Recreating a Native Application in React Native
— Feasibility of Using React Native With Bluetooth &
Background Processing

Oscar Lifh
lifh.oscar@gmail.com
Blekinge Institute of Technology

Petrus Lidholm
petrus.lidholm@outlook.com
Blekinge Institute of Technology

June 4, 2018
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the bachelor degree in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Oscar Lifh
E-mail: lifh.oscar@gmail.com

Petrus Lidholm
E-mail: petrus.lidholm@outlook.com

University advisor:
Emil Folino
Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

Developing apps for both Android and iOS has previously necessitated two different code bases in the platforms' native languages. Creating an app would be a quicker and easier process if they could be written once and run on both platforms, and such solutions have been appearing over the last few years. One of them is React Native, which we will investigate in this paper.

To investigate React Native’s capabilities, we are going to look into the feasibility of porting the Android and iOS versions of an already existing fitness app, developed by a software consulting company. The original app uses Bluetooth-functionality to record users’ heart-rate during exercise. They want to know if they can switch to React Native and a single code-base for future installments, lessening the workload and making it more maintainable.

In order to find out if it is advisable to recreate the app with React Native we attempt a port of the app, looking at performance, functionality, and the code-base. The code-base investigation focuses on what parts can become completely platform independent and where we need to fill in the gaps with code targeted to a certain platform, and how large our port is in relation to the equivalent code in the original app.

We end up finding that performance is severely worse on iOS, with much higher CPU utilization when the Bluetooth functionality is in use. On Android the difference is noticeable but not quite as big. For functionality we could get everything working with a single code-base except for handling Bluetooth while the app is in the background on Android. The code-base is mostly platform independent, and where it is not this is due to differing Bluetooth implementations for the platforms. It is also larger than either of the original apps, but smaller than the two put together.

Lastly, we conclude that React Native has a largely platform-independent code-base, and for simpler apps where less complex functionality is needed we suspect the code-base can be completely platform-independent. The cause and remedies to iOS performance ought to be further investigated, but React Native is capable for this particular use-case.

Keywords: React Native, iOS, Android, Bluetooth
Contents

1 Introduction ................................................. 3
   1.1 Background ........................................... 3
   1.2 Purpose ................................................ 5
   1.3 Scope .................................................. 5

2 Research Questions (RQs) .................................. 6
   2.1 Motivation ............................................. 6

3 Research Methodology ................................. 7
   3.1 Performance ........................................... 7
   3.2 Functionality .......................................... 8
   3.3 Code-Base Differences ............................... 9

4 Literature Review ........................................ 10

5 Results ...................................................... 13
   5.1 Performance (RQ 1a) .................................. 13
   5.2 Functionality (RQ 1b) ............................... 18
   5.3 Code Base Differences (RQ 2) ..................... 21

6 Analysis & Reflection .................................. 23

7 Conclusion ................................................. 29

8 Validity Threats ........................................ 30

9 Future Work .............................................. 31
1 Introduction

Creating a mobile application (app) for both Android and iOS usually involves creating what is essentially two different native apps, one for each platform. The company behind the app we attempt to port to React Native in this report is a software consulting firm with about 1300 employees across various offices in five different countries. They offer testing, quality assurance, development of software, among other things. They also have their own products, like the app we are to examine in this report, and during the development of it they have been made aware of what it is like to work with two code-bases for what is meant to be a largely identical app for two different platforms. The app, which is designed to help motivate patients who have received a prescription to engage in more physical activity, was and is being developed for iOS and Android separately, which in this case has lead to the different versions diverging somewhat in functionality and especially in terms of visuals. Because of this, they want to investigate into possibilities of switching to React Native and a largely shared code base, retaining the same functionality as the two native apps. We attempt to do this by recreating the existing apps with React Native; recreating or “porting” it.

1.1 Background

Previously, releasing an app on both iOS and Android meant developing two separate native versions of the app, with both logic and front-end using each platform’s native languages and technologies—that is Java/Kotlin for Android and Swift/Objective-C for iOS—which are not cross-platform. But keeping two code-bases (or even four if we count the front-end languages used for defining the layout) has some disadvantages. Back-end wise, every bit of functionality to be added needs to be written twice, likely by different developers. If the app owner cares for their product, they also need to test each version, and ensure that it works as intended, as well as the same way across both platforms. The front-end is much the same way, although keeping a similar looking interface between platforms is perhaps not as important as identical functionality. What can be done about this?
With the arrival of cross-platform development alternatives like: *web* apps, which are just a web page accessed through the phone’s browser—no storage space required, and no app updates as all data is fetched from a server through the phone’s browser; *hybrid* apps, also web pages but stored locally and contained within a web view component\(^1\,^2\) in a native app, an example of which is Ionic\(^3\); *cross-compiled* apps, which are written in some other language and then transpiled to the native languages, like C# in the case of Xamarin\(^4\); and lastly, *interpreted* apps, where the code is in a cross-platform interpreted language with some bridge to native APIs, as with Titanium\(^5\) or React Native, the focus of our case study; there seems to be a solution to the segmented code base conundrum—several, in fact.

