Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science
Master's Programme in Software Engineering and Digital Transformation

Master’s Thesis

Tommi Nivanaho

DEVELOPING A CROSS-PLATFORM MOBILE APPLICATION WITH REACT NATIVE

Examiners: Prof. Ajantha Dahanayake
M.Sc. (Tech.) Ilkka Toivanen

Supervisors: Prof. Ajantha Dahanayake
M.Sc. (Tech.) Ilkka Toivanen

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science
Master's Programme in Software Engineering and Digital Transformation

Tommi Nivanaho

Developing a cross-platform mobile application with React Native

Master’s Thesis

2019

71 pages, 20 figures, 2 tables

Examiners: Prof. Ajantha Dahanayake
M.Sc. (Tech.) Ilkka Toivanen

Keywords: mobile applications, mobile development, react native, cross-platform

With the splintering of the mobile application market, a new demand has risen for technologies that allow mobile application development for multiple platforms simultaneously. The current market leading solution React Native framework was selected for further study. In the thesis mobile application development and React Native were studied by conducting a literary review into the subjects. A full featured React Native application was also produced with the aim of gaining insight into using the framework in a mobile application development. The literary review and implementation project proved that while React Native still has some lingering problems due to it still being an unfinished product, it is certainly a viable and usable solution for developing mobile applications.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma
Master's Programme in Software Engineering and Digital Transformation

Tommi Nivanaho

Järjestelmäriippumattoman mobiilisovelluksen kehittäminen React Nativella

Diplomityö

2019

71 sivua, 20 kuvaa, 2 taulukkoa

Työn tarkastajat: Prof. Ajantha Dahanayake
DI Ilkka Toivanen

Hakusanat: mobiisovellukset, mobiilikehitys, react native, järjestelmäriippumaton

Mobiilisovellusmarkkinoiden jakautuminen on nostanut pinnalle uuden kysynnän teknologioille, jotka mahdollistavat mobiilisovelluskehityksen useammille alustoille yhtäaikaisesti. Tässä diplomityössä kyseisten teknologioiden tämänhetkinen markkinajohtaja React Native valittiin jatkotutkimusta varten. Ensimmäiseksi mobiilisovelluskehitystä ja React Nativea tutkittiin suorittamalla kirjallisuuskatsaus. Tämän lisäksi diplomityötä varten kehitettiin täysimittainen React Native -mobiilisovellus. Kehitysprojektin tarkoituksena oli hankkia lisätietoa ja kokemuksia kyseisen teknologian käytöstä mobiilikehityksessä. Työn aikana selvisi, että vaikka React Nativella on vielä omat ongelmansa, se on silti toimiva ja käytettävä ratkaisu mobiilisovelluskehitykseen.

ACKNOWLEDGEMENTS

I want to thank my family, friends, colleagues and supervisors from both Lappeenranta University of Technology and Visma Consulting. I have gained lots of motivation from all of you to finish this thesis and my studies.

While my studies have not always been easy and certainly not swift, I have enjoyed my years here in Lappeenranta and have learned so much during my time here. I'd even say that some of the best experiences of my life have happened while studying here.

With that in mind, it is almost with a wistful feeling that I finally leave this chapter of my life behind, but one must always believe that even greater things are still to come.

-*Tommi*

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
ANR	Application Not Responding
AOT	Ahead-Of-Time
API	Application Programming Interface
ART	Android Runtime
CLI	Command Line Interface
CSS 3	Cascading Style Sheets 3
DOM	Document Object Model
ETRS89	European Terrestrial Reference System 1989
GUI	Graphical User Interface
HAL	Hardware Acceleration Layer
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JDK	Java Development Kit
JIT	Just-In-Time
JVM	JavaScript Virtual Machine
NPM	Node Package Manager
OS	Operating System
POJO	Plain Old Java Object
RGB	Red Green Blue
RGBA	Red Green Blue Alpha
RQ	Research Question
SDK	Software Development Kit
SWOT	Strengths, Weaknesses, Opportunities, Threats
UI	User Interface
URL	Uniform Resource Locator
UWP	Universal Windows Platform
VM	Virtual Machine
WAP	Wireless Application Protocol

WGS84	World Geodetic System 1984
XML	Extensible Markup Language

1 INTRODUCTION

1.1 Background

In today's mobile phone market there are two clear leaders, Google's Android and Apple's iOS platforms. These two mobile operating systems (OS) have gained a nearly complete market domination, where Android currently holds 74,4 percent market share globally and iOS 19,6 percent. Only 6 percent market share is left for smaller competitors. This trend of duopoly has been growing over the recent years. [1]

Table 1. Mobile OS worldwide market share.

Year	2015	2016	2017	2018
Android	59,8%	66,3%	71,6%	74.4%
iOS	22,8%	19,6%	19,7%	19,6%
Series 40	5,7%	2,6%	0.9%	0,4%
Windows	2,3%	2,2%	1,1%	0,6%
Other	9,4%	9,3%	6,7%	5,0%

Both the Google's and Apple's OSs have their own mobile application ecosystems and channels for selling and distributing these applications, Google Play Store and Apple App Store. In the year 2017 over 28 billion applications were downloaded from Apple App Store, while Google Play Store's downloads exceeded over 64 billion. It is interesting to note that despite the download numbers, the global application revenue in 2017 was 38,5 billion dollars on Apple App Store, far exceeding the 20,1 billion dollars of revenue generated on the Google Play Store. [2]

This splitting of the mobile application market greatly incentivizes mobile developers to develop and release their applications for both of the market leading OSs. While the developers may reach almost three quarters of the users by releasing an application only on the Android platform [3], they could potentially be missing out on the generally larger revenue generated by the smaller iOS market. Or if the developers choose to only release

their applications on the App Store in the hopes of greater revenue [4], they will still only reach about a quarter of their target audience.

Developing an application for Android and iOS platforms individually could potentially almost double the resources needed to complete the project. Android applications are usually coded using either Java or Kotlin, while iOS applications are developed using Objective-C or Swift. This means that directly sharing code between platforms is out of the question. In addition to the programming language difference, both OSs have many specific quirks related to how they operate, that must be considered when developing software for them.

There are a few different technologies developed over the years to reduce the costs related to multi-platform application development. One of the earliest such technologies is the Apache Cordova framework, an open source continuation of its predecessor, the PhoneGap framework. Cordova and other cross-platform frameworks based on its design allow developers to use Cascading Style Sheets 3 (CSS 3), HTML 5 and JavaScript [5] to build platform independent applications that are then wrapped inside a platform native component that renders them. The applications built this way are called hybrid applications because they are neither true native applications nor web applications [6]. The architectural solution of Cordova is displayed in Figure 1.

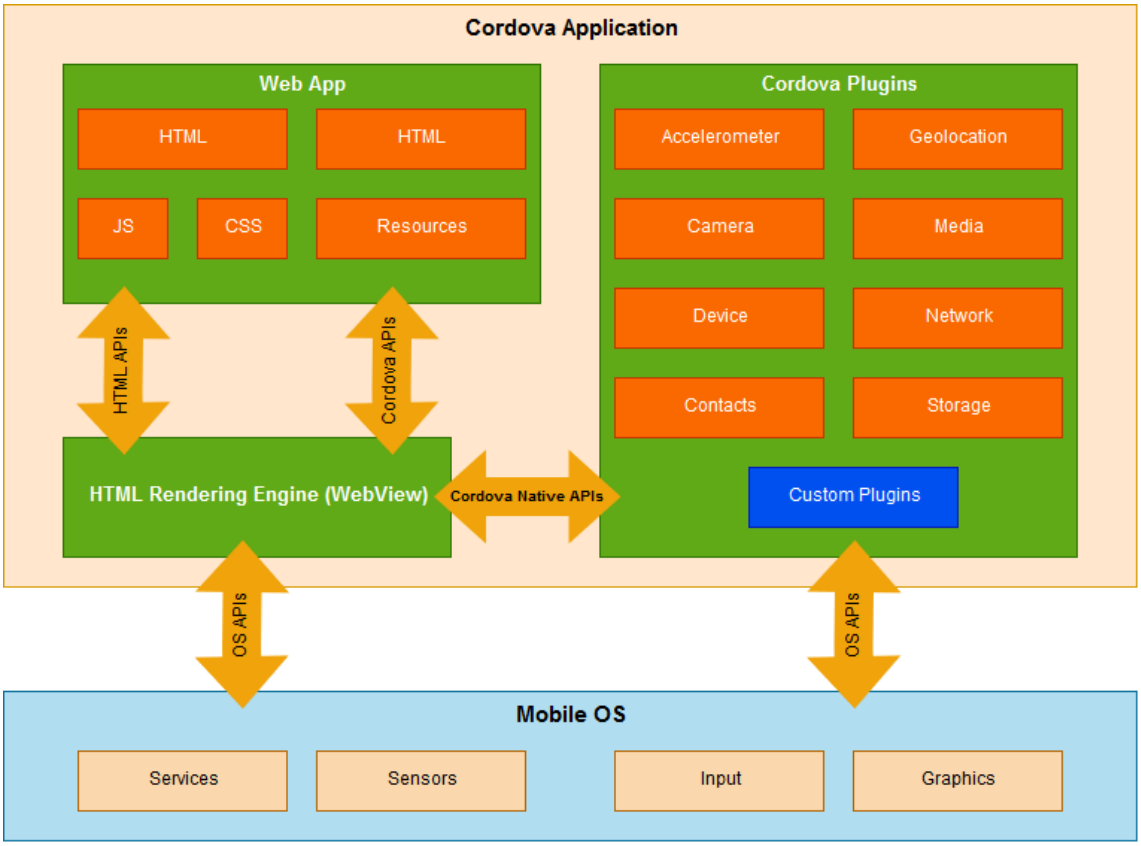


Figure 1. Cordova architecture. [7]

The hybrid application approach is usually very tempting to developers in the business sense, costs and time required to complete an application development project are reduced greatly with the effective use of a single codebase for all platforms [8]. There are some trade-offs however that come with the hybrid applications. It can be difficult to achieve the look and feel of a platform native application with hybrid applications, since they are by their very nature web applications running in a native wrapper. Hybrid applications can also have performance issues since they use the platform native WebView component to render and are bound to its limitations [6]. Performance issues can further be exacerbated by the single threaded nature of JavaScript.

In 2015 Facebook released their own framework called React Native, aimed to tackle the problem of cross-platform mobile development. React Native doesn't use HTML 5 or CSS like hybrid applications, although it also uses JavaScript as the programming language of choice. React Native has since become one of the most popular cross-platform development frameworks [9].

React Native's architecture features a bridge between the applications logic written JavaScript and the platform native layer. This means that the React Native user interface (UI) components are linked to the Native views, improving the application performance. React Native also has multi-threaded design that allows the JavaScript to run in a separate thread from the UI and the Native modules [10] [11, pp. 26-27]. With these design choices and others, the aim of React Native framework is to solve most of the problems present in the previous cross-platform mobile development solutions. React Native framework's architectural design is presented in Figure 2.

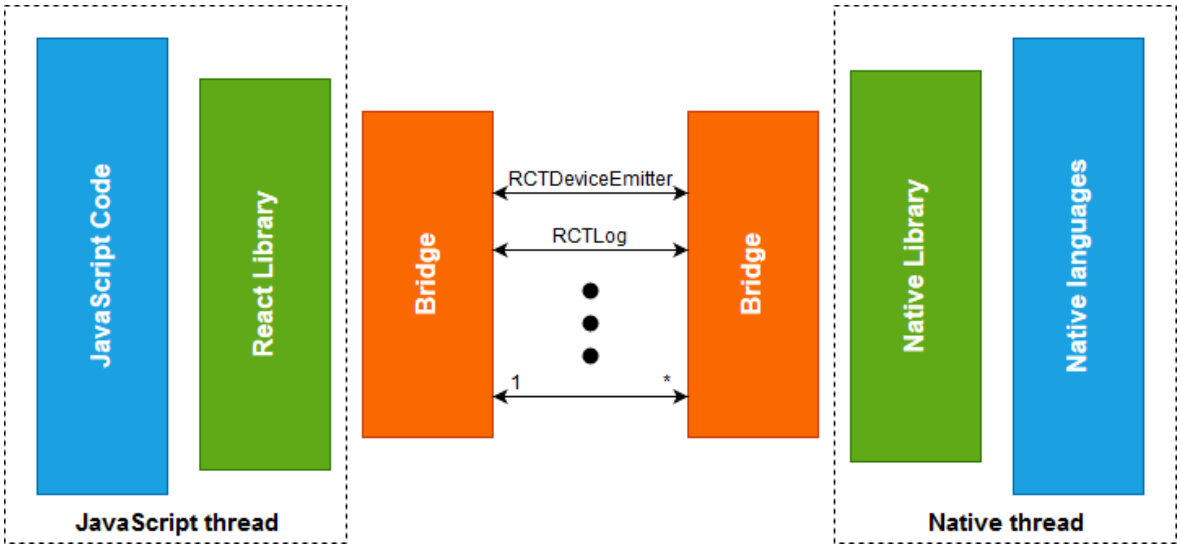


Figure 2. React Native's architectural design. [12]

1.2 Motivation and goals

The motivation for the thesis is to research the characteristics, features and challenges of using the React Native framework in mobile application development. The information gained from the thesis will be useful in the future when determining if a cross-platform development project should be completed using the React Native framework or are there any clear problems that will deter future use.

The goal of the thesis is to develop a full-fledged React Native application for Traffic Management Finland Group and use that project as subject of study. This application is developed by the author in his role as a software engineer in Visma Consulting, along with

the other members of the development team. The mobile application will provide a new way for end users to access publicly available information about the status of Finland's transportation network, provided and collected by Traffic Management Finland Group.

The application will also provide users with an option to leave and view geographically targeted questions or feedback directed to Traffic Management Finland Group. Traffic Management Finland Group provides similar, but separate web services and the aim of the mobile application is to be an easily available way to access these services on mobile devices, while leaving a possibility open to expand the features that are provided by the application.

The application's source code will be openly available to the public by the end of development. The completed application will be available on Android and iOS platforms and will be released on the Google Play Store and the Apple App Store.

