COMPARISON OF REACT NATIVE AND EXPO

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation, Master's thesis

2023

Hugo Hutri

Examiner(s): Associate Professor Jussi Kasurinen

Teemu Taskula M.Sc. (Tech.)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Hugo Hutri

Comparison of React Native and Expo

Master’s thesis

2023

57 pages, 20 figures, 5 tables and 0 appendices

Examiner(s): Associate Professor Jussi Kasurinen and Teemu Taskula M.Sc. (Tech.)

Keywords: cross-platform, mobile development, React Native, Expo, developer experience

Cross-platform mobile development frameworks are a popular way to create mobile apps these days. Two popular options are React Native and Expo, which offer slightly different ways to create Android and iOS apps using React. Expo provides an abstraction layer on top of React Native, enabling a better developer experience and simpler process.

This work compares the features, advantages and disadvantages of popular cross-platform mobile development frameworks React Native and Expo. Their abilities and suitability for various projects are evaluated and it is determined whether Expo can be considered a better option in certain situations. The results of the work show that although Expo offers many advantages, its suitability for a specific project depends on the specific requirements and needs of the project, but today the features it offers are sufficient and more straightforward for many projects.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT School of Engineering Science

Tietotekniikka

Hugo Hutri

React Nativen ja Expon vertailu

Diplomityö

2023

57 sivua, 20 kuvaa, 5 taulukkoa ja 0 liitettä

Tarkastaja(t): Apulaisprofessori Jussi Kasurinen ja Teemu Taskula M.Sc. (Tech.)

Avainsanat: alustariippumattomuus, mobiilikehitys, React Native, Expo, kehittäjäkokemus

Alustariippumattomat mobiilikehityskehykset ovat tänäpäivänä suosittu tapa luoda mobiilisovelluksia. Kaksi suosittua vaihtoehtoa ovat React Native ja Expo, jotka tarjoavat hieman erilaiset tavat luoda Android ja iOS sovelluksia Reactin avulla. Expo tarjoaa abstractiokerroksen React Nativen päälle, mikä mahdollistaa paremman kehittäjäkokemuksen ja yksinkertaisemman prosessin.

Tässä työssä verrataan suosittujen alustariippumattomien mobiilikehityskehysten React Nativen ja Expon ominaisuuksia, etuja ja haittoja. Niiden kykyjä ja soveltuvuutta erilaisiin projekteihin arvioidaan ja selvitetään, että voidaanko Expoa pitää parempana vaihtoehtona tietyissä tilanteissa. Työn tulokset osoittavat, että vaikka Expo tarjoaa monia etuja, sen soveltuvuus tiettyyn projektiin riippuu projektin erityisvaatimuksista ja tarpeista. Expo tarjoaa kuitenkin nykypäivänä riittävät ominaisuudet useisiin projekteihin ja yksinkertaisemman kehittäjäkokemuksen verrattuna React Nativeen.

ACKNOWLEDGEMENTS

I am very grateful to Taito United Oy for giving me the opportunity to work on a really interesting topic for my thesis and for their help and support all along. I also would like to thank my supervisors Jussi Kasurinen and Teemu Taskula for their feedback and mentoring through the process.

SYMBOLS AND ABBREVIATIONS

Abbreviations

API	Application Programming Interface
CI/CD	Continuous Integration and Continous Delivery
CLI	Command-line Interface
EAS	Expo Application Services
ECMAScript	European Computer Manufacturers Association Script
GPS	Global Positioning System
iOS	iPhone Operating System
JS	JavaScript
Npm	Node Package Manager
OS	Operating System
OTA	Over-the-air
SDK	Software Development Kit
SVG	Scalable Vector Graphics
TS	TypeScript
XML	Extensible Markup Language

Table of contents

Figures

Tables

1 Introduction

Smartphone apps have been gaining increasing popularity over the years, with Android and iOS being the two dominant mobile operating systems. Traditionally, developers have been using native tools to create applications for these platforms. For Android, the native tools have been Android Studio and applications could be written using languages such as Kotlin, Java, and C++ (Google, no date). For iOS, the tools have been Xcode and applications could be written using languages such as Objective-C, Swift, and others (Sinicki, 2016). However, nowadays many developers seek to develop cross-platform applications, and this thesis conducts a developer experience comparison of two popular frameworks: React Native and Expo.

1.1 Background

Developing one application and then running it on both operating systems seems tempting, and that is what many frameworks try to accomplish. Developing one app instead of two might also make a lot of sense from the business perspective since one might be able to develop one cross-platform application much quicker and cheaper than two completely native applications, where the work done in one project cannot be used in the other (Miquido, 2022). With some frameworks, like React Native, web developers can easily transfer their knowledge from web development with React and JavaScript to mobile development with React Native. This is also one huge driving factor that pushes attention toward cross-platform options.

Cross-platform mobile development frameworks have become increasingly popular in recent years, with many developers choosing to use these frameworks to build their applications. Some of the most widely used frameworks include React Native, Flutter, Xamarin, Cordova, and NativeScript (*Cross-platform mobile frameworks used by global developers 2021*, no date). Each of these frameworks offers its own unique set of features and benefits and can be used to create a wide variety of applications. In this work, the focus will be on React Native and Expo, two popular frameworks that have gained significant traction in the development community.

One of the most widely used cross-platform tools is React Native which is going to be the focused framework in this work. Another tool that has been gaining popularity is Expo, which is a framework to build React Native apps, and it can be thought of as an abstraction layer on top of React Native. React Native and Expo are both popular cross-platform mobile development frameworks. React Native allows developers to build native apps using JavaScript and React components, while Expo provides a toolchain that simplifies the creation and distribution of cross-platform applications. Expo also offers a platform-neutral API (Application Programming Interface) that eliminates the need for developers to write platform-specific code.

1.2 Motivation and goals

The goal of the thesis is to find out the differences between React Native and Expo, and how well they can adapt and solve the needs of the industry in 2023. Later on, this thesis will provide comparisons on how well, and to which extent, can Expo handle tasks that can be done using React Native. Advantages that each framework has will be compared, and if it would be wise to start developing new projects with Expo instead of React Native.

The motivation behind the work is to find a solution to whether or not to switch from React Native to Expo in new projects, and if it makes sense to convert old projects to Expo. This thesis will also help to understand the benefits of both approaches in 2023 since both of them have been evolving a lot during recent years, especially Expo. One of the main problems with Expo compared to React Native has been that it has a more limited set of features and capabilities, as was described in a blog post by Yura Kruhlyk in 2018 (Kruhlyk, 2018). That article was written in 2018, and since then Expo has had a lot of time to evolve. As mentioned earlier, Expo simplifies the development process, but according to Borozenets, it does not provide access to native modules made with Java and Objective-C same way as React Native (Borozenets, 2022), therefore some features can be more difficult to implement. However, Expo recently introduced a new configuration system called Expo Config Plugins (*Config Plugins*, no date), which allows developers to add custom native modules and components to their Expo apps and expand their capabilities. This new feature has helped to address some of the limitations of Expo and makes it a more viable option for developers who need access to more advanced features and capabilities.

This thesis will be answering to the following research questions:

1. What are the biggest limitations and benefits of moving from React Native to Expo?
2. How is the developer experience with Expo compared to React Native?

1.3 Scope and limitations

The scope of this thesis is limited to comparing the developer experience and limitations of React Native and Expo when developing cross-platform mobile applications.

This study has several limitations. First, the results of this study may not be generalizable to all developers or all applications, as the specific skills and experience of the developers, as well as the characteristics of the application, may affect the results. Second, this study only considers two specific frameworks, React Native and Expo, and does not evaluate other cross-platform mobile development frameworks. Finally, this study only focuses on two specific aspects of cross-platform mobile development: the effort required to develop an application, and the developer experience of the process. Other factors, such as cost, maintainability, and scalability, may also be relevant and should be considered in future research.

1.4 Structure of the thesis

Chapter 1 contains the introduction and the goals of the topic. Chapter 2 presents the history of mobile application development and introduces React Native and Expo in more detail. Literature review and other related work will be explored in chapter 3 as well as other comparisons between mobile development frameworks. Other studies that have compared different software technologies, such as programming languages, will also be reviewed. The findings of these studies will be considered in the context of our analysis. In chapter 4, this thesis will focus on explaining the research methods and findings, such as feature implementations on both frameworks. Results of the comparisons will be presented in chapter 5. The discussion of the comparisons of these two mobile development frameworks will be presented in chapter 6, as well as the implications of the findings for developers in the mobile app industry, and directions for future research in this area will be suggested. The last part of the thesis, chapter 7, conclude and summarise the thesis.

2 Mobile application development

In this chapter, an overview of the history and current state of mobile application development, with a focus on cross-platform frameworks, will be provided. In the first part, the rise of Android and iOS as dominant mobile operating systems will be discussed, and then the development of cross-platform frameworks like React Native and Expo. Lastly, different ways to manage an Expo project will be presented and compared.

2.1 History and current state of mobile application development

Android and iOS are currently the most popular mobile operating systems. Both of them have their own ways of developing mobile applications. Apple released iPhones in 2007 (Snell, 2022) and Android was released slightly after that during the same year (Chitu, 2007). These two have been dominating the market since, and they both have had their own tools for developing the applications.

Cross-platform mobile development frameworks were created in response to the rising popularity of mobile devices and the need for more effective methods of creating applications that can operate on several platforms. These frameworks enable programmers to produce applications with a single codebase that functions on both iOS and Android (*What is cross-platform mobile development? | Kotlin*, no date). React Native, Flutter, and Xamarin are some of the frameworks that were first used. These frameworks have developed and grown throughout time, giving programmers more potent tools to build complex, feature-rich apps. As they offer a way to reach a larger audience without having to maintain different codebases for each platform, cross-platform mobile development frameworks are a common choice for many developers today.

2.1.1 React Native

React Native is a widely known example of a cross-platform mobile development framework and according to Statista around 40% of developers have been using it during the last three years (*Cross-platform mobile frameworks used by global developers 2021*, no date). Facebook first made React Native available to developers in 2015, and it enables them to create native apps that can run on both Android and iOS with a single codebase (Occhino, 2015). The framework enables developers to create apps utilizing well-known web technologies like JavaScript and React components because it is built on top of React, a prominent JavaScript toolkit for creating user interfaces. React Native has become increasingly well-liked over time, and many businesses, including Meta, Microsoft, Shopify, and Discord, now use it to create their mobile applications (*Showcase · React Native*, no date). The developer community has also rapidly accepted the framework, and there are many third-party libraries and tools available to expand its functionalities. In fact, React Native Directory currently lists 1178 packages for React Native (*React Native Directory*, no date).

2.1.2 Expo

Expo is a toolchain built around React Native that streamlines the creation and distribution of cross-platform software. In addition to a managed build environment, and tools for testing and debugging, Expo offers a variety of tools and services that may be used to develop, build, and publish React Native applications. Additionally, Expo offers a platform-neutral API that frees developers from having to write platform-specific code in order to access native device functions like the camera, GPS (Global Positioning System), and push notifications (*API Reference*, no date). The ability to build features and functionality into their applications rather than worrying about platform-specific aspects makes it simpler for developers to create apps that can work on both Android and iOS. Based on the Expo documentation, Expo introduces some features that bare React Native does not have, which can be seen in Table 1 below.

Table 1: List of features that Expo provides but React Native does not (What is Expo, no date)

Feature	**With expo**	**Without expo (bare React Native)**
Develop complex apps entirely in JavaScript.	Yes	No
Write JSI native modules with Swift & Kotlin.	Yes	No
Develop apps without Xcode or Android Studio.	Yes	No
Create and share example apps in the browser with [Snack][build/introduction/].	Yes	No
Major upgrades without native changes.	Yes	No
First-class TypeScript support.	Yes	No
Install natively compatible libraries from the command line.	Yes	No
Develop performant websites with the same codebase.	Yes	No
[Tunnel][/workflow/expo-cli/#tunneling] your dev server to any device.	Yes	No

From Table 1 it can be seen that Expo has provided multiple useful features for developers to use. However, it should be kept in mind that the comparison was provided by Expo (*What is Expo*, no date), and it can be biased towards Expo and it is only designed to highlight its advantages over bare React Native. Some of the points in the table could be argued against, for example, the statement that bare React Native does not have first-class TypeScript support. That statement is no longer true, because starting with React Native version 0.71, React Native will have TypeScript as the default (*First-class Support for TypeScript · React Native*, 2023). Besides that, the table gives a clear list of interesting features, which makes Expo very appealing. Developing the application entirely in JavaScript and without Xcode and Android studio can make the development process compelling, since eliminating or reducing platform-specific code has been the main ideology behind cross-platform development frameworks, and Expo clearly embraces that.