These cross-platform solutions do have their drawbacks, and no one solution is the best in all regards. But, going by comparisons made by Willocx et al \[^1\], native seems to be the best in all areas regarding performance on Android, followed by *cross-compiled* and *interpreted* neck and neck, but the situation on iOS is more complex. React Native, a newcomer announced in 2015 \[^2\], is one of the *interpreted* solutions, belonging to the same class as one of the better performers on Android. Although not included in the previously mentioned comparisons, it seems to perform almost identically to native apps on iOS and somewhat worse than Android ones \[^3\]. We have chosen to focus solely on React Native; judging by number of installs it is popular at the moment \[^4\] (not all of these are cross-platform solutions; Kotlin is not) and it has promising performance potential.

React Native is broadly used by thousands of apps on the app market, including both well-established companies and smaller start-ups \[^5\]. Skype, Instagram, and Airbnb are just a few of the well-known names that have decided to work with React Native, so there evidently is interest in the framework.

The benefit of using a cross-platform solution is saving time and money; having a single code-base could mean halving the amount of developers and time spent to create an app, and problems with getting the app to behave uniformly across the platforms will not be a problem. Also, JavaScript is one of the most popular languages in the software development world, according to Stack Overflow’s Developer Survey from 2017 \[^6\], so there are likely to be a large amount of developers to choose from when creating an app. If this approach is usable, then a lot of time and money could be saved.

---


\(^3\) Ionic, example of the hybrid approach: [https://ionicframework.com](https://ionicframework.com)

\(^4\) Xamarin, example of the cross-compiled approach: [https://www.xamarin.com](https://www.xamarin.com)

\(^5\) Titanium, example of the interpreted approach: [https://www.appcelerator.com/Titanium/](https://www.appcelerator.com/Titanium/)
1.2 Purpose

Our aim is to produce a guide for those interested—including the company—in moving existing apps to/creating an app from scratch in React Native. It could be used to evaluate the feasibility of creating a React Native app, and get an idea of when it is advisable, and even possible, and when it is not. The immediate value to the customer is that they can use our findings as a basis for deciding whether or not to convert their current app into a React Native one, as well as use React Native for future projects.

Value of Contribution

Most of the value derived from this case study will most certainly be to the company and their product. They will be given a document that will guide them on whether or not their app would be possible on a cross-platform code-base, and with that and the result of this study, the company will be able to make an informed decision regarding React Native.

The study may also contribute to other developers interested in the area. Investigation into specific functionality may be of value to those interested in the feasibility of functional parameters such as background processes or Bluetooth in React Native.

1.3 Scope

We will be looking specifically at what limitations the use of React Native imposes regarding functionality. Seeing as there is an advantage to have as much of the code base be platform-independent, we will also investigate what parts and how much of the code base we can move into JavaScript.

However, recreating the whole application will not be necessary to be able to test the feasibility of migrating the app. We instead has specified four key functionalities in the existing application provided to us to compare with our implementation in React Native: Notifications, Graphs, Bluetooth and Background processes. These four functionalities were carefully chosen as they together form the functionality required for a minimum viable product (MVP) version of the original app.
2 Research Questions (RQs)

These are the research questions we aim to answer through our case study:

1. What implications are there of using React Native as opposed to native apps in regards to:
   (a) Performance
   (b) Functionality, specifically:
      i. Notifications
      ii. Graphs
      iii. Bluetooth
      iv. Background processes

2. How is the code-base affected by using React Native as opposed to the technologies native to Android and iOS?
   (a) Size compared to original app
   (b) Ratio of platform independent code

2.1 Motivation

Performance is to be tested to see if it suffers from using React Native as opposed to native apps. These performance metrics may be of limited use since the numbers depend on both the original implementation and ours; however, they could show that performance comparable to native is possible with React Native and, at the very least, show that React Native can achieve sufficient performance according to the company (a judgment we leave up to them).

By comparing the four main functionalities mentioned in RQ 1b of the already existing app with the one we will developed, we shall be able to test the feasibility of React Native in the perspective of these parameters. Turning our attention to notifications (RQ 1(b)i), in-app notifications: which are created and triggered locally on the device entirely within the app, are present in many apps—including this one. This is why we want to see if and how we can create them with React Native. RQs 1(b)ii and 1(b)iv are closely related as the background process the app uses is to keep a Bluetooth connection and save data received through Bluetooth when the app is in the background.

As a follow up to the implemented functionalities the specific code-bases in React Native will be compared to iOS and Android respectively in RQ 2. By comparing differences in size of code between the different platforms, we shall be able to evaluate if React Native app’s reduces total amount of code written and how much is platform dependent.
3 Research Methodology

By porting the product to React Native, we are going to investigate the areas related to our research questions, namely: how the performance differs, if the key functionalities are possible to recreate (preferably with React Native), and information on the code-base(s) are shaped.

3.1 Performance

Performance will be measured using the built in Apple Instruments\textsuperscript{6} for iOS and the Android Profiler\textsuperscript{7} for Android. These tools provide monitoring for several parameters regarding the performance of the application. We will use the tools as present in Xcode version 9.3 and Android Studio version 3.1.2, and conduct testing on an iPhone SE running iOS 11.2.2, and an Android device Huawei P20 Lite running Android 8.0.0. The React Native version we will be using is 0.52.1. We are to test the following:

- CPU measurements
- Memory measurements

These will be tested on the overview screen which we are to recreate. It allows users to scan for compatible Bluetooth devices, connecting to a chosen device, displaying the current pulse value as well as graphs. We are to monitor CPU usage and memory consumption while the app is in one of two modes:

- Scanning. This occurs on the overview screen when a device is not connected and the user presses a button to begin scanning for devices. Found devices are added to a list of available devices. Measurements can start as soon as the user have pressed the button to start scanning.

