadvantages with this approach include the difficulty to achieve native resembling user interfaces, and since hybrid applications are executed inside the browser engine the performance is inferior compared to native applications (Raj and Tolety, 2012).

### 2.2.3. Cross-compilation approach

Hartmann et al. (2011) describes cross-compiling as a technique that separates the build environment from the target environment. In mobile development, cross-compiling means that the developers use a framework that provides an API that is platform-independent. Hartmann et al. argue that the API should use a mainstream programming language such as Javascript, Java, or the like. Using the API, the developer builds the application’s different parts: the user-interface, data-structure, and business logic. The cross-compiler then transforms the code into native applications for the targeted platforms, see Figure 3. Cross-platform tools such as RhoMobile (RhoMobile, 2016) and Xamarin (Xamarin, 2016a) use cross-compilation. Xamarin has for example been used to build the mobile application for easyJet (easyJet, 2016).
According to Hartmann et al. (2011) this cross-platform approach has several advantages. The first and foremost advantage is performance since the application is running natively on the platform. The user experience is also advantageous since the application behaves like a regular application in regard to the user’s mindset. Additionally this approach gives access to the device specific capabilities like the camera. A major disadvantage however is that cross-compilers are difficult both in implementing and keeping consistent with the divergent mobile platforms and operating systems being used (Hartmann et al., 2011).

### 2.2.4. Interpreted approach

The fourth approach utilise an interpreter on the target device that interprets the application code at runtime. The interpreter is often a virtual machine running on the device that executes the application like a physical machine (Hartmann et al. 2011). Through an abstraction layer the native API can be accessed (see Figure 4), exposing native features to the developer. Since each platform has its own platform specific interpreter it is possible to use native user interface elements (Raj and Tolety, 2012). One tool that utilise this approach is Adobe AIR (Adobe, 2016) which for example was used to port the game Machinarium (Machinarium, 2016) to phones.
The advantages with this approach, similar to cross-compilation, is that the native user interface access allows building applications that feels native, that device capabilities can be utilized as a result of the abstraction layer, and that it can be distributed through application stores (Raj and Tolety, 2012). According to Hartmann et al. (2011) this solution provides superior portability in comparison to cross-compilation because the interpreter is easier maintained when support must be added to new device features. With regard to performance, the interpreted approach outshines both web and hybrid. However, because the application code is interpreted during runtime there may be a performance overhead when compared to cross-compiling. Another challenge is that the approach limits the developers to the features provided by the used framework or tool (Raj and Tolety, 2012).

The four approaches’ characteristics are summarised in Table 1.
Table 1. Characteristics of web, hybrid, cross-compilation, and interpreted approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Native UI</th>
<th>Device capabilities access</th>
<th>Performance</th>
<th>Application store distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>No</td>
</tr>
<tr>
<td>Hybrid</td>
<td>No</td>
<td>Yes</td>
<td>Low</td>
<td>Yes</td>
</tr>
<tr>
<td>Cross-compilation</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
<td>Yes</td>
</tr>
<tr>
<td>Interpreted</td>
<td>Yes</td>
<td>Yes</td>
<td>Mid</td>
<td>Yes</td>
</tr>
</tbody>
</table>

2.3. Cross-platform tools

While there are several cross-platform tools on the market, two will be used for the purpose of this project: React Native and Xamarin. In this section these will be described.

2.3.1. React Native

React Native (Facebook, 2016a) is a framework for building smartphone applications using React (Facebook, 2016b). React is a JavaScript library for building user interfaces for web applications. It was first developed by Jordan Walke, a software engineer at Facebook, and deployed in Facebook’s newsfeed 2011 and Instagram’s website during 2012. The project was released as open-source in May 2013 (Krill, 2015). According to Libscore (2016) it has since then been used in for example the development of both Netflix’s and Paypal’s websites.

React began as a JavaScript port of XHP. XHP (Facebook, 2016c) is an extension of PHP, a server-side scripting language, developed by Facebook for easier front-end development and to avoid cross-site scripting attacks. It does however have problems with optimisation, which is the reason Jordan Walke initiated the development of React. The idea was to limit the amount of DOM manipulation, which is direct manipulation from the code after execution, as this is a very costly type of operation. React does this by only re-rendering a website’s content when its state changes (Dawson, 2014).

In March 2015 React Native was released by Facebook. Just like React, React Native uses JavaScript and the same type of re-rendering method to reduce the amount of DOM manipulation. The idea was to allow web style development in place of the more traditional style native development for mobile applications. At React Native’s release only iOS was supported (Asay, 2015). But as of the 14th of September 2015 the release for Android support was officially announced (Witte and von Weithershausen, 2015). This allows for the possibility to develop for both Android and iOS simultaneously, by allowing developers to use the same JavaScript codebase for both platforms.
Applications written with React Native resemble cross-compiled applications in some sense: they use native views and components potentially resulting in a native look and feel. But in contrast to cross-compilation, this approach has no compilation step. React Native applications are also similar to hybrid applications in that they are written in JavaScript and downloaded as native applications. Like the hybrid approach React Native requires an extra component for code execution, but instead of using a WebView React Native utilize JavaScriptCore. JavaScriptCore is a JavaScript engine running as a virtual machine on the device that renders the JavaScript into native-specific components (Occhino, 2015). React Native should thus rather be categorized as an interpreted approach, since the application code is translated to native at runtime.

2.3.2. Xamarin Platform

Xamarin Platform is a cross-platform tool for application development released by Xamarin in February 2013. In addition to iOS and Android, it also supports development for Windows Phone. Xamarin Platform uses C# as its code base and utilizes cross-compiling for cross-platform functionality. This means that the code is compiled to native applications for the respective platforms. According to Xamarin (2016a) this results in performance that excels cross-platform solutions that interpret the code at runtime. They estimate that developers may achieve 75% shared code base for all platforms and nearly 100% in creating user interfaces using Xamarin.Forms.

Xamarin.Forms is a natively backed toolkit for user interface development. It allows developers to create user interfaces that can be shared in all three platforms. Because the interfaces are rendered with the help of native controls, applications made in Xamarin.Forms can achieve the specific native look and feel for each platform. It is possible to use the API’s of the underlying platform in combination with Xamarin.Forms to utilize all native-specific user interface features. Developers may choose between two techniques when using Xamarin.Forms. Either implementing the user interface exclusively in C# code or using Extensible Application Markup Language (XAML), a declarative markup language used for describing user interfaces (Xamarin 2016b). Xamarin.Forms is however only recommended for applications that use little or no platform-specific functionalities and where shared codebase is more important than custom user interfaces (Xamarin 2016c). For the purpose of this thesis, Xamarin.Forms was used as it allowed a shared code base for different platforms.