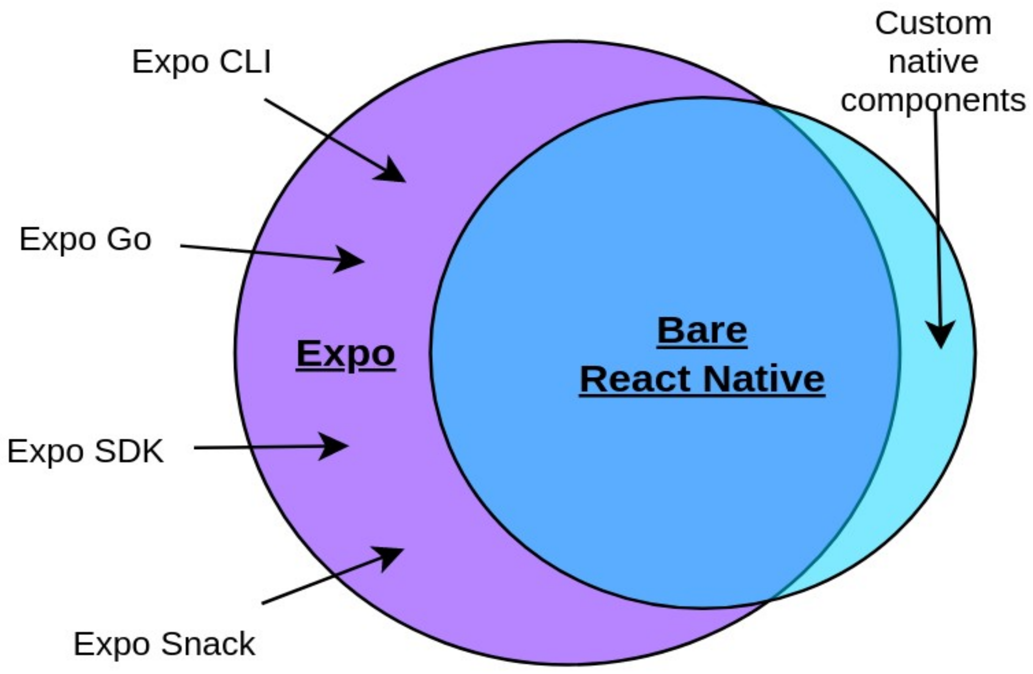


Figure 1. Venn diagram on how Expo and React Native are related.

However, the relationship between the two frameworks is often misunderstood, with some developers assuming that they are interchangeable or that Expo is simply a version of React Native. In reality, React Native is a framework for building fully native mobile applications using JavaScript and native mobile components, while Expo is a toolchain that makes it easier to develop, build, and distribute React Native applications (*React Native · Learn once, write anywhere*, no date; *What is Expo*, no date). The relationship between the frameworks is illustrated in Figure 1, which shows how React Native and Expo have a common middle ground, which is most of the React Native framework. The figure shows also how Expo's ecosystem has its own tools outside React Native, such as Expo Go and Expo CLI (Command Line Interface). Even though Spencer Carli described that "Expo lives as a superset of React Native" in his LogRocket article (Carli, 2021), it might not actually be fully true, since React Native has direct access to platform-specific code. Therefore the figure above was created to describe the relationship between the two and to illustrate some of the tools and services that differentiate them.

While the two frameworks are closely related, they serve different purposes and have distinct features and benefits that developers should consider when choosing a framework for their cross-platform mobile development projects.

2.2 Managing Expo

In order to gain a better understanding of how Expo allows developers to manage the apps, three different ways will be presented: Fully managed, custom development client, and Expo prebuild. This subchapter will introduce these three ways and highlight the differences between the two.

2.2.1 Fully managed Expo

Expo fully managed app is the simplest way to develop an Expo application, since it requires developers to write just TypeScript or Javascript and a JSON (JavaScrip Object Notation) configuration file, and Expo is able to handle the rest. For these managed applications Expo provides a runtime, which runs inside the Expo Go mobile application and consists of ECMAScript (European Computer Manufacturers Association Script) Standard Library, React Native, and Expo SDK (Software Development Kit). This allows the developers to use Expo’s runtime and the Expo Go app, where the application can be previewed easily. (Vatne, 2021) This is the default way when a new Expo project is created, and it can be extremely easy for developers to use because they do not need to worry about the native code or native modules.

2.2.2 Custom development client

Another way to develop Expo applications is to use a custom development client, which means that a custom version of the Expo Go application is created with additional functionality. With a custom development client, the team would get similar benefits as with Expo Go, while having native features that are not part of the default Expo Go client and it allows to go “beyond the standard runtime provided in Expo Go” as described in the

Expo article by Davis (Davis, 2022). Expo development clients were introduced in the article by Davis in 2021, and they were already in use in teams like Brex and Valve. This way of developing Expo applications might be more relevant for software product companies since they could have one team for the client application and another for the actual application. However, this might not be suitable for smaller teams, and the next option could be a more appealing alternative.

2.2.3 Prebuild

Prebuild is the third option, where Expo prebuild generates the native code for the project before the app is compiled as explained in Expo documentation (Prebuild, no date). For example with Android, the prebuild can be run before the “android” command, like “npx expo prebuild && npx expo run:android” which generates the android and ios directories and then runs the Android app. With prebuild, developers can use config plugins to customize the directories and files containing native code for both iOS and Android. These folders cannot usually be accessed with the fully managed solution, because it relies on the API:s provided by Expo.

Expo prebuild documentation recommends creating config plugins which are a core part of the prebuild workflow. According to Expo Config Plugins documentation (*Config Plugins*, no date) plugins can configure, generate and edit native code in the project, such as AndroidManifest.xml and Info.plist files, etc, to customize the behavior of the application. With config plugins, developers can create native and platform-specific functionalities that are not implemented in the Expo ecosystem yet.

3 Related work

In this chapter, previous research on the subject will be reviewed, including a comparison of native and cross-platform development methods. First, a study that compared the experience, performance, and other qualities of different mobile development approaches will be reviewed, including native and cross-platform methods. Other studies that have compared subjective measures, such as comparisons of programming languages will also be considered. The methods used in these studies will be introduced and the results of their analyses will be compared.

3.1 Comparisons of mobile development platforms

Nawrocki, P et al. Compared both native ways and several cross-platform frameworks for mobile application development in their article written in 2021. In the article, they compared, for example, ease of development and some performance-related metrics like application size, start-up time, and memory usage. Compared frameworks and implementations were: native iOS, native Android, React Native, Flutter, and Xamarin. (Nawrocki *et al.*, 2021)

The developer experience, which is one of the main focuses of this thesis, was mentioned briefly in the Nawrockis research. They used the Stack Overflow Developers survey, which gave some insight and data on developer preferences, even though this method was said to be subjective. For the developer experience, they analyzed the following areas: quality of IDEs and tools, feedback cycle length, quality of libraries, and availability of features in the framework. Their study showed that React Native was creating most of the developer experience problems when compared to other frameworks. The study did not include Expo, so it can be assumed to be part of React Native in this context, so the research does not provide enough information to determine whether these problems would go away or become more challenging when using Expo. That is also one more reason to study these two frameworks, so the difference in developer experience and in ease of development can be shown.

The second study compared React Native and Flutter too. The research was conducted by Ekrem Gülcüoğlu et al. in 2021 and it compared the popularity of both tools over time as well as the syntax of both approaches (Gülcüoğlu, Ustun and Seyhan, 2021). The study presented an implementation of different features of programming languages and mobile application frameworks in order to determine their effectiveness and efficiency. The study compared the performance, user interface, and testing capabilities of the two platforms.

The main thing found when looking at these studies was that most of the time the mobile development frameworks are measured by the end result, and not so much by how the process of developing a mobile app goes from the developers perspective. Although some studies did consider the developer experience or the ease of development, it was usually a rather small part of the research. This might be because most of the time companies and organizations might want to focus on the product itself, and the developer experience is not the main interest. This literature review on the comparison of mobile application frameworks indicates that there might not be enough scientific research on true developer experience comparison of these technologies.

3.2 Other comparisons

All of the other studies were about comparing mobile development platforms and frameworks, but those were mostly comparing quantitative characteristics of the platforms or frameworks. One context where more qualitative comparisons can be found is the research of the best programming language for a given situation.

3.2.1 Comparison of good first programming languages

One of the studies comparing programming languages is the article “What is a Good First Programming Language?” written by Diwaker Gupta. In the article, he compared multiple programming languages and what it would be like to learn them. Characteristics he considered in the work were for example simplicity, orthogonality, regularity, turnaround time, and debugging support. (Gupta, 2004)

Simplicity considered how intuitive a programming language was to understand for beginners, and the study stated that functional languages were harder to read and that most procedural programming languages were easier to understand since they followed the natural semantics of control flow. The second example, orthogonality, meant that a language should have only a few ways to do one thing since a beginner might be intimidated by the options. Orthogonality can be considered in this thesis too since both mobile development frameworks might offer a way to do one task in multiple ways. For example, Expo might offer a way to implement a feature by using config plugins, by using a package provided by Expo, or by using a third-party package for the task. One question arising from this topic is how many ways to do a task is too many.

In this work, simplicity can be considered when comparing React Native and Expo, and it will be one of the main factors for the final decision if both frameworks seem to be capable of accomplishing the same results. Other than that, the study conducted by Diwaker Gupta had a quite different target, since it wanted to give insight on the best first programming language. However, it is not the objective of this thesis to target the mobile frameworks toward new programmers, and the requirements will differ significantly since it can be assumed that the developers are already familiar with React or similar development approaches.

3.2.2 Comparison of quality in JavaScript and TypeScript projects

Because React Native and Expo are very similar in nature it can be quite tricky to compare these two. Expo could be thought of as a superset of React Native, which has not been the case with the other technologies mentioned in previously mentioned research articles. When keeping that in mind, some possible comparisons arise: comparing JavaScript and Typescript, and comparing C and C++. A study conducted by Justus Bogner and Manual Merkel (Bogner and Merkel, 2022) compared TypeScript and JavaScript and how they affect the software quality. In their study, they conducted a repository mining study based on 604 GitHub projects. They analyzed the repositories and over 16 million lines of code (LoC) and then collected the following results from the analysis as mentioned in their report:

a) code quality (# of code smells per LoC)
b) code understandability (cognitive complexity per LoC)
c) bug proneness (bug fix commit ratio)
d) bug resolution time (mean time a bug issue is open)

Repository mining was the main methodology behind this research, and that could definitely be one approach that could be used in comparing React Native and Expo too. Since Expo and React Native share so much in common, as do TypeScript and JavaScript, the comparison of quality, understandability, bug proneness and bug resolution time could give insight into how the developer experience is since better quality code would probably make the developer experience better when compared to worse quality.

In the article, Bogner and Merkel explored two research questions, which basically were about if TypeScript applications have better quality than JavaScript applications, and if usage of the "any" type affects the TypeScript code quality in a positive or negative way. Based on the TypeScript documentation (*Handbook - Basic Types*, no date), any type is a way to opt-out from type checking and the type can be anything. In the article, Bogner and Merkel explored how the any type affected the quality but is not directly comparable to this thesis and the direct comparison between JavaScript and TypeScript is more relevant.

Bogner and Merkel measured code quality by counting code smells, which are, according to Yamashita, indicators of bad code quality and can decrease software maintainability (Yamashita, 2013). They used a static analysis tool to count the number of code smells on both platforms, and then calculated the number of code smells per line of code. In the context of React Native and Expo, the code smells might not be that important, since the frameworks only provide ways to do tasks, and it is not directly related to patterns in the code.

Code understandability was the second point in the article, and it was measured with cognitive complexity. Yamashita produced also results as cognitive complexity per line of code. This could possibly be measured in mobile development repositories too, since the frameworks underneath, such as React Native or Expo, have the possibility to affect the code understandability. Some packages might make the program easy to read, whereas missing access to native files could make the solutions quite difficult to understand.

The third part was bug proneness, which was measured by looking at how frequent bugs and fixes were in the repository. This was done by analyzing the commit messages, and counting messages containing “bug” and “fix”. With this data extraction method, they were able to calculate bug fix commit ratio for every project. This part, as well as code understandability, is something that is worth considering when comparing mobile platforms. Bug proneness is an important topic to measure in the future when mobile developers have had more time to develop Expo projects with modern features. After that, the bug fix commit ratio could be calculated and some insight possibly gained.

The last consideration in Bogner’s and Merkel’s article was bug resolution time. It was defined as the “mean duration from bug issue opening until the last issue comment“, where the bug issues were retrieved from GitHub, and the issues containing the word “bug” in a label, title, or in the description were counted. This was as well interesting idea for studying the software quality, and it could be taken into account in mobile framework comparison as well.

