Linköpings universitet | Institutionen för datavetenskap
Examensarbete på avancerad nivå, 30hp | Datateknik
2016 | LIU-IDA/LITH-EX-A--16/050--SE

React Native application development

– A comparison between native Android and React Native

William Danielsson

Handledare : Anders Fröberg
Examinator : Erik Berglund

LINKÖPINGS UNIVERSITET

Linköpings universitet
SE–581 83 Linköping
013-28 10 00 , www.liu.se

Abstract

Creating a mobile application often requires the developers to create one for Android och one for iOS, the two leading operating systems for mobile devices. The two applications may have the same layout and logic but several components of the user interface (UI) will differ and the applications themselves need to be developed in two different languages. This process is gruesome since it is time consuming to create two applications and it requires two different sets of knowledge. There have been attempts to create techniques, services or frameworks in order to solve this problem but these hybrids have not been able to provide a native feeling of the resulting applications.

This thesis has evaluated the newly released framework React Native that can create both iOS and Android applications by compiling the code written in React. The resulting applications can share code and consists of the UI components which are unique for each platform. The thesis focused on Android and tried to replicate an existing Android application in order to measure user experience and performance. The result was surprisingly positive for React Native as some user could not tell the two applications apart and nearly all users did not mind using a React Native application. The performance evaluation measured GPU frequency, CPU load, memory usage and power consumption. Nearly all measurements displayed a performance advantage for the Android application but the differences were not protruding.

The overall experience is that React Native a very interesting framework that can simplify the development process for mobile applications to a high degree. As long as the application itself is not too complex, the development is uncomplicated and one is able to create an application in very short time and be compiled to both Android and iOS.

First of all I would like to express my deepest gratitude for Valtech who aided me throughout the whole thesis with books, tools and knowledge. They supplied me with two very competent consultants Alexander Lindholm and Tomas Tunström which made it possible for me to bounce off ideas and in the end having a great thesis. Furthermore, a big thanks to the other students at Talangprogrammet who have supported each other and me during this period of time and made it fun even when it was as most tiresome.

Furthermore I would like to thank my examiner Erik Berglund at Linköpings university who has guided me these last months and provided with insightful comments regarding the paper.

Ultimately I would like to thank my family who have always been there to support me and especially my little brother who is my main motivation in life.

Contents

List of Figures

1 Introduction

Since the release of the first smart phone, the use and demand of mobile applications has increased rapidly. Many businesses have been established by providing a service through an application but some businesses want an application in order to prove that they are contemporary or because their competitors have one. This has led to a huge amount of applications being created but there is one significant problem, the app needs to be supported by both Android and iOS. Even though the application itself is the same, the developers still need to develop two applications which requires unnecessary time and skills. In the summer of 2015, Facebook released a framework called React Native which is used to build an application in React and then compile it to either Android or iOS.

1.1 Motivation

This project and thesis has been performed in cooperation with Valtech Sweden as a part of their Talent program. It is very common for Valtech Sweden to create web services for their customers and often the customer also desires a mobile application. This results in Valtech developing a web application together with two mobile applications, one for Android and one for iOS. Even though this is not a serious problem for Valtech to provide for their costumers due to their wide range of expertise in different areas, it is still a problem. If a customer wants to have a website and an application for iOS and Android, the developers need to have knowledge in both web development, Swift which is the programing language for iOS and Java which is used in developing Android. This would most likely result in three different teams, a costly setup for the customer and Valtech would need to have available consultants in all these areas. If Valtech and other developers would begin to use React Native, they would only need one team that can create all three services with more or less the same code. This would shorten the developing time enormously and furthermore they would only need to hire developers with expertise in React.

1.1.1 Valtech

Valtech is a global IT consultant company with around 1800 employess and was founded in 1993 in France. Valtech Sweden is Valtech's subsidiary in Sweden with 258 employees and their main office is in Stockholm. Valtech strives to be a digital partner and combines

creativity, technique and a flexible product development strategy. The Talent program offers an internship containing education and an 8-week project where the talents experiences the life as a real consultant. Following this program, the students are allowed to perform their thesis at Valtech.