2.4. Related research

Following the introduction of alternatives to native application development, the subject has been thoroughly researched. There are several studies comparing native development with one or more cross-platform approaches.

Charland and Leroux (2011) compare the development of native mobile applications with web based applications. The authors specifically investigate the hybrid framework PhoneGap with which applications are built in HTML, CSS and JavaScript. Their
Comparison is based on three aspects. First, native code vs. web code is discussed. Because of large differences in languages and software development kits between platforms, it is difficult for developer teams to reach out to multiple platforms when developing natively. Although possible, it requires a vast set of skills and backgrounds. According to the authors, the only real common denominator of the different platforms is that these provide a mobile web browser that may be accessed programmatically. They argue that this makes it easier developing applications for multiple platforms using the same code base. The second part concerns the user experience. Each platform has its own user interface conventions that the users expect to find in an application. Hence, to achieve the best possible user experience, the developers must adhere to these conventions both when designing and writing code for applications. The authors conclude that cross-platform tools therefore must provide the possibility of adapting to these conventions and argue that it is possible using web technologies. The third aspect is performance which Charland and Leroux call a cornerstone of the user experience. They quantify performance using latency and execution time and contend that while it is not yet possible to achieve the same performance using web technologies as native, it is getting closer and closer. The primary conclusion of the paper is that developers must see to the application’s primary objectives, the business and development conditions, and the current opportunities of web technologies when deciding between web and native development (Charland and Leroux, 2011).

Raj and Tolety (2012) review different cross-platform approaches which they classify as web, hybrid, interpreted, and cross compiled. They discuss the approaches’ respective advantages and challenges in achieving native resembling applications with respect to distribution, maintenance, access to device specific capabilities, performance, and design. The study furthermore suggests criteria for choosing the right approach, claiming that developers must carefully evaluate what kind of application they are developing. Raj and Tolety categorize mobile applications into Server data driven applications where the business logic resides on a server, Sensor/IO applications which mainly use the device hardware, Standalone applications where data is either produced locally or remotely but processed on the device, and Client-server applications where data processing is done both on server and client. They suggest different cross-platform approaches depending on the aforementioned categories. The general conclusion of the study is that cross-platform approaches are preferred when the targets are multiple platforms, but that developers should carefully select the right one based on the application type and relevance (Raj and Tolety, 2012). However, the authors do not address the specific tools available for development, but just merely mentions them. Additionally, the authors do not support their conclusions with any quantifiable tests or case studies.

A similar review is done by Amatya and Kurti (2013) in their meta study on cross-platform approaches. The primary goal of their paper is to identify and address the trends of cross-platform mobile development at the time. To do this, the authors conduct a literature search in two digital libraries, IEEE Xplore Digital Library and ACM Digital Library. By using the keywords “mobile” and “cross-platform” along with a set of
inclusion criteria, the authors select 17 papers for the following literature survey. They provide an outline of all papers, including problem descriptions, proposed solutions, and the technologies used. Comparing these aspects of the reviewed papers, trends are identified and analyzed. The authors conclude that web-based mobile applications gained momentum because of the evolution of HTML5, but that these applications have trouble achieving native performance. Instead, the authors suggest that the emerging hybrid approaches to cross-platform development at least in theory brings a better alternative to native development than pure web solutions. The main reason discussed is that hybrid approaches allows access to native features and device capabilities, while still utilising web based tools (Amatya and Kurti, 2013).

Heitkötter et al. (2012) compares a set of cross-platform application development tools with the development of native applications. The authors begin by outlining current approaches to cross-platform development and select three for their evaluation based on popularity among developers. In the second part, the proposed framework of evaluation criteria is described. They claim that the criteria is derived from discussions with experts from different software firms, literature research, typical topics found in online communities, and the authors’ own experience. The framework is divided into two perspectives. The first concerns the infrastructure which covers the life-cycle of an application, including usage, operation, and functionality. The second perspective involves criteria directly related to the development process, including testing, debugging, and the integrated development environment (IDE). In the third part of their study, the authors apply the criteria on the selected cross-platform approaches, as well as native development for both iOS and Android. They conclude that the tool PhoneGap performs best of the evaluated tools, unless if the applications must resemble native applications as much as possible for which others performed better. They also point out that the maturity of the cross-platform tools suggest that developers should consider cross-platform tools even though they are developing for just one platform. The reason for this is the low barriers involved when developing applications using web technologies, which are standardised, reasonably simple yet powerful, and popular (Heitkötter et al. 2012). The criteria have inspired some of the criteria used in this study. While this study essentially share the approach of Heitkötter et al. they naturally do not mention React Native since the framework was released after the study.

Dalmasso et al. (2013) also evaluates different cross-platform development tools. Using an experimental approach, the authors compare the tools with regard to performance. They begin their paper with an outline of desirable requirements of cross-platform tools, after which the authors select three for the performance assessment. These are PhoneGap, Sencha Touch and Titanium (Appcelerator, 2016). The performance test is done using a test application consisting of buttons that fires different types of web requests. The response of the requests are then parsed and displayed. The performance is measured in terms of memory usage, CPU usage, and battery consumption. The authors conclude that the application built with PhoneGap performs better with respect to all three metrics, but that the other two provides a better user interface (Dalmasso et al., 2013). However, the
test applications are exclusively implemented for Android and they propose that further assessments should include iOS and other platforms.

A number of recent theses of different levels concern the topic as well. Jonsson (2015) does a survey of current cross-platform tools and assess the performance of three tools with respect to execution time, battery consumption, and application size. Android is however the only evaluated platform. Friberg (2014) evaluates a set of cross-platform approaches by comparing his own experience of developing test applications with the scrutinised tools along with previous scientific findings. Pålsson (2014) conducts a performance evaluation of two cross-platform tools and compares them with native development. Additionally, a cost evaluation is done to investigate costs involved when adapting to cross-platform tools. The author develops test applications for the performance assessment and uses questionnaires as well as interviews for the cost evaluation.