Overall, at least code understandability, bug proneness, and bug resolution time could potentially be valuable metrics when comparing similar mobile development platforms. The study compared TypeScript and JavaScript, which are very similar in nature, just like Expo and React Native, and that could mean that the metrics used in the study could be useful in a comparison like this thesis. However, that kind of study requires a lot of data, and this study is made more like a case study, which means that the metrics are not directly applicable to a comparison between Expo and React Native.

4 Methods and features to be implemented

This chapter will introduce the features that will be implemented using both frameworks and present the ways how the features were implemented, or how they could be implemented. The selected features are very commonly used, and they will be necessary for many applications and therefore exploring how they can be implemented is important. This chapter will focus on how the implementations differ and compare against each other. In the last subchapter, the current trend with Expo plugins and packages will be explored.

4.1 Features

The list of features was selected based on the company's needs. These features are selected since they are very common features that new and future applications might require, and the company will probably need most of them when developing a new mobile application. In order to provide a comprehensive comparison of React Native and Expo, features that are widely used and require a variety of techniques for implementation were chosen. These features can be seen in Table 2 below.

Table 2: List of common features of mobile applications

Splash screen and app icons	Adding customized slash screen and app icons
Localization	Adding support for translating texts, dates, and other localizable fields
Device info	Getting relevant device and application information, such as application version, device brand, and type
Network info	Getting the state of the device's network connection
Permissions	Defining, checking and requesting permissions from the OS
Publishing	The process of publishing the application to application stores
Environment variables /	Using different variables based on the current environment,

Config variables	such as development and production environments
Routing	How the navigation is handled in the application
Over the air updates	Providing over-the-air updates, such as typo fixes, to the applications

This thesis will explore how these features can be implemented using both React Native and Expo, and compare the approaches taken by each framework. By implementing these features, better understanding of the capabilities of each framework and their suitability for different types of applications will be gained. The developer experience when using each framework will also be evaluated, and the ease of use as well as the availability of documentation and support will be compared. Additionally, any limitations or challenges that may arise when using React Native and Expo to implement these features will be identified.

Some of these features also might require some amount of editing of the native files, which are in “android/” and “ios/” folders when developing with React Native. The “ios” folder contains iOS specific native code, and the “android” folder contains Android-specific code, and React Native requires Xcode and/or Android Studio to develop for those platforms (*Setting up the development environment · React Native*, no date). For example, the android folder contains a “manifest” folder for AndroidManifest.xml, a “java” folder for Java source code, a “res” folder for resources like icons, and a gradle scripts folder for configuration (Hanif, 2020).

However, with Expo, these folders are not directly accessible, but one workaround would be to use config plugins. Config plugins allow developers to add custom native code to their Expo projects without having to eject from the Expo environment. This enables developers to customize their applications and access native functionality without losing the benefits of using Expo. Additionally, config plugins can be used to modify the behavior of Expo's APIs, allowing developers to further customize their applications to meet their specific needs. Overall, config plugins provide a way for developers to extend the capabilities of Expo while still being able to take advantage of its streamlined development process. Based on the features presented later in this chapter, it can be examined if the complexity of either of these frameworks causes too much complexity.

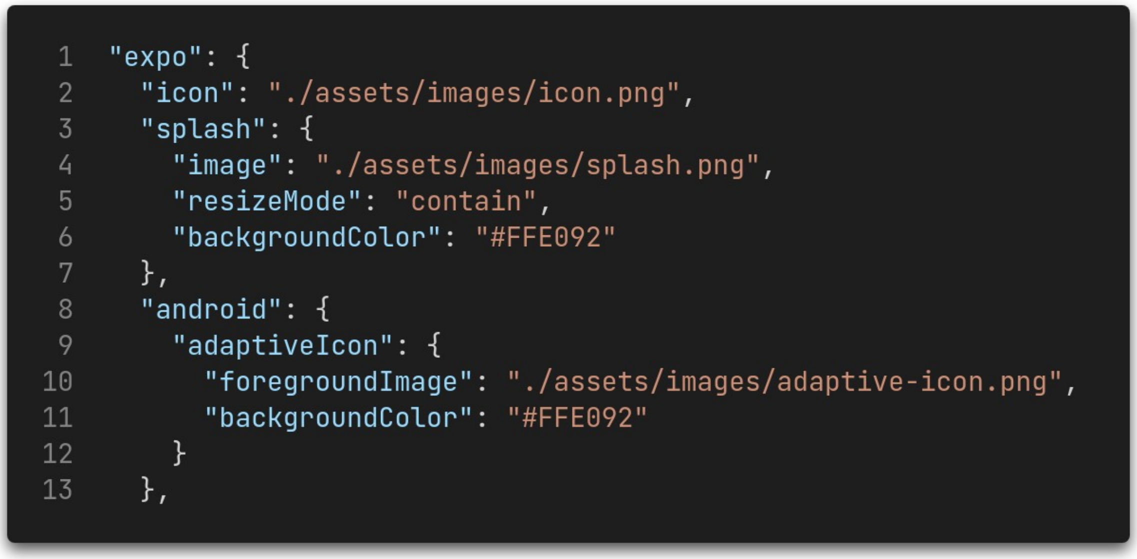
The implementation of each feature will be evaluated using a set of metrics that take into account various factors, such as the speed of implementation, the clarity and readability of the solution, and the overall developer experience. These metrics will provide a comprehensive assessment of the effectiveness of each feature, allowing us to compare and contrast the performance of React Native and Expo in implementing these features. Additionally, any additional challenges or limitations that may arise when implementing these features will be considered, such as the need for editing native files or the use of config plugins. By taking all of these factors into account, a better understanding of the strengths and weaknesses of each framework in implementing common features will be gained. In other words, the measured features are:

- How fast it was to implement?
- How clean and readable the solution is?
- How was the developer experience?

4.1.1 Splash screen and app icons

A splash screen and an app icon are important visual elements of a mobile application. The splash screen is the initial screen that is displayed when an app is launched, and it typically displays the app's logo or name.

A splash screen is the view that will be shown to the user when the app is opening, but has not yet fully loaded. App icons are platform-specific icons that will be usually displayed in the application menu or on the home screen, but Android and iOS both require the icon in different formats, so both platforms will need their own files.



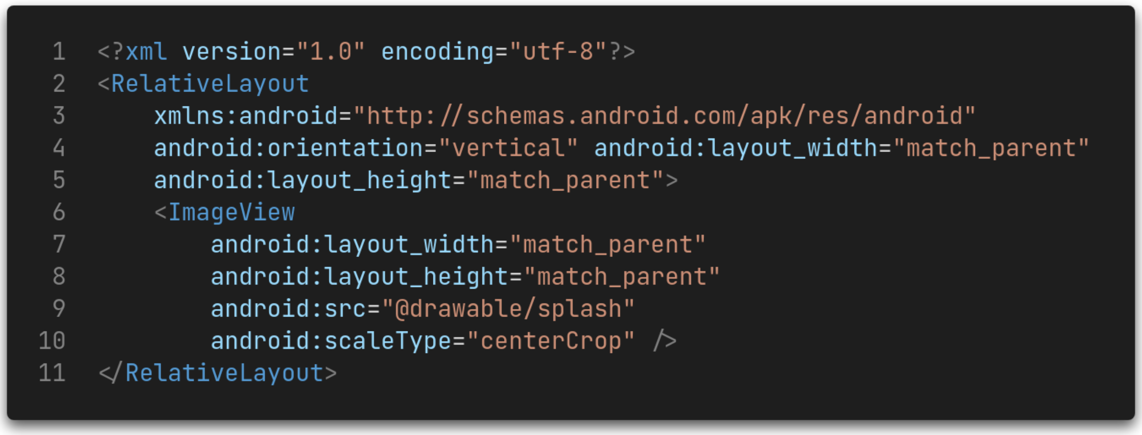
```
"expo": {
  "icon": "./assets/images/icon.png",
  "splash": {
    "image": "./assets/images/splash.png",
    "resizeMode": "contain",
    "backgroundColor": "#FFE092"
  },
  "android": {
    "adaptiveIcon": {
      "foregroundImage": "./assets/images/adaptive-icon.png",
      "backgroundColor": "#FFE092"
    }
  },
```

Figure 2. The app config for a splash screen and app icons in Expo

Adding the splash screen and app icons can be categorized under the same task since they have very similar processes. Both of these depend on the platform used, but Expo made that task very easy. Adding the splash screen and the icons took three (3) steps:

1. Adding the files in the correct folder, which was “assets/images/” for this project.
2. Editing the “app.json” file, by changing the default paths as shown in Figure 2.
3. Reloading the app

For React Native, the splash screen was much a more complicated task, since it was mostly done by directly modifying android and iOS files. Adding the splash screen for Android required multiple steps from editing native files to installing extra packages. There were multiple different guides found online, but all of them were a little different, which makes the developer experience more challenging. If a developer has not added a splash screen before or does not have experience with native mobile development, this will be a relatively harder task, compared to the implementation in Expo.



```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/splash"
        android:scaleType="centerCrop" />
</RelativeLayout>
```

Figure 3. launch_screen.xml for the splash screen in Android

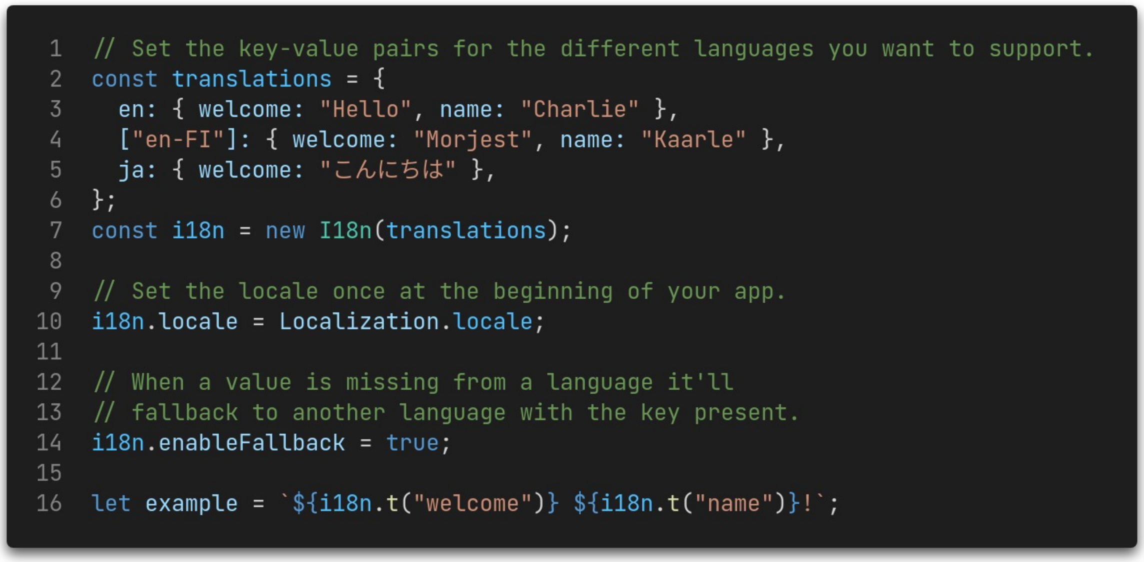
1. Adding the splash screen files in the correct folder, which was “android/app/src/main/res/drawable” for this project.
2. Editing colors.xml, styles.xml, and launch_screen.xml as shown in Figure 3.
3. Editing MainActivity.java
4. Installing “react-native-splash-screen”
5. Hiding the splash screen in React
6. Reloading the app

The steps were only for Android and other iOS-specific steps need to be taken to get the splash screen working on iOS. These steps were based on a guide made by LogRocket (Etukudo, 2022).

Overall, Expo was a clear winner on this task, since it was much easier and quicker to implement. React Native did not really offer much help, and the implementation was made mostly with native files. One advantage of React Native was that if Expo implementation was not enough, the native files could offer more customization. However, that is not usually that important in the vast majority of applications. The developer experience was good when using Expo, and confusing when using React Native, since there were multiple guides on how to add a splash screen and they all seemed slightly different and harder to follow.