- Connected. After choosing a device, the app will attempt to connect to it. This may take some time, and we will begin taking measurements when the device is connected and we start receiving pulse values.


\textsuperscript{7}Android Profiler: https://developer.android.com/studio/profile/android-profiler.html
For both of these modes, we will take measurements when the app is in the *foreground*, that is it is visible on screen and active; and when it is in the *background*, which means the app is not visible, either because some other app is open, no app is open, or the screen is turned off. The combination of these two modes and two states leaves us with a total of four scenarios which we refer to as: Foreground Scanning, Foreground Connected, Background Scanning, and Background Connected. These scenarios were decided by us.

For the scanning scenarios, we predict both CPU usage and memory consumption will be largely stable, and thus we can simply take measurements as soon as the usage stabilizes. The connected scenarios will likely be stable but exhibit peaks when pulse values are taken in, since these are processed and stored in memory, which means we will have to look at the average usage, with peaks taken into account.

The process of gathering measurements for any scenario is to start the app, wait until everything is loaded and performance is stable, then trigger the scenario, sampling the metrics for CPU usage and memory consumption over a minute. We then calculate the average usage over it.

The scenarios will be run on both the original native apps, as well as our recreated React Native version. We will then compare the two, to get an idea on how performance differs in our port. This does not give a definitive answer about how React Native affects performance, but it could give us an idea if React Native is suitable and is capable of performing on par with the original native apps.

### 3.2 Functionality

For most (if not all) use-cases, getting the functionality you want using a combination of native and React Native code is most likely possible—what is interesting is to see how much can be accomplished with React Native alone. We investigate this by porting the app to React Native and seeing where we can achieve the same functionality without native code.

In contrast with performance, where we can simply compare numbers, the findings here will be somewhat less defined; we can say with certainty whether or not one number is greater than another, but a statement on whether or not a functionality is possible with React Native is more vulnerable to ignorance. We may come to situations were we find that we could not solve it using React Native, whilst (unbeknownst to us) there was a solution. In these cases, where a question of a functionality’s possibility is answered in the negative, we shall be clear why it is so, and preferably provide sources.
3.3 Code-Base Differences

For code base size comparisons (RQ 2a), we will be taking each component from the original code that has a representation in our JavaScript code, and compare the number of lines. To avoid size differences caused by code style and documentation, we will not count: comments (whether single line or block); import statements; and lines that contain nothing but white space and opening/closing brackets, parentheses, or end of line characters, alone or in combination. Examples of lines that will not count towards the total are:

- `{}
- `}
- `)
- `);`  
- `// This is a comment`
- `/* This is a comment that spans multiple lines */`

We call the function of calculating the number of lines of code for a given code base LoC.

For the ratio of platform independent code (RQ 2b) we will count the lines in the same manner as for size comparisons and then simply calculate the ratio,

\[ r = \frac{\text{LoC}(\text{independent})}{\text{LoC}(\text{all})}. \]
4 Literature Review

There are several cases of studies [3], [7], [8] regarding the topic we are to investigate. The listed studies involve comparisons between a React Native app recreated from an earlier Android version. But what they all have in common is that the recreated app is of a very basic standard, meaning one that does not take advantage of native app technologies, such as sensors or Bluetooth communication.

This is why we are aiming to investigate the possibility of recreating more complex app with native app technologies involved. Due to the nature of our investigation (that is, finding out what functionality is possible), we need to—in addition to searching for research papers and articles—dive into documentations and packages to answer our questions.

Performance

As mentioned, the performance comparisons we have seen [1], [3] indicate that React Native is worse than native on Android and comparable to it on iOS. In these cases, as well as our own, the measurements were made with a certain app, and as such are not general. We do not yet know how the app’s performance will be as a React Native app—that will be part of the investigation.

Notifications

The React Native documentation only provides information regarding how to set up notifications for iOS apps with the PushNotificationIOS component (which, despite its name, also supports local in-app notifications) [9], with no Android-equivalent documentation. However there are packages that do support both platforms. The react-native-push-notifications package [10] utilizes the built in PushNotificationIOS package that follows React Native for the iOS part, and has its own implementation for the Android part. Another package, react-native-notifications [11], on the other hand has a fully custom implementation for both platforms. The key difference is that even though both packages allow to write JavaScript, only react-native-push-notifications offers creating notifications with platform independent functions—compared to the other where one has to set up notifications separately for each platform. From package documentation it appears that the react-native-notifications package allows for more customization, making use of the specifics of each platforms notification system.
Graphs

the app uses various graphs to visualize user activity. These can be zoomed in and scrolled through using touch, and while there are packages to display graphs\textsuperscript{12, 13} we have not found any literature proving support of interactions in either of them. We have found a package \textit{React Native Charts Wrapper}\textsuperscript{14} that acts as a wrapper to MPAndroidChart\textsuperscript{8} and Charts\textsuperscript{9} (which is an iOS implementation of the former), the same packages used in the native apps. We do not yet know if this wrapper is complete and has these interactive capabilities—that will be part of the investigation.