The fact that none of these theses or articles evaluates React Native is an incentive for this project. Furthermore, whereas most of the theses and articles attempt to answer quite specific questions, sometimes only including one platform in their study, the idea for this thesis is to provide a broader perspective with a combination of quantifiable and qualitative methods.
3. Method

In order to approach the problem statement, as defined in the introduction, and reach valid conclusions, a set of methods have been used. These include a study on related research, implementation of some simple components natively and in both React Native and Xamarin, and implementation of a larger application in React Native. These will be explained below together with the measurements used in the evaluation and comparison.

3.1. “Hello, World!”

The first step in evaluating the different approaches to application development is building a simple “Hello, World!” application, as this is often seen as a natural starting point for any programmer (Hussung, 2015). There are different interpretations of the program and it can be implemented in many different mediums. In the context of this thesis “Hello, World!” is interpreted as a mobile application which displays the text “Hello, World!” at the center of the screen. This test is done to evaluate setting up the environments for development as well as the development of a simple application. Additionally, the size of the respective applications will be used for evaluation. This is done to measure and compare the size overhead of the different approaches.

There are of course different ways to develop native applications. For example, iOS applications can be developed using either Objective-C or Swift. For the purposes of this thesis Swift has been chosen for native iOS development and Java has been chosen for native Android development.

3.2. Performance evaluation

When selecting a computer system, performance is obviously an important factor. Regardless if seen from the users, the administrators, or the designers’ point of view, evaluating the performance of different computer systems is needed to select the best suited one with respect to a specific use case. To do this, the evaluator must carefully decide the right performance measures, the right environments for the measurements, and the right techniques (Jain, 2008).

3.2.1. Performance measures

Before conducting the performance study of a system, one must naturally choose a set of performance metrics or criteria. The first step may be to select what to scrutinize from the services provided by the system. When executed, these services will either perform correctly, incorrectly, or straight up refuse execution. If the service executes correctly, the performance of the service can be measured using three metrics: the execution time of the service, the rate of which it is performed, and the resources consumed during the execution (Jain, 2008).
According to Corral et al. (2012) there are a number of different metrics to choose from when evaluating the performance of a mobile application. They argue that execution time, memory usage and battery consumption are all useful aspects to consider in the performance assessment. Dalmasso et al. (2013) additionally evaluates the CPU usage. For the purpose of this thesis, the execution time will be the primary focus in the performance evaluation. This was decided partly because the execution time is in direct relation to the user experience. The amount of time it takes for an application to perform certain tasks may harm the user experience and is thus of high importance when choosing the right development approach.

To measure the execution time of a service or action, it can be convenient to use stopwatches. Stopwatches are usually provided by programming languages and frameworks. The stopwatch can easily be started when the sought execution begins and stopped when it has finished. To avoid including other executions such as boot time, the timer should be placed so that it only encapsulates the code of interest. This way, the execution time of a particular code can be accurately measured (Wadleigh and Crawford, 2000). For the purpose of this thesis, this technique will be used by starting the stopwatch right before the measured algorithm code and stopping it when the code finishes.

3.2.2. Measurement environment

To make a fair comparison all tests on the respective platforms were made using the same testing devices. Even though the environments provide device emulators, the physical devices described in Table 2 were chosen for the measurements. In addition, all devices were running in flight mode with no other applications running during the measurements.

Table 2. Devices used for testing.

<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 6s 16GB</td>
<td>iOS 9.3</td>
<td>Dual-core 1.84 GHz Twister</td>
<td>2 GB</td>
</tr>
<tr>
<td>Android</td>
<td>Sony Xperia M4 Aqua</td>
<td>Android 5.0 (Lollypop)</td>
<td>Quad-core 1.5 GHz Cortex-A53 &amp; Quad-core 1.0 GHz Cortex-A53</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

3.2.3. Bubble sort

To measure and compare the performances a sorting algorithm has been implemented using React Native, Xamarin, and both platforms’ native methods. This has been done through two separate tests comparing execution times, one for Android and one for iOS, both comparing the different frameworks.
There are several different sorting algorithms. For the purpose of this test Bubble sort was chosen as it is well established, simple to implement, and also because it is inefficient as it scales, thus magnifying potential differences. Bubble sort belongs to the $O(n^2)$ sorting algorithms with both average and worst case time complexity of $O(n^2)$ (Chang, 2003). The premise is that this should lead to quick and comprehensible results. The basic outline for the algorithm is the same for whatever language of implementation and in the case of this test each implementation was based on the piece of pseudocode seen in Figure 5, as described by Algolist (n.d.).

Arrays containing the numbers 0-9 of the sizes 0, 100, 1 000, 5 000, 10 000, and 20 000 elements were randomly generated (20 arrays for each size) to test the Bubble sort implementations. Each array was sorted by its respective implementation 10 times, resulting in 1 200 runs per implementation and 3 600 runs per platform. The execution time of each run was stored in milliseconds and then used to calculate the mean value for each array size, implementation, and platform.

3.3. Native user experience

According to Heitkötzter, Hanschke, and Majchrzak (2012) most smartphone users seek native resembling applications. This means that deciding whether to use a cross-platform framework for application development depends on its ability to build applications with a good user experience, and which behave like applications built natively. User experience can be defined as “[...] a momentary, primarily evaluative feeling (good-bad) while interacting with a product or service” (Hassenzahl, 2008 p.2). It is the subject side of the use of a product. To achieve a good user experience, human needs such as stimulation and relatedness must be taken into account. This may include response time when pressing buttons or scrolling in a list.
To evaluate this metric, a simple application has been built natively, using Xamarin, and using React Native for iOS and Android respectively. A number of people have then been asked to test the applications and answer questions regarding how they experienced them.

3.3.1. Application design

The application includes three views. The first view consists of an application menu used for navigating to the other two views. The second view includes six buttons representing different emotions. Pressing a button renders an image in the top of the view corresponding to the button’s emotion. This view’s purpose is to evaluate the feeling of interacting with buttons and rendering different images. The third view is a simple to-do list. The user may add tasks using a text input and a submit button. When submitted, the task is stored in the local storage of the device and subsequently displayed in a list in the same view. The user may scroll through the tasks and delete records by pressing the done button for the specific task. This view’s purpose is to evaluate how the users experience commonly used features such as text inputs, storing and deleting content in a database, and browsing a list of items. The views can be seen in Figure 18 (Android) and Figure 19 (iOS).