1.2 Research questions

Since the announcement of React Native in 2015, some blogposts and tutorials have been posted online in order to help developers create applications using the framework. However, the amount of articles about React Native is almost nonexistent and most of the available information originates from Facebook themselves. The motivation of this thesis is to evaluate React Native and compare it to Android in terms of development, user experience and performance, which leads to the following questions:

1. How difficult is it and does React Native have the support to develop an application that is indistinguishable from an Android application?
2. How well does an application created in React Native perform compared to a native application in Android?

The first part of the first question is focused on the implementation and development of the application to be created. This can not be answered in numbers but with the experience obtained throughout the development phase and the end-result when the period of development is done. The second part will evaluate the ability of replicating an application in React Native and how well the application performs regarding achieving the same look and feel as a native application. The process for evaluation this aspect is further explained in section 3.3.1. The second question will be answered by measuring different aspects of the two applications once they have been finished. Even though there are many ways of measuring performance, this thesis will focus on the ones mentioned in 3.3.2.

1.3 Aim

The aim of this thesis is to evaluate the React Native framework and how well the applications created by the framework correlates to natives. By developing an application in React Native, the thesis will evaluate the development potentiality and the simplicity for beginners to create their own app. Moreover, this paper will analyse the performance of a React Native application by comparing it to a similar one in Android. The result of these two aspects will result in a solid and fair conclusion whether or not React Native is a framework worth investing in or if it is not able to replace writing application in the native approach.

1.4 Delimitations

Performing a complete evaluation of a framework is a large task, especially with React Native that is able to compile to both Android and iOS. Due to the limited time, this thesis will only focus on the comparison of Android due to previous knowledge in developing for Android even though the support for iOS in React Native is more complete at this time. Furthermore there are many aspects of how one could evaluate a framework, some of these are mentioned in 1.5, but this thesis will focus on the two questions listed in 1.2. The paper will investigate the possibilities in developing an application using React Native by trying to recreate an existing Android application. Instead of spending time developing a new application in Android, an already existing app will be obtained. The obtained application will be simple, in reasonable size for one to be able to recreate it and not performance optimised in order to

retrieve a just result. Furthermore the two applications will be compared in means of performance. There are innumerable ways of measuring performance of an application but this thesis will examine the GPU frequency, CPU load, memory usage and power consumption in a similar way as Arnesson[2].

1.5 Related Work

As earlier mentioned, the amount of academic reports about React Native is nonexistent despite the framework was released nearly a year ago. Since it is a new framework, it is understandable that there will be a shortage of books regarding the subject and it is mostly through the documentation and blogposts where one can obtain tutorials and knowledge about React Native. However, even though there are no previous work about React Native itself, there are a lot of articles related in terms of similarities of evaluating a hybrid framework and how to perform the evaluation.

Arnesson published a paper[2] in 2015 and compares native Android to two hybrid frameworks, Codename One and PhoneGap. Arnesson created a similar application using the three different methods and evaluated the performance of the three applications. The application performed different sorting algorithms, wrote and read from a database, created a list and sorted that list and finally it used the GPS to determine the position of the user. All of these functionalities were divided into activities and Arnesson used the tools PowerTutor[38] and Trepn Profiler 5.1 to measure different aspects of the application. By measuring the CPU load, memory use, application size, energy consumption and execution time, Arnesson was able to perform a thorough test and obtained a fair and complete result.

Moreover, there are other ways of comparing frameworks and Sommer compared different frameworks with native by using the FURPS model[14]. Sommer evaluated the application created by the framework and the framework itself on five attributes; Functionality, Usability, Reliability, Performance, and Supportability. Each attribute were given a rating from 1 to 5 and resulted in the framework receiving an average score. Even though this is an effective way of evaluating a framework on more than solely performance, the values are given by Sommer himself which may result in a biased outcome.[33]

In *Experimental Comparison of Hybrid and Native Applications for Mobile Systems*[19], Seung-Ho Lim compares a native application to a web application by creating a social network service and focusing on user interface and the efficient utilisation of device capabilities. To evaluate the user interface, Lim investigates the response time of rendering the user interface and how many network operations were made in order to obtain and load the data. However, since a web application can not access the camera or the battery level, Lim could not evaluate how efficient the device capabilities were since the hybrid version could not even obtain their functionality.