4.1.2 Localization

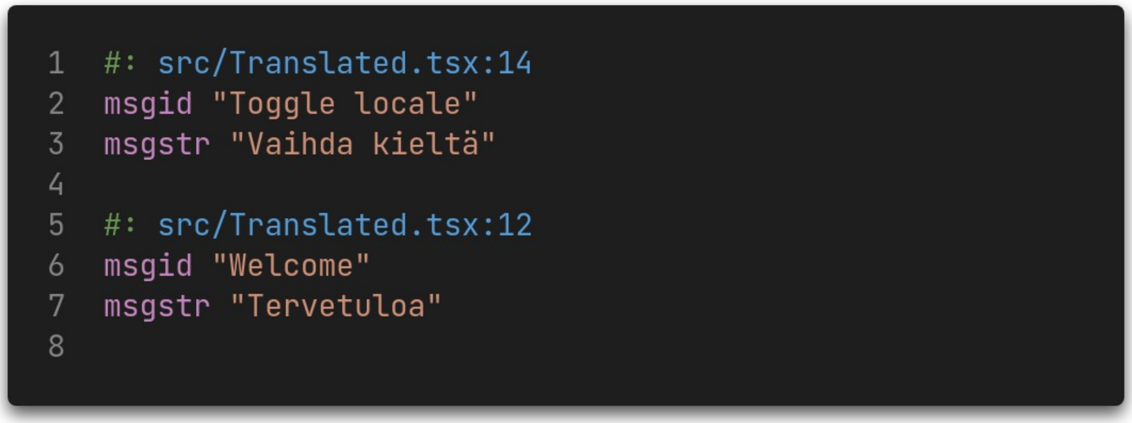
Localization is an important part of applications and the translation process should be easy and clean. The localization should not clutter the code too much, and it should be easy for developers to add them.



```
// Set the key-value pairs for the different languages you want to support.
const translations = {
  en: { welcome: "Hello", name: "Charlie" },
  ["en-FI"]: { welcome: "Morjest", name: "Kaarle" },
  ja: { welcome: "こんにちは" },
};
const i18n = new I18n(translations);

// Set the locale once at the beginning of your app.
i18n.locale = Localization.locale;

// When a value is missing from a language it'll
// fallback to another language with the key present.
i18n.enableFallback = true;

let example = `${i18n.t("welcome")} ${i18n.t("name")}!`;
```

Figure 4. Localization in Expo with the expo-localization package

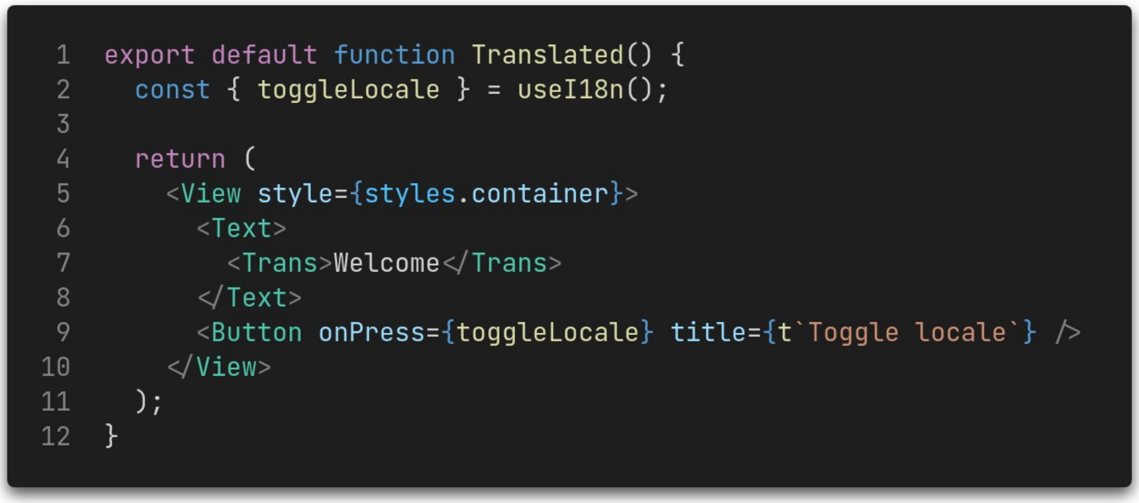
Expo provided an npm (Node Package Manager) package for localization and a demo code (Expo, no date). Figure 4 presents a slightly modified version of the demo, but that was the basic setup for the localization. However, the developer experience can be greatly improved by adding custom React hooks for getting and updating the locale, and by wrapping some of the code to help make the usage cleaner. By introducing macros to simplify the translation process, the developer experience can be made much more welcoming after the initial setup. Macros are not Expo or React Native specific thing to implement, but they will be presented next.



```
#: src/Translated.tsx:14
msgid "Toggle locale"
msgstr "Vaihda kieltä"

#: src/Translated.tsx:12
msgid "Welcome"
msgstr "Tervetuloa"

```

Figure 5. PO-file for two translated messages

In addition to the previous method, the localization can be done in multiple ways. One of these ways is by using LinguiJS, which is “A readable, automated, and optimized (5 kb) internationalization for JavaScript” (LinguiJS, no date). Adding LinguiJS to a project has clear benefits since it can make the code more readable and easier to manage. The translations can be added as separate PO (Portable Object) files as seen in Figure 5, which are translation files that are text-based and meant to be edited by humans (*Files (GNU gettext utilities)*, no date; *Catalog formats — LinguiJS documentation*, no date).



```
 1  export default function Translated() {
 2    const { toggleLocale } = useI18n();
 3
 4    return (
 5      <View style={styles.container}>
 6        <Text>
 7          <Trans>Welcome</Trans>
 8        </Text>
 9        <Button onPress={toggleLocale} title={t`Toggle locale`} />
10      </View>
11    );
12  }
```

Figure 6. Using LinguiJS macros to translate the messages

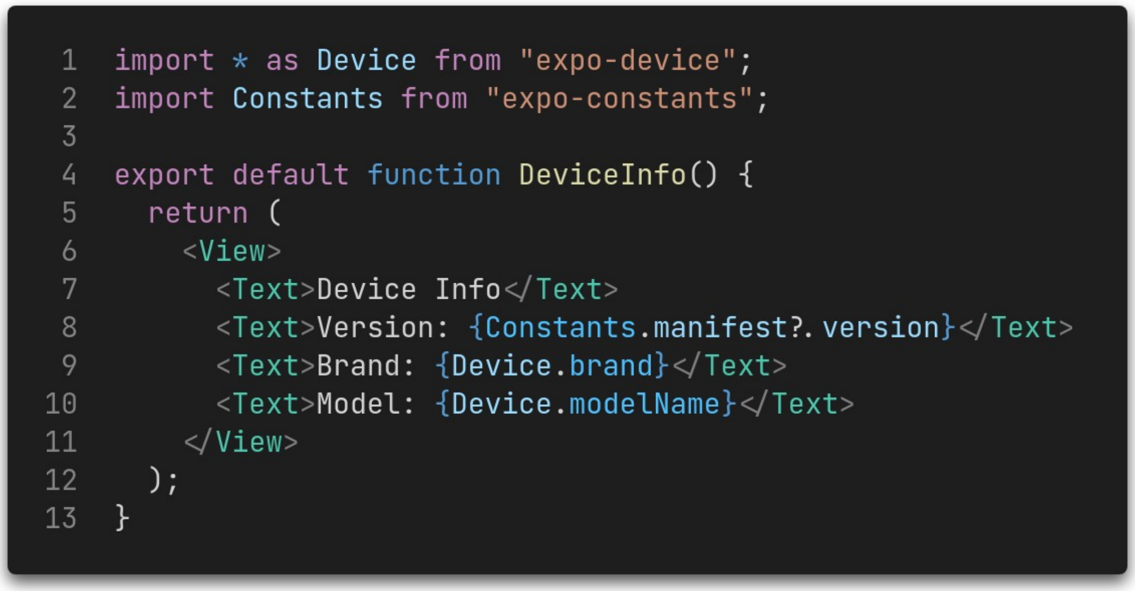
LinguiJS allows the developer to make the translation process easier. Since the messages can be easily written inside the Trans-element or with the “t”-tag to translate template literals as can be seen in Figure 6 on lines 7 and 9. Implementing this feature was much more complex than some other features mentioned previously, but it was fairly similar with both platforms.

The localization in React Native is fairly similar when using LinguiJS, with some minor differences in the configuring. And since the implementation Expo was done earlier than React Native, the process was much smoother since all the challenging parts were already discovered. Expo provided “expo-localization” package but it was not used in this implementation for React Native, since the LinguiJS was seen as a more suitable tool for the localization process.

To conclude the localization, both of the frameworks required multiple files to be edited and a much more in-depth understanding of tools like Babel and LinguiJS. To make the developer translation experience easier quite a bit of utility code needed to be written, but eventually both frameworks could handle this very similarly. This implementation was certainly specific, and other localization tools and methods exist, but it is not in the scope of this thesis and the point of this subchapter was to present a way of implementing it.

4.1.3 Device info

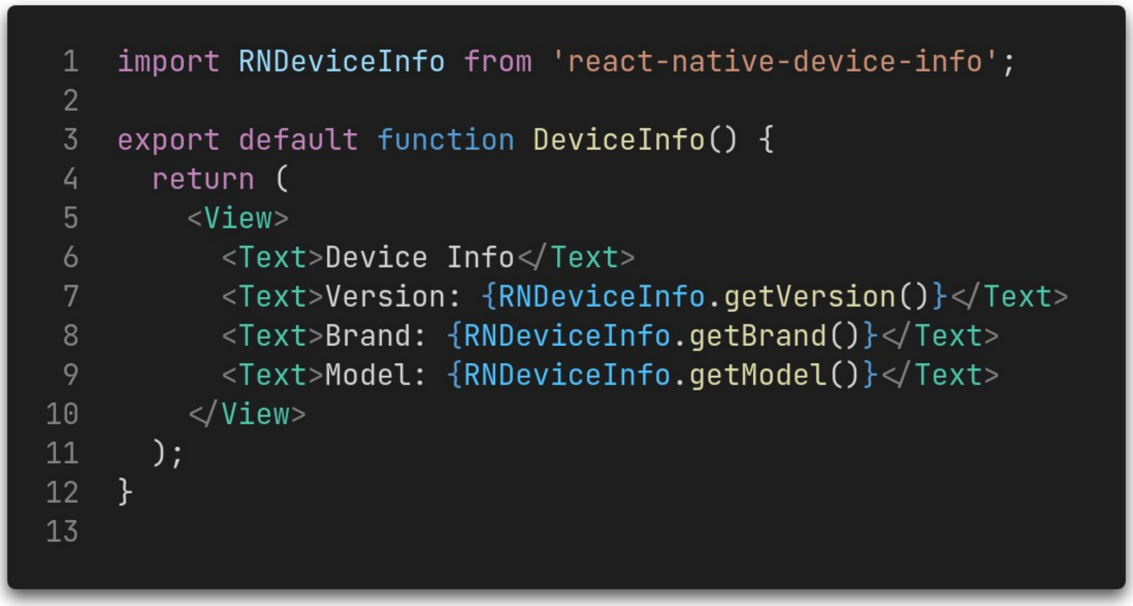
Mobile applications may require access to various values related to the device on which they are running, such as the brand, model, and current application version. These values can be useful for a variety of purposes within the app, such as displaying them in the app's settings or using them to implement device-specific features. For example, an app may need to know the device's model in order to optimize its layout for different screen sizes. Additionally, the current application version can be useful for version tracking and update management. This subchapter will explore the available options for that.



```
import * as Device from "expo-device";
import Constants from "expo-constants";

export default function DeviceInfo() {
  return (
    <View>
      <Text>Device Info</Text>
      <Text>Version: {Constants.manifest?.version}</Text>
      <Text>Brand: {Device.brand}</Text>
      <Text>Model: {Device.modelName}</Text>
    </View>
  );
}
```

Figure 7. Getting device information with the package from Expo

Expo provides a way to access device information with the expo-device package. It offers slightly different features than the React Native equivalent, but it basically lets the developer access almost identically. To access some other values that are not directly related to the device, such as the application version, it is possible to use expo-constants package. Even though these values are found in separate packages, it might arguably make the code cleaner, since the actual device values can be found from the device package and other constants from the constants package.



```
import RNDeviceInfo from 'react-native-device-info';

export default function DeviceInfo() {
  return (
    <View>
      <Text>Device Info</Text>
      <Text>Version: {RNDeviceInfo.getVersion()}</Text>
      <Text>Brand: {RNDeviceInfo.getBrand()}</Text>
      <Text>Model: {RNDeviceInfo.getModel()}</Text>
    </View>
  );
}

```

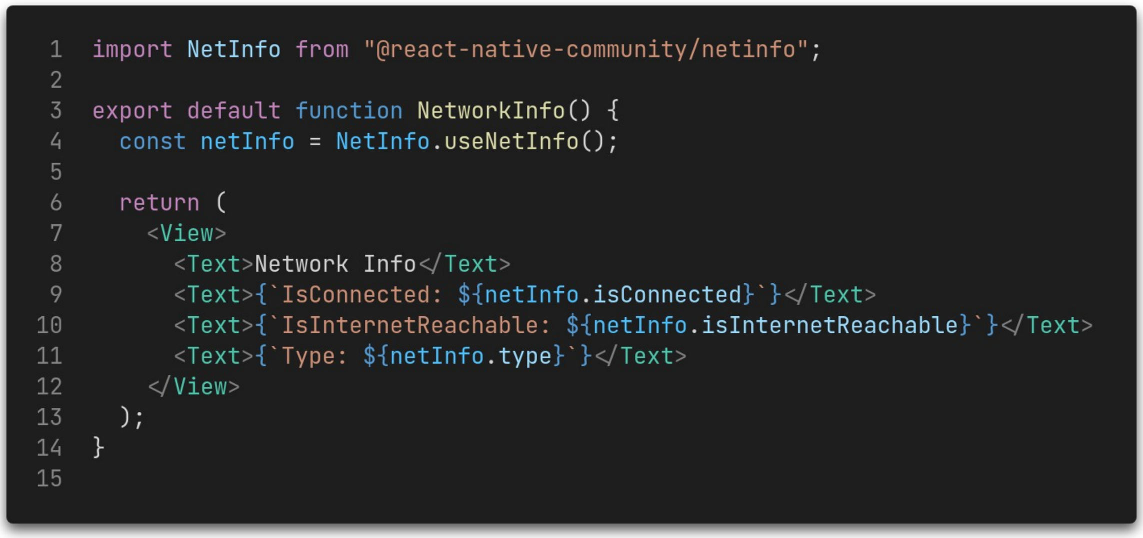
Figure 8. Getting device information in React Native

A common solution to this task for React Native is the react-native-device-info package, which offers very similar functionality as the expo-device package. The same values can be accessed with methods and the overall experience is the same. The main differences to the version provided by Expo are that this version provides only one package and the values are accessed with methods instead of variables.