The research questions for this master's thesis are:

- What are the characteristics and features of React Native?
- How to implement React Native in mobile application development?
- What challenges can arise when React Native is used to develop cross-platform mobile applications?

The research method utilized in this thesis to answer the aforementioned research questions is comprised of a literature review into the subject of cross-platform frameworks and mobile development, an implementation of cross-platform mobile development project and an evaluation of the information gained.

1.3 Structure of the thesis

The first chapter contains the introduction to the topic that is covered in the thesis. The second chapter is a literature review on the topic of mobile application development. The purpose of this chapter is to analyze the history and the current state of mobile application development and platforms. The second chapter also includes a brief overview of cross-platform frameworks.

The third chapter continues the literature review and delves deeper into the characteristics and features of the React Native framework. The next chapter explains the implementation of React Native into the mobile application development. This fourth chapter contains information about the tools, techniques and methods used in the React Native development project.

The fifth chapter is reserved for an evaluation the information gained during the literary review and implementation phase of the thesis. The conclusion for the thesis is found on the sixth and final chapter.

2 MOBILE APPLICATION DEVELOPMENT

2.1 A brief history of mobile applications

The history of mobile phones can be traced back to 1973 when the first public mobile phone call was made in New York by Martin Cooper of Motorola [13]. It would however take another ten years before personal handheld mobile phones became publicly available in the form of Motorola DynaTAC 8000X [14, p. 25]. The device is displayed in Figure 3.

Figure 3. Motorola DynaTAC 8000X from 1983. [14]

Those first mobile phones didn’t run any mobile applications, however, and the first recognizable mobile applications were introduced to the world on a device called the Psion Organiser, released in the 1984. The Psion Organiser was advertised as the "world's first practical pocket computer" [15]. It allowed the end-users to write and run their own programs in a primitive programming language called POPL [16].

The first device that could combine the features of mobile phones and pocket computers was the IBM Simon Personal communicator in 1994 [17]. Simon’s operating system was the

Datalight ROM-DOS that was engineered for embedded systems. The device had multiple pre-installed applications such as a calendar, calculator and electronic notepad [17]. Third party applications could also be installed to the device. It also featured a touch screen. Due to these features IBM Simon can be called as the first smartphone [18, p. 172]. IBM Simon is seen in Figure 4.

Figure 4. The first smartphone, IBM Simon Personal Communicator with a charger. [19]

The next big development in world of mobile applications was the introduction of the Wireless Application Protocol (WAP). Introduced first in 1999, WAP is a technical standard that enabled a new way to access information over a mobile wireless network [20]. WAP allowed end users to download applications among other content provided by the mobile operators. The WAP standard was developed and maintained by an industry consortium called the WAP Forum founded by Nokia, Ericsson, Motorola and Unwired Planet [21].

The modern era of mobile applications can be said to have really began when Apple Inc. released the first iPhone in 2007 [22, p. 3]. The iPhone had many of the features that are considered standard in today's smartphone, such as multi-touch screen, multitasking and rotating screen that matched the device rotation. These features weren't necessarily new, but Apple had managed to gather them to an attractive package that enticed consumers, selling

6,1 million iPhones during five financial quarters [23]. The first-generation iPhone is displayed in the Figure 5.

Figure 5. The 2007 first-generation iPhone. [24]

Even more significant event to the history of mobile applications, or “apps” as they are sometimes called, was the release of the Apple’s digital distribution platform App Store in 2008 [25]. The App Store and the prior release of the iOS Software Development Kit (SDK) enabled third party developers to build, distribute and sell their applications to all iOS users. This was the beginning of the iOS application ecosystem that is still in use today.

Not too much after the release of the iPhone, the mobile phone industry experienced another high impact event with the unveiling of Android mobile operating system in 2007 and the release of the first Android device in 2008 [26]. The operating system was originally developed by Android Inc. but was later purchased by Google in 2005. An example of HTC Dream, the first commercially available Android device is shown in the Figure 6.

Figure 6. HTC Dream, the first commercially available Android device. [27]

The Android operating system largely matched the capabilities of Apple's iOS, and as mentioned earlier, the two would later become the biggest contenders for the mobile phone market. And as Apple had done a few months earlier in the same year, Google opened their own Android Market digital distribution service for Android applications in October 2008 [28]. This service would later be rebranded as Google Play and is the base of the Android application ecosystem.

2.2 iOS application development and architecture

As mentioned above, iOS is the operating system that is used in mobile devices developed by Apple, such as iPhone smart phones, iPad tablet computers and iPod touch mobile media devices. The development environment required to develop native applications for the iOS consists of an Intel-based Macintosh computer running an OS X operating system [29]. These environment requirements keep iOS developers wholly within the Apple ecosystem.

Apple provides a software suite called Xcode for iOS and OS X software development. The main feature of this software suite is an Integrated Development Environment (IDE) bearing the same name. Other features provided in the suite are Apple's developer documentation, a

built-in Interface Builder and depending on the Xcode version it either has GNU Compiler Collection, LLVM-GCC compiler or Apple LLVM Compiler [30]. Xcode also includes the iOS software development kit (SDK) that contains the tools and resources needed for application development.

iOS SDK officially supports two programming languages, Objective-C and Swift. Objective-C was the original language used in the iOS application development prior to the introduction of Swift in 2014. Unlike Objective-C, Swift is specifically developed by Apple to be used in its operating systems, including iOS [31]. In 2018, Swift surpassed Objective-C in popularity among developers [9].

As the iOS acts as intermediary between the mobile device hardware and a high-level software application, it provides a set of system interfaces and frameworks that can be used by the developers so the applications can communicate with the hardware. These interfaces and frameworks can be divided into four separate layers as shown in the Figure 7.

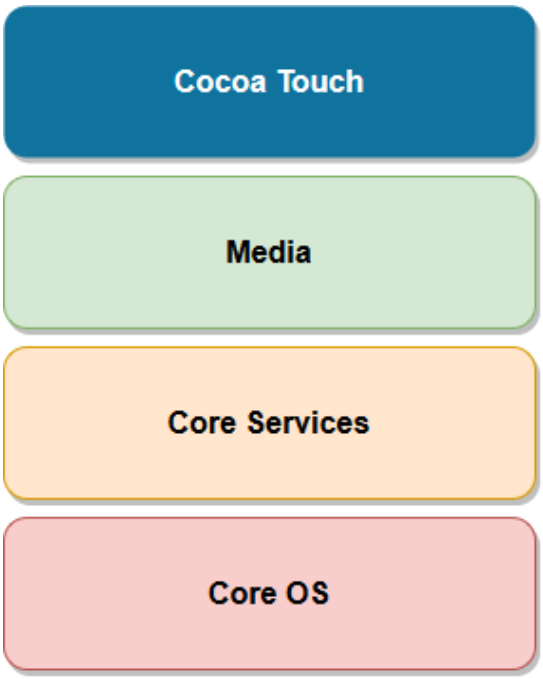


Figure 7. Layers of iOS. [32, p. 8]

The topmost layer of the iOS architecture is the Cocoa Touch layer. This layer hosts several key frameworks for building iOS applications, such as the UIKit framework that provides

essential infrastructure needed for graphical and event-driven iOS applications. Therefore, native iOS applications are incapable of functioning without UIKit linkage. All in all, UIKit and other frameworks of the Cocoa Touch layer combine to offer vital and iconic features such multitasking, touch input, notifications, accessibility functions and others [32, pp. 11-21].

The Media layer provides a large set of tools to implement multimedia experiences such as graphics, audio and video to iOS applications. Some technologies that are included in this layer are: Core Graphics framework that is the native drawing engine of iOS, OpenGL ES used for handling the hardware accelerated 2D and 3D rendering, Core Audio framework collection providing audio playback and recording interfaces and AV Foundation that provides video playback and recording capabilities [32, pp. 22-34].

The second deepest layer is the Core Services layer in charge of managing the fundamental system services used by iOS applications. Other layers such as the Cocoa Touch is dependent of the Core Services for some of its functions. Some of the features and frameworks provided by this layer are: iCould Storage allowing reading and writing of data from/to cloud storage, Data Protection that enables built-in encryption support for sensitive data, Address Book framework that provides programmatic access to user account's contact database and Core Location framework delivering location and heading information to applications [32, pp. 35-47].

The Core OS layer provides the low-level features that are the basis for most the of the higher-level layers' functionality. This layer provides only a few frameworks that applications can directly use, such as the Accelerate Framework containing interfaces for performing digital signal processing, linear algebra and image-processing, the Network Extension framework used for Virtual Private Network configuration and the Generic Security Services framework that provides a standardized set of security services to applications [32, pp. 48-51].

The layered approach to architecture in iOS allows simple applications to be developed relative easily, while also providing access to lower level functionality when appropriate. Higher level layers offer greater abstraction and ease of use while the low-level ones expose

more features not available in higher layers. Developers are encouraged to use the services provided by a higher-level layer over those provided by the lower level layers. [32, p. 8]

2.3 Android application development and architecture

Google's Android operating system can be found running on a variety of mobile devices and unlike iOS, its use is not limited to devices produced by its developer, Google. Also, unlike iOS, the development environment doesn't require a computer running a one specific operating system. Android applications can be developed on Windows, OS X and Linux based development environments [33].

An official IDE called Android Studio is available to Android developers. Android Studio is provided by Google and it is based on JetBrains' IntelliJ IDEA. Android Studio offers a full set of IDE features, many of which are specifically suited for mobile application development and building. Some features included in the Android Studio are the Gradle-based build support, lint tools to help developers maintain application performance and combability, ProGuard implementation for code obfuscation, app-signing, a layout editor, performance and memory profilers, and deep code analysis [34].

Android as a platform is an open source software stack based on Linux operating system [34]. The platform's architecture is divided into six major component categories. These categories are, counting from the highest to the lowest: System Apps, Java API Framework, Native C/C++ libraries, Android Runtime, Hardware Abstraction Layer (HAL) and finally the Linux Kernel [35]. The Android platform software stack is arranged as shown in the Figure 8.

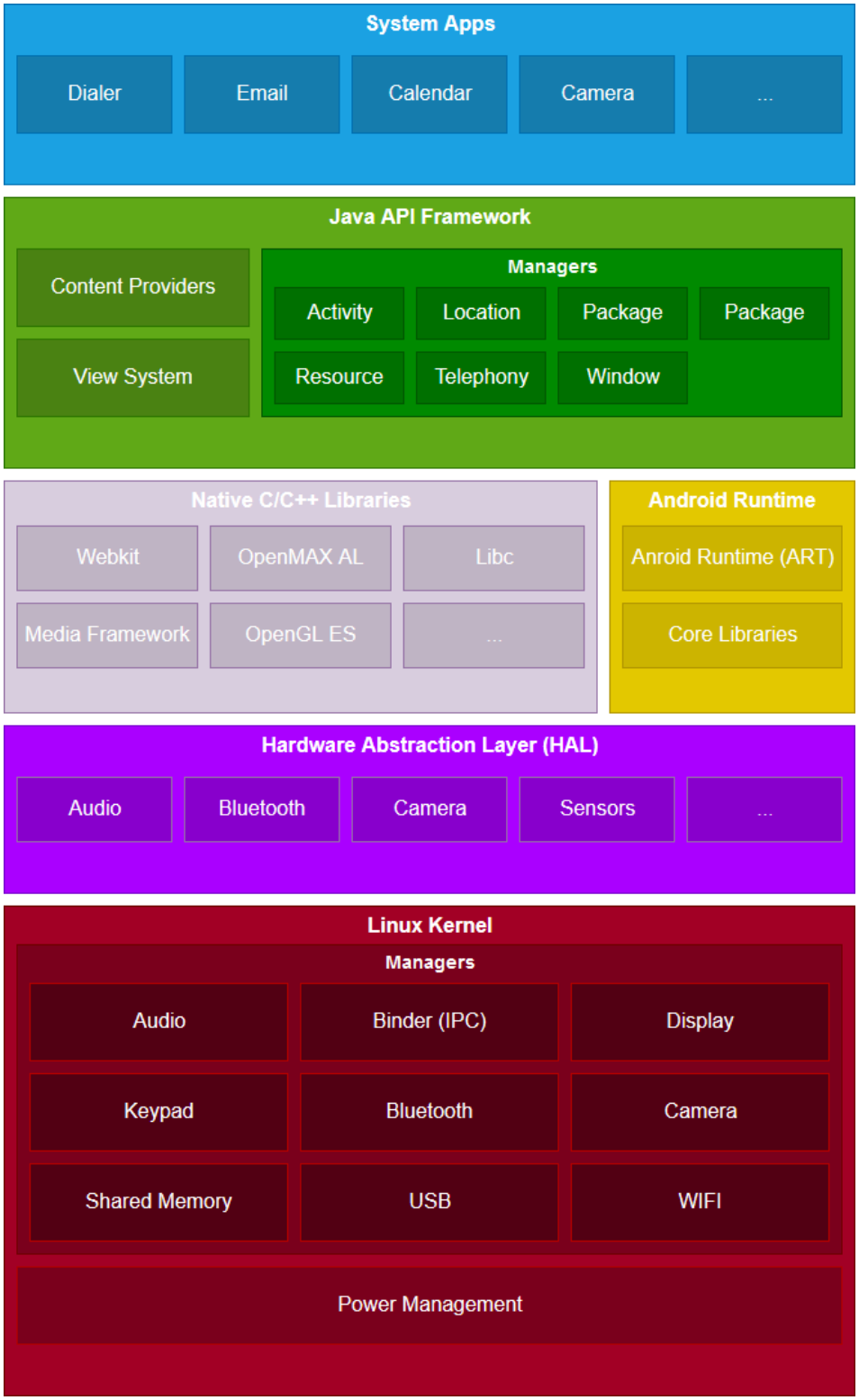


Figure 8. The Android software stack. [35]

System Apps layer of the Android architecture contains the core applications that are included in the Android platform. These applications provide basic mobile device

functionality such as email, SMS messaging, calendars, internet browsing and contacts. Although these core applications are provided with the platform, they can be replaced by third party applications as system defaults by the user. These applications also provide features that third-party applications can use, allowing them to call upon these System Apps for functionality that is not included in the third-party application itself. [35]

The Java API Framework layer exposes the Android OS features to the developers. These Application programming interfaces (API) allow Android developers to easily use core, modular system components and services that together enable the building of an Android application. These building blocks contain such elements as the View System that is used to build the UI for the application, the Resource Manager that can be used to access non-programmed resources such as graphics and localization files, the Activity Manager that manages the applications lifecycle and provides applications navigation, and other key services. [35]

The next layer of the Android platform architecture has two distinct entities. The first one is a collection of native libraries written in C and C++. Many Android system components are built using native code and as such require C and C++ libraries. The Java API Framework mentioned earlier exposes some functionality of these libraries to applications. An example of the available functionality is the OpenGL ES library that provides support for two-dimensional (2D) and three-dimensional (3D) graphics in applications. [35]

Android Runtime (ART) makes up of the second half the layer. In newer Android devices each application has its own ART instance. This instance is used to run the applications processes, separating each application's operation to separate virtual machines (VM). Android Runtime is developed to operate multiple VMs in low memory environments such as mobile devices. This is achieved by using a specific bytecode format called DEX that is optimized for Android and has minimal memory footprint. In older Android versions a VM called Dalvik was used as the Android runtime. [35]

The second deepest layer in the Android platform's software stack is the Hardware Abstraction Layer. This layer contains the interfaces that expose device hardware functions to the higher-level Java API Framework. Several library modules that each implement an

interface for a specific hardware component make up the HAL. Android system loads the needed library module for the hardware component when a request for it is made through the Java API Frameworks. [35]

The Linux Kernel is situated at the bottom of the software stack and is the foundation of the Android platform. It provides functionality such as threading and low-level memory management to the higher layers [35]. The advantages of using the Linux kernel as the base for Android are the security features it provides such as the application sandboxing and UNIX-style filesystem permissions [36], and its wide-spread use in the industry, making it easier for mobile device manufacturers to develop hardware drivers.