3.3.2. User testing with interviews

To evaluate the user experiences of applications developed with the different frameworks user tests were conducted. The test consisted of testing three applications (Native, React Native, and Xamarin) for either Android or iOS. Users were given instructions on what to do, specifically for each view, after which they were told to play around freely. 10 users were selected for each platform, resulting in 10 tests being performed per platform and 20 tests in total.

The selected users were by their own admission used to the platform for which they were selected and the order in which the frameworks were tested were randomized for each user. The users consisted of both males and females, aged 20-28, all studying at Uppsala University.

After the tests the users were interviewed about their experiences. The interviews were semi-structured meaning that the interview questions were defined on the basis of an underlying theme, but without being strictly leading. Instead, the interviews were allowed to diverge with regard to what the respondent believed was relevant to the theme. It was throughout the test unknown to the testers in which framework each application was created, to avoid possible bias (Kvale, 2007).

The following questions were asked after each separate application test:

- Did the application ever not act as you anticipated? If so, in what way?
The purpose of this question was to highlight what the tester found most significant about the experience. With it being a broad, non-specific question the idea was to not influence the user’s answer.

- Did you experience any delay using the application? If so, in which part?

A more specific question in case delay was not mentioned in the answer to the previous question.

When all three applications had been tested by the user the following question was asked:

- Did your experiences of the three applications differ? If so, in what way?

The purpose of this question was to highlight any differences between the applications the user might have experienced that were not covered by the previous questions.

The goal with qualitative research methods such as interviews, is to acquire a deep knowledge of a phenomena through a limited amount of respondents. Therefore the selection of interviewees affects the study substantially. (Holme and Solvang, 1997) With this in mind respondents had to have prior experience of using smartphone applications.

### 3.4. Real-life case

There are certain aspects of React Native not as quantifiable as the ones presented in earlier sections. One aspect would be how well React Native works as part of a smartphone application project. The more qualitative part of the study looks to evaluate this, and was conducted during the development of a mobile application at the Uppsala based company Interactive Solutions, an IT-company with expertise in web services, smartphone applications, and online gaming, to further understand how React Native performs in a real-life case.

The application’s initial concept was for it to be a social platform for people taking part in their local nightlife, with interaction between users being limited to their geographical location. The application’s specification was not written with this thesis in mind. Meaning the development was conducted the same way as in the company’s other projects, to the standards of a real-life case with respect to deadlines and quality. Two back-end developers and a designer employed by Interactive Solutions were also involved in the project. Meaning that this study focused on the front-end development of the application.

The idea was for the application to have a number of features, such as posting updates, viewing updates from other users posted within a certain geographical radius, notifying friends of where one is going out, viewing where friends are going out, viewing other user’s profiles, and messaging other users. This was achieved by the use of geolocation, connection to both local storage and a remote database, and Facebook integration. The application was built for Android and iOS with all its features fully supported for both
platforms, as per request of Interactive Solutions. It is planned to be released on Google Play and the App Store for download.

Heitkötter et al. (2011) propose a framework of metrics to use when evaluating cross-platform approaches. The framework covers both development and infrastructure perspectives, with metrics including the maturity of the development environment, ease of development, and whether the approach supports different platforms equally well. The framework has inspired the metrics used for the purpose of this thesis: documentation, community, maintainability, testing, learnability, and cross-platform potential. These metrics are explained briefly in Table 3. All these metrics will be discussed based on experience from developing the application, in comparison to the authors’ previous experiences of native development.

*Table 3. Metrics used in the evaluation of React Native.*

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Evaluation of Facebook’s documentation of React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community</td>
<td>Evaluation of React Native’s community and definition of its role.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>How easy it is to maintain the application following React Native updates during and after development.</td>
</tr>
<tr>
<td>Testing</td>
<td>Description of React Native’s testing and what it means to development.</td>
</tr>
<tr>
<td>Learnability</td>
<td>Measure of how easy it is for developers to learn and use React Native.</td>
</tr>
<tr>
<td>Cross-platform potential</td>
<td>Measure of React Native’s capacity to develop cross-platform applications.</td>
</tr>
</tbody>
</table>
4. Results and analysis

Here the results are presented in the same order as described in the Method chapter.

4.1. “Hello, World!”

Here the results of implementing “Hello, World!” for each framework are presented. This is done by first describing each framework’s setup for this thesis (for easier verification of results), followed by the actual implementation and a description of the implementation.

4.1.1. Native Android setup

There are several IDE’s for Android development that work on different operating systems. For this thesis the IDE Android Studio 1.5.1 was used on OS X El Capitan 10.11.4. To set up the environment the Android Studio DMG was downloaded from the official website (Android, 2016) and installed. The Android Studio version used was 1.5.1. No previous settings were imported and all options during installation were set to default.

Other than Android Studio the following were also installed to enable development:

- Java SE Development Kit 8u77
- Android SDK 24.4.1

Testing on the Android test device was enabled through Settings > Developer options. Developer options are since the release of Android 4.2 hidden by default and revealed in Settings > About phone by tapping Build number seven times. Testing was done through USB debugging (connecting the test device to the Mac with a USB-cable), enabled in Developer options.
4.1.2. Native Android implementation

Figure 6. Screenshot of Android version of Hello, World! run on the Android test device.

```
1. package com.example.stuart.helloworld;
2. 3. import android.graphics.Point;
4. import android.os.Bundle;
5. import android.support.v7.app.AppCompatActivity;
6. import android.view.Display;
7. import android.view.Gravity;
8. import android.widget.LinearLayout;
9. import android.widget.TextView;
10. public class MainActivity extends AppCompatActivity {
11.     @Override
12.     protected void onCreate(Bundle savedInstanceState) {
13.         super.onCreate(savedInstanceState);
14.         Display display = getWindowManager().getDefaultDisplay();
15.         Point size = new Point();
16.         display.getSize(size);
17.         int width = size.x;
18.         int height = size.y;
19.         LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(width, height);
20.         TextView helloView = new TextView(this);
21.         helloView.setText("Hello, World!");
22.         helloView.setLayoutParams(params);
23.         helloView.setGravity(Gravity.CENTER);
24.         setContentView(helloView);
25.     }
26. }
```

Figure 7. “Hello, World!” code for Native Android.