Bluetooth

Both Android and iOS has built in support for Bluetooth Low Energy (BLE) in pure native apps\textsuperscript{10, 11}. With an application developed in a cross-platform framework like React Native the built in support for BLE does not exist. Although React Native does not provide any Bluetooth functionality out of the box, there are packages that do, such as \textit{ble-plx}\textsuperscript{15}. \textit{ble-plx} has a \textit{BleManager}\textsuperscript{16} with support for listening to Bluetooth events whilst the app is in the background. This may or may not be enough to keep the Bluetooth connection whilst the app is in the background.

Background Processes

Collecting heart-rate data while the app is in the background (which is to say the user has exited it or opened up some other app) is also something that is required for the app. We suspect \textit{ble-plx} may be sufficient for our use case, but we still want to find out how background processes—those that run whilst the app is in background-mode and not visible to the user—work on the platforms.

Both Android and iOS are somewhat careless, and can decide to shut down apps when they enter the background. You as a developer have no say in this—whether your app lives or dies is up to the operating system on your device\textsuperscript{17, 18}. There exist cases where background processing is needed, and both platforms support background processing of certain types, but their approaches are very different.

\textsuperscript{8}MPAndroidChart https://github.com/PhilJay/MPAndroidChart
\textsuperscript{9}Charts https://github.com/danielgindi/Charts
\textsuperscript{10}Bluetooth Low Energy: https://developer.android.com/guide/topics/connectivity/bluetooth-le.html
\textsuperscript{11}Bluetooth for Developers: https://developer.apple.com/bluetooth/
Android uses what is called a service [19]. Services let you have long-running operations that do not provide a user interface. These services will continue to run even if the app is in the background. There are two types of services: foreground and background (the app uses the background variety, which should work in this case). Despite the names they can both run whether or not the app is in the background. Creating a service would mean holding the connection in native code, and we would like to have as much of the code in React Native. Therefore, the ideal way would be to communicate with React Native, and handle events there. For some events, like when we want to save pulse values, we need to be able to do this when the app is in the background. React Native allows this through Headless JS [20], and this seems to be the way to go.

There are other packages that allow tasks to be run in the background on Android [21], [22] but these are limited to running every 15 minutes and for a limited time. The app continuously makes measurements, so these are not applicable. This seems like an instance where we need to implement a independent solution for each platform.

On iOS, there are several background modes supported [23]:

- apps with shorter tasks that should not be interrupted has the possibility to get some background time
- apps that initiate downloads in foreground can hand over download management to the system
- apps that need to support certain tasks continuously in the background, and uses one of the accepted tasks provided by Apple, including location, audio, remote notifications, etc.

Fortunately BLE is supported in the third category, since the app works with a Bluetooth accessory that needs to deliver updates on a regular basis through the Core Bluetooth framework [23]. In order to let the OS know about use of BLE one has to carefully configure it within the apps info.plist file. Nothing else is necessary and the application will continue running in background as long as it receives Bluetooth updates. This likely means we can keep the connection upon exiting the app, without much more than a small change in a configuration file.
5 Results

In our literature review we found packages that seemingly provided the functionality we needed, without the need for native code, although there were instances that we thought would require it. So how did it actually end up being? Let us go through the research questions one at a time.

5.1 Performance (RQ 1a)

How well did React Native perform in terms of memory consumption and CPU usage towards the two native apps? Could React Native keep pace with native, or is performance negatively affected? Let us look at our measurements for the different scenarios as defined in section 3.1. As we will see, performance will be significantly worse on iOS when connected to a Bluetooth-device. This might be due to our usage of the chosen package (ble-plx v0.8.0), or our usage of it.

We ended up measuring over 3 runs representing a minute of data. We then calculated the average usage over a minute of time, without any user interaction; the app was left “idle”. There was no real idea behind how many iterations of tests that had to be run, our reasoning behind 3 runs was mainly because the results where stable and almost identical every run. This may be a validity threat as the precision of the results could be bad.
Our performance measurements for iOS revealed an alarming increase of both memory consumption and CPU usage. With an increased memory consumption of up to ~203% and CPU usage up to ~54%.

Measurements have shown that the native app (written in swift) has a better overall performance of the two apps regarding memory consumption. It was expected, based on the literature review made on a performance comparison by Willocx et al. [1] that native would do better regarding memory consumption, but it was not expected to differ at these rates. Their metrics on an app built for iOS with a framework called Titanium resulted in a 16.56 MB ≈ 129% increased consumption of an idling device (high-end) in comparison to our React Native of 24.2 MB ≈ 162% increased consumption, see figure 1. For a processing device with the same app measurements gave 20.09 MB ≈ 125% increased consumption in comparison 25.1 MB ≈ 203% increased consumption on our implementation respectively. Titanium is perhaps not fair to compare against as it is a totally different framework, but as mentioned in the literature review it does use the same technical solution as React Native, resulting in a broader knowledge regarding feasibility of the interpreted technique overall.
Measuring CPU usage has lead to results of interest, specifically for React Native. As soon as Bluetooth is being utilized, overall CPU usage increases to appalling numbers, see figure 2. Normally, CPU is used whilst the app is launching, or going into idle mode—and not whilst the app has been processing for a while [1] which is indicated by the 0% usage in figure 2.