2.4 Cross-platform frameworks

As mentioned in the introduction chapter, the splitting of the mobile application market in 2010s has fueled the development of new technologies to aid multi-platform mobile application development. These cross-platform frameworks each offer their own approach to solving the same core-challenge of writing cross-platform applications that are seamlessly indistinguishable from platform native applications.

This section of the thesis presents a quick overview of five different cross-platform frameworks. The five frameworks selected are React Native, Xamarin, NativeScript, Ionic and Flutter. These frameworks were selected because they are all relatively popular and still being actively updated, thus offering a good cross-section of the current landscape of cross-platform frameworks.

The approaches taken by each of the frameworks was investigated by taking a look into what coding language each framework uses, how does it compile the code and how does it render its views. In addition, the year of the first public release of the framework and its estimated popularity using version control hosting service GitHub's "star" system will be presented. The stars given to GitHub projects are essentially indicators of users marking the project as an item of interest. The data collected is presented in Table 2.

Table 2. A comparison between popular cross-platform frameworks.

Framework	**Language**	**Compilation**	**Rendering**	**Popularity (x10 ^{3})**	**Public release**
React Native	JavaScript. [37]	Interpreter/JIT [38]	Native UI controllers [39, p. 21]	76,7 [40]	2015 [39, p. 3]
Xamarin	C# [41]	AOT/JIT [42]	Native UI controllers [43]	6,1 [44]	2011 [45]
NativeScript	JavaScript/TypeScript [46]	Interpreter/JIT [47]	Native UI controllers [46]	16,8 [48]	2015 [49]
Ionic	HTML, CSS, JavaScript/TypeScript [50]	JIT/WKWebView [51]	HTML, CSS [51]	37,9 [52]	2013 [53]
Flutter	Dart [54]	AOT/JIT [55]	Skia Graphics Engine [56]	62 [57]	2017 [58]

As the data collected shows, JavaScript and its subset TypeScript are popular language choices in the frameworks, with three of the five frameworks using some combination of them. Xamarin's use of C# and Flutter's use of Dart make the distinction here.

The code compilations are mostly done using Just-in-time (JIT) or Ahead-of time (AOT) compilation, usually depending on the platform used. JIT compilation is a run-time compilations technique that compiles the code during the execution of a program [59]. AOT compilation on the other hand compiles the code before execution. The use of different techniques in different platforms is explained by the absence of writable executable memory in iOS applications [38].

Most of the frameworks studied use Native UI controllers for rendering views. Using native UI controllers enables user experience close to native applications. Ionic framework opts for using the combination of Hypertext Markup Language (HTML) and CSS to recreate native behavior. Flutter on the other hand relies on the Skia Graphics Engine to provide rendering support.

When comparing the estimated popularity using the numbers provided by GitHub, two of the frameworks are clearly more popular than others, React Native and Flutter. It is interesting to note that the most popular frameworks are also the latest ones to the market, perhaps signifying that it was only during this latest generation of cross-platform frameworks that they became viable enough to gain mass popularity among mobile developers.

3 REACT NATIVE FRAMEWORK

3.1 Overview

As mentioned in the introduction chapter, React Native is a cross-platform framework developed by Facebook, first released in 2015. React Native framework enables mobile developers to write real native mobile applications simultaneously for iOS and Android [11, p. 17]. It uses JavaScript as its coding language of choice and bears many similarities to web development libraries, allowing skill transference from web to mobile development.

React Native was originally an internal side project to React, a JavaScript web development framework in development by Facebook. React was first used by Facebook in 2011 for a newsfeed feature and in 2012 it was used for Instagram. The framework was made open source in 2013 and its popularity began to increase over the years. With the now established popularity of React in web development and Facebook's need to increase native support, React Native was made available to public in March 2015. [39, p. 3]

React Native has gained a lot of developer support from the community and even Microsoft and Samsung have committed to supporting React Native in their platforms. Nowadays React Native has been used to create many popular mobile applications such as Facebook, Airbnb, Discord, Instagram and Skype. [39, p. 3]

In addition to the technical features mentioned earlier in the introduction chapter such as the framework's architectural bridge between the applications logic written JavaScript and the platform's native layer, the multi-threaded design separating the JavaScript and the UI threads and the Native modules in their own threads, React Native offers other benefits to developer teams adopting it. One important benefit is that React Native allows developers to write native code and call it from the main code written in JavaScript [39, p. 4]. This enables developers to implement code that uses some native device capabilities not supported by default by React Native and use it from the cross-platform code or even re-use some previously developed native modules.

Another benefit React Native has compared to native mobile development is the ability to dynamically update applications without submitting a new application version to Google and Apple for inspection and approval. The caveat to this is that only the parts written in JavaScript can be updated dynamically and any changes to native code require a release of a new version. However, as usually most of the code in a React Native application is written in JavaScript and the frequently long approval times of Google and especially Apple for new application versions, this feature can enable great improvements to development team's ability to swiftly make changes to applications, such as reacting to reported bugs. [39, p. 4]

React Native includes several built-in developer tools that can be accessed using the in-application developer menu that is available whenever the application is not in release mode. From the developer menu application can be instantly reloaded without the need to recompile it. Application development can be accelerated further by enabling the automatic reloading features available in the menu. The first one of these is the Live Reload feature that automatically reloads the application with new code changes whenever they are made. This functionality can be enhanced even further by using the Hot Reloading feature that allows the application to persist its state through reloads. [60]

The developer menu also has features that help developers to more easily debug their applications. The Debug JS Remotely feature can be used to debug application's JavaScript code by either using Chrome Developer Tools found in the Google Chrome web browser or a custom JavaScript debugger [60]. The graphical user interface (GUI) can be debugged using the Inspector feature. It enables developers to see information about the GUI elements by selecting them from the screen. The built-in performance monitor and sampling profiler features help with the application's performance optimization [61]. The usage of the developer menu inside an application is shown in the Figure 9.

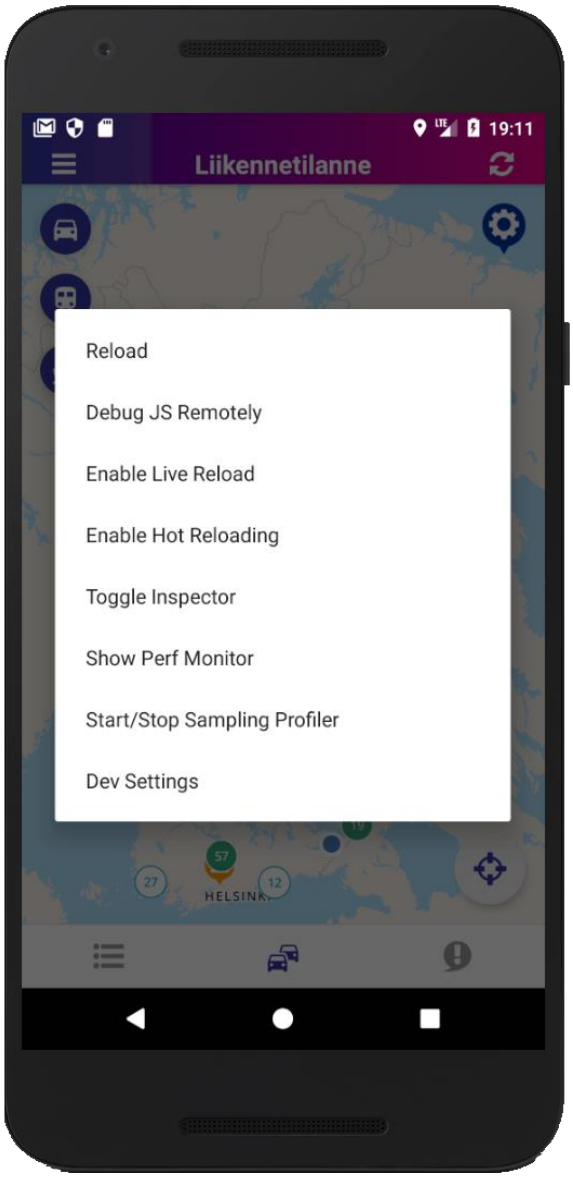


Figure 9. The built-in developer menu in a React Native application.

Many of the features available in the developer menu are made possible by the JavaScript bundler Metro used by React Native. It keeps a graph of all the JavaScript files needed by the React Native application and compiles these required files into a single file. Metro bundler also translates all JavaScript code that the application's host platform doesn't understand into a format that it does and converts assets into objects that can be used by React Native components. Metro is started along with a Node.js server when a React Native application is started in development mode and the bundled file is served to the application. [62]

3.2 React Native concepts

3.2.1 Virtual Document Object Model

Document Object Model (DOM) is an inverted tree structure that represents all the objects in a single document and like many other aspects of React Native, its roots are in the world of web development [39, p. 18]. The DOM consists of an object at the head of the tree that has children, and those children have their own children under them and so on. Each object is tagged with a HTML tag that represents its role in the tree, such as “head” or “body” [39, p. 19].

When changes are made to the displayed view either by code or user action, the DOM is then updated, and the updated DOM is used to render the changed content. Even seemingly small changes in the content can cause big changes in the DOM and as a result force big portions of the view to be re-rendered. This can cause problems as extensive re-rendering can be slow, making changes in the conventional DOM complicated and expensive performance wise. [39, p. 19]

As the view displayed is parsed and a DOM tree with nodes corresponding to the tags in the view is created, a second tree is also created. This render tree includes all the styling information belonging to the view tags. When this style information is processed, a synchronous attach method is called and when a node needs to be deleted, a synchronous detachment method is called. [39, p. 19]

Any changes to one node may lead to changes in any number of other nodes because the view must be recalculated and re-rendered. All of these operations will trigger a potentially expensive synchronous call, possibly resulting in hundreds or thousands of operations. The synchronous nature of these possibly expensive method executions can cause the aforementioned performance issues. [39, p. 19]

Virtual DOM is a solution to the performance problem caused by the DOM updating and re-rendering. It is a layer of abstraction on top of the conventional DOM. Although the virtual

DOM is not used to calculate or render nodes, it is still an inverted tree structure that is made of simple and lightweight JavaScript objects, comparable to Plain Old Java Objects (POJO).

The pivotal difference between conventional DOMs and virtual DOMs is the way they handle changes. When a virtual DOM is changed, an operation is executed that tries to batch all necessary changes into one patch, so only one change to the actual DOM is needed. This functionality is achieved by extensive use of diffing algorithms, that are algorithms that aim to explicitly produce a shortest possible edit script for the given changes [63]. Using this method makes virtual DOM much more efficient when compared to conventional DOM. [39, p. 19]

By using the virtual DOM React framework is able to intelligently differentiate the difference between the existing DOM and those incoming changes made by code. By keeping the update passes to the actual DOM at minimum, React is able to increase efficiency. The efficiency gains are more significant as the complexity of the displayed content increases. React Native adds to this by having the virtual DOM represent platform-native components that are drawn to the screen using platform-native methods. [39, p. 20]

3.2.2 Rendering bridge

The rendering bridge is as a core aspect of React Native framework that works underneath the virtual DOM. The rendering bridge is a platform specific component that is tasked with translating the virtual DOM to a form that the host platform is capable of handling. The bridge communicates directly with the native API's and uses them to create and render native components. [39, p. 21]

This means that React Native can potentially be used with any host platform if there is a rendering bridge component available for that platform. React Native is currently distributed with render bridges for iOS and Android, but anyone can write and distribute render bridges to other platforms, if they so choose [39, p. 21]. For example, Microsoft has released its own support for building React Native applications for Windows 10, Windows 10 Mobile and Universal Windows Platform (UWP) [64].