Figure 6 shows a screenshot of the native Android version of the “Hello, World!” application. Figure 7 shows the implementation, written in Java. When the application is executed it runs the class MainActivity (line 11). Within this class there is a method called onCreate (line 14), which is called as soon as the application launches. The method is divided into three steps. The first step retrieves information about the device’s display
(line 17), stores information about its size in the object size (line 18 and 19), stores the width and height of the screen in separate variables (line 20 and 21), and creates a LinearLayout params based on the width and height (line 22). The second step initializes the TextView helloView (line 24), sets its text to “Hello, World!” (line 25), sets its size according to the earlier created params (line 26), and centers the text (line 27). In the last step the contentView of the application is set to helloView (line 29).

4.1.3. Native iOS setup

To begin developing native iOS applications, the developer must first and foremost have access to a Mac along with an Apple ID. This is needed to install the official iOS IDE Xcode from the application store. The version used was Xcode 7.3. The default installation of Xcode provided all necessary tools for developing iOS applications and the setup of the project was thus very straightforward.

Testing a native iOS application on a device requires a provisioning profile which basically ties the developer’s Apple ID to the specific device. When attempting to run the application for the first time on a device, Xcode detects that a provisioning profile is missing. This occurred as part of this project, but was easily resolved by pressing the Fix Issue button that appeared. This automatically conducted all the actions needed for creating required certificates and registering the device. When this was finished, the application compiled and ran on the testing device.

4.1.4. Native iOS implementation

Figure 8. Screenshot of iOS version of Hello, World! run on emulator through Xcode.
Figure 8 shows the native iOS version of the “Hello, World!” application. Figure 9 shows the implementation, written in Swift. Two constants are initialized, screenSize (line 7) and label (line 9). The constant screenSize is created using UIScreen, which is part of UIKit (line 1) and allows the program to retrieve information about the device’s screen size. The size of label is set to the size of the screen using screenSize (line 9) and its text is set to “Hello, World!” (line 10). The text is displayed by adding label to the main view (line 12). The alignment of the text is set to the center of the screen (line 11) so that “Hello, World!” is displayed as seen in the image and not in the top left corner. All of this is done within the method viewDidLoad(), which by definition is executed once the view has loaded to the device’s screen.

4.1.5. React Native setup

The earlier described setups for native development on Android and iOS are both necessary for cross-platform development. Specifically for Android development with React Native the following packages have to be installed (through the SDK Manager):

- Android SDK Tools 24.3.4
- Android SDK Platform-tools 23.0.1
- Android SDK Build-tools 23.0.1
- SDK Platform
- Intel x86 Atom_64 System Image
- Intel x86 Atom System Image

The React Native version used was 0.24. The setup of React Native requires a few different pieces of software. Firstly Homebrew (Brew, 2016) was installed, through which Watchman 2.9.6 (Facebook, 2016d) and Flow (Flow, 2016) were installed. Then Node.js 4.4.3 (NodeJS, 2016) was installed after which React Native could be installed with the following command in the Terminal:

$ npm install -g react-native-cli
Since React Native doesn’t provide an IDE for development this needs to be installed separately. For the purposes of this thesis *Webstorm 11* and *PHPStorm 10* were used. While these are not free, evaluation versions are available for 30 days respectively.

The React Native project was then initialized with the terminal command:

```
$ react-native init HelloWorld
```

This created all the needed files for a working sample application, including the native project files for both Android and iOS.

Running the React Native application on Android was done using the Mac terminal. The device was connected through USB and its connection to the Mac was verified by the command:

```
$ adb devices
```

If the device was in the output list, it was possible to run the application with the command:

```
$ react-native run-android
```

To test on an iOS device the same procedure as with native iOS, including provisioning, was needed.

If the application running on the device is connected to the Mac’s development server, it is possible to almost instantly refresh the application on the device. This meant that testing newly edited JavaScript code was done in seconds. To utilise this feature when running on the device, additional settings was required. For Android, the following command was needed:

```
$ adb reverse tcp:8081 tcp:8081
```

And for iOS the Mac’s IP address was added to the native project. It was furthermore required that the device and Mac were connected to the same WiFi. In practice, the refresh ability was accessed by shaking the phone which would display a developers menu with the option *Refresh JavaScript*. 
4.1.6. React Native implementation

Figure 10. React Native version of Hello, World! run on iOS emulator through Xcode (left) and Android test device (right).

```
1. 'use strict';
2. import React, {
3.   AppRegistry,
4.   Component,
5.   StyleSheet,
6.   Text,
7.   View,
8.   Dimensions,
9.   } from 'react-native';
10. class HelloWorld extends React.Component {
11.   render() {
12.     return {
13.       <View style={styles.helloView}>
14.         <Text>Hello, World!</Text>
15.       </View>
16.     }
17.   }
18. }
19. }
20. }
21. 
22. const styles = StyleSheet.create({
23.   helloView: {
24.     width: Dimensions.get('window').width,
25.     height: Dimensions.get('window').height,
26.     alignItems: 'center',
27.     justifyContent: 'center',
28.   }
29. });
30. AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

Figure 11. “Hello, World!” code using React Native.
Figure 10 shows the React Native versions of the “Hello, World!” application. With the iOS version to the left and the Android version to the right. Figure 11 shows the implementation, written in JavaScript. A few components are imported from the React Native framework (line 4 and 10) to be able to execute the program. Among these are Component (line 5) in order to build the application’s layout with different components, StyleSheet (line 6) in order to customize the look of each component, Text (line 7) in order to include text, View (line 8) in order to separate the layout into different views, and Dimension (line 9) in order to retrieve the dimensions of the device’s screen.

The class HelloWorld (line 12) is executed when the application runs. Within this class there is the method render(), telling the device what to display on the screen when HelloWorld is executed. In render() there is a Text component with the text “Hello, World!”. This component is wrapped in a View component, giving the text an area to be displayed. This View component’s style is set to helloView, which can be found in the StyleSheet styles (line 22). Here the width and height are set to the dimensions of the screen and the alignment is set to the center of the view both horizontally (alignItems) and vertically (justifyContent).

4.1.7. Xamarin setup

The setup of the Xamarin IDE, Xamarin Studio, was made through Xamarin’s official website (Xamarin, 2016a). The same installer was used for both Android and iOS setup, yet the support for different platforms can be installed separately. For Android, Xamarin requires the latest Android SDK which the installer downloads itself if it is missing. For iOS the developer must prior to installation manually have installed the latest iOS SDK and Xcode. When developing for iOS, Xamarin requires either development on a Mac or on Windows but with access to a Mac. Using the latter option, the Mac works as a host or online service for iOS builds. This was not needed for this thesis since development was done exclusively on Macs.

To build a Xamarin.Forms application in the Xamarin Studio, it is simply selected when initializing a project. This generates a single view application made in C#. This application needed very slight changes to achieve the “Hello World” view seen in Figure 12. The Xamarin.Forms version used was 2.1.0.6529 with Xamarin Studio 6.0.

Xamarin has its own Android emulator that may be used for testing while it for iOS uses the standard Xcode emulator. When testing a Xamarin.Forms application on an Android device, the developer must follow the exact same steps as when testing a native application. The same goes for testing on an iOS device, including provisioning. Additionally when testing on an iOS device, the developer must create a dummy Xcode project that shares the id of the Xamarin project. When building the application for the different devices, the code is compiled to native code. This compilation naturally takes time and is required to test even the slightest code change.
4.1.8. Xamarin.Forms implementation

Figure 12. Xamarin version of Hello, World! run on iOS emulator through Xcode (left) and Android test device (right).