Finally, Johansson and Andersson compares different frameworks to each other in their article[15]. The frameworks are PhoneGap, Unity3D, GameMaker, and Qt and the comparison is made by evaluating the battery consumption and the time it takes to loop 10 million times. The tools for measuring the power consumption is not described but it seems as they solely observed the phones own statistics in order to observe the battery drainage. The time for performing the loop was done by manually timing the process. This way of measuring the battery consumption and performance is quite old-fashioned since the phone's statistics might not provide an accurate result and neither would manually timing a process.

1.6 Report Structure

Chapter 1 contains the introduction to the thesis and describes the motivation for the thesis and what similar work exists regarding related subjects. Chapter 2 consists of relative theory for the areas in this thesis. It will provide more specific details and theory about Android and

its development in section 2.1, the current state of cross-platform development in section 2.2 and explain what React is and how it is used in 2.3. Moreover it will provide the necessary details of how React Native works in section 2.4. Subsequently section 2.5 explains Realm, the database solution used in the React Native application.

Furthermore, chapter 3 will describe how the work was carried out by explaining the pre-study and the application used for the replication. Section 3.2 will describe how the application in React Native was developed. It will contain code examples from the foundations in the application and how some problems were solved. Section 3.3 will describe the different approaches for evaluating the application and section 3.4 contains the tools used in this thesis. Subsequently, the procedure for analysing the data retrieved from the tests is described in section 3.5.

In chapter 4 the results from the test regarding the replication is displayed and subsequently the comparison of the applications regarding performance is presented. Lastly, in chapter 5 the results and the development experience are discussed and chapter 6 contains the conclusion of this thesis and suggestions for further work.

2 Theory

2.1 Android

Android is an open source operating system, based on a single modified Linux kernel and is owned by Google. Google is responsible for the development together with the Open Handset Alliance (OHA).[24] The operating system was initially thought to be a platform for digital cameras when the development by Android Inc. began in 2005. However, Google bought the company and three years later the first smartphone running Android was sold. In 2015, nearly 83% of the mobile market uses Android and 1.5 billion[1] applications are downloaded by Android users every month.[31]

2.1.1 Architecture

Android operating system is a stack of software components and as shown in figure 2.1, the Android architecture consists of four layers: Linux kernel, Libraries and Android runtime, Application framework and Applications.