The rendering bridge is situated between the other two main components, native modules written in Java for Android and Objective-C or Swift for iOS and the JavaScript virtual machine (JVM). The virtual machine is the component responsible for running the JavaScript code that makes up the React Native application's non-native parts. The JVM and native modules run on separate threads and use the bridge to communicate with each other using a custom protocol. [39, p. 21]

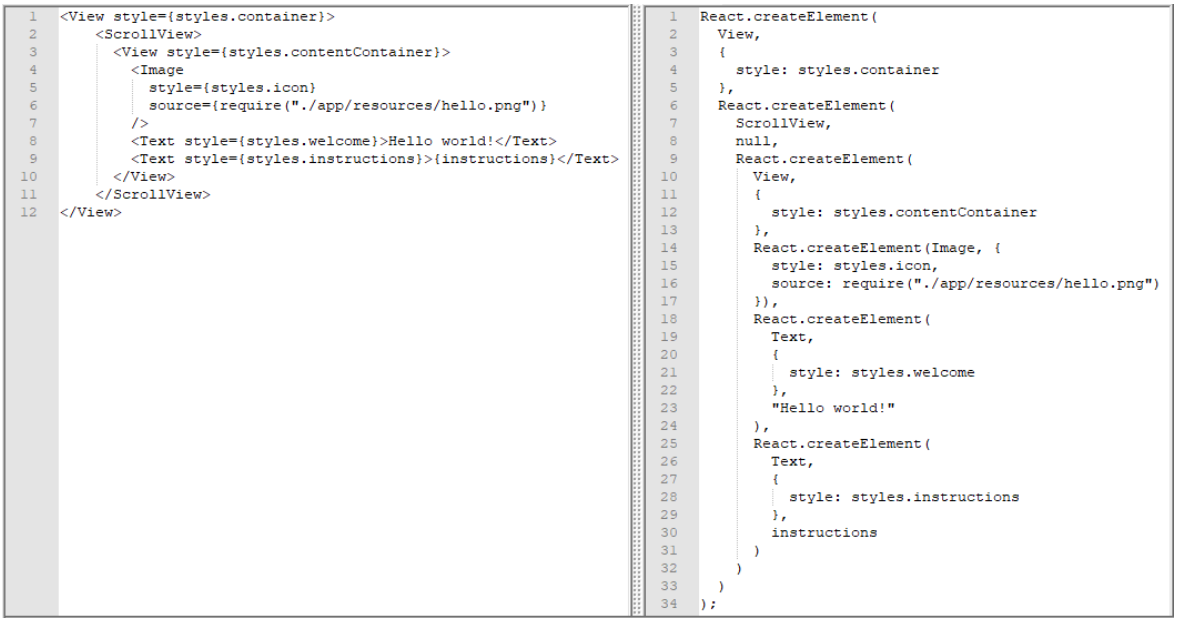
The virtual machine is provided by the high-performance JavaScript Core engine that is also used in the Safari browser on the iOS platform. On Android and other possible platforms that don't have it, the engine is bundled with the React Native application and for those cases the size of the application is naturally larger. [39, p. 21]

On application startup, a native code component that launches the JVM is run at first [39, p. 21]. The JVM then loads the application specific code that has been bundled into single file. The JavaScript code running on its own thread then instructs the native modules to via the rendering bridge to create the needed components to display the wanted application view. The native modules then use the bridge to inform the JVM once their operations are completed [39, p. 22].

3.2.3 JSX

React Native is typically written in JSX code. JSX stands for JavaScript XML and it has elements from JavaScript, CSS and Extensible Markup Language (XML) which itself resembles HTML. JSX was designed as an extension to JavaScript language syntax as a way to embed XML inside JavaScript. [39, p. 22]

React Native processes the JSX code and transforms it to pure JavaScript that is executed at runtime. Because of this functionality, it is possible to write React Native applications by only using JavaScript, but this is not commonly done. Using JSX makes writing React Native applications more straightforward and less verbose compared to using pure JavaScript and for this reason JSX has become the de facto way to write React Native applications [39, p. 22]. A comparison of achieving the same rendered result using JSX (on the left) and pure JavaScript (on the right) is displayed in Figure 10.



```
<View style={styles.container}>
    <ScrollView>
      <View style={styles.contentContainer}>
        <Image
          style={styles.icon}
          source={require("./app/resources/hello.png")}
        />
        <Text style={styles.welcome}>Hello world!</Text>
        <Text style={styles.instructions}>{instructions}</Text>
      </View>
    </ScrollView>
</View>
```

```
React.createElement(
  View,
  {
    style: styles.container
  },
  React.createElement(
    ScrollView,
    null,
    React.createElement(
      View,
      {
        style: styles.contentContainer
      },
      React.createElement(Image, {
        style: styles.icon,
        source: require("./app/resources/hello.png")
      }),
      React.createElement(
        Text,
        {
          style: styles.welcome
        },
        "Hello world!"
      ),
      React.createElement(
        Text,
        {
          style: styles.instructions
        },
        instructions
      )
    )
  )
);
```

Figure 10. A comparison between JSX and pure JavaScript to achieve the same result.

As seen in the Figure 10, JSX uses tags similar to HTML or XML and they may be self-closing or have an opening and closing tag, depending if they contain content. Additionally, JSX tags may have different attributes such as value or style. JSX may also have varying types of children such as text, JavaScript expressions or other tags. Although JSX uses markup language style tags like HTML, JSX allows embedding of tag-based markup snippets into the code and not the other way around [39, p. 23]. JSX also discourages the separation of markup language, application code and style code into their own separate files [39, p. 24].

3.2.4 Components

The concept of components is one of the core principles of React and therefore, React Native. React Components are self-contained sections of code used to encapsulate the characteristics of a visual element. These characteristics are the element's current state, attributes and the functionality it contains. As React Native components are mapped to suitable native platform components, they can be seen as a way to instruct the native layer how the underlaying native component should look like and how it should work. [39, p. 24]

React Native provides a large number of built-in components. These components include ones from such categories as basic components, user interface, list views, iOS and Android

platform specific components and others [65]. The React Component class acts as base for all React Native components regardless of its use or origin and as such, they all extend it, either directly or indirectly following the principles of object-oriented programming [39, p. 24]. An example of a component that wraps other native and third-party components within itself is displayed in Figure 11.

```
import React from "react";
import { StyleSheet, TouchableOpacity } from "react-native";
import MaterialIcon from "react-native-vector-icons/MaterialIcons";
import MaterialCommunityIcon from "react-native-vector-icons/MaterialCommunityIcons";

import { colors } from "../../resources";

import { BUTTON_SIZE, BUTTON_MARGIN, ICON_SIZE } from "./constants";

export default class MapTabButton extends React.Component {
  render = () => {
    const communityIcon = this.props.iconType === "community";
    return (
      <TouchableOpacity
        onPress={() =>
          this.props.onSelection(this.props.id, !this.props.selected)
        }
        style={styles.mapTabButton}>
        {communityIcon && (
          <MaterialCommunityIcon
            style={styles.mapTabButtonIcon}
            name={this.props.icon}
            size={ICON_SIZE}
            color={
              this.props.selected ? colors.white : colors.disabledMapButtonBlue
            }
          />
        )}
        {!communityIcon && (
          <MaterialIcon
            style={styles.mapTabButtonIcon}
            name={this.props.icon}
            size={ICON_SIZE}
            color={
              this.props.selected ? colors.white : colors.disabledMapButtonBlue
            }
          />
        )}
      </TouchableOpacity>
    );
  };
}

const styles = StyleSheet.create({
  mapTabButton: {
    backgroundColor: colors.primaryBlue,
    borderRadius: 0.5 * BUTTON_SIZE,
    alignItems: "center",
    justifyContent: "center",
    width: BUTTON_SIZE,
    height: BUTTON_SIZE,
    marginBottom: BUTTON_MARGIN,
    elevation: 10,
    shadowColor: colors.black,
    shadowOpacity: 0.8,
    shadowRadius: 2,
    shadowOffset: {
      height: 2,
      width: 0
    }
  }
});
```

Figure 11. An example of React Native component.

All React Native components must provide a rendering method that returns some valid JSX code used to describe the component, as is seen in the Figure 11. The render method always

returns a single component that encloses all other elements between it's opening and closing tags. Other custom made or third-party components may also be embedded in the parent component's JSX code. [39, p. 24]

React Native components can have attributes that describe them and event handlers in the form of props. The important concept of props will be discussed further later. Components may also have methods that enable other parts of code to call it to perform all manner of functions. [39, p. 24]

3.2.5 Component Life Cycle

As mentioned in the previous subchapter, React Native components have only one required method, the render method. This requirement is in place because without the rendering method, components would not be displayed, defeating their intended purpose. There are, however, other methods that are called during the component's lifecycle events, but it is not mandatory to explicitly implement them. The component's lifecycle is displayed in Figure 12. [39, p. 26]

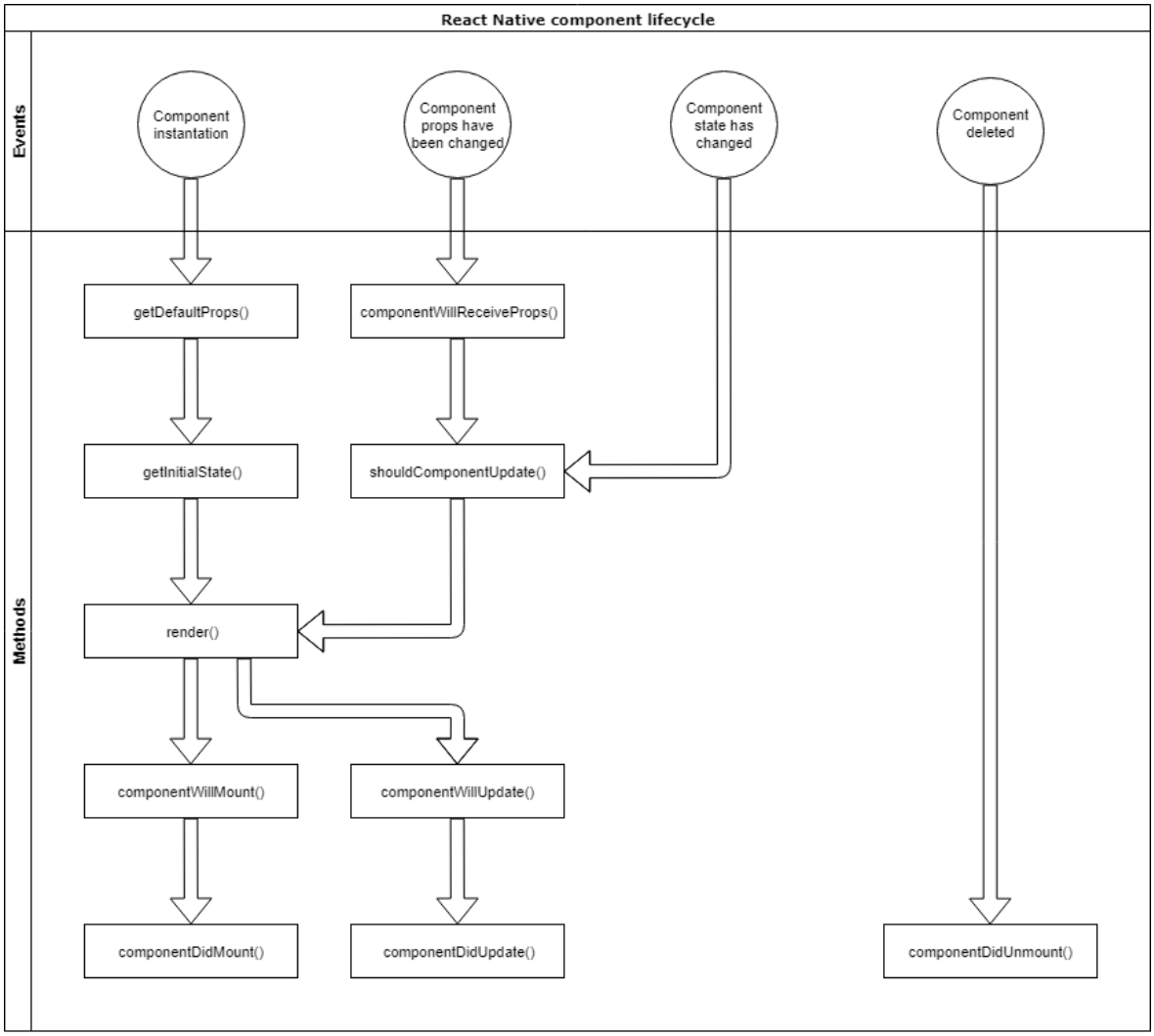


Figure 12. React Native component lifecycle illustrated as a flowchart. [39, p. 25]

As can be seen from the Figure 12, the getDefaultProps method is first called when a React Native component is instantiated from the JSX code and the getInitialState method is called after it. These methods are related to the handling of the component's props and state, and as a result the component has its initial values for any possible internal data. These concepts will be discussed later in the next sub-chapter. [39, p. 26]

After the initialization methods the component's render method is called. After the rendering is finished and the component is added or "mounted" to the virtual DOM, the componentDidMount method is called, signaling that the component is live and active. This method can be used to handle all the necessary startup logic of a component, such as data

fetching. After this initial phase of the component's lifecycle a few different things can happen that determine what the next method to be called is. [39, p. 26]

First, the component's internal attributes, the previously mentioned props, may change and the componentWillReceiveProps method is called, indicating that the component is about to receive new props. The next method to be called is the shouldComponentUpdate method that is used to examine the current state of the component's internal attributes and data to determine if the virtual DOM needs to be updated following the prop change. [39, p. 26]

If the shouldComponentUpdate method's return value is true, the rendering method is called again. The rendering method is designed to take into account the component's current state, so it returns the updated visualization of the component. After the rendering method has finished its job drawing the component, componentWillUpdate method is called and the view is updated with the newly rendered component. Lastly, componentDidUpdate method is called at the end of the render life cycle. [39, p. 26]

Other events that will trigger the component's rendering lifecycle are the changing of component's state and the deletion or unmounting of the component. When setState method is called, the rendering lifecycle is similar to when the component's props are updated. After the component's state is changed, the lifecycle will skip to shouldComponentUpdate method continue from there. If the component is deleted, it is removed from the virtual DOM and componentDidUnmount method is called. [39, p. 26]

3.2.6 Props and State

The previously mentioned concepts of component props and state are closely related to each other as they both represent the data a component has. The difference between the two is that while props are usually viewed as being static, the component's state is more fluid and changing. In other words, props generally define attributes of a component that are determined upon the component's creation and state contains the component's current data that can change more often during the component's lifecycle. [39, p. 27]

An example of these concepts can be seen in the custom “ExampleTextInput” component that is displayed in the Figure 13. The custom component has its own state and it wraps a framework native “TextInput” component. The “TextInput” component is given props such as “editable” and “maxLength” that specify its behavior when it is mounted on the view. One of the props given is the “onChangeText” that contains a function expression that changes the parent component’s state when the child’s value changes. The parent’s state is also tied to the child as it receives the parent’s state attribute “text” as its value prop.

```
1   import React from "react";
2   import { TextInput } from "react-native";
3
4   export default class ExampleTextInput extends React.Component {
5     constructor(props) {
6       super(props);
7       this.state = { text: "Placeholder text" };
8     }
9
10    render() {
11      return (
12        <TextInput
13          editable={true}
14          maxLength={40}
15          onChangeText={text => this.setState({ text })}
16          value={this.state.text}
17        />
18      );
19    }
20  }
```

Figure 13. A custom React Native component wrapping a framework native component.