```
public class application: Application
{
    public application ()
    {
        MainPage = new ContentPage {
            Content = new Label {
                Text = "Hello, World!",
                VerticalOptions = LayoutOptions.CenterAndExpand,
                HorizontalOptions = LayoutOptions.CenterAndExpand,
            }
        };
    }
}
```

Figure 13. “Hello, World!” code using Xamarin.Forms with C#

Figure 12 shows the Xamarin versions of the “Hello, World!” application. With the iOS version to the left and the Android version to the right. Figure 13 shows the implementation, written in C#. After importing relevant packages the main class of the application is defined (line 6). The root page of the application is defined using the variable MainPage from where the content of the page is built. In an application with a
more complex user interface, the Content variable would be assigned a layout class. Depending on the structure of the user interface different layouts may be used, including StackLayout where all content is placed in a line horizontally or vertically. In this very simple application however, the only content is a Label (line 11 and 16). The label is assigned three attributes; the text of the label (line 13) as well as its vertical (line 14) and horizontal (line 15) position.

4.1.9. Setup and development comparison

As far as writing the code was concerned, no great difference was experienced between the different frameworks. The number of lines differ, with Native Android at 25, Native iOS at 10, React Native at 27, and Xamarin at 19 (empty lines not included). However this could be due to the authors’ varying experience with the different languages. And even when not taking that into account the number of lines is not necessarily a good indicator for difference in size when it comes to larger projects. It can therefore be concluded that for development of simple applications the choice of framework should be based on language preference and desired platforms. Someone with experience within for example web-development and JavaScript wanting to develop an application for both Android and iOS would by this logic be advised to use React Native.

It is however important to point out that these results do not regard development of applications much more complex than the different versions of “Hello, World!” presented in this chapter. Furthermore it is also possible that certain types of more complex applications simply cannot be built with some frameworks.

4.1.10. Application size comparison

After installation of the “Hello, World!” applications on its respective platform and test device, the application size of each application was documented. On iOS this was simply retrieved from the device’s settings page and on Android from the list of installed applications. All applications were deployed as release versions, which means that debugging functionalities are excluded from the build and that the size therefore is more reliable. The different sizes can be viewed in Table 4.

<table>
<thead>
<tr>
<th>Platform / Framework</th>
<th>Native</th>
<th>Xamarin</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>5.8 MB</td>
<td>16.1 MB</td>
<td>20.2 MB</td>
</tr>
<tr>
<td>iOS (Swift)</td>
<td>16.2 MB</td>
<td>32.9 MB</td>
<td>7.6 MB</td>
</tr>
</tbody>
</table>

Table 4. “Hello, World!” application size for the respective platform and framework.
For Android, the native application used 5.8 MB of storage, while the Xamarin and React Native used 16.1 MB and 20.2 MB respectively. Using Xamarin resulted in a 178% size overhead over native and using React Native the overhead was 248%. Note that the React Native application was the largest in comparison to its iOS counterpart which was the smallest. The result implies that for Android, a Native application will outperform both cross-platform approaches for simple applications with regard to storage consumption.

For iOS, the native application consumed 16.2 MB of storage. React Native used the least with 7.6 MB and Xamarin the most with 32.9 MB, a negative size overhead of -114% over the native application for the former and a positive overhead of 103% for the latter. Since the size overhead of using React Native over native is negative and the overhead for Xamarin is large, the application size should be considered when selecting the development approach. The result above implies that React Native for iOS will outperform the contenders with respect to application size, at least when developing simple applications.

However, it should be noted the application sizes of the “Hello, World!” applications does not reveal the size overhead when developing complex applications. To conclude the top performer in that case, additional tests should be conducted by developing applications with different complexity and functionality.

4.2. Performance evaluation

The results of the Bubble sort implementations introduced in section 3.2 are presented below in graphs and a table for each platform. For each platform and framework the average execution time in milliseconds is given for the different array sizes. Since the tests for Android and iOS were carried out on different devices no comparison has been made between platforms, but rather between the frameworks.
4.2.1. Android

Figure 14. Mean values of execution times in linear scale for Bubble sort on Android test device.

Figure 15. Mean values of execution times in logarithmic scale for Bubble sort on Android test device.
Table 5. Confidence intervals of execution times for Bubble sort on Android test device.

<table>
<thead>
<tr>
<th>Array size / Framework</th>
<th>Native Android</th>
<th>Xamarin Android</th>
<th>React Native Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 elements</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0.0426572 ms</td>
</tr>
<tr>
<td>100 elements</td>
<td>0 ms</td>
<td>0.233329 ms</td>
<td>0.170925 ms</td>
</tr>
<tr>
<td>1 000 elements</td>
<td>0.424196 ms</td>
<td>1.45635 ms</td>
<td>1.37876 ms</td>
</tr>
<tr>
<td>5 000 elements</td>
<td>1.76197 ms</td>
<td>1.83030 ms</td>
<td>11.5912 ms</td>
</tr>
<tr>
<td>10 000 elements</td>
<td>1.60565 ms</td>
<td>2.52776 ms</td>
<td>42.9763 ms</td>
</tr>
<tr>
<td>20 000 elements</td>
<td>18.5788 ms</td>
<td>6.65002 ms</td>
<td>47.4184 ms</td>
</tr>
</tbody>
</table>

From Figure 15 it can be seen that already at arrays with 100 elements the difference between React Native and the other frameworks is noticeable. And looking at Figure 14 it is clearly evident that React Native scales a lot worse. Xamarin performs better than React Native but is still also worse than the native framework. The results are statistically verified in Table 5, which shows that there is no overlap between the different frameworks confidence intervals.

4.2.2. iOS

Figure 16. Mean values of execution times in linear scale for Bubble sort on iOS test device.
Figure 17. Mean values of execution times in logarithmic scale for Bubble sort on iOS test device.

Table 6. Confidence intervals of execution times for Bubble sort on iOS test device.

<table>
<thead>
<tr>
<th>Array size / Framework</th>
<th>Native iOS</th>
<th>Xamarin iOS</th>
<th>React Native iOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 elements</td>
<td>0 ms</td>
<td>0 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>100 elements</td>
<td>0.00587143 ms</td>
<td>0 ms</td>
<td>0.252554 ms</td>
</tr>
<tr>
<td>1 000 elements</td>
<td>0.0219667 ms</td>
<td>0.153326 ms</td>
<td>0.376060 ms</td>
</tr>
<tr>
<td>5 000 elements</td>
<td>0.225642 ms</td>
<td>0.181593 ms</td>
<td>2.71765 ms</td>
</tr>
<tr>
<td>10 000 elements</td>
<td>1.02099 ms</td>
<td>0.487073 ms</td>
<td>14.2675 ms</td>
</tr>
<tr>
<td>20 000 elements</td>
<td>2.47675 ms</td>
<td>1.57610 ms</td>
<td>34.4227 ms</td>
</tr>
</tbody>
</table>