[1] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

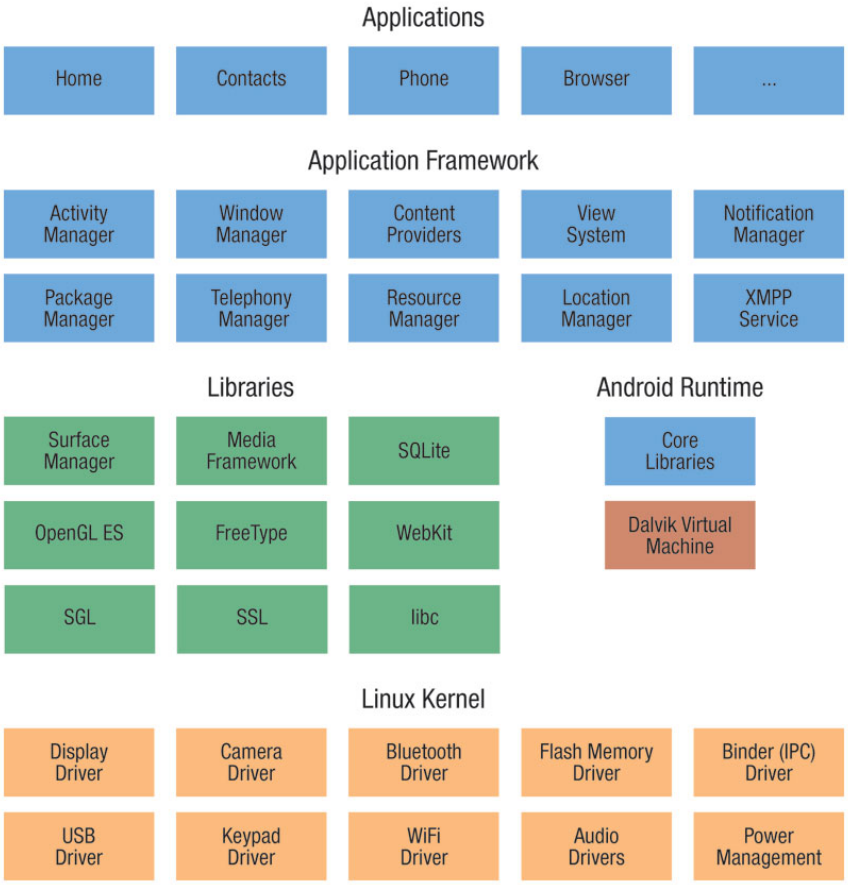


Figure 2.1: Architecture of Android[36]

Linux kernel

Android is based on a Linux 2.6 kernel which is modified in order to run in a phone. Like many operating systems it serves functionalities as internal storage, internet protocol, process management, device management and other core services. The Android OS interacts with the device's hardware at this layer and the modules and drivers are mostly written in C programming languages.[31][21][32]

Libraries and Android runtime

On top of the Linux kernel layer are the native libraries and the Android runtime module. The native libraries are libraries used by the applications and are an important link connecting the applications to the features which Android offers. The libraries are written in C or C++ language and are called through a Java interface which enables accessing different varieties of data or audio in the phone.[31][24][3]

Android runtime is a combination of a set of core libraries and the ART (Android Runtime) and has replaced the Dalvik Virtual Machine. The Java Core Library provides the most essential functions and ART is used for running the applications. The Dalvik VM was a modified Java Virtual Machine and was register-based and optimised in order to run in mobile phones since it had low power and memory requirements. Moreover the DVM allowed the applications to run in its own instance of the machine. This produced a reliable system since the DVM enabled multiple instances of the virtual machine to be created concurrently which provided isolation, memory management and support for threading. The Dalvik Virtual Ma-

chine ran .dex files that are a transformation of .class and .jar files and are executed with lower resources. However in Android 5.0, DVM was completely replaced by ART that is now the the managed runtime and was specifically created for Android.[35] ART transforms bytecode, compiles it to native instructions and are then executed in the runtime environment. It uses the same .dex-files as Dalvik but replaced the .odex-files with Executable and Linkable Format (ELF) executables. These ELFs are the only files executed when the application is compiled which results in the elimination of different application execution overheads. ART uses ahead-of-time (AOH) compilation that compiles the whole application to machine code when the app is installed. This amongst other things improves the execution efficiency, memory allocation and reduces the power consumption.[31][21][1]

Application framework

The application framework offers abstractions of the native libraries which are combined with capabilities of Dalvik. Furthermore it provides the application programming interfaces (APIs) and other higher-level services in form of Java classes. One important block of the application framework layer is the Activity Manager which handles the life cycle of the applications.[31][21]

Applications

The applications layer is the top layer in the architecture and is the layer where the applications are located. The applications can be pre-installed and provide the basic functionalities of the phone as making phone calls or browsing the internet but the application layer also handles the ones that are downloaded or under development. These applications are written in Java and are subsequently compiled into machine code to be installed on the smartphone.[31][3][32]

2.1.2 Development

Creating a new Android project in an integrated development environment (IDE) such as Android Studio or Eclipse provides a basic application where the Android SDK aids the developer to structure the work and is mandatory for Android developers. The Android SDK takes advantage of a Java programming language called Java Android Library which contains all the packages, application framework and class libraries the developer requires in order to create an Android application. The syntax is the same as the original Java when it comes to operands, iterations and selections but there are some specific Android classes and packages such as the Activity and View Class.[32][22][3]