CPU usage is expensive, as it draws power which causes severe battery drain. In fact, an idling iOS app has 10 times greater power draw over sleeping apps, whilst 10% CPU usage has an impact of 2 times greater power draw on idling apps, followed by 10 times greater power draw on idling when it reaches 100% CPU usage [24]. This is worrying as our implementation with React Native reaches up to 54% CPU usage on average Bluetooth communication. This may be a result of our lack of knowledge regarding the package ble-plx and how it should correctly be implemented to optimize performance. There is also a possibility that the package per se is not well optimized, which could be a candidate for future work to investigate into.
The performance on Android is another story. Our measurements revealed a performance more in line with the original app, even outperforming it in terms of lower memory consumption.

![Memory usage on Android](image)

The results of memory measurements on Android, see figure 3, show that memory consumption in our app is actually lower than in the native one, which may be explained by the native app having a few more features and thus more to keep track of. *We want to note that the Background scanning scenario does not really occur in our app, and it is not actually at 0 MB of memory usage as indicated in the graph.*

We noticed during measurements that the reported memory consumption steadily climbs, with a tenth of a megabyte every other second. This happens with both the port and the original app, and while we are not sure why the memory climbs like this, we know that it is either a problem present in both apps, or it likely has to do something with the profiler (perhaps it is keeping logs on the device’s memory).
In terms of CPU usage, our React Native app’s performance is much closer to native on Android, with a difference of at most 4.5 percentage units between the two, see figure 4. We found no reliable way to get figures for CPU usage over a set period of time, so the figures are approximate readings by us based on data extracted via Android Profiler.
5.2 Functionality (RQ 1b)

Was it possible to get it working identically to original app regarding functionality, or is there something missing?

Notifications (RQ 1(b)i)

As stated in the literature review, React Native does not support notifications for Android, a package is required to get notifications on both platforms. As the app only utilizes basic notifications, we decided to go with the less advanced package named `rn-push-notifications` [10] as it—based on the literature review—has advantage regarding platform independent code, and we see no necessities in the more advanced package at the moment.

For Android, to present local notifications whilst the app is in background, one has to implement it either by creating a local notification from a Headless JS Task [20], or to schedule the notification to a specific time or interval. Scheduling notifications is supported by Android, if the correct configuration has been added to Android Manifest [25].

For iOS, the package uses the implementation of notifications that follows along with react-native core [9]. Some configuration to AppDelegate file in the iOS specific code allows for sending notifications, but to allow for background notifications one has to activate notification background execution [23].

Graphs (RQ 1(b)ii)

The aforementioned React Native Charts Wrapper [14] was sufficient for our use cases. We were able to get the same graphs, with largely the same look and interactivity as the ones in the original app. As a consequence of the library being based on the same libraries used in the app, most of the attributes to modify the graph shared almost identical names, making styling a process of simply going through the source code and copying and pasting, with minor adjustments. The only difference appearance-wise is that we have not managed to get a bar chart to show the axis and lines when no data is present. We are not sure if this is something we have missed when configuring the graph or something that is missing from the library. In our case it is not important to the customer (and a compromise we think most would be willing to accept).
Bluetooth (RQ 1(b)iii)

We thought based on the literature review, that the **ble-plx** package [15] might be all we needed for our Bluetooth needs on both platforms: scanning for devices, connecting, reconnecting, subscribing to heart beat updates. This turned out to be true, and you can use it for this—unless you want to keep your connection and subscriptions when the app is no longer in the foreground on Android. The **ble-plx** package does have support for “background mode” [20], which presumably refers to iOS’s “background execution”\(^{12}\); we did manage to keep the connection going on iOS. We could not get it to work on Android however, and it seems the developers are aware of this as they mention that it is “unlikely” to get it to work without using native code [27].

In short, foreground Bluetooth functionality is available without the use of native code on Android, and thus we needed to find another solution. We could—and do—use it for both background and foreground on iOS.

Background processes (RQ 1(b)iv)

With **ble-plx** unable to keep the connection going in the background for us on Android, we had to find another solution. Thus, in order to keep listening to the Bluetooth band we had to start an Android service, which in turn requires native code for Android. Originally, we thought a background service would do, but we ran into trouble on some devices, as the service would stop after around 30 minutes. As it turns out, Android 8.0 brought about some limitations to background services [28], and we made the switch to a foreground service. Using this, we could continue to monitor the Bluetooth device even if Android kills our activity/app, and the service restarts independent of the app’s UI.

In order to communicate Bluetooth events to React Native, we emit our own events from the native code that we subscribe to in React Native. Receiving these events only work when in the foreground, so any updates other than to the UI, such as saving pulse values to a database, either need to be done in native code, or—as we have done it—with **Headless JS**.

For iOS we could simply use **ble-plx** as is, without native code. We still have to make some adjustments for when the app goes to the background, which was expected based on the literature review, as iOS does support background processing of BLE data if it uses the Core Bluetooth Framework [23]. Fortunately the **ble-plx** is based on **RxBluetoothKit** which is backed by RxSwift and CoreBluetooth [29].