In the end, these packages can offer very similar information about the device. One difference worth noting is that expo also provides some async methods, such as “getDeviceTypeAsync()”, which require additional code to handle because they involve waiting for the asynchronous promises to resolve.

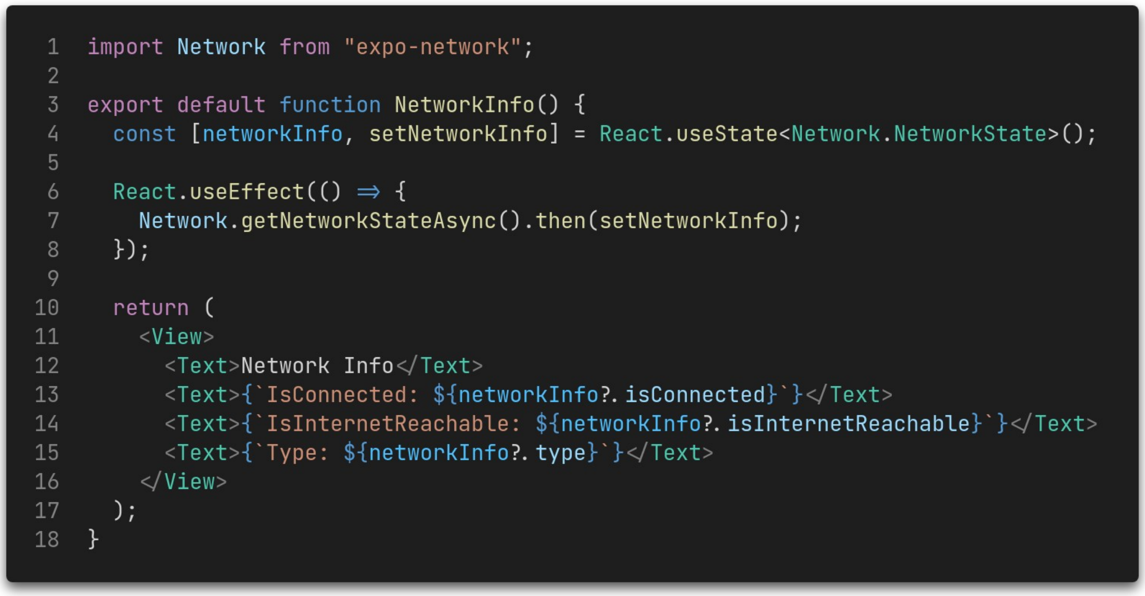
4.1.4 Network info

Getting network information is useful when the user needs to have internet access to perform certain actions. One possible and perhaps common use case is when the application needs to display an alert or a pop-up when the connection is lost. This could be done using listeners to listen for changes in the connection, or the state could be polled once in a while.



```
import NetInfo from "@react-native-community/netinfo";

export default function NetworkInfo() {
  const netInfo = NetInfo.useNetInfo();

  return (
    <View>
      <Text>Network Info</Text>
      <Text>{`IsConnected: ${netInfo.isConnected}`}</Text>
      <Text>{`IsInternetReachable: ${netInfo.isInternetReachable}`}</Text>
      <Text>{`Type: ${netInfo.type}`}</Text>
    </View>
  );
}
```

Figure 9. Using a hook from the netinfo package to get the state.



```
import Network from "expo-network";

export default function NetworkInfo() {
  const [networkInfo, setNetworkInfo] = React.useState<Network.NetworkState>();

  React.useEffect(() => {
    Network.getNetworkStateAsync().then(setNetworkInfo);
  });

  return (
    <View>
      <Text>Network Info</Text>
      <Text>{`IsConnected: ${networkInfo?.isConnected}`}</Text>
      <Text>{`IsInternetReachable: ${networkInfo?.isInternetReachable}`}</Text>
      <Text>{`Type: ${networkInfo?.type}`}</Text>
    </View>
  );
}
```

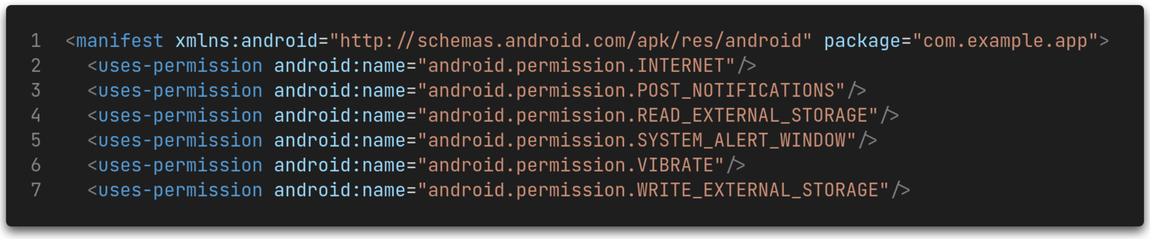
Figure 10. Using the expo package to get the network information.

With Expo, it is possible to use either “expo-network” or “netinfo” –package from the React Native community. The expo-network package, which was demonstrated in Figure 10, did not have a way to add listeners, and it only provided a way to get the state. The other package, netinfo, provided a way to add a listener or to use a Reach hook as shown in Figure 9, so that method was preferred since it made it easier to update the state with just one line of code.

With React Native, the feature can be implemented just like in Figure 9, because the package can be used in both React Native and Expo. To conclude, both of them can accomplish the same thing, but in addition, Expo has also its own package.

4.1.5 Permissions

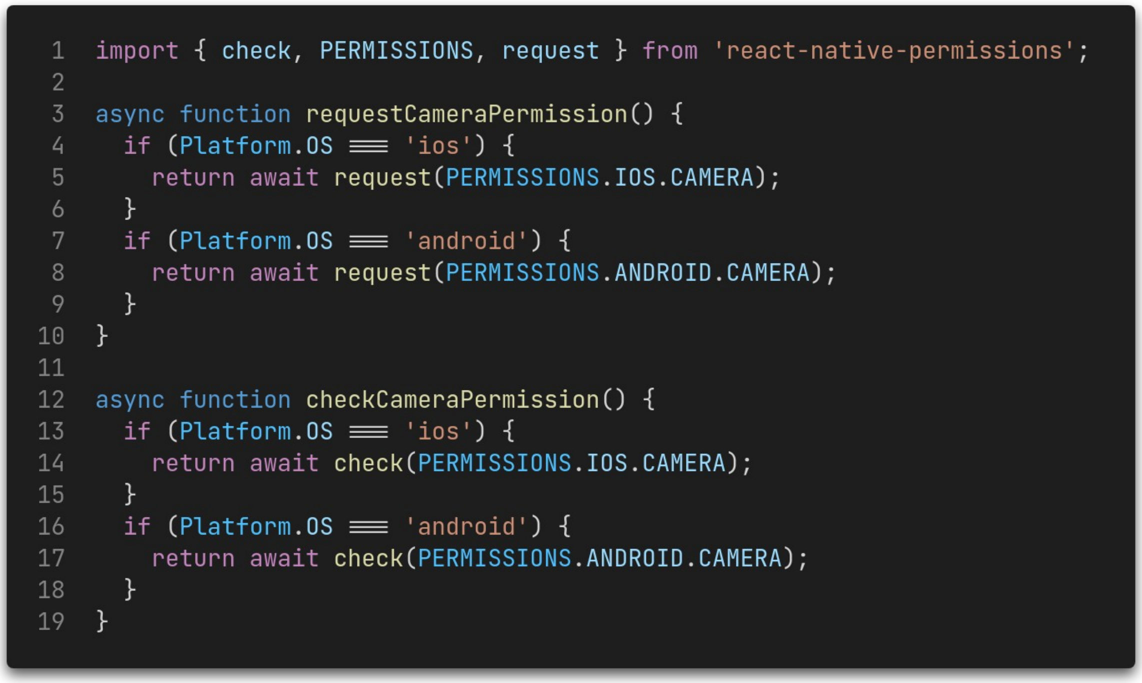
Permissions are handled by platform-specific implementations, and therefore they are not pure JavaScript. That means that developers can write the permissions with native code when they are using React Native, but when they are using Expo they need to rely on different solutions.



```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.example.app">
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
  <uses-permission android:name="android.permission.VIBRATE"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Figure 11. Android permissions in AndroidManifest.xml

Android permissions are added in the AndroidManifest file in XML format (Extensible Markup Language) as can be seen in Figure 11. In the documented in the react-native-permissions repository, iOS permissions are defined by first updating Podfile to list the needed permissions and then running “pod install” (Acthernoene, 2022). The next step for iOS is to update Info.plist with the permission usage descriptions. The documentation also clarified how the iOS and Android permissions flows differ, which is an important topic to cover when developing permissions to these platforms. The usage of the npm package will be presented in the following part of this subsection but it basically abstracts the requesting and checking of the permissions for both of these operating systems. It combines the permission responses, which generally are: unavailable, denied, granted, limited, and blocked.



```
import { check, PERMISSIONS, request } from 'react-native-permissions';

async function requestCameraPermission() {
  if (Platform.OS === 'ios') {
    return await request(PERMISSIONS.IOS.CAMERA);
  }
  if (Platform.OS === 'android') {
    return await request(PERMISSIONS.ANDROID.CAMERA);
  }
}