When an Android project has been created, the developer can compile the Java code using the Android SDK and the development environment. This will result in an application consisting of an Android Package (APK), a compressed collection of the code compounding the application, and the application can be installed and run on a real or virtual device. Every version of Android has a different Android Virtual Device (AVD) that can be configured and launched in the IDE. The AVD is an emulator which contains the specific smartphone OS and is convenient if the developer lacks an Android device.[32][22][24]

The design and implementation of the user interface in Android uses similar UI components and concepts as native Java UI however there are some differences. All user interface components in Android are built on Views and the UI in the applications consists of Activities which represents the user interface. The Activity contains many Java components and the styling is done through XML in a similar way as when designing a web page using HTML and CSS.[19][24]

2.2 Cross-Platform Development

Over the last years, different techniques and frameworks have risen in order to provide a solution for creating a cross-platform application. The goal is to only develop one service which can either be accessed or deployed to different operating systems and provide a homogeneous and native feeling.

One popular technique is to create a responsive website, an application on the web accessible by mobile devices through their browser. By using features in HTML5 and CSS3 together with the popular front-end library Bootstrap[5], the web application will have an UI which is easy to use with either a computer or a mobile device. The user interface is responsive as the dimensions and layout of components are styled in relation to the height and width of the accessing device's screen. This results in a web application with an UI as an ordinary website when accessed by a computer but is more similar to an application when accessed by a mobile device.[11][17]

An alternative is to use a hybrid framework in order to create an application that can be used by any operating system. This approach is a combination of using web and native development since the application is built by using web techniques but is executed, rendered and displayed as a native application by using a WebView[7]. The capabilities of the device are exposed by an abstraction layer as a JavaScript Application Programming Interface which allows the hybrid application to access functionalities and features of the device. However, even though the development cost of hybrid applications are significantly lower than natives, the hybrids can not provide the same native user experience and therefore they have not been successful in replacing native development.[16][15]

2.3 React

React, sometimes referred to as React.js, is a JavaScript framework developed by Facebook and released as open source in 2013 in order to aid the development community to build interfaces. Facebook were in need of a framework that could solve their problem with complex user interfaces which had data that changed over time. React provides the "view" part of the development paradigm model-view-controller (MVC) and as a framework which serve the V in MVC, one can believe that React only works on the client side. However, it can also be rendered on the server side, resulting in an inter-operable communication between the two sides. Tom Occhino, who is an engineer at Facebook, said "React wraps an imperative API with a declarative one. React's real power lies in how it makes you to write code".[26] A declarative programming style describes what to do but not how it should be done, resulting in less code while an imperative programming style however describes how to do it.[12][10]

2.3.1 Virtual DOM

Once a web page is loaded into a web browser, a Document Object Model (DOM) is created containing that web page. A DOM is a representation in form of a tree structure and displays the structure of the current web page and its state using HTML elements. When an action is performed on the web page, for example the user navigates to another page or the application receives data from a server, the new page is either recreated or the content of the DOM is manipulated by JavaScript. The latter is done when using a single page application (SPA) but DOM manipulation is expensive and has become a mantra. The code base will become hard to maintain and the mutations are slow since they cannot be optimised for speed. In order to receive the synchronisation between the applications UI state and the data model's state, a two-way data binding can be used. This can be achieved by using key-value observing which is used by Knockout.js and iOS and the other one is dirty checking which the framework Angular by Google is using. This would require a linking function in order to look at what data has changed and imperatively make changes to the DOM to keep it updated.[10][27][18]