At arrays with 100 elements React Native starts performing worse than both native and Xamarin, as seen in Figure 17. In Figure 16 it is also apparent that React Native scales a lot worse than the other frameworks. Up until 100 elements Xamarin performs slightly better than native. However at 1 000 elements and beyond native performs better than Xamarin. The results are statistically verified in Table 6, which shows that there is no overlap between the different frameworks confidence intervals.
4.3. Native user experience

In this section the implemented application used for testing is described. Followed by the results of the user tests which are analyzed to get an understanding of the overall user experience.

4.3.1. The application

The specification of the application used in the test required it to contain three views with different functionality. The main view from which the user navigates, an emotions view in which different images are rendered based on button presses, and a simple to-do list where tasks are added and deleted. The resulting application for Android can be seen in Figure 18 and for iOS in Figure 19.

Figure 18. Screenshots from the native version of the Android user test application.
4.3.2. Test results

For iOS none of the users experienced any differences between the different versions of the application, stating that all views performed well in all three versions. For Android however 5 out of 10 users experienced somewhat of a delay when selecting the different emotions in the React Native application. 3 users also noted a slight delay in the images loading on Xamarin. Apart from that no differences were experienced between the different Android versions and the users all said that the to-do list ran smoothly for all three versions.

The result for iOS shows that both the React Native and Xamarin applications provide a user experience similar to the native one. However, the results for Android indicate that neither framework can achieve a native user experience, but with Xamarin being closer.

4.4. Real-life case

Development of the application at Interactive Solutions was successful in that the requirements of the requirements specification were all met. In this section, experiences from the development process are presented and analysed based on the metrics defined in Section 3.4. No strict comparisons are made with other frameworks, as in previous sections. Instead the experiences of working with React Native are compared to previous experiences of native development.
4.4.1. Documentation

The documentation for React Native includes how to get started, a few guides, and covers the available components and API’s to varying degrees. Compared to the documentation for Android Studio and Xcode, React Native’s documentation places much more emphasis on showing through example, which in many cases was helpful in understanding React Native. However it is not nearly as thorough as the native frameworks’ documentations and did on a number of occasions during development prove insufficient. In some of these cases other developers had to be relied upon if help was needed for understanding. In the case of creating animated transitions between the application’s different views however, no one seemed to have implemented it the way in which the specification for the application required. To understand the component and implement the animations, React Native’s source code had to be studied. Some of the provided animations, not mentioned in the documentation, turned out to be incomplete, in which cases custom animations had to be built. Overall the documentation is not as complete as its native counterparts.

4.4.2. Community

Being an open-source project, the community is vital to React Native. In this context the community includes the founders of React Native and all its users. As founders Facebook have done and continue to do a lot of work. It is also their decision what is included in every update. But a lot of work is also done on React Native by its users, in finding bugs and improving and developing new components.

It has been concluded that the community also serves an educational purpose. As a result of the lacking documentation and React Native still being in its early days, members coming together to help each other has proved important both for the communities shared knowledge and the knowledge of individual developers. This became a major part in the development of the application, both using and contributing to new components as well as exchanges with Facebook regarding issues connected to React Native. This is a significant difference from developing natively and should be taken into account by developers considering using React Native.

4.4.3. Maintainability

It became clear that some of the basic components of React Native were somewhat unstable, in that they would display unforeseen behaviour. For example the application’s navigation bar would inherit unwanted functionality from previous views when navigating between views (as demonstrated in Figure 20). When this was the case as in the rightmost image of the figure, it was possible to navigate to the same page repeatedly causing both a strange user experience and memory issues. The issue was found to be caused by React Native’s source code, and not the implementation by the authors of this report. It was furthermore reported as an issue to the official developers of React Native and consequently recognized as a problem in their code.
Problems like this would generally require unofficial solutions, either provided by a third party or developed from scratch. In some cases this proved to be a time consuming process of searching for, implementing, and testing new code. And if an implemented solution is not accepted and integrated into future versions of React Native, it is not guaranteed to work following new updates, which for the time being are provided every second week. This is of course true when contributing to any open-source project. However in working with React Native during the project it was a problem encountered repeatedly, making it noteworthy.

In each new version there were bug fixes, performance enhancements, and often completely new features. In the beginning of the application development, some basic features used in native application development were not yet officially implemented for React Native. One example was the pull to refresh functionality for lists. This functionality allows the user to update lists of data, such as news feeds, by pulling it down. At first this was solved by using a third-party module in which a self-developed solution was implemented. This solution broke at times when updating to new React Native versions, caused by its dependence on deprecated code. Maintaining the application was consequently difficult and sometimes downgrading to old versions was required for the
application to run as intended. However, after a time an official pull to refresh solution was released. And by implementing the official feature, the maintenance became easier.

The frequent updates and implementation of new features proved that React Native is still young and continuously evolving. The authors’ experience implies that in time, React Native will provide all the needed components used in native application development. When dependence on third party modules diminishes, maintainability will increase.

4.4.4. Testing

With the refresh-ability, executing new code without having to recompile the code or reinstall the application, the number of tests that would otherwise have been done during development has increased. This in turn had an effect on development, especially implementing design and searching for bugs, allowing for a more “trial and error” based approach when needed. One example is the application’s chat which had to be redesigned from scratch during development, as the proposed design did not satisfy Interactive Solutions when implemented. The process of fine tuning the results was much smoother because of the refresh ability as different dimensions of the components could be compared efficiently. The result can be seen in Figure 21.