---

\(^{12}\)Background Execution iOS [https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html](https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html)
So, we effectively have two code bases for this functionality, but there are advantages: we do not need to maintain two sets of native code, and also, if ble-plx gets updated to handle the service situation for us on Android, we could simply update the JavaScript code and remove the native Java code. It would be favorable to use ble-plx in the end as it has an already implemented support for both platforms regarding the connection and handling of Bluetooth communication. The problem is that the package is not fully implemented with full support and cannot be used at the moment. This is a common problem with packages for React Native. It is limiting regarding customization, but allows for less code duplication.
5.3 Code Base Differences (RQ 2)

One of the problems React Native promises to resolve is the separate code bases for the different platforms. We want to know if unifying the code bases makes for a bloated code base, and how large it is compared to the combined sizes of the platform specific code bases. We also want to know the ratio of platform independent code to platform dependent code, to see how much of the code still needed to be platform specific.

Size Difference (RQ 2a)

We will first look at the entire React Native code base and compare its size to the respective code for the different native platforms.

![Graph over code base sizes](image)

Figure 5: Graph over code base sizes. The Android and iOS figures are taken from those parts of the original apps’ code who has some equivalent in our port.

From figure 5, we can tell that the individual code bases for the original apps are slightly smaller than the React Native code base. The React Native code base is smaller than the two combined however, at ~ 60% of their combined size.
Ratio of Platform Independent Code (RQ 2b)

How much of the ported app’s code ended up being platform independent, and how much of it is code targeted to Android or iOS?

![Code Sizes](image)

Figure 6: Code sizes from our ported version of the app. The “Platform independent” bar represents the total React Native code from the project, excluding the code that targets a specific platform. We separate code based on the targeted platform, not the language it’s written in; for the Android-targeted code we had to use Java (and an insignificant amount of React Native to tie it together), but for the iOS equivalent we used solely React Native.

As figure 6 shows, the majority of code is platform independent, with the total 1814 being distributed as follows: 1249 lines of platform independent JavaScript code, 440 lines of Android-targeted Java code, and 125 lines of iOS-targeted JavaScript code. On Android we had to both write native code and construct a bridge to it, whilst for iOS we simply used a package. For this reason, the iOS-targeted code is much smaller than the Android-targeted equivalent. The ratio of platform independent code $r$ as described in the research methodology in section 3.3 is then

$$r = \frac{\text{LoC}(\text{independent})}{\text{LoC}(\text{all})} = \frac{1249}{1814} \approx 69\%.$$ 

All of the code that targets a specific platform are due to the different approaches we had to take for Bluetooth to work (see 5.2).
6  Analysis & Reflection

More than just stating the results, we would like to analyze and reflect on them, as well as discuss some insights we gained that were not necessarily tied to any research question.

Functionality

We managed to get all functionality researched to work in React Native. But there is some parts of interest that should be addressed.

Bluetooth

For this particular research we tried to implement Bluetooth with a package labeled ble-plx. We had to implement Bluetooth in native code for Android, as the package was incapable of handling Bluetooth in background mode on Android, resulting in a platform-specific solution for each platform. As we encountered huge performance issues with Bluetooth for iOS and with already platform specific solution, it is perhaps better to implement a native solution for iOS as well. This assumption is not representative for all React Native packages, but carefully overlooking performance whilst using other packages is preferred.

Notifications

The choice of package is not a simple decision to face, as the two packages mentioned in the literature review is suited for different use cases. If an app would use basic notifications without platform-specific customization, it is perhaps easier to go with the more basic package of the two. It is very minimal, with a few platform specific parameters, which do not affect each other, and could be included in the same function call for both platforms.
Performance

Going from the literature review and especially comparisons by Hansson and Vidhäll [3] we expected the app to perform worse on Android than on iOS compared to the performance of the original native apps. For us, this was the other way around. Measurements shown that our React Native app for iOS increases memory consumption over the native one of up to 203%. Our numbers are greater than measurements we found in reviewed literature, as their results shown that the iOS version did perform equally to the native one. We assume, based on the results shown in figure 1 and 2 that our implementation of Bluetooth and the package ble-plx is the cause of this, as the performance issue only occurs whilst connected to the Bluetooth smartband.

For Android, memory consumption is lower with our React Native port than that of the original app. As we briefly mentioned in the research methodology (see 3.1), these values may be misleading and explained by the latter having more going on under the hood, rather than our port innately using less memory. The figures we found on CPU usage are also much better than those on iOS, being at most ~ 4.5 percentage units above the native Android app.

Code-base Differences

We did get a lot of platform-specific code, but this was solely due to supporting running things (in this case Bluetooth) in the background on Android. Everything else: Bluetooth in the foreground, graphs, fetching data through REST-calls and notifications was possible with only React Native and existing packages. The results we got from measuring the code-base of our app may not be representative of what a ratio between platform-specific and platform-independent code would be for the average app.