React uses a concept called the Virtual DOM instead of a two-way data bind. The Virtual DOM selectively renders subtrees of the nodes in the DOM based on the current state, resulting in a minimal amount of manipulation required in order to keep the page updated. The Virtual DOM is a representation of the actual DOM and is an abstraction which grants the ability to treat the JavaScript and the DOM as if they were reactive. React will store the state of the application internally and only perform the DOM manipulation when the state has changed. When the state of the data model has changed, the virtual DOM and React will re-render the user interface to a virtual DOM representation. Thereafter, React will compare the two virtual DOMs to each other in order to identify the differences and what is required to be manipulated. These differences are thenceforth updated and rendered into the real DOM. This process is called reconciliation and is all due to the creation of the components. When the component is initialised for the first time, the *render* method is called and a string of markup is produced and injected to the DOM. When the data changes, the *render* method is called once again. The differences of the return value from the previous and the new call is the minimal set of changes needed to be applied.[10][27]

2.3.2 Components and JSX

React is often seen as a UI library and expedites the creation of UI components that are interactive, reusable and stateful. These components are the building blocks of React and are similar to functions which have property and state parameters, these are described later. The creation of components can be done with either plain JavaScript or by using JSX. JSX is a JavaScript syntax extension for XML based object representation and has the benefit of creating large trees that are easy to read. Furthermore, React can either create HTML tags as strings or React components in form of classes.[27]
The syntax of rendering a component into the web page is simple:

```
var divStyle = {color: blue};

React.render(
  <h1 style={divStyle}>Hello, world!</h1>,
  document.getElementById('myDiv')
);
```

The first argument of the *render* method is the component that should be rendered and the second argument is what DOM node it should be injected to. The code above will create a header of type h1 with blue text colour and insert it into the division which has the id "myDiv". However, another method called *createClass* can be used in order to first create a custom component which later on can be rendered. Furthermore, attributes called props can be added to the components and can be utilised in the *render* method in order to present dynamic data.[27]

```
var MyComponent = React.createClass({
  render: function(){
    return (
      <h1>Greetings, {this.props.name}!</h1>
    );
  }
});
```

This component can now be rendered into the DOM by calling the render method:

```
React.render(
  <MyComponent name="User"/>,
  document.getElementById('myDiv')
);
```

States are the core of the interactivity of React and is used in order to merge data into components. This is done by using the *setState* method that passes the data to a component which is then re-rendered.[10][27]

2.4 React Native

At the React.js conference in 2015 Facebook introduced their new framework React Native, a framework they thought would revolutionise the way mobile applications are created. When React Native was released, there was only support for iOS but since then the support for Android has been added and is still expanding. Facebook have started to become more open-source and is the approach they have chosen for React Native. Even though the source is not completely open yet, Facebook attempt to achieve this and contemplates that the community will contribute to improve the framework.

The main purpose of React Native is simple, a developer should not require the knowledge or need to spend superfluous time in order to create a mobile application since at least two applications need to be developed in order to support both iOS and Android. Since different platforms have different looks, feels, and capabilities, there can not be an application which is homogeneous on all operating systems. However since it is the graphical interface that differs, the development could base on the same language but have the graphics be rendered differently depending on the targeted platform and be real native components. Facebook call this approach "learn once, write anywhere" which describes what React Native is all about. The technology of React Native is based on React, described in section 2.3, and the advantages of React is shipped to the framework which applies it to native applications. Instead of running React in the browser and render divs and texts which can be seen in section 2.3.2, React Native runs in an embedded instance of JavaScriptCore (iOS) or V8 (Android) inside the applications and render to higher-level platform-specific components. JavaScript components are declared by using a set of built-in primitives supported by iOS or Android components.[25][37]