async function checkCameraPermission() {
  if (Platform.OS === 'ios') {
    return await check(PERMISSIONS.IOS.CAMERA);
  }
  if (Platform.OS === 'android') {
    return await check(PERMISSIONS.ANDROID.CAMERA);
  }
}
```

Figure 12. Checking and requesting permissions in React Native

In React Native, defining the permissions was done by editing the native files, as discussed above, but checking and requesting the permissions is done with "react-native-permissions" npm-package. With the package, the process is relatively straightforward, since the developer can use "request" and "check" functions, which return the permissions responses or statuses mentioned earlier. This is presented in Figure 12, where the application checks and requests the permission state.

Expo on the other hand has a more complicated situation when it comes to permissions. Requesting and checking permissions can be like in React Native (Figure 12), but the different part is when it comes to defining or listing the permissions. Because Expo does not have direct access to native files, a workaround needs to be used. Some expo packages have built-in permission tools, for example, the MediaLibrary (*MediaLibrary*, no date), which provides functions and hooks to request the permissions. In one way, this can provide a clean and simple way of handling the permissions by leaving the handling part to the package, but this is not a universal solution. If a developer wants to write highly specific code or implementation of some feature, the permissions provided by these packages might not be enough. Some other examples in addition to MediaLibrary are Camera, Brightness, Contacts, and many other packages. From a developer's viewpoint, this way of dividing the permission handling into individual packages makes it easy to use because the permission handling is part of the package already, but a more generalized and centralized solution would be welcome.

4.1.6 Publishing

Publishing and continuous integration and delivery (CI/CD) are important aspects of mobile application development. CI/CD is a process that involves automatically building, testing, and deploying code changes to production environments. In the context of mobile application development, this means automating the process of building and deploying apps to app stores or other distribution channels. CI/CD tools allow developers to automate these processes and easily manage multiple build configurations for different platforms.

Publishing in App Center is an important feature that is very specific, but higly relevant for this thesis. App Center is a cloud-based mobile development platform that helps developers build, test, and deliver their applications. It is a tool offered by Microsoft and is particularly useful for building cross-platform applications using frameworks such as React Native and Expo (*Visual Studio App Center*, no date a). With App Center, developers can easily set up continuous integration and delivery (CI/CD) pipelines, which allow them to automatically build and deploy their applications to various platforms.

To publish a React Native or Expo application in App Center, developers must set up a build configuration in App Center and specify the repository and branch to build from. They must also configure the build process by providing necessary scripts and dependencies. To publish, a release must be created in App Center and the build must be specified. Release channels and target audience must also be specified. The process for publishing a React Native or Expo application in App Center is generally similar, with the main difference being in the build configuration and build process.

However, App Center does not provide ways to build fully managed Expo apps. The App Center Github repository has had an open issue requesting support for fully managed Expo applications for almost four years (*Support Expo React Native Apps in App Center · Issue #189 · microsoft/appcenter*, no date), but the issue still remains open. To use the App Center, the developers need to use some other way to manage their Expo application for now.

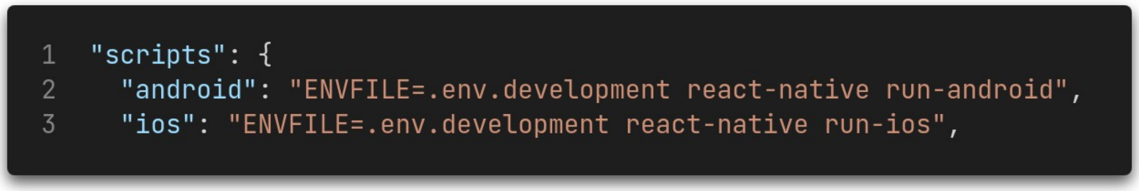
In a React Native application, developers may need to configure the build process by providing additional build scripts and dependencies, as well as specifying any environment variables that are required. This is because React Native applications have more flexibility and access to native files compared to Expo applications, which means that there may be more steps involved in the build process.

One other option worth mentioning is Bitrise. It is a continuous integration and delivery (CI/CD) platform (*Welcome to Bitrise documentation!*, no date), and it has very good support for React Native (*How to set up a React Native app on Bitrise - Bitrise*, no date). According to Khoa Pham (Pham, 2020), Bitrise makes the CI process easy and a “pretty good UI to add and edit steps”. Bitrise supports Expo too, and it has a detailed step-by-step guide to make that straightforward (*Getting started with Expo apps*, no date).

4.1.7 Environment and config variables

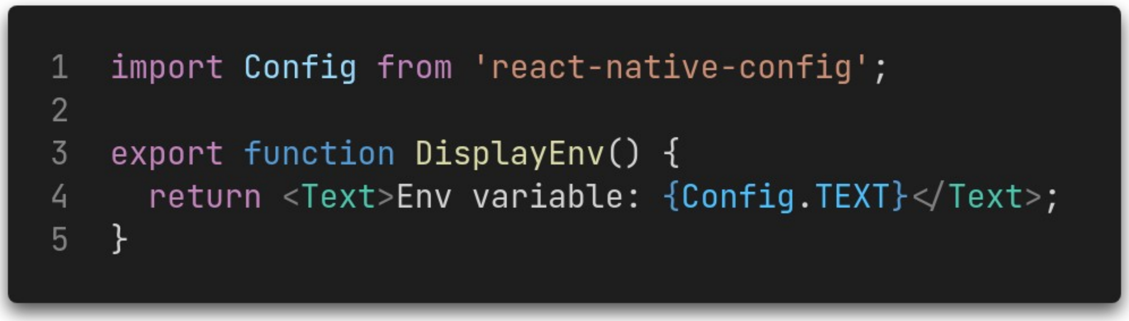
Environment variables are an important part of mobile development because they allow you to store configuration data that can be used by your app at runtime. This can be useful for things like storing API keys, enabling or disabling certain features, or specifying different behavior for different environments (e.g., development, testing, production). (Innocent, 2022)

In React Native, there were multiple options to choose from, but one of those was react-native-config. Setting the variables was a fairly simple process, even though the developer experience could have been more streamlined. At first, the environment files needed to be created, such as .env.development and .env.production. In this implementation, the files contained only one variable so keep the process simple. By following the steps in the documentation (*react-native-config*, no date), one line was added to the android build.gradle -file, start scripts were updated and then variables were accessed in the JS/TS file.



```
"scripts": {
  "android": "ENVFILE=.env.development react-native run-android",
  "ios": "ENVFILE=.env.development react-native run-ios",
```

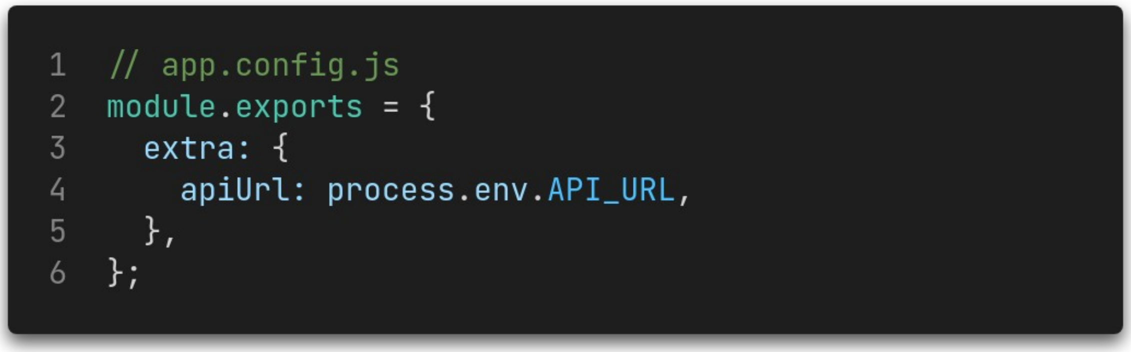
Figure 13. Scripts to start the development environment.



```
import Config from 'react-native-config';

export function DisplayEnv() {
  return <Text>Env variable: {Config.TEXT}</Text>;
}
```

Figure 14. Using an environment variable in a React component.

To allow the program to know which environment to use, the name of the environment file was passed in the android and ios start scripts as can be seen in Figure 13. Similar way the environment file can be added to the build step when creating the releases for example for production or testing. For this implementation, three files were created: .env.production, .env.development, and .env, and they worked well with the react-native-config npm package. In the React component, the usage was simple because it was only a matter of importing the Config object from the package and accessing the variables from there, which was demonstrated in Figure 14.



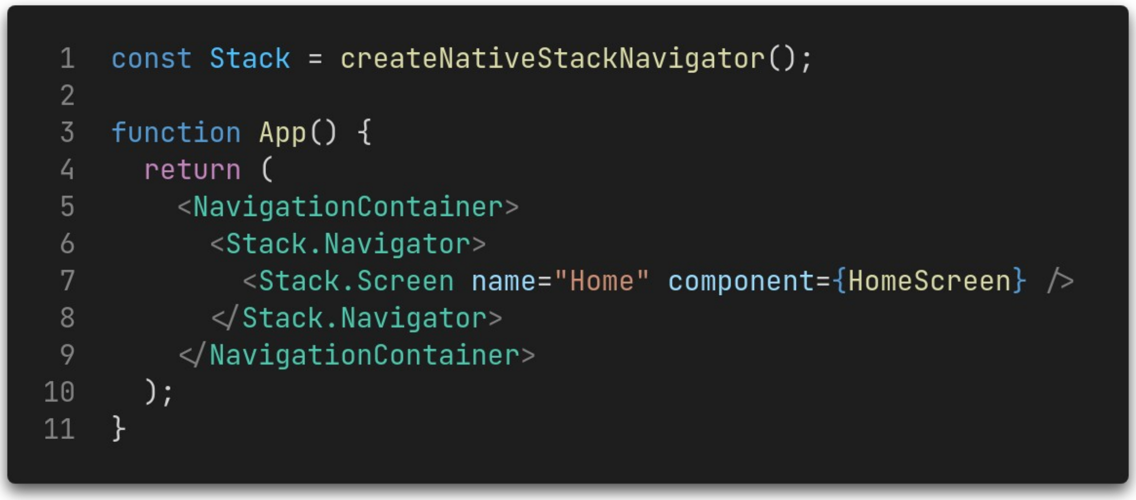
```
// app.config.js
module.exports = {
  extra: {
    apiUrl: process.env.API_URL,
  },
};
```

Figure 15. Dynamic app config in Expo with a value for API url

Expo on the other hand listed multiple different ways of using environment variables in their documentation (*Environment variables in Expo*, no date). One of these ways was to use a dynamic app config, which is presented in Figure 15. This provides a simple way to implement config variables for the application to use. Expo provides a way to read the extra properties with the expo-constants package. For example, to use the apiUrl defined above, the developer can write Constants.expoConfig.extra.apiUrl and access the value. For loading the .env files, Expo documentation recommends using direnv and .envrc.

4.1.8 Routing

In a mobile application, the user usually needs to navigate between multiple screens. React Navigation package was created to solve this task in React Native applications. It has almost 670 thousand weekly downloads from npm, which is 56% of weekly React Native downloads (~1.19 million) (*@react-navigation/native*, no date; *react-native*, no date). That makes it the most popular React Native navigation library. However, Expo has been developing a new way of handling routes and it has a router library that is in beta at the time of writing, and it should bring “the best routing concepts from the web to native iOS and Android apps” (*Introduction* | *Expo Router*, no date).

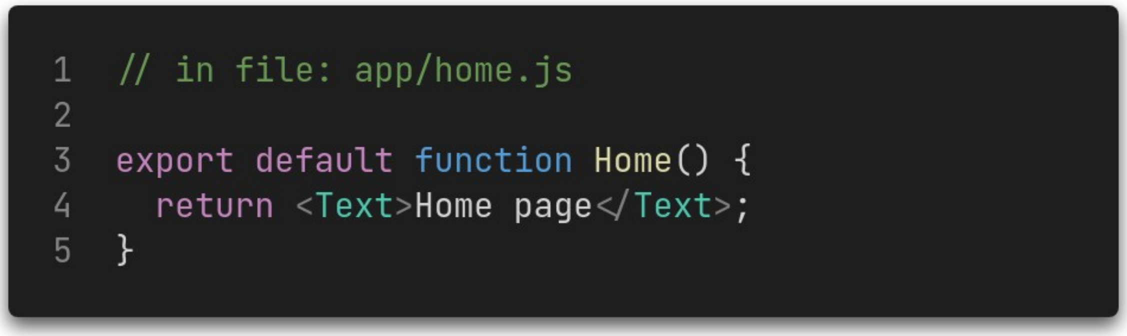


```
const Stack = createNativeStackNavigator();

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Figure 16. Demo of a native stack navigator with React Navigation

The default way of creating navigators is React Navigation, and Figure 16 shows an example from (*Hello React Navigation* | *React Navigation*, no date) on how to implement simple stack navigation. Based on the React Navigation documentation, this approach consists of Navigator, which is a React component that renders the right screens, and Screens are the actual screen elements or views to be displayed in the app.

Expo Router on the other hand has decided to go with file-based routing, where the structure and names of the files affect the routes and every file in the app directory is a route for the application. (*Introduction | Expo Router*, no date) This means that developers need to think differently when creating the file structure, and the routing knowledge could be transferred from web development frameworks to mobile development since frameworks like Next.js and SvelteKit use filesystem-based routers (*Basic Features: Pages | Next.js*, no date; *Routing • Docs • SvelteKit*, no date). Expo Router is built on top of React Navigation, so the truly native navigation is still working under the hood, bringing the animations and other native elements of the navigation to the library.



```
// in file: app/home.js

export default function Home() {
  return <Text>Home page</Text>;
}
```

Figure 17. Home screen example in file-based routing in Expo Router

Figure 17 demonstrates the file-based routing, where the Home component is placed in the app directory and then the router automatically knows how to handle it. Some of the examples from the documentation were:

- app/home.js matches /home.
- app/settings/index.js matches /settings.
- app/[user].js matches any unmatched path like /evanbacon or /expo. (*Introduction | Expo Router*, no date)

However, this package is still in beta and changes can occur, so it might not be relevant to swap the existing React Navigation to Expo Router, but it certainly offers an interesting option to the current solutions. Expo Router is not yet available in the Expo Go app, meaning that if developer want to use it with Expo Go, they need to wait for a future release. This is something that needs further research in the future when the package has evolved a bit more and has left the beta stage.