If you consider creating a React Native app then you may be able to get away with no native and platform-specific code as long as you are fine only executing code when the app is in foreground. If you already have an app for one or both platforms and you need background execution, — or some other functionality that requires native code—then chances are you can re-use parts of the code you already have, and tie it together with the shared React Native code-base. Developers might find they need to write native code as they start to add certain functionality. We advise you search for packages (in addition to the documentation) to see what functionalities are available without having to write it themselves. If a functionality would need a native implementation and re-use is an option, then the difficulty of re-use depends on how coupled those sections of code are to the rest of the native code-base, so it is not necessarily an easy process. In our case there was some struggle to remove what we did not need or could not use and get it to work correctly.
An overabundance of packages (and the lack of functionality out-of-the-box)

You may have noticed that we have mentioned a lot of different packages. We have found the vast collection of packages very helpful, but they may be symptomatic of a problem with React Native—the lack of expected functionality available with React Native alone. In native application development you have certain functionality available from the get-go in the development kit: for instance both Android’s SDK and iOS’s App Services provide navigation and notifications. But with React Native you have to find packages to get these very basic and, for many if not most applications, integral functions. If you need to use some feature not provided directly by React Native you often have multiple packages to choose from, and no “official” solution. Since these packages are available by the generosity of other people (who may be anyone from an employee at a software company to a lone coder) who are not always required to follow certain standards, the quality of the code and the availability of documentation varies widely between packages. It is not reasonable to expect React Native to provide solutions for every need, but we did expect that it had basic things like notifications and navigation already built in. As it stands, the lack of these features makes us feel like React Native is not as mature a tool as we initially thought.

The need for Android and iOS competence

Developing even a very basic app will likely make you come in contact with either iOS or Android. Installing packages often require you to make changes in the build configuration files for both platforms, as well as writing code. For Android, generating a production version of your application also requires changes in build configuration files. If you are developing an app which targets iOS, then the same requirements for regular native iOS development need to be fulfilled—the app can only be built on a Mac.

We have experienced problems regarding services on Android, as we lacked any knowledge or experience. This has brought us some pain, and although we did have access to the source code, it had other dependencies on Java code that we had to remove, so getting the same functionality has not been straightforward. More Android experience would certainly have made the initial period of development smoother, due to our somewhat advanced use-case.

For simpler use-cases, we do think one can rely almost solely on React Native, but you will most likely need to have people with Android and iOS experience at hand (if you are targeting both platforms). For iOS development you also need a Mac. We feel all of this slightly diminishes the appeal of React Native.
Messy Development Process

At a first glance it may appear to developers as an easy task getting started in React Native. Documentation is easy to follow along and there are not a lot of steps involved, but it has some drawbacks. The command line interface for the most basic setup, `Create-React-Native-App`, has a dependency towards `Expo`, an application for wrapping your developed application within it. Which makes it possible to be run cross platform. This is fine for simple applications that does not use any native packages or dependencies. Facebook’s Documentation on getting started describe this as follows.

“Create React Native App is the easiest way to start building a new React Native application.” - Facebook

However, problems start to shine through React Native’s well polished guide with descriptions mentioning the “easiest way” to start building applications, as it is not applicable to a wide variety of apps. Instead, another set of more advanced instructions are provided in order to build projects with native code. A key difference in these instructions is the need of installing `Android Studio` and `Xcode` in order to build for Android and iOS respectively. This leads to having three Integrated Development Environments (IDEs) running in order to fully develop React Native. It is worth mentioning that running native IDEs is only necessary whilst developing native extensions, and not for development with JavaScript, but they still have to be installed in order to build the apps.

As if three IDEs would not be enough of a mess, React Native also ships with `Chrome Developer Tools`, which allows to debug the running application within a browser, causing developers to have to navigate to a browser as well. Fortunately, there are plug-ins for the most popular JavaScript IDEs to allow debugging within them.

Business Value

As we have seen, the total code-base is smaller with our React Native app than the two original apps combined. Being able to recreate both apps with less code leads us to believe the whole app could be recreated/ported with fewer lines of code than the total of the two apps, which should mean it would be faster to develop with React Native. We also do not have two code-bases, which each have to be tested and controlled to see that they behave the same way, further decreasing the time spent on development and quality assurance.
Answering the research questions

Based on our results and our analysis, these are our conclusions about each research question:

1. What implications are there of using React Native as opposed to native apps in regards to:

   (a) **Performance.** The CPU usage is higher (which is worse) on both platforms for the React Native app compared to the original apps, and the effect is most apparent on iOS. Since the severe increases of CPU usage are in the scenarios where a device is connected, we think that the cause lies somewhere in the package we used or in our usage of it.

   For memory, the consumption is higher in all scenarios on iOS. It was expected—based on the literature review—that the native implementation would do better regarding memory consumption, but it was not expected to differ at these rates.

   On Android, we see decreased memory consumption compared to the app. This goes against what the literature review lead us to believe, but it may be explained by the original app having functionality that our port does not. With a little more going on, it is not strange to see a bit more memory being used, so we think that memory consumption is comparable to the original native app on Android when using React Native.