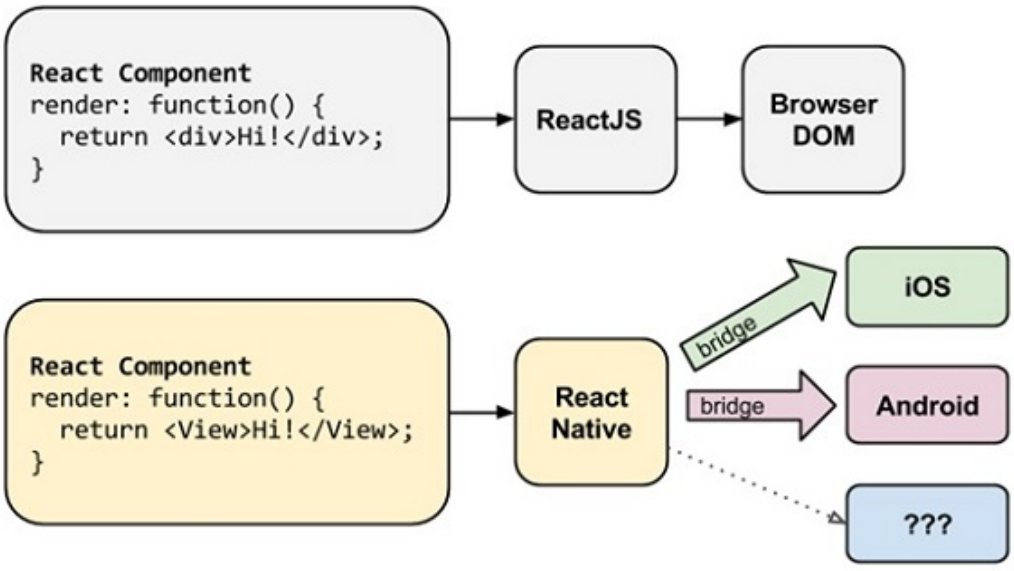


Figure 2.2: Rendering in React and React Native [9]

There have been many attempts of providing the development community with a hybrid framework but none has the complete set of features offered by React Native. Unlike most other mobile cross-platform development approaches, React Native is not a hybrid solution. Unlike Phonegap, Titanium, and Ionic, React Native does not rely on rendering inside Web-

Views or tries to mimic HTML & CSS. The layout is performed on a separate thread, resulting in the main thread being able to perform animations undisturbed. Since an application in React Native does not use WebViews, it is possible to build applications with a native responsiveness. Moreover instead of pushing one codebase to all platforms, a shared codebase is written and deployed to the platforms and only some sections as graphical elements and platform-specific components are written separately for each targeted platforms.[25][28]

2.4.1 The core of React Native

As earlier stated, React Native is able to render the React Native components to real native Views for Android or UI Views for iOS. This is possible due to the abstraction layer known as the "bridge" which enables React Native to invoke the rendering APIs in Java for Android or Objective-C for iOS. Furthermore, the framework reveals the interface of JavaScript, allowing the application to access platform-specific features as the battery or the location. Section 2.4 mentioned the different threads used in React Native and there are actually three main threads on which React Native is based on: the shadow queue, where the the layout is handled; the main thread, where the UI rendering is performed; the JavaScript thread, where the scripts are running. These threads are responsible for handling different events in the application.[20][8]

2.4.2 Features

A few features of React Native have been briefly introduced earlier in the paper but there are several features of React Native which makes it even more attractive to the development community. The creation and explanation of creating the native UI components are described later in section 2.4.3.

First of all, React Native has support for asynchronous execution of operations between the JavaScript code in the application and the native platform but also permits threading in the native modules. This allows many different operations to be performed in the background while not blocking the user interface. It also allows the developer to debug the code while running the application by for example using the Chrome Developer Tools.

Secondly, React Native handles screen interaction by implementing a system which handles touches in a complex view with high level features. Since the gesture recognition is more advanced on a mobile device than on the web, there are many different actions which can be interpreted by the touch such as scrolling, tapping, sliding and so forth. Moreover, there can be multiple touches being performed concurrently. The users notice the difference between a web app and a native since every touch should display what will happen when its released and the user should be able to cancel the action by dragging their finger away. React Native has solved this problem by adding an abstract *Touchable* and *TouchableHighlight* implementation that assimilates properly with scrolls and other elements without requiring additional configuration.