4.1.9 Over-the-air updates

Over-the-air (OTA) updates refer to the ability to deliver updates to a mobile application wirelessly, without the need for the user to manually download and install the update from an app store. This can be especially useful for fixing bugs or adding new features to a mobile application without requiring the user to take any action. In the context of React Native and Expo, both platforms support OTA updates for applications that are built and deployed using their tools.

Based on a guide by Microsoft React Native can use CodePush, which is a cloud service that enables developers to deploy updates directly to their users' devices. CodePush is a cloud service provided by App Center that allows developers to deploy updates to their React Native mobile apps directly to their users' devices. It functions as a central repository for updates such as changes to JavaScript code and images. By using the provided client SDKs, apps can query for updates from the repository. CodePush enables developers to have a more direct and controlled way to engage with their users and address bugs, add small features, or deploy updates without the need to rebuild a binary or go through public app stores. (*Visual Studio App Center*, no date b)

With Expo, in addition to CodePush, another option for deploying updates to React Native apps is Expo's EAS (Expo Application Services) Updates feature. EAS Updates is a hosted service specifically designed for projects using the expo-updates library. EAS Updates allows developers to quickly fix small bugs and make updates in between app store submissions by allowing the end-user's app to replace non-native elements – i.e. elements made with only JavaScript – with new updates containing bug fixes and other improvements. All apps running the expo-updates library are able to receive updates through EAS Update. (*EAS Update*, no date) While both CodePush and EAS Updates offer the ability to deploy updates directly to users' devices, they differ in their implementation and target audience. CodePush is available for all React Native apps created on App Center (*Visual Studio App Center*, no date b), while EAS Updates is specifically geared towards projects using the expo-updates library. (*EAS Update*, no date)

Overall, both React Native and Expo offer OTA update capabilities, which can be a useful tool for developers to quickly and easily deliver updates to their users without requiring any action on the part of the user.

4.2 The trend with Expo plugins and packages

Because the goal of this thesis is to determine if Expo is able to replace React Native in the company applications, this subchapter will explore the current status of its development. One important factor to consider when deciding whether Expo is a suitable tool to use in the future, is to look at how it has been progressing so far. Expo's popularity and development will be reviewed and some estimations for the future are given based on the data.

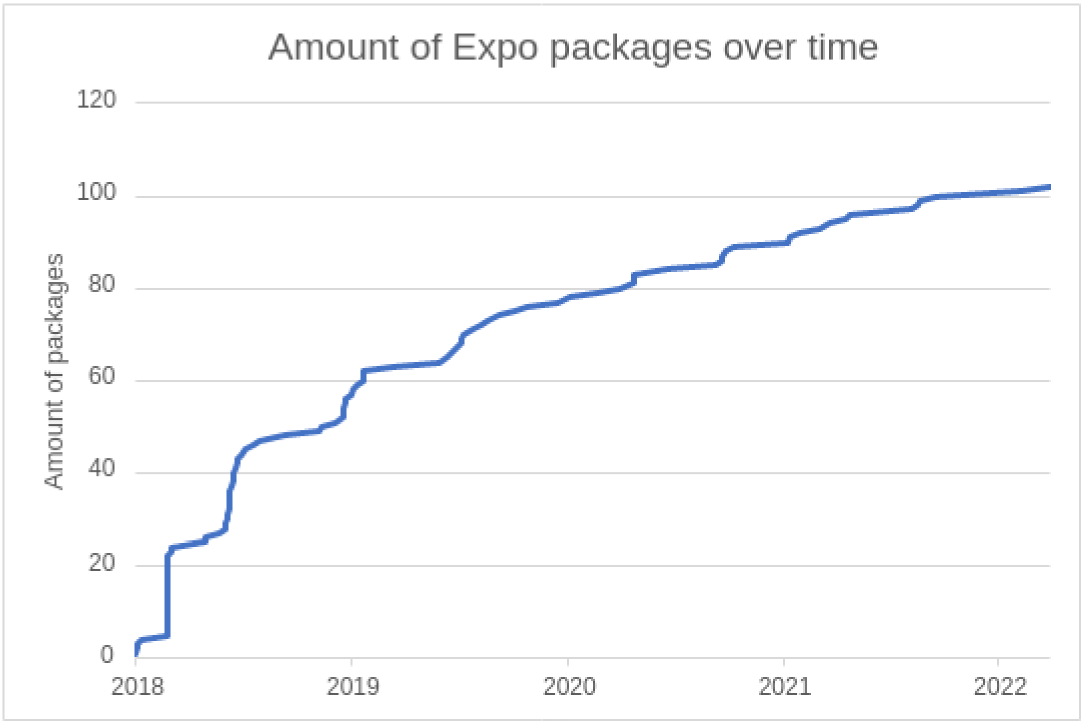


Figure 18. Amount of Expo packages over time

Expo has been providing packages for different use cases for a few years, such as expo-camera, expo-constants, expo-contacts, and expo-permissions to name a few. The amount of these packages over time can be seen in Figure 18, which was generated based on the Expo repository, where all packages are stored in the packages folder. The date for each package was determined based on the first commit to each package. This method might not give truly accurate information on when the package was exactly created, and the folder might have internal packages, but it still gives relevant insight into how the general progress has been. From the graph, it could be predicted that new expo implementations and packages will keep coming since the development has not reached a plateau yet, where it would mostly just update existing packages. That means that developers could be optimistic about the future of the Expo development.

The data also revealed some insight into what could be coming next because the latest package was expo-maps, which is currently unpublished and experimental package, which means that Expo is still working on important packages to improve its already vast ecosystem. However, this means that some important features might not be conveniently available as Expo’s package yet, but as shown in the previous comparisons in the

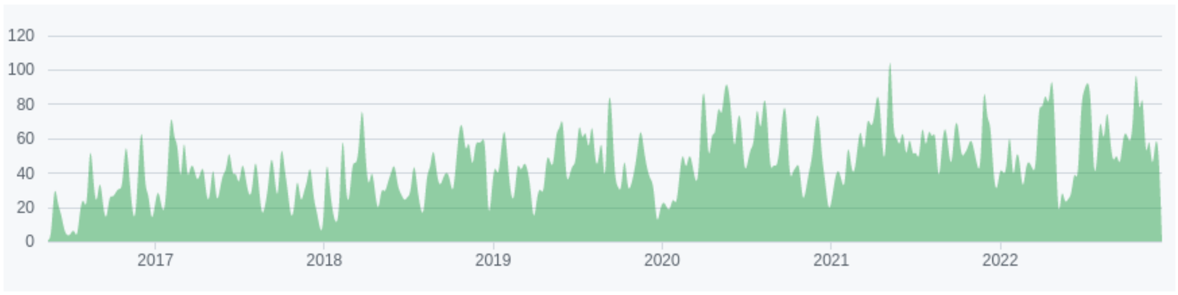


Figure 19. Contributions to the main branch in the Expo repository (Contributors to expo/expo, no date)

The developer activity in the Expo project seems promising when looking at the contribution activity graph in Figure 19. The image shows “contributions to main, excluding merge commits and bot accounts” (*Contributors to expo/expo*, no date). If the developer activity would be clearly decreasing, it could indicate that the project might not be growing or it would become more inactive.

5 Results

The results of the comparison of React Native and Expo showed that both frameworks offer a range of features and benefits that make them suitable for building a wide variety of applications. Many times the implementations were almost identical, but sometimes Expo had also its own package for the task. Expo's own packages felt also more simplified and more abstracted, like for example setting the splash screen, which was just a matter of editing one JSON config file and adding the images, whereas it was a much more complicated task in bare React Native. This chapter will address the comparison of these frameworks by presenting three different results of direct 1-to-1 comparison of the feature implementations. The comparisons are:

1. Thoughts and feedback about the feature implementations and the experiences
2. Lines of code required for each feature
3. Number of edited files for each feature

These comparisons can provide valuable insight into the features and the implementation differences of the frameworks. The results of each of the three comparisons are presented in separate tables below.

Table 3: Experiences with both frameworks when implementing the features

Task	**Results**
Splash screen and app icons	Adding a customized splash screen and app icons was clearly an easier task on Expo since it required only editing the JSON file, whereas it was much more complicated in React Native.
Localization	Both of the frameworks were similar in this task, and the implementations were mostly platform-independent.
Device info	Both of the frameworks allowed easy access to the relevant device and application information, such as application version, device brand, and type, and the results are down to preference at the end. However, Expo had asynchronous

	methods making the implementations a bit more complicated.
Network info	Both of the frameworks provided similar tools to access the information just like the device info.
Permissions	Defining permissions was very different on both platforms since React Native required editing native and platform-specific files. Expo on the other hand had abstracted the process
Environment variables / Config variables	Configuring environment variables was also quite different on both platforms, but it was a little bit easier on Expo.
Routing	Currently, the most popular routing solution is used in both frameworks, but Expo had an interesting idea for the future with the file-based routing. However, that is still in beta.

The general feedback and the experience of both frameworks can be seen in Table 3. It highlights the key differences and conclusions about each feature in both frameworks. Both frameworks had most of the time similar or even the same solution for the features, but the most notable differences were, for example, the splash screen and app icons, and permissions. These features required completely different approaches on both platforms.

Table 4: Lines of code used for each task

Task	**Lines of code with React Native**	**Lines of code with Expo**
Splash screen and app icons	20+ (without images)	5 (without images)
Localization	-	-
Device info (with 3 values)	4	9
Network info (with 3 values)	5	5 or 8
Permissions	10	2+
Environment variables / Config variables	6	5+
Routing	-	-

One of the selected ways to measure the developer experience is to count the lines of code needed for a given feature as seen in Table 3. This will give numerical insight into how complex different solutions have been, and looks like Expo has usually fewer lines of code compared to bare React Native. It is worth noting that the implementations were done for Android phones only, and some of them will require additional changes in order to make the iOS version work in the bare React Native app. Such changes are for example splash screen and permissions. This is another point in favor of Expo, because these features will work on both platforms without additional lines of code. The plus symbol ("+") indicates that more lines could be added depending on the situation. In the feature "Splash screen and app icons" the plus means that iOS will require additional lines, but in some features, like in Table 5 for the feature "Localization" they mean that both of the features will have multiple edited files, and the actual number of files can be anything. The hyphen symbol ("-") is used to indicate features where the measurement did not make sense or was not possible.

Table 5: Files edited for each task

Task	**# of edited files with React Native**	**# of edited files with Expo**
Splash screen and app icons	5+	1
Localization	1+	1+
Device info	1	1
Network info	1	1
Permissions	2+	1-2+
Environment variables / Config variables	4+	3+
Routing	-	-

The number of edited files can be seen in Table 5, and it shows that when developing with Expo, developers do not need to edit so many files compared to bare React Native. The difference comes from platform-specific native files, that needed editing when using React Native.

One key difference that was identified is that React Native provides more flexibility and access to native files compared to Expo, without writing config plugins to insert native code. This means that developers using React Native have more control over the native features and functionality of their applications, and can more easily implement complex or custom features. This also means that React Native may require more effort and expertise to set up and maintain, and may require developers to write more platform-specific code. However, Expo’s config plugins have also shown that they are capable of implementing necessary native features or modifications. Based on the implementations, Expo was able to succeed in every task it was presented with. Some tasks were more challenging with it but the majority were easier.

6 Discussion

After comparing React Native and Expo cross-platform mobile development frameworks, the results show that it is clear that both have their own unique strengths and limitations. React Native offers a more robust set of features and capabilities, giving developers greater control over the native functionality of their applications, but this also means that they need to write more lines of code and edit more files, including platform-specific files. On the other hand, Expo provides a simpler and more streamlined development process, and its platform-neutral API eliminates the need for platform-specific code. Some features, like app icons and splash screens, were much easier to implement in Expo, but some features were more complicated since Expo had an opinionated way to handle those, such as the permissions. Also, some highly specialized native functionality can be more challenging to implement with Expo since then the developer would need to use config plugins. The results of this thesis showed that Expo was able to handle the same tasks as bare React Native, but with fewer lines of code and with fewer files. Some of the React Native features required editing platform-specific files in the android and ios folders, whereas Expo allowed the developers to write the same code for both platforms.

The first research question was about the biggest limitations and benefits when moving from React Native to Expo. The results showed that the biggest benefits were the reduced complexity and the ease of development when using Expo. Features required fewer files and lines of code, and the implementation process was streamlined since Expo had abstracted many of the tasks. The second research question was about the developer experience, and based on the results, Expo provided a nicer developer experience. If a developer comes with a web development background – as they usually do in the company – the Expo is much easier to manage, since there is less worrying about the platform-specific code in android and ios folders.