The component’s prop values are usually contained within braces, as in JSX everything inside the braces is considered a JavaScript expression and the outcome will be interpreted and resolved as a result. Static values surrounded by quotes can also be used, but the using braces is far more powerful and the more common method. The component’s props can be accessed when needed using the props attribute with “this.props”. [39, p. 27]

As seen in the Figure 13, React Native components can have optional constructor functions that are usually used when dealing with the component’s state. The constructor receives an

object containing all the component's props as a parameter and as React Native is based on object hierarchy, the "super(props)" has to be called first to give the parent component a chance to handle the props. The constructor can then be used to set the default state values. [39, p. 29]

After the state's default values have been set in the constructor, it is not encouraged to directly manipulate them, but to use the setState function. The setState function merges the passed object intelligently with the current state object and those values not included in the passed object remain unchanged. As mentioned in the previous subchapter, the setState function triggers the component's rendering function that in turn renders the component using the new state values. Bypassing the setState function and directly manipulating the state can lead to the component being inconsistent and its displayed view not being up to date. [39, p. 29]

3.2.7 Styling

Styling the components in React Native differs a bit from what is usually considered standard in web or mobile development due to the nature of JSX. As mentioned before, because React Native components are supposed to be self-contained, the component's code and markup language should all be contained within the same source file. This is also true for any styling that is applied to the component and the subset of CSS used in JSX is usually incorporated into the component. However, this is not forced in any way. [39, p. 30]

The subset of CSS used in React Native is a simplified version of the CSS used in web development. The layout is primarily based on flexbox and many parts unnecessary to mobile development are left out. Another difference to the CSS used in web development is that there are no cross-browser issues when styling a view, the styling always works the same regardless of platform. [39, p. 30]

In addition to the differences already mentioned, the biggest variation from React Native styling to the styling used in web-based development comes in the form of inline styling. A React Native component has a prop called "style" that is used to pass a JavaScript object containing the styles being applied to the component. This object can either be directly

defined in the inline or provided using a StyleSheet object supplied by React Native that acts as an abstraction. [39, p. 31]

The StyleSheet object's "create" method accepts a JavaScript object containing the styles and returns a new instance of a StyleSheet object containing the defined styles. This method is used more often, and an example of its usage is seen in the Figure 11. [39, p. 31]

While using the StyleSheet object method, CSS properties are also defined as attributes of that object and they are separated with commas just like any other JavaScript object, unlike stylesheet definitions normally used in web development. Values must also be contained within quotes in some cases, according to JavaScript syntax. In addition to the "create" method, StyleSheet object also provides a "flatten" method that can be useful in combining an array of style objects into a single StyleSheet object. [39, p. 31]

3.3 Current state and the future of development

React Native is currently still under development and received its latest update, version 0.59.8 in May 2019. This continues the ongoing trend of the framework receiving a new major version about every three months with numerous minor updates in the between [66]. There is currently no information available about the version 1.0, either in the form of a release date or planned features.

While Facebook and a few other parties are still mainly responsible for the development of the React Native framework, there has been a concentrated effort to shift more of the framework's development into the hands of the developer community. This has been done by developing a better and more formal structure for collaboration between the community developers and Facebook. A new set of standards will be developed and enforced to help with the framework's development and to maintain the quality of the code. [67]

Facebook has also started to remove non-core components from React Native in an effort to reduce the surface area of the framework. These previously included core components will be now transferred to the community, who will be responsible for their maintenance. The

goal of this component pruning is to allow for faster development of the core framework. The WebView component is one example of these now community-maintained components. [68]

All in all, React Native seems to move forward at a steady pace, even though the long awaited 1.0 release version appears to still be a ways off. The development is shifting more to the developer community and while this may seem like a first move by Facebook to start distancing itself from the project, there is no concrete evidence that this is the case. The public interest in the framework remains high as evidenced by the Google Trends graph about React Native shown in the Figure 14. The public interest over time shown in the graph is measured using the Google's search engine activity.

Figure 14. A Google Trends graph about React Native. [69]

4 IMPLEMENTATION

4.1 Introduction

This chapter of the thesis describes the mobile application development project completed using the React Native framework. The goal of the project was to produce a working and release ready cross-platform mobile application used for monitoring the Finnish traffic situation. The chapter has information about the project's initialization, development environment and tools, project structure and component hierarchy and solutions used for several key aspects of the application.

4.1.1 Project initialization

The implementation of the traffic situation mobile application began with the initialization of a React Native project. Before the initialization process could start, the development environment had to be prepared correctly, and all required dependencies had to be installed. The required dependencies for React Native are Node runtime environment, the React Native command line interface (CLI), Python2 language, a Java Development Kit (JDK), and since the main development platform was a computer running Windows operating system, Android Studio [70].

After the dependencies were installed, the project initialization itself was started with React Native CLI using the command "react-native init placeholderProject" with the actual name of the project replacing the placeholder name. The initialization process creates the basic project folders and files, downloads the needed modules using Node Package Manager (NPM) and creates Android and iOS native projects needed to run the application on those platforms.

When using the initialization CLI command, it is possible to specify the React Native version for the project using "--version" argument. If no argument is given, the latest available version is used. When the development of the traffic situation application began, the latest version was 0.53.0 and it was used as the default option. During development the React Native version was updated to 0.57.0 for better performance, features and stability.

4.1.2 Development environment and tools

As mentioned before, the traffic situation application was mainly developed on a Windows 10 machine. Because iOS development is limited only to Apple devices, a secondary development machine running macOS High Sierra operating system was used to test and build the iOS versions of the applications. The more powerful main development machine hosted the Node.js server running the Metro JavaScript bundler and the secondary development machine was configured to connect to the development server on the main machine. This allowed for any code changes to be instantly reflected on both platforms for the ease of development.

The IDE used for the JavaScript part of the development was Visual Studio Code with a few helpful extensions. React Native Tools extension provided easy access to React Native debugging tools and a way to quickly run React Native commands directly from the IDE command palette. ESLint and Prettier extensions were enabled and configured to ensure good code formatting, readability and style. These configurations force such rules as naming case conventions, inline styling prohibitions, ECMAScript 6 variable keywords and more. The configuration file example for ESLint can be seen in the Figure 15.

```
{
  "globals": {
    "babelHelpers": true
  },
  "extends": ["eslint:recommended", "plugin:react/recommended"],
  "plugins": ["react", "react-native", "prettier", "import"],
  "settings": {
    "react": {
      "createClass": "createReactClass",
      "pragma": "React",
      "version": "16.5"
    },
    "import/cache": { "lifetime": 5 },
    "propWrapperFunctions": ["forbidExtraProps"]
  },
  "env": {
    "es6": true,
    "react-native/react-native": true
  },
  "parser": "babel-eslint",
  "parserOptions": {
    "sourceType": "module",
    "ecmaVersion": 6,
    "ecmaFeatures": {
      "jsx": true
    }
  },
  "rules": {
    "camelcase": 1,
    "no-console": 0,
    "no-var": 1,
    "prefer-const": 1,
    "semi": 2,
    "arrow-body-style": [1, "as-needed"],
    "react/prop-types": 0,
    "react-native/no-unused-styles": 2,
    "react-native/split-platform-components": 2,
    "react-native/no-inline-styles": 2,
    "react-native/no-color-literals": 2,
    "prettier/prettier": 1,
    "import/first": 2,
    "import/order": [
      2,
      {
        "groups": ["builtin", "external", "parent", "sibling", "index"],
        "newlines-between": "always"
      }
    ]
  }
}
```

Figure 15. An example ESLint configuration file.

For Android and iOS native code development, the IDEs used were Android Studio and Xcode respectively. Testing of the application on these platforms during development was mostly performed using emulators and simulators provided by Android SDK and iOS SDK. This ensured rapid feedback to code changes and ease of testing without the need to physically switch devices mid-development, especially while using the Live Reload and Hot Reloading features provided by React Native. Physical mobile devices were occasionally used for testing, especially when a release candidate build was being tested.

4.2 Structure

4.2.1 Project structure

The structure of the project can be separated at the highest level to four distinct parts. These four sections were automatically created in the project initialization and contain the boilerplate code for a React Native project, acting as a starting point for a more specialized application.

First, there was an “app” folder that contained the React Native JavaScript code itself that defines all the application logic, functionality and appearance options. In addition to the React Native JavaScript code there were the “android” and “ios” folders containing the platform specific native projects that enable the React Native application to be run on those platforms.

Finally, there was a folder for all the modules installed using NPM. Among those modules there were the React Native core modules and their dependencies that without the application would not function, and also all the optional third-party modules that were used in the application. The project structure is presented in Figure 16.

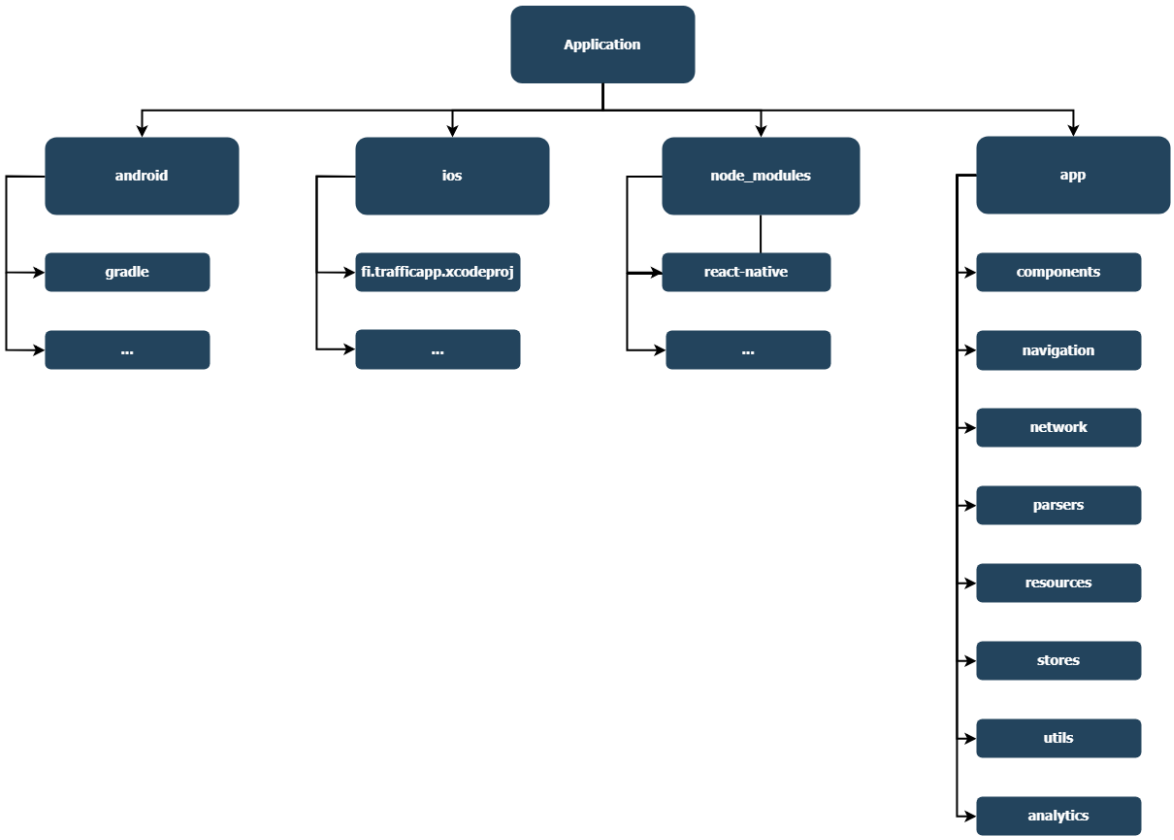


Figure 16. The traffic situation application's project structure.

While React Native doesn't force any particular application structure other than having the App.js file as the main entry point to the application [39, p. 87], the "app" section of the project was further divided into smaller divisions, each responsible for one aspect of the application's functionality. This was done to ensure modularity and ease of code maintenance.

The part containing the largest amount of code was the "components" section that had all the custom-built React Native components that made up the application. Next, there was the "navigation" section that contained all the application navigation components, navigation routes and stack information that defined how the user could navigate the application's screens.

The third and fourth segments in the "app" section were both used to handle the applications incoming and outgoing data. The "network" section contained all the different types of data fetchers and data caching logic for when the application needed to communicate with various

different application programming interfaces (API). The classes in the “parsers” section for their part handled transforming the inbound and outbound data to a form that could be used by the receiving party, either the mobile application or the corresponding API.

The resources needed by the application were stored in the “resources” section of the project structure. These resources were things like fonts, color-codes and images that were packaged with the application during builds and they were used to give the application a custom look and feel. The “stores” section of the project contained all the data stores that were used to save and distribute the application’s data.

Lastly, there were the “utils” folder containing all the shared utility classes used by other parts of the application and the “analytics” folder that had the code related to anonymous user data gathered using Google’s Analytics service. The analytics data was gathered to map out user behavior within the application and to report possible errors, crashes or Application Not Responding (ANR) situations.

4.2.2 Component hierarchy

As mentioned previously, React Native only dictates that App.js acts as the main entry point to the application and all other components are wrapped within it. Everything else about the hierarchy is left to the individual developer teams to decide. In the traffic situation application, the hierarchy of components could be regarded to be similar to an inverted tree structure, much like DOM. Each component usually wrapped its own child components that in turn had their own children or “branches”.

Directly below the App component in the hierarchy was Provider, a wrapper component from the MobX library that introduced an observable global state for the application. Below the Provider component there were AppDrawer and Drawer components providing the application with an openable drawer style side menu that was used as a quick access point to the application settings in the UI.