   (b) **Functionality, specifically:**

   i. **Notifications.** We were surprised by the lack built-in support for notifications, which is only available for iOS. We did find packages of varying complexity, with a simpler one being tested by us. We know that basic in-app notifications are possible, and from studying the packages we conclude notifications that make use of platform-specific notification configurations are possible through the use of these packages.

   ii. **Graphs.** We found a package that acted as a bridge to the original packages being used in the original apps. We could then recreate the graphs with the same look and interactivity as in the original apps.

   iii. **Bluetooth.** Bluetooth support is not built-in and we had to use a package for it. This package worked for both platforms when the app was in the foreground.
iv. **Background processes.** On Android you can use packages that allow shorter tasks to be run with a 15 minute interval. This was not often enough for us, as we needed to keep the Bluetooth device connected and continuously listen for pulse values. We came to the conclusion that we had to create an Android **service**, with the use of native code, and send messages to React Native about the connection state and pulse values.

On iOS, background processing of several predefined types are supported if they fulfill specific requirements for each. Bluetooth for example, has to use the Core Bluetooth Framework provided by Apple Inc. **ble-plx** does use it in the background and is therefore supported regarding background processing of Bluetooth connection.

2. How is the code-base affected by using React Native as opposed to the technologies native to Android and iOS?

(a) **Size compared to original app.** We ended up with an app \( \sim 60\% \) the size of the equivalent parts of the native apps combined. It is thus smaller than the total code-base of the two apps, although slightly larger than each of the native code-bases by themselves.

(b) **Ratio of platform independent code.** Solely due to the Bluetooth implementations we had to write code that was not cross-platform. Because of this, we have \( \sim 70\% \) of the code base that is platform independent.
7 Conclusion

We have seen that a React Native app running on Android performs comparably to the native app in terms of CPU-usage and memory consumption, whilst the iOS version is much worse than the native app. CPU-usage on iOS reaches up to $\sim 50\%$ with React Native compared to the native app’s $\sim 0\%$, and significantly increased memory consumption, the reason for which may be that the package itself is poorly optimized, or our lack of knowledge on how to utilize it properly.

For functionality, we were able to get everything but Bluetooth working without native code on Android, and even then it was only when we needed to keep a connection in the background. For the same problem on iOS we used a targeted approach but without the use of native code. For most use-cases then, where your app does not need to do advanced tasks more than fetching, sending, and displaying data, we suspect you can get away without any native code.

Compared to the original native apps, the code base is slightly larger than either one of the native app code bases on their own, but smaller than the two put together ($\sim 60\%$). The ratio of our React Native port’s code that is platform independent is about $70\%$, and the reason for it not being $100\%$ is solely due to having to implement a Bluetooth solution using native code for Android, the equivalent code for iOS uses no native code. The indication is that switching to React Native could decrease development time and thus save money.

In the end, we could recreate the app in React Native; CPU usage was higher, especially on iOS, which is related to the Bluetooth part, and the memory consumption was about double that of the original on iOS, whilst being comparable and slightly lower on Android. Regarding functionality React Native is capable for our use-case—it just needs some help from native code. Our conclusion is then that—for the considerations discussed in this report and given that the performance issues for iOS could be resolved—we feel switching to React Native is feasible and would decrease development time, which would benefit the company.
8 Validity Threats

Here we list possible faults with our investigation.

Ignorance of available solutions

We ended up implementing one Android and one iOS-targeted approach for Bluetooth as we did not find any suiting packages that could manage our use case. Perhaps there is some package that could solve this for us with one set of platform independent JavaScript code. We cannot be sure that we have researched all packages available.

Slightly incorrect performance measurements

Whilst measuring performance for the two platforms, we used Android Profiler for Android and Apple Instruments for iOS. We had no previous experience with these tools and had a hard time getting the tools correctly configured and properly working for our test cases.

Another important part to mention is the number of runs we did for each performance metric. We decided to do the metrics with 3 runs per case representing a minute of sampling data each. This was solely due to results being—as anticipated—largely stable. Which may have resulted in poor precision of data.

The performance measurements may also point in favor of React Native as we did not implement the app fully, which resulted in the Native apps doing slightly more work. We cannot be sure that these smaller functionalities missing is not affecting the results in a bad way.

Possibly bad iOS Bluetooth implementation

Poor performance on iOS leads us to believe that we might utilize the ble-plx package in a bad way. We did not have the time to become experts on the package used for Bluetooth, which may have caused us to utilize the available functionalities provided by the package in a way that is not optimal. This could be interesting to investigate further into
9 Future Work

We have almost exclusively covered only that which was of relevance to the app we set out to create. But there are a lot more questions to answer regarding React Native.

Other functionality

As for now, finding out if your use-case requires native code or not is something one had to investigate oneself (apart from the functionality we have already covered). For those interested in using React Native it would be convenient to have a document with the common functionality listed, and whether or not it is available without native code, be it with packages or React Native itself. Examples of such functionality are gyroscope, camera access, and sound playback.

Bluetooth for iOS

As we have seen some negative results on our performance tests regarding iOS and the ble-plx package that is being used, there is a lot of questions regarding the package used. We cannot be sure that our implementation of the Bluetooth for iOS is written correctly from a performance perspective. Therefore, performance testing of ble-plx would be interesting to see.

Comparison between the package and a natively implemented Bluetooth solution would be interesting as well, to be able to see whether the package is the issue behind the lack of performance. And whether or not React Native really is capable of handling bluetooth for iOS.
References