Expo also abstracts the upgrade process of React Native by providing the “expo-cli upgrade” command. The command handles the upgrading React, React Native, Expo, and other known dependencies and therefore makes the upgrading easier. The command also updates app.json config file, validates the project, lists the tasks it does, and highlights useful information after running the command. Based on the Expo documentation, the upgrade process of native apps is “extremely challenging and users often either upgrade their app incorrectly” or miss some crucial changes (*Prebuild*, no date). The documentation recommends using the prebuild upgrading because it is similar to a pure JavaScript upgrading process, where the package versions are just updated and then the project is generated.

6.1 Trends

One of the trends that arose during the research was that when React Native required editing multiple native files for a common task, Expo had created a package for that. Chapter 4.2 presented the current progress and development of Expo packages, and that shows that developers of the Expo framework have been actively creating new packages. That could mean that the trend of creating packages to abstract difficult platform-specific features will continue. This prediction is good news to developers who use Expo since it means that the usage will become easier and easier over time, and the needs for config plugins will be smaller.

6.2 Limitations and future research

The main limitation of this study is that it might not be generalizable to every project, because the features were selected based on the company’s needs, and more specialized features were not covered. Additional research is needed to cover the topic more in-depth. The second major limitation is that the comparison was done using only Android devices, with only brief mentions of iOS implementations.

Future research on these topics could include how well Expo config plugins remain stable and how much they cause technical debt since they can edit native folders with config plugins. How well these plugins stand the test of time and remain functional is one question, and how relevant they will still be in two or five years. This topic could be important to re-evaluate in the future.

Additional research could also go in-depth on the performance of these implementations and the differences between them. Expo could add more performance and memory issues since it is an abstraction layer on top of React Native, but smartphones also keep getting more powerful and the performance differences might not be that relevant in the end. A more specific potential area of study could be a more detailed exploration of the performance of Expo Config Plugins. This could include studies of real-world applications built with Expo to measure and assess its performance in different scenarios.

Another potential area of research could be the adoption of Expo among developers. This could include surveys and interviews with developers to assess their attitudes towards Expo and its features when compared to React Native. Some software engineering surveys might provide some insight into this question already, but more in-depth questionnaires might be useful.

7 Conclusions

This thesis provided insight into the benefits of Expo and how does it compare to bare React Native applications. Multiple implementations of different features were compared by implementing the selected set of features using both frameworks, and the developer experience was explored.

In terms of their potential for use in the industry, both frameworks show promise. React Native has already established itself as a popular choice among developers, and its robust feature set makes it well-suited to a wide variety of applications. Expo, on the other hand, has recently introduced new features such as Expo Config Plugins, which may make it a more viable option for developers today, since it opens up the possibility to edit and access the platform-specific files. Before config plugins, developers needed to fully rely on Expo’s own packages for native modules, such as accessing camera and Bluetooth.

Ultimately, the decision of which framework to use will depend on the specific needs and goals of each project, but for the needs of the company, Expo seems to be enough for most projects, since no major challenges were faced. However, applications that require highly specific or complex native implementations might be better to be done with React Native, but even then truly native tools, such as Android Studio and Xcode could be considered. For developers seeking a more powerful and flexible toolset with a small compromise in developer experience, React Native may be the better choice. However, for those looking for a simpler and more streamlined development process, Expo could be the more suitable option. In either case, it is important for developers to carefully evaluate their options and choose the framework that best fits the project’s needs.

The current direction where Expo is heading looks promising, as new packages have been added to the Expo ecosystem. In the future, more and more of the potentially missing features will be implemented and Expo will grow to be a more and more attractive alternative to bare React Native, but even now, it clearly provides a remarkable developer experience.

References

Acthernoene, M. (2022) ‘ react-native-permissions’. Available at: https://github.com/zoontek/react-native-permissions (Accessed: 17 December 2022).

API Reference (no date) *Expo Documentation*. Available at: https://docs.expo.dev/versions/latest (Accessed: 29 December 2022).

Basic Features: Pages | Next.js (no date). Available at: https://nextjs.org/docs/basic-features/pages (Accessed: 23 January 2023).

Bogner, J. and Merkel, M. (2022) ‘To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub’. arXiv. Available at: http://arxiv.org/abs/2203.11115 (Accessed: 1 January 2023).

Borozenets, M. (2022) ‘React Native Init vs Expo 2022: What Are the Differences?’, 22 April. Available at: https://fulcrum.rocks/blog/react-native-init-vs-expo (Accessed: 8 January 2023).

Carli, S. (2021) *Building cross-platform apps with Expo instead of React Native, LogRocket Blog*. Available at: https://blog.logrocket.com/building-cross-platform-apps-expo-instead-of-react-native/ (Accessed: 2 January 2023).

Catalog formats — LinguiJS documentation (no date). Available at: https://lingui.js.org/ref/catalog-formats.html (Accessed: 15 December 2022).

Chitu, A. (2007) ‘Google Launches Android, an Open Mobile Platform’, *Google Launches Android, an Open Mobile Platform*, 5 November. Available at: https://googlesystem.blogspot.com/2007/11/google-launches-android-open-mobile.html (Accessed: 13 November 2022).

Config Plugins (no date) *Expo Documentation*. Available at: https://docs.expo.dev/guides/config-plugins (Accessed: 25 January 2023).

Contributors to expo/expo (no date). Available at: https://github.com/expo/expo/graphs/contributors (Accessed: 18 December 2022).

Cross-platform mobile frameworks used by global developers 2021 (no date) *Statista*. Available at: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (Accessed: 19 December 2022).

Davis, T.C. (2022) *Introducing: Custom Development Clients*, *Medium*. Available at: https://blog.expo.dev/introducing-custom-development-clients-5a2c79a9ddf8 (Accessed: 25 January 2023).

EAS Update (no date) *Expo Documentation*. Available at: https://docs.expo.dev/eas-update/introduction (Accessed: 30 December 2022).

Environment variables in Expo (no date) *Expo Documentation*. Available at: https://docs.expo.dev/guides/environment-variables (Accessed: 30 January 2023).

Etukudo, E. (2022) *Building a splash screen in React Native*, *LogRocket Blog*. Available at: https://blog.logrocket.com/splash-screen-react-native/ (Accessed: 6 December 2022).

Expo (no date) *Localization*, *Expo Documentation*. Available at: https://docs.expo.dev/guides/localization (Accessed: 14 December 2022).

Files (GNU gettext utilities) (no date). Available at: https://www.gnu.org/software/gettext/manual/html_node/Files.html (Accessed: 15 December 2022).

First-class Support for TypeScript · React Native (2023). Available at: https://reactnative.dev/blog/2023/01/03/typescript-first (Accessed: 25 January 2023).

Getting started with Expo apps (no date) *Bitrise Docs*. Available at: https://devcenter.bitrise.io/en/getting-started/getting-started-with-expo-apps.html (Accessed: 4 February 2023).

Google (no date) *Application Fundamentals*, *Android Developers*. Available at: https://developer.android.com/guide/components/fundamentals (Accessed: 12 November 2022).

Gülcüoğlu, E., Ustun, A.B. and Seyhan, N. (2021) ‘Comparison of Flutter and React Native Platforms’, *Journal of Internet Applications and Management* [Preprint]. Available at: https://doi.org/10.34231/iuyd.888243.

Gupta, D. (2004) ‘What is a good first programming language?’, *XRDS: Crossroads, The ACM Magazine for Students*, 10(4), pp. 7–7. Available at: https://doi.org/10.1145/1027313.1027320.

Handbook - Basic Types (no date). Available at: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html (Accessed: 1 January 2023).

Hanif, M. (2020) ‘React Native Structure Folder — For Simplicity’, *Medium*, 13 October. Available at: https://hanifmhd.medium.com/react-native-structure-folder-for-simplicity-e2345c87ee5f (Accessed: 25 January 2023).

Hello React Navigation | React Navigation (no date). Available at: https://reactnavigation.org//docs/hello-react-navigation (Accessed: 23 January 2023).

How to set up a React Native app on Bitrise - Bitrise (no date). Available at: https://bitrise.io/blog/post/how-to-set-up-a-react-native-app-on-bitrise (Accessed: 4 February 2023).

Innocent, C. (2022) *Understanding React Native env variables*, *LogRocket Blog*. Available at: https://blog.logrocket.com/understanding-react-native-env-variables/ (Accessed: 30 December 2022).

Introduction | Expo Router (no date). Available at: https://expo.github.io/router/docs (Accessed: 23 January 2023).

Kruhlyk, Y. (2018) *Expo vs Vanilla React Native Development: What to Choose*, *Apiko | Learn*. Available at: https://apiko.com/blog/expo-vs-vanilla-react-native/ (Accessed: 29 December 2022).

LinguiJS (no date) *LinguiJS - Seamless internationalization in Javascript — LinguiJS documentation*. Available at: https://lingui.js.org/ (Accessed: 14 December 2022).

MediaLibrary (no date) *Expo Documentation*. Available at: https://docs.expo.dev/versions/latest/sdk/media-library (Accessed: 17 December 2022).

Miquido (2022) *Native vs Cross-Platform Mobile App Development: A Comparison - Miquido Blog*, *Miquido*. Available at: https://www.miquido.com/blog/native-vs-cross-platform-app-development/ (Accessed: 19 December 2022).

Nawrocki, P. *et al.* (2021) ‘A Comparison of Native and Cross-Platform Frameworks for Mobile Applications’, *Computer*, 54(3), pp. 18–27. Available at: https://doi.org/10.1109/MC.2020.2983893.

Occhino, T. (2015) ‘React Native: Bringing modern web techniques to mobile’, *Engineering at Meta*, 26 March. Available at: https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/ (Accessed: 19 December 2022).

Pham, K. (2020) ‘Using Bitrise CI for React Native apps’, *Fantageek*, 14 March. Available at: https://medium.com/fantageek/using-bitrise-ci-for-react-native-apps-b9e7b2722fe5 (Accessed: 4 February 2023).

Prebuild (no date) *Expo Documentation*. Available at: https://docs.expo.dev/workflow/prebuild (Accessed: 25 January 2023).

React Native · Learn once, write anywhere (no date). Available at: https://reactnative.dev/ (Accessed: 29 December 2022).

React Native Directory (no date) *React Native Directory*. Available at: https://reactnative.directory (Accessed: 19 December 2022).

react-native (no date) *npm*. Available at: https://www.npmjs.com/package/react-native (Accessed: 23 January 2023).

react-native-config (no date) *npm*. Available at: https://www.npmjs.com/package/react-native-config (Accessed: 30 December 2022).

@react-navigation/native (no date) *npm*. Available at: https://www.npmjs.com/package/@react-navigation/native (Accessed: 23 January 2023).

Routing • Docs • SvelteKit (no date). Available at: https://kit.svelte.dev/docs/routing (Accessed: 23 January 2023).

Setting up the development environment · React Native (no date). Available at: https://reactnative.dev/docs/environment-setup (Accessed: 25 January 2023).

Showcase · React Native (no date). Available at: https://reactnative.dev/showcase (Accessed: 19 December 2022).

Sinicki, A. (2016) *Developing for Android vs developing for iOS - in 5 rounds, Android Authority*. Available at: https://www.androidauthority.com/developing-for-android-vs-ios-697304/ (Accessed: 12 November 2022).

Snell, J. (2022) *iPhone (2007) review: A game-changer years in the making, Macworld*. Available at: https://www.macworld.com/article/186335/original-iphone-review-2.html (Accessed: 13 November 2022).

Support Expo React Native Apps in App Center · Issue #189 · microsoft/appcenter (no date) *GitHub*. Available at: https://github.com/microsoft/appcenter/issues/189 (Accessed: 4 February 2023).

Vatne, B. (2021) *Expo managed workflow in 2021, Medium*. Available at: https://blog.expo.dev/expo-managed-workflow-in-2021-5b887bbf7dbb (Accessed: 25 January 2023).

Visual Studio App Center (no date a). Available at: https://appcenter.ms/ (Accessed: 18 December 2022).

Visual Studio App Center (no date b). Available at: https://learn.microsoft.com/en-us/appcenter/distribution/codepush/ (Accessed: 30 December 2022).

Welcome to Bitrise documentation! (no date). Available at: https://devcenter.bitrise.io/ (Accessed: 4 February 2023).

What is cross-platform mobile development? | Kotlin (no date) *Kotlin Help*. Available at: https://kotlinlang.org/docscross-platform-mobile-development.html (Accessed: 19 December 2022).

What is Expo (no date) *Expo Documentation*. Available at: https://docs.expo.dev/introduction/expo (Accessed: 19 December 2022).

Yamashita, A. (2013) ‘How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study’, in *2013 IEEE International Conference on Software Maintenance*. *2013 IEEE International Conference on Software Maintenance (ICSM),* Eindhoven, Netherlands: IEEE, pp. 566–571. Available at: https://doi.org/10.1109/ICSM.2013.97.