Next there was SwitchNav, a navigation component providing two branching routes in the application, the OnboardingScreen and RootStack components. OnboardingScreen was high

level component that wrapped components that were used to display new users an onboarding tutorial for the application. RootStack was another navigation component used to direct the applications navigation flow.

SettingsStack was one the two branching components from RootStack, also a navigation component. It wrapped four high level components: SettingsScreen, LocationSettingsScreen, UserDataSettingsScreen and NotificationSettingsScreen. Each of these components wrapped components used to display a distinct aspect of the application settings.

The other component branching from RootStack was TabNav, a navigation component providing the main navigation tabs for the application UI and wrapping the three main navigation stacks of the application. These stacks were wrapped in the HomeStack, MapStack and FeedbackStack components.

Homestack wrapped the HomeScreen and SearchFavoritesScreen components used to display traffic notifications, favorites and a view to house a search functionality to make searches into the available data. WeatherCameraScreen was a component that was shared between the HomeStack and MapStack due to there being a way to access it from both navigation paths.

Additional components that MapStack wrapped were MapScreen, TrainStationView, TrainView, HarborScreen and MapFilterOptions. These components were used to present a map displaying traffic situation information, train stations, trains, harbors and options to filter all the available data.

The final navigation component branching from TabNav was FeedbackStack. This stack wrapped the FeedbackScreen that provided users a way to view old feedbacks on an interactive map and an option to submit their own. MapFilterOptions was shared with MapStack due there being a similar map displayed in both. The high-level component hierarchy discussed in this subchapter is displayed in a tree diagram form in the Figure 17.

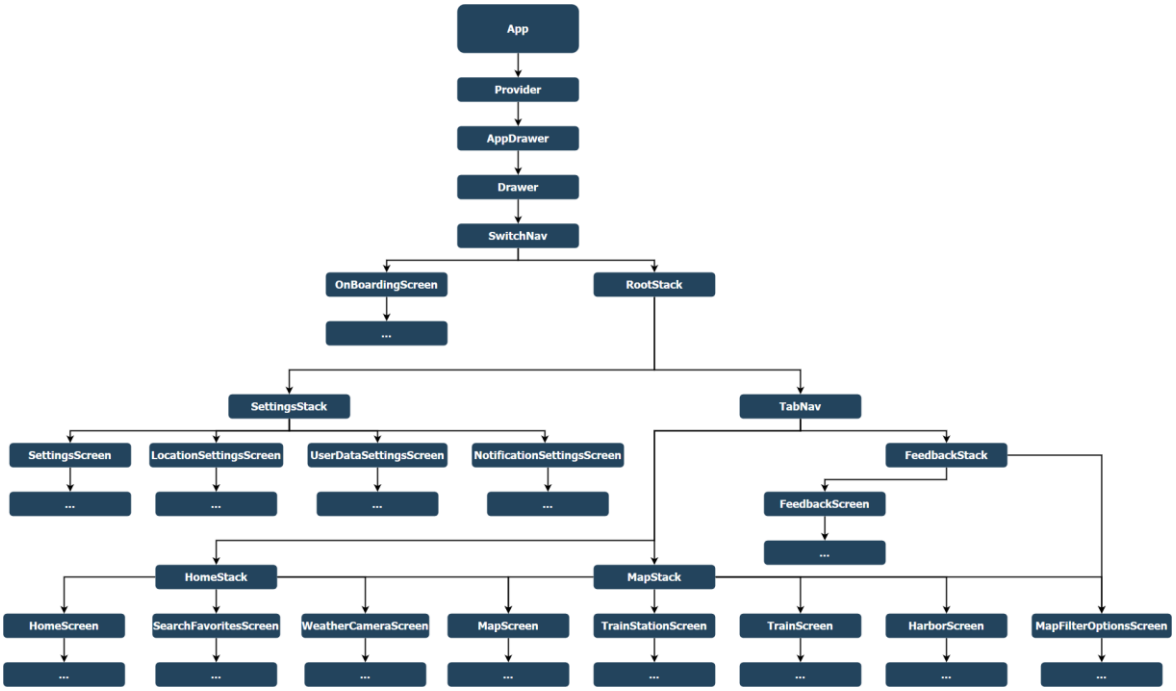


Figure 17. The high-level component hierarchy.

4.3 Application navigation

As mentioned in the previous subchapter, the traffic situation application was divided into multiple view configurations or “screens” as they usually are called in a mobile context. While having multiple different screens in mobile applications is in no way unusual, React Native doesn’t provide official navigation utilities or components to manage the presentation of and transition between the screens [71].

To handle this aspect of the application a third-party navigation library React Navigation was used. The React Navigation library is a community developed and maintained, open source solution that is even recommended as an option for application navigation in the official React Native documentation [71]. However, there are also many other navigation libraries available for React Native.

React Navigation provided the traffic situation application the screen transition functionality and application navigation history management. It also provided the support for gestures and animations that gave the application the native feel users are used to when they interact with native Android and iOS applications.

Using the navigation library, the application could push and pop items from the navigation stacks and move between different navigation routes when needed. To provide this functionality, React Navigation supports various kinds navigators, each suited for a different use case. The navigators used for the traffic situation application were the switch navigator, the tab navigator and the stack navigator.

The switch navigator's purpose is to only ever show one navigation screen at a time and as such, it doesn't handle back actions [72]. This was the navigator used in the traffic situation application to direct users either to the onboarding screen designed to provide them with a short overview of the applications features or to bypass the onboarding experience completely, depending on if they had used the application before or not. The component containing the switch navigator was the SwitchNav component seen in Figure 17.

The tab navigator is designed to allow users to easily switch between different routes. The navigation tabs can be at the bottom or at the top of the screen. The navigation routes within the tab navigator are initialized lazily so the screen components will not be mounted until they are first displayed [72]. In the traffic situation application, a bottom tab navigator was used to allow navigation between the routes that that contained the application's main features, the home stack, the map stack and the feedback stack. As seen in Figure 17, TabNav was the component wrapping the tab navigator. Figure 18 displays the navigation bar at the bottom of the application.

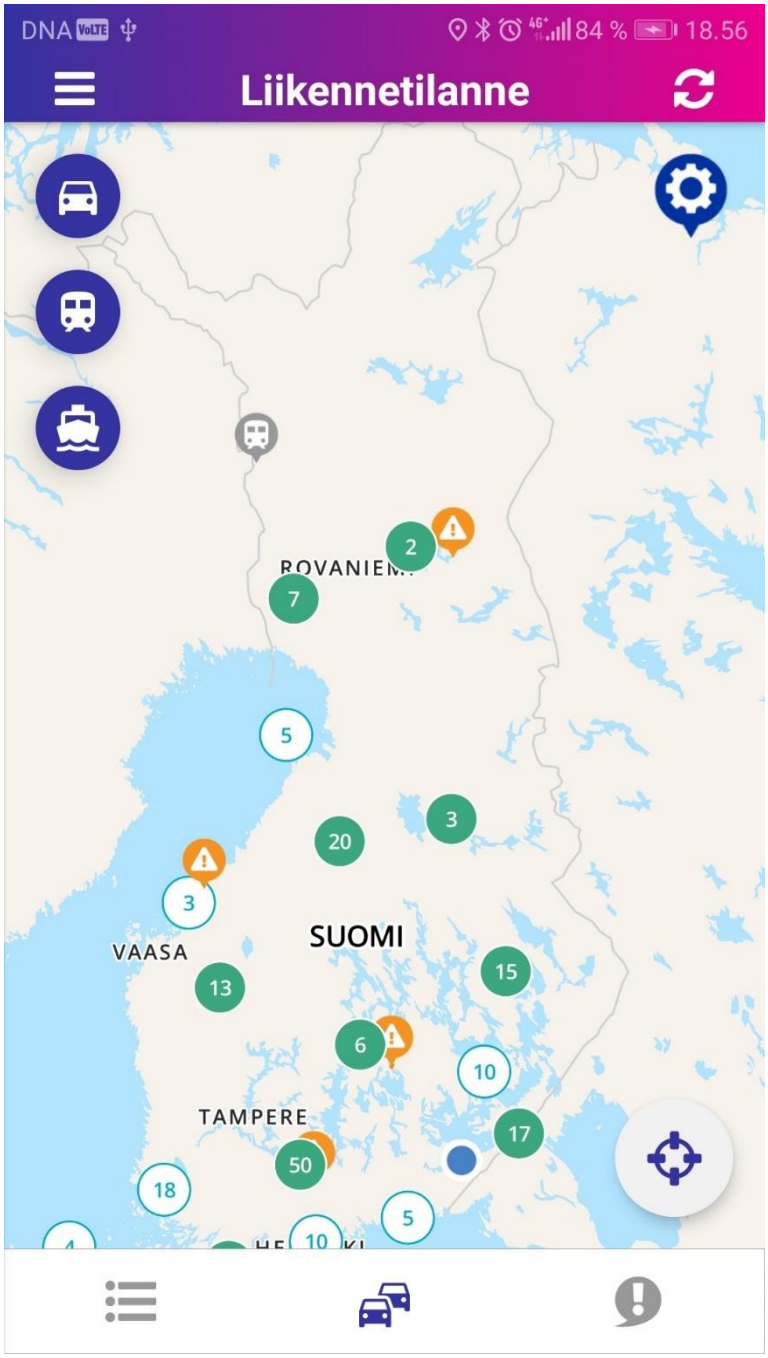


Figure 18. The tab navigator is displayed at the bottom of the application.

The most used navigator in the traffic situation application was the stack navigator. This navigator is used to provide navigation stack where the routes' screens can transition between each other and where each new screen displayed is set at the top of the current stack [72]. The three main screens of the application mentioned previously were placed in navigation stacks, because their business logic was best suited for the two directional back and forth transitions provided by the stack navigator.

In React Navigation, each navigator's routes are configured using a route configuration object and an options object. The route configuration object is used to provide the navigator with the screens or routes it can transition to. The optional route options object can be used to further configure the navigator [72]. In traffic situation application it was used for setting a navigation route as the initial route and a specific navigation mode, for example. Every screen in a navigation route is also passed a navigation prop. This prop contains numerous support functions that can dispatch navigation actions to the navigator [72].

To pass data between navigation routes, the navigate and push functions of the navigation prop that are used to initiate transitions, allow an optional second argument. This parameter argument makes it possible to inject data to the destination route [45]. The data passed can be read with the use of getParam function [72]. In traffic situation application this functionality was used extensively, for instance to pass data from the main map screen to the map feature specific info screens.

When using React Navigation to transition between screens, the normal React Native the component lifecycles are a bit altered. A component's componentDidMount and componentWillUnmount functions will not always be called when expected because depending on the navigation route configuration, the component can still be mounted while not displayed. React Navigation makes four new events available to allow the application to be able to respond to the changes in navigation. These events are: willFocus, willBlur, didFocus and didBlur. [72]

4.4 Networking

Networking functions of the traffic situation application were written into their own helper classes, separate from the components that used them. These helper classes were collected under the networking section of the application, as seen in the Figure 16. Each part of the application's business logic had its own networking helper class responsible for handling its specific networking needs. An example of this would be a need to fetch information about weather cameras for the map screen, or posting a feedback created in the feedback screen to the feedback API.

As JavaScript is inherently single threaded and traffic situation application had many situations where multiple codependent networking operations were simultaneously executed, all networking operations were wrapped inside promises. Promises in JavaScript represent a result of an asynchronous operation, such as fetching data over network. A promise will be either fulfilled or rejected and code can be written to handle either eventuality. In the application this commonly meant either continuing with the current functionality or showing the user an error and aborting the current operation.

The networking functionality itself inside the promises was handled using the JavaScript fetch API. The fetch API provides an interface for handling request and response objects involved with network requests. The API supports multiple Hypertext Transfer Protocol (HTTP) request methods and header types. Each use of fetch was also bundled with Google Analytics tracking events to gather anonymous application use data.

Another aspect related to networking in traffic management application was the network request data caching. To reduce the amount of data transferred needlessly, as there were several use cases where real time data was not needed, a configurable cache was used to store network request data. The cache was configured to store data a certain, data type specific time and offer it as a request response when applicable.

4.5 Data storage

The caching functionality mentioned in the previous subchapter was achieved by using an open source third-party library React Native Cache Store. The library itself works as a wrapper around React Native's default AsyncStorage key-value storage system. The wrapper adds the aforementioned individual key-value cache expiration that the traffic situation application used.

The React Native's AsyncStorage is a persistent and asynchronous storage system. It is simple and unencrypted and can provide access to data in a global context within the application. Because of AsyncStorage's simplistic and global nature, it is recommended not

to access it directly, but to use additional abstraction on top of it [73]. As mentioned, in the traffic situation application this was achieved by using a wrapper library.

The native code side that AsyncStorage relies upon is different depending on the platform. On iOS, the native code that backs up AsyncStorage stores small data values in a serialized dictionary and bigger values in their own, separate files. On devices that run Android, either RocksDB or SQLite database solution is used to run AsyncStorage, based on availability. [73]

Another storage related aspect of the traffic situation application is the global data stores provided by the previously mentioned third-party MobX library. As explained in the React Native concept chapter, React native renders the applications state into a tree of components. MobX library provides a new mechanism to store and update data that can then be used by React Native to render those components. This mechanism uses a reactive virtual dependency state graph that is only updated when a rendered observable's state changes, to synchronize application state with React Native components [74, p. 20].

The different store classes of the traffic situation application, namely FavoritesStore, FeedbackStore and StateStore, situated in the "store" section of the application hierarchy seen in the Figure 16, utilized MobX and React Native Cache Store to store and manipulate data. With MobX, changes to the application data would update the application's UI globally when needed and with React Native Cache library the data could be preserved even through application restarts.

4.6 Data parsing

The traffic management application relied quite heavily on data retrieved from various third-party APIs and there were multiple cases where the data had to parsed to a more easily usable format. In an effort to keep the more UI oriented components as simple as possible, all the data parsing logic was written into separate parser classes as seen in the Figure 16.

Much of the parsing was centered around the data retrieved from an ArcGIS based mapping and analytics platform. The data retrieved from the ArcGIS APIs was comprised of map feature collections used to display location-based information on the traffic situation map. Each feature collection contained an arbitrary number of features, each with its own type of specific attributes.