![Figure 21. The application’s chat view, displaying a conversation between two nightclub enthusiasts.](image-url)
Whether or not this ended up having any effect on the result cannot be concluded. It could be the case that it had a negative effect in that the developers’ overall understanding of the project ended up being smaller than it otherwise would have been. As a trial and error approach could lead to developers simply testing different solutions until something works. The opposite could however have been true as the refresh ability helped in learning how React Native works (as discussed in the section below). Either way it did in many cases speed up the development and the possible negative side effects can easily be prevented by keeping them in mind.

4.4.5. Learnability

React Native allows for development of mobile applications with JavaScript. This is a language traditionally used for web development and will be unfamiliar to most in the context of mobile application development. For organisations wanting to use React Native it is therefore important to know how learnable it is, either for web developers with JavaScript experience wishing to develop mobile applications or mobile application developers looking to develop cross-platform.

Prior to the project the authors had little experience of web development and some experience of mobile application development (Android and iOS). So during the real-life case, early development was mainly about understanding JavaScript in a mobile application setting. This proved to be relatively straightforward and no problems were encountered suggesting that neither web developers nor application developers would have any issues adjusting to React Native.

One of the reasons for the ease of understanding React Native was its refresh ability, described in the previous section. Being able to play around with components and quickly implement small changes was very helpful in understanding how they were built. Something that wouldn’t have been as efficient if learning how to develop natively for example. The refresh ability also made debugging faster and easier. Adding logs or alerts for detecting errors could be added and run without recompiling as the problems appeared.

4.4.6. Cross-platform potential

Using React Native as a means for cross-platform development it became apparent that it is somewhat restricted. During development one of the main problems was the lack of some components for Android (most likely due to Android support being added after iOS support). Occasionally, features were according to documentation cross-platform while in reality required different implementations for different platforms. In addition, at times the alleged cross-platform functionality was undermined by the lack of support for certain methods. One case where the support differed appeared when building views that utilized the device keyboard. When the keyboard appears, it may be needed to adapt the user interface to prevent vital parts to become hidden behind the keyboard. This was required for a number of views in the application, for example when writing a new comment (see
When the user press the text input the device keyboard appears from the bottom. The send button must then be moved from the bottom of the screen to right above the keyboard. This was solved by using a listener that detects if the keyboard will show or hide and adapting the position of the button accordingly. At first, this worked perfectly for iOS but not for Android. After troubleshooting it was found that the support for listeners was different between the two platforms and that it was required to use different platform-specific solutions.

While this particular case in the end did not affect the resulting functionality and user experience, it is noteworthy that the support differs between the two platforms. In other instances it was sometimes worked around by simply adapting the design specification and in other instances by developing natively. Over the course of the project support has however improved and will probably continue to do so. And in general during the development of the application components have been built without cross-platform related issues, leading to time being saved having not been required to build and test these components twice.
5. Discussion and conclusions

During the last ten years the market for smartphones has grown drastically. Because of the current state of the market, with different operating systems, many smartphone applications need to be developed for several platforms. In this thesis the framework React Native has been evaluated and compared to native development and Xamarin, as a means to develop cross-platform applications. This has been done with quantifiable as well as qualitative methods, including a performance test, a user test, and a real-life case. The case was issued by the Uppsala based company Interactive Solutions, and entailed developing the front-end part of an Android and iOS application, ready to be released after the project. In this section the results are discussed and an attempt is made to answer the questions presented in Section 1.

Firstly, results of the real-life case show that React Native can potentially be used successfully for the development of cross-platform applications. The real-life case’s requirements specification was met without any real compromises having to be made due to any cross-platform restrictions of React Native. Nothing can be said about the public reception of the application, as Interactive Solutions as of now have not deployed the application. But they are themselves pleased with it and in the end time was saved in not developing the application twice natively.

The results of the Bubble sort tests however indicate that React Native does not scale as well as neither the native framework nor Xamarin, performance-wise. And for Android the results of the user tests indicate that simple applications also seem to perform slightly better when built natively rather than with the cross-platform frameworks.

This would in summary mean that there are several factors to be considered in choice of framework for development of applications for Android and iOS. One being the complexity of the application to be developed as well as the priority of performance and time plan. A complex application might be noticeably better optimized when developed natively. But it can on the other hand take longer to develop. If size of the released application is a priority complexity of the application also needs to be taken into account, as simple applications seem to require less space when built with React Native. Another factor to consider is programming language preferences, as the different approaches require knowledge in different languages. It is also important to be aware of the fact that React Native is open-source. Meaning that development with React Native can mean having to rely upon unofficial support from the React Native community.

Future research could look to conduct similar performance tests as the one presented in this thesis, but with different algorithms or different test devices. Perhaps an algorithm like Quicksort could be tested to see how big the differences are when using a better optimized algorithm. Another approach for future research would be to implement native code for the algorithm in a React Native application and compare it to a native application, as this is a possibility in React Native not covered by this thesis. It could also be interesting to conduct similar user tests but with larger sample sizes to produce more
reliable results. Furthermore future research could look more deeply into React Native’s feature to refresh application during testing and what it means to development. This is something the native framework and Xamarin do not provide and it could possibly have unknown effects on development results.
6. References