While the data format of the feature collections was compatible with the map library used in the traffic situation application so the features could be shown in the map, in most cases the map features themselves contained data that had to be parsed into more readable format in order to display it in the other UI elements. An example of this was the need to parse specific weather station data such as wind speed and direction from multiple different sensor values returned with the feature.

Another notable area of the applications business logic that required data parsing was the feedback service. The feedbacks API used a data format not directly compatible with the map library used, unlike the ArcGIS APIs. This meant that the data retrieved from the feedback API had to be broken down, parsed and built into features that could be used. Also, because the feedback API was used for posting data from the application to the service, the user inputted data gathered from the UI components also had to be parsed into acceptable format before sending it to the API.

This included operations such converting location coordinates from one coordinate system to another. The API returned coordinates in European Terrestrial Reference System 1989 (ETRS89) format while the map library worked with the World Geodetic System 1984 (WGS84). The coordinate conversion was accomplished using an open source library PROJ that is designed to perform cartographic projection conversions.

While in general the data parsing operations in traffic management application were quite simple cases of finding the required data from the data received by using matching attribute keys and combining the found data into suitable data objects, there were cases where this was not enough. For example, in cases like when some sets of data had to be combined together without duplicate entries using some of the feature's attribute as the determining factor, a utility function like "unionBy" from Lodash was used. Lodash is a third-party

JavaScript library that provides numerous functions to simplify the handling of objects, arrays, math and so forth. An example of the use of Lodash function in data parsing can be seen in Figure 19.

```
/**
 * Combines main (primary) and secondary train station data, removing
 * duplicate stations (from secondary data).
 *
 * @param {object} mainStationFeatureCollection - Main train stations.
 * @param {object} secondaryStationFeatureCollection - Secondary train stations.
 * @return {object} Combined train stations as a feature collection.
 */
static combineTrainStations = (
  mainStationFeatureCollection,
  secondaryStationFeatureCollection
) => {
  const features = _.unionBy(
    mainStationFeatureCollection.features,
    secondaryStationFeatureCollection.features,
    "properties.LYHENNE"
  );

  return MapUtils.createFeatureCollection(features);
};
```

Figure 19. An example of a parse function using a Lodash utility function.

4.7 Styles and constants

In traffic situation application, StyleSheet objects, discussed previously as one of the concepts of React Native, were used to define component styles. StyleSheet objects were used as opposed to inline styling to improve the code quality and readability by moving styles away from the render function and giving them meaningful names. Performance wise, making the stylesheets objects was also beneficial because this allows referring to them by identifier, instead of creating a new style object every time a component is rendered.

Colors used in the application's styles were defined in their own resource file, separate from the specific component styles because several components shared the same basic color schemes. By handling all the colors in the same centralized constants enabled quick, easy and comprehensive color changes when they were needed during development, while also maintaining the required color consistency between different components. The colors

themselves were defined in the constants using the “Red Green Blue” (RGB) and “Red Green Blue Alpha” (RGBA) color models in a functional notation, as opposed to hexadecimal that is also supported by React Native. React Native also follows the CSS3 specification by having large array of predefined colors available [75]. These colors can simply be utilized by using their names as the color style value.

String constants used in the traffic situation application were also located in their own source file and a localization library called React Native Localization was used to divide the strings into different localization groups. This way the application would use the devices default localization if it was available, or default to a one that was present. The string resource was exported from the application resource files and could be imported anywhere it was needed.

Application’s configuration constants were made available from the config file located at the application folder’s root. These configuration constants were used to handle all of the base Uniform Resource Locators (URL) of the APIs used by the application and the access tokens required by some the APIs. For security reasons, the access tokens themselves were not hardcoded into the configuration source file, but they were imported there from a separate environment variables file at the times the application was built.

4.8 Project outcome and observations

The traffic situation application project proceeded as well as could be hoped. There were no major setbacks during the project and a release ready application was produced while staying within the project’s budget and schedule. React Native was observed to be quite pleasant to work with and the time spent with development project left a generally positive image of the framework’s capabilities. However, there were some worthwhile observations made and lessons learned during the project.

During the development it quickly became clear that there are large amounts third party libraries available for React Native. This was a great boon for the project as it enabled more rapid development due to not having to produce every required component in-house. However, not all third-party libraries are created equal and some had to be switched after it was found out that they did not fill the specifications the traffic situation application required.

For example, the primary map component library that was used had to be switched to another since it didn't have sufficient performance to display thousands of items at once.

Another lesson learned later in the development was that while React Native doesn't set any limitations to the size or functionality of a single component, it is better to split up the code into smaller units and create more sub-components instead of letting a few become monolithic. It is easy to start adding new functionality to the main components and not see that they quickly become sprawling and hard to understand. There were times especially during the latter half of the project where components had to be refactored into several smaller ones to maintain the code readability and quality.

Debugging the React Native application in general was quite simple, but at times it was observed that React Native could display some genuinely non-descriptive error messages that were hard to understand and as such made the pinpointing of the root cause quite hard. There were also some mysterious and difficult to solve bugs that were finally found out to be caused by a race condition in the framework's native implementation on some devices, but not all. This also brought up the point of testing the application on both platform as there could easily be some unknown differences between them.

Fortuitously, React Native was observed to have lots of community resources available. Most problem situations could be solved either by referencing the framework's documentation or by a simple web search. The framework's popularity could really be seen here as pretty much every problem encountered during the development was already previously reported and solved by the developer community.

5 EVALUATION

The purpose of this thesis was to gain insight on mobile application development for multiple platforms simultaneously by using a cross-platform framework solution. The React Native framework used in the study was selected because it is an established and popular cross-platform solution that offered a good representation of the capabilities and challenges of a modern cross-platform framework.

To help to define the direction of the study, following research questions (RQ) were set for this master's thesis:

- What are the characteristics and features of React Native?
- How to implement React Native in mobile application development?
- What challenges can arise when React Native is used to develop cross-platform mobile applications?

Answers for these questions was sought by first conducting a literary review into the characteristics and features of React Native. Further questions were answered by implementing React Native into a mobile application development project with the end goal of a fully functioning mobile application available on multiple platforms. Summarized answers for each research question gained with these methods are explained below.

RQ1: *What are the characteristics and features of React Native?*

Fairly comprehensive picture of React Native's characteristics and features was gained during the literature review presented in this thesis. In its core, React Native is a mobile cross-platform framework derived from its web-based parent framework React. The mobile framework allows developers write native mobile applications concurrently for multiple platforms.

React Native offers several novel solutions compared to its competitors such as the architectural bridge between the applications logic written JavaScript and the platform's native layer, the multi-threaded design separating the JavaScript and the UI and the Native modules, and the support for native code modules. The use of JavaScript as the framework's

language of choice and its origins in web development frameworks have made it easy for developers to make use of their existing web development skills while working with React Native. These design choices and its fairly mature state have established the framework as the most popular cross-platform framework.

The most important technical features of React Native identified in the literary review can be condensed into three terms: Virtual DOM, JSX and React Components. Virtual DOM is a React Native specific adaptation of the DOM structure, used to represent all the objects in a view. DOM has its roots in web development and the Virtual DOM is an abstraction on top of it, offering increased performance. JSX is the language that React Native is commonly written in and it has elements from JavaScript, CSS and XML. JSX is transformed into pure JavaScript to be executed at runtime. React Components are self-contained sections of code used to represent visual elements. Components encapsulate the elements' attributes and have their own lifecycles, styles, states and props.

RQ2: *How to implement React Native in mobile application development?*

The implementation of React Native into mobile application development proved to be fairly simple and mature process. The development environment setup process was not excessively long and only a few dependencies were required before project initialization. The project initialization itself was an automated process completed using React Native CLI tool. It should be noted that the implementation process was tested using both Windows and macOS platform with similar results.

Development itself was also relatively fast as any previous experience in web development makes React Native feel familiar from the get-go. While React Native doesn't have an official IDE, Visual Studio Code proved to be well-suited for the task with the numerous "quality of life" extensions available that helped to keep the code quality high and consistent. The extensions also made it easy to access the React Native's built-in developer tools. Especially the Live Reload and Hot Reload features proved to be invaluable with the increased rate of visual feedback received after code changes were made.

Lastly, during development it became clear that as the most popular cross-platform framework on the market, React Native has a large spectrum of third-party libraries available. Significant reductions in development time could be achieved by using readymade, open source libraries rather than developing certain features in-house. These libraries were easily and quickly installable through NPM.

RQ3: *What challenges can arise when React Native is used to develop cross-platform mobile applications?*

The largest challenge of using React Native in mobile application development that was identified during the thesis project was the fact that React Native is still not a finished product. While seemingly mature in many aspects, the framework is still under development and doesn't have a stable 1.0 release version in sight. This means that sometimes there are breaking changes between React Native versions or problems with package combability.

Updating the React Native version of an already ongoing project is not simple or easy either. During the traffic situation application project, the React Native version was upgraded from version 0.53 to version 0.57 and it proved to be quite a challenge. There are a few tools provided by React Native to automate the upgrade process, but at least in this case those tools didn't work. The version had to be upgraded by manually implementing each change, file by file, from version to version in a very time-consuming process.

Another challenge that was observed while using React Native was that while it is a cross-platform framework and in theory all platforms can share the same codebase, there are still some platform specific behavior that has to be taken into account, be it in styles or in the framework's use of native features. Developers should not blindly trust that just because a feature is tested to be working on one platform, it will work on the other.

It should also be noted that it is likely that from time to time during development, there will still be a need to solve some problems using native code. This means that while JSX is mostly used in React Native projects, developers still need the skills to work with platform native languages and environments.

Using the observations made in this thesis about the implementation of React Native into a mobile application development project, a general Strengths, Weaknesses, Opportunities, and Threats (SWOT) matrix is presented in the Figure 20.

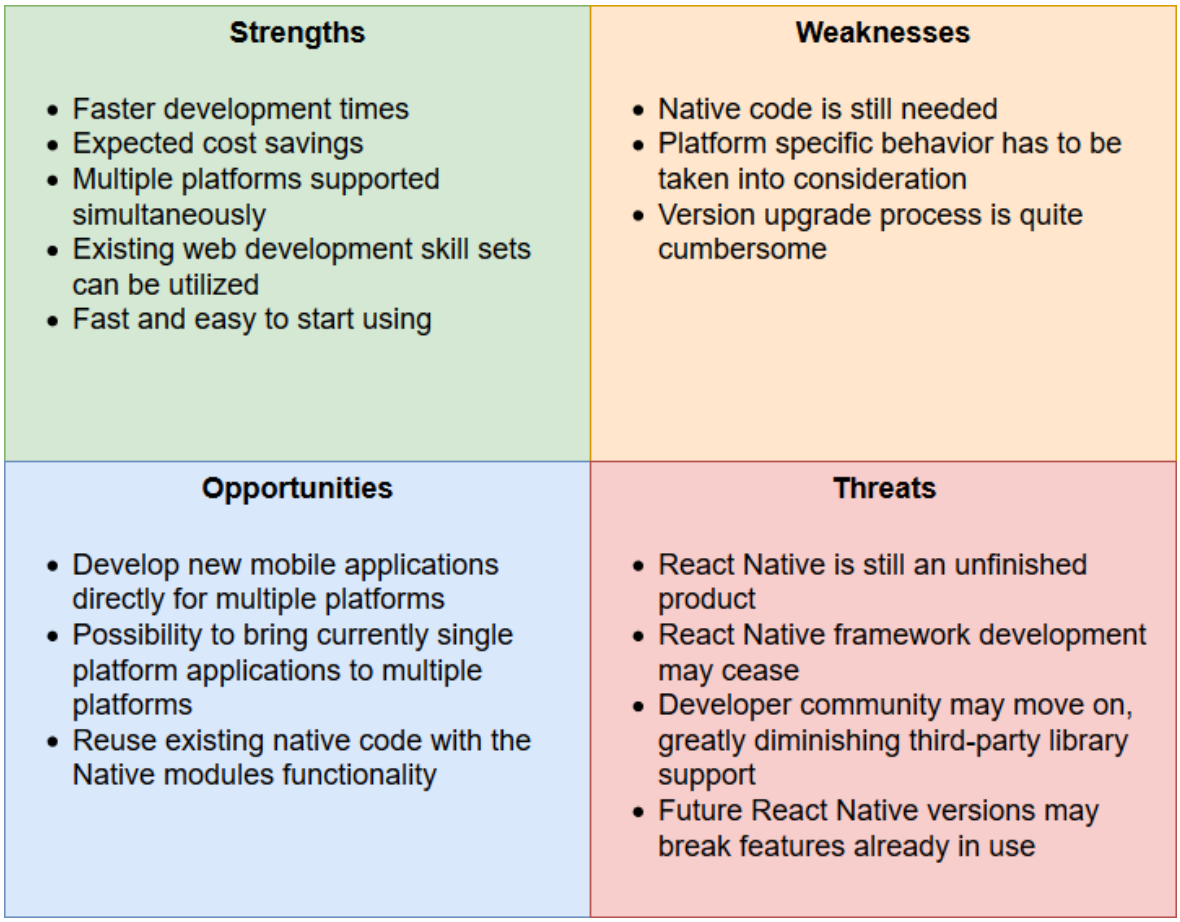


Figure 20. A general SWOT matrix about the implementation of React Native into a mobile application development project.

6 CONCLUSION

The splitting of the mobile application market between the two biggest competitors, Apple's iOS and Google's Android, has created a need for technologies that enable quick and efficient development for multiple platforms simultaneously to reach the widest target audience possible. In addition, the applications developed using these cross-platform technologies should offer a level of user experience that is comparable to the platform native applications. There are several different technologies developed for this purpose, each of them offering its own solution for the daunting dilemma.

For this thesis, one of the cross-platform technologies were selected for further study. Facebook's React Native was selected due to it being the current market leading cross-platform technology with the stated aim of solving most of the problems present in the other cross-platform development solutions. The study began with a literary review into mobile application development in general and more specifically into React Native framework. To gain more firsthand experience and information about the framework, a real mobile application was developed using it.

The literary review and implementation project proved that while React Native still has some lingering problems, mainly due to it still being an unfinished product, it is certainly a viable and usable solution for developing mobile applications simultaneously for multiple platforms. With the low barrier to entry for initial development, large available community support and adequate technical documentation, the application development project done for the thesis succeeded with no major setbacks or slowdowns.

6.1 Limitations and further research

The scope of this thesis was limited to mainly researching and testing React Native on the two market leading platforms, Android and iOS. Even though third parties such as Microsoft have released their own support for building React Native applications on their platforms. The native code side of the framework was also very minimally studied during the thesis, with the Native Modules feature left mainly untouched.

Furthermore, more emphasis could have been put into performance comparisons between an application developed using React Native and a platform native application. All of the three issues just mentioned would good lines of new research to the topic, with the performance testing being especially interesting subject due to React Native's claims that applications developed with it are real mobile applications comparable to true native applications

SOURCES

[1] StatCounter Global Stats, "Mobile Operating System Market Share Worldwide," [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide/. [Accessed 28. 10. 2018].

[2] SensorTower, "Global App Revenue Grew 35% in 2017 to nearly $60 Billion," [Online]. Available: https://sensortower.com/blog/app-revenue-and-downloads-2017. [Accessed 28. 10. 2018].

[3] BusinessWire, "With Expectations of a Positive Second Half of 2018 and Beyond, Smartphone Volumes Poised to Return to Growth, According to IDC," [Online]. Available: https://www.businesswire.com/news/home/20180829005143/en/Expectations-Positive-2018-Smartphone-Volumes-Poised-Return. [Accessed 6. 11. 2018].

[4] TechCrunch, "Apple's App Store revenue nearly double that of Google Play in first half of 2018," [Online]. Available: https://techcrunch.com/2018/07/16/apples-app-store-revenue-nearly-double-that-of-google-play-in-first-half-of-2018/. [Accessed 6. 11. 2018].

[5] Intel, "The Development of Mobile Applications using HTML5 and PhoneGap on Intel Architecture-Based Platforms," [Online]. Available: https://software.intel.com/en-us/articles/the-development-of-mobile-applications-using-html5-and-phonegap-on-intel-architecture-based-platforms#phonegap. [Accessed 6. 11. 2018].

[6] S.-H. Lim, "Experimental comparison of hybrid and native applications for mobile systems," *International Journal of Multimedia and Ubiquitous Engineering,* vol. 10, no. 3, pp. 1-12, 2015.

[7] The Apache Software Foundation, "Cordova Overview," [Online]. Available: https://cordova.apache.org/docs/en/latest/guide/overview/. [Accessed 28. 10. 2018].

[8] A. Khandeparkar, R. Gupta and B. Sindhya, "An Introduction to Hybrid Platform Mobile Application Development," *International Journal of Computer Applications,* vol. 118, no. 15, pp. 31-33, 2015.

[9] Stack Overflow, "Developer Survey Results 2018," [Online]. Available: https://insights.stackoverflow.com/survey/2018. [Accessed 11. 11. 2018].

[10] Facebook, "React Native Docs, Performance," [Online]. Available: https://facebook.github.io/react-native/docs/performance. [Accessed 11. 11. 2018].

[11] B. Eisenman, Learning React Native, USA: O'Reilly Media Inc., 2016.

[12] R. Soral, "React Native vs Ionic: Comparing performance, user experience and much more," [Online]. Available: https://www.simform.com/react-native-vs-ionic/. [Accessed 10. 28. 2018].

[13] Motorola, "Motorola Demonstrates Portable Telephone, Motorola Communications Division press release, April 1973," [Online]. Available: https://www.motorola.com/sites/default/files/library/us/about-motorola-history-milestones/pdfs/DynaTAC_newsrelease_73_001.pdf. [Accessed 11. 11. 2018].

[14] B. Fling, Mobile Design and Development, USA: O'Reilly Media Inc., 2009.

[15] New Scientist, *No. 1422*, pp. 42-43, September 1984.

[16] D. Pountain, "A Plethora of Portables," *Byte U.K.*, vol. 9, no. 12, pp. 413-420, 1984.

[17] C. O'Malley, "Simonizing the PDA," *Byte,* vol. 19, no. 12, pp. 145-148, 1994.

[18] J. Lumsden ja E. Koblentz, Human-Computer Interaction and Innovation in Handheld, Mobile and Wearable Technologies, UK: IGI Global, 2011.

[19] Bcos47, "The IBM Simon Personal Communicator and charging base," Wikimedia Commons, [Online]. Available: https://en.wikipedia.org/wiki/IBM_Simon#/media/File:IBM_Simon_Personal_Communicator.png. [Accessed 2. 12. 2018].

[20] A. H. Johnson, "WAP," *Computerworld,* vol. 33, no. 44, p. 69, 1999.

[21] B. Emmerson, "Is WAP wobbling?," *Communications International,* vol. 27, no. 2, p. 52, 2000.

[22] S. Morrissey and T. Campbell, iOS Forensic Analysis for iPhone, iPad, and iPod touch, USA: Apress, 2010.

[23] Apple Inc., "Apple Reports First Quarter Results," [Online]. Available: https://web.archive.org/web/20090701210028/http://www.apple.com:80/pr/library/2009/01/21results.html . [Accessed 6. 1. 2019].

[24] R. Fernandez, "A vector render of the 1st generation iPhone.," Wikimedia Commons, [Online]. Available: https://en.wikipedia.org/wiki/IPhone_(1st_generation)#/media/File:IPhone_1st_Gen.svg. [Accessed 6. 1. 2019].

[25] C. Bonnington, "5 Years On, the App Store Has Forever Changed the Face of Software," Wired, [Online]. Available: https://www.wired.com/2013/07/five-years-of-the-app-store/. [Accessed 1. 6. 2019].

[26] R. Gao, "Android and its first purchasable product, the T-Mobile G1, celebrate their 8th birthdays today," Android Police, [Online]. Available: https://www.androidpolice.com/2016/09/23/android-first-purchasable-product-t-mobile-g1-celebrate-8th-birthdays-today/. [Accessed 6. 1. 2019].

[27] A. NDE, "HTC Dream mobile phone with AZERTY keyboard for French market," Wikimedia Commons, [Online]. Available: https://commons.wikimedia.org/wiki/File:HTC_Dream_Orange_FR.jpeg. [Accessed 6. 1. 2019].

[28] J. Biggs, "Android to Get Its Own App Market," TechCrunch, [Online]. Available: https://techcrunch.com/2008/08/28/android-to-get-its-own-app-market/. [Accessed 6. 1. 2019].

[29] J. Nutting, F. Olsson, D. Mark and J. LaMarche, Beginning iOS 7 Development: Exploring the iOS SDK, USA: Apress, 2014.

[30] P. Kanoi ja I. Payal, ”History and Features of a developer tool in iOS: XCODE,” *International Journal of Advanced Research in Computer Science*, osa/vuosik. 4, nro 6, pp. 113-116, 2013.

[31] J. Jackson, "Apple unveils Swift, a new programming language for iOS, Mac," PCWorld, [Online]. Available: https://www.pcworld.com/article/2358541/apple-unveils-swift-a-new-programming-language-for-ios-mac.html. [Accessed 27. 1. 2019].

[32] Apple Inc, "iOS Technology Overview," 2014. [Online]. Available: https://docplayer.net/7123963-Ios-technology-overview.html. [Accessed 20. 1. 2019].

[33] Google, "Android Studio, System requirements," [Online]. Available: https://developer.android.com/studio#downloads. [Accessed 7. 5. 2019].

[34] Google, "Android Studio User Guide," 2019. [Online]. Available: https://developer.android.com/studio/intro/ . [Accessed 20. 2. 2019].

[35] Google, "Platform Architecture," 2019. [Online]. Available: https://developer.android.com/guide/platform. [Accessed 10. 2. 2019].

[36] Google, "System and kernel security," 2019. [Online]. Available: https://developer.android.com/guide/platform. [Accessed 19. 2. 2019].

[37] Facebook, "React Native," [Online]. Available: https://facebook.github.io/react-native/. [Accessed 5. 5. 2019].

[38] Facebook, "JavaScript Environment," [Online]. Available: https://facebook.github.io/react-native/docs/javascript-environment. [Accessed 5. 5. 2019].

[39] F. Zammetti, Practical React Native: Build Two Full Projects and One Full Game using React Native, USA: Apress, 2018.

[40] GitHub, "React Native," [Online]. Available: https://github.com/facebook/react-native. [Accessed 6. 5. 2019].

[41] Microsoft, "Introduction to mobile development," [Online]. Available: https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/introduction-to-mobile-development. [Accessed 6. 5. 2019].

[42] L. O'Brien, C. Dunn and B. Umbaugh, "Xamarin.Mac ahead of time compilation," [Online]. Available: https://docs.microsoft.com/en-us/xamarin/mac/internals/aot. [Accessed 6. 5. 2019].

[43] C. Dunn, D. Britch, M. Koudelka, T. Opgenorth, M. de Los Santos, M. McLemore and F. Eilertsen, "Architecture," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/xamarin/android/internals/architecture. [Accessed 6. 5. 2019].

[44] GitHub, "Xamarin," [Online]. Available: https://github.com/xamarin. [Accessed 6. 5. 2019].

[45] J. Allen, "The Death and Rebirth of Mono," InfoQ, [Online]. Available: https://www.infoq.com/news/2011/05/Mono-II. [Accessed 6. 5. 2019].

[46] Progress Software Corporation, "How NativeScript Works," [Online]. Available: https://docs.nativescript.org/angular/core-concepts/technical-overview. [Accessed 6. 5. 2019].

[47] Progress Software Company, "What is iOS Runtime for NativeScript," [Online]. Available: https://docs.nativescript.org/core-concepts/ios-runtime/Overview. [Accessed 6. 5. 2019].

[48] GitHub, "NativeScript," [Online]. Available: https://github.com/NativeScript/NativeScript. [Accessed 6. 5. 2019].

[49] V. Stoychev, "NativeScript First Public Beta Release is Now Available!," [Online]. Available: https://www.nativescript.org/blog/nativescript-first-public-release. [Accessed 6. 5. 2019].

[50] Drifty, "All about Ionic," [Online]. Available: https://ionicframework.com/docs/v1/guide/preface.html. [Accessed 6. 5. 2019].

[51] CruxLab, "Xamarin vs Ionic vs React Native: differences under the hood," [Online]. Available: https://cruxlab.com/blog/reactnative-vs-xamarin/. [Accessed 6. 5. 2019].

[52] GitHub, "Ionic," [Online]. Available: https://github.com/ionic-team/ionic. [Accessed 6. 5. 2019].

[53] Drifty, "Ionic Core Concepts," [Online]. Available: https://ionicframework.com/docs/v3/intro/concepts/. [Accessed 6. 5. 2019].

[54] Google, "Technical Overview," [Online]. Available: https://flutter.dev/docs/resources/technical-overview. [Accessed 6. 5. 2019].

[55] Google, "Flutter FAQ," [Online]. Available: https://flutter.dev/docs/resources/faq. [Accessed 6. 5. 2019].

[56] Google, "Flutter System Architecture," [Online]. Available: https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/edit#slide=id.p. [Accessed 6. 5. 2019].

[57] GitHub, "Flutter," [Online]. Available: https://github.com/flutter/flutter. [Accessed 6. 5. 2019].

[58] Google, "Flutter releases," GitHub, [Online]. Available: https://github.com/flutter/flutter/releases/. [Accessed 6. 5. 2019].

[59] S. Oaks, Java Performance: The Definitive Guide, USA: O'Reilly Media, Inc, 2014.

[60] Facebook, "Debugging," [Online]. Available: https://facebook.github.io/react-native/docs/debugging. [Accessed 3. 3. 2019].

[61] Facebook, "Performance," [Online]. Available: https://facebook.github.io/react-native/docs/performance. [Accessed 3. 3. 2019].

[62] R. Sharma, "Role of Metro Bundler in React Native," [Online]. Available: https://medium.com/@rishabh0297/role-of-metro-bundler-in-react-native-24d178c7117e. [Accessed 3. 3. 2019].

[63] W. Miller and E. W. Myers, "A File Comparison Program," *Software- Practice and Experience*, vol. 15, no. 11, pp. 1025-1040, 1985.

[64] E. Rozell, "Creating Universal Windows Apps with React Native," Microsoft, 2016. [Online]. Available: https://www.microsoft.com/developerblog/2016/05/26/creating-universal-windows-apps-with-react-native/. [Accessed 20. 3. 2019].

[65] Facebook, "Components and APIs," [Online]. Available: https://facebook.github.io/react-native/docs/components-and-apis.html. [Accessed 20. 3. 2019].

[66] GitHub, "React Native releases," [Online]. Available: https://github.com/facebook/react-native/releases. [Accessed 22. 5. 2019].

[67] L. Sciandra, "The State of the React Native Community in 2018," [Online]. Available: https://facebook.github.io/react-native/blog/2019/01/07/state-of-react-native-community. [Accessed 22- 5- 2019].

[68] H. Ramos, "Open Source Roadmap," [Online]. Available: https://facebook.github.io/react-native/blog/2018/11/01/oss-roadmap. [Accessed 22. 5. 2019].

[69] Google, "Google Trends for React Native," [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=React%20Native. [Accessed 22. 5. 2019].

[70] Facebook, "Getting Started," [Online]. Available: https://facebook.github.io/react-native/docs/getting-started. [Accessed 2. 4. 2019].

[71] Facebook, "Navigating Between Screens," [Online]. Available: https://facebook.github.io/react-native/docs/navigation. [Accessed 15. 4. 2019].

[72] React Navigation, "API Reference," [Online]. Available: https://reactnavigation.org/docs/en/api-reference.html . [Accessed 19. 4. 2019].

[73] Facebook, "AsyncStorage," [Online]. Available: https://facebook.github.io/react-native/docs/asyncstorage. [Accessed 2. 5. 2019].

[74] P. Podila and M. Weststrate, MobX Quick Start Quide, UK: Packt Publishing Ltd, 2018.

[75] Facebook, "Color Reference," [Online]. Available: https://facebook.github.io/react-native/docs/colors. [Accessed 9. 5. 2019].