Furthermore, similar to React, React Native uses JSX which is explained in section 2.3.2 and inline-styling in order to create and style components without complicating the structure or code. In React, the styling is specified as an object and is added to the element inside the HTML tag which can also be seen in the code in section 2.3.2. However, React Native uses a layout model from the web called Flexbox. Flexbox simplifies the process of building common UI layouts and its StyleSheet abstraction offers the possibility to declare the layout and styling along with the component inline. The usage of Flexbox can be seen figure 2.6. Lastly, React Native is inspired by React where components are the building blocks, which is described in section 2.3 and 2.3.2. The similar approach applies for React Native as the application is constructed by native modules and UI components. React Native has support for the common platform APIs and native UI components but also the support to create custom native modules and views if it is needed.[28][8][9]

2.4.3 Development

In order to install React Native, Homebrew is required to be installed and is then used to install Node.js. By using the node package manager (npm) which is a supplement to Node, the developer can install React Native with the command line. In order to run a React Native application in Android, Java JDK and Android SDK together with Android SDK Build-Tools 23.0.1 need to be installed on the computer. With React Native installed, a new React Native application can be created and be run by yet again entering a new command. The steps of the commands can be seen in figure 2.3.[28]

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"

$ brew install node

$ npm install -g react-native-cli

$ react-native init TestProject

$ cd TestProject

$ react-native run-android
```

Figure 2.3: Commands to set up and create a React Native project

The newly created project TestProject contains:

- package.json - A file containing the relevant metadata for the project or project dependencies.
- node_modules/ - Contains the dependencies and the CLI tool, *node_modules/react-native/local-cli/cli.js*, which controls the project. This script executes another helper script *node_modules/react-native/init.sh* that assembles the boiler plate code.
- index.android.js - The React Native main file of the project.
- android/ - Contains android specific code and is executed together with the code from index.android.js, resulting in a React Native application.
- iOS/ and index.ios.js - The iOS version of the two aforementioned file and folder.

As stated, the file *index.android.js* is the main file of the code and contains basic functionality from the beginning. First of all, the imports are listed. This states that the application is React Native and lists the imports needed as components, stylesheets, images etc. These imports can then be used to create components which can be rendered in the application. If the developer requires more imports, these can be added to the import object.

```
import React, {
  AppRegistry,
  Component,
  StyleSheet,
  Image,
  Text,
  View
} from 'react-native';
```

Figure 2.4: Example of an import object used in React Native

As earlier mentioned, a React Native application is built by using React components in form of JavaScript objects which are in charge of rendering and automatically updates the user

interface. The first component that is created is a custom class component, the component that is the first screen of the app. This component includes the render-function containing the layout written in JSX. In the code in figure 2.5, a View is created as the container and encloses two text elements and an image. The content for the texts and the source to the image is retrieved from the object in the array MOCKED_DATA.

```
var MOCKED_DATA = [{
  name: 'William',
  born: '1992',
  images: {thumbnail: 'http://goo.gl/W06A0l'}
  }];

class TestProject extends Component {
  render() {
    var data = MOCKED_DATA[0];
    return (
      <View style={styles.container}>
        <Text style={styles.nameText}>{data.name}</Text>
        <Text style={styles.bornText}>{data.born}</Text>
        <Image style={styles.image}
          source={{uri: data.images.thumbnail}} />
      </View>
    );
  }
}
```

Figure 2.5: The creation of a component in React Native

In the code above, the components have a style attribute referring to a JSON object in an instance of a *React.StyleSheet* that uses Flexbox. The styling of the components in figure 2.5 is presented in figure 2.6. The container have the property *flex* which specifies the length of the item relative to the rest of the flexible items in the same container. Moreover it aligns its child elements in the center and gives it a very pale blue background color. The texts are given different font sizes, colors and margins and finally the image is given a specific height and width.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  nameText: {
    fontSize: 20,
    margin: 10,
  },
  bornText: {
    color: '#333333',
    marginBottom: 5,
  },
  image: {
    width: 53,
    height: 81,
  },
});
```

Figure 2.6: JSON object containing the styling for the component by using Flexbox

Lastly, in order to specify for React Native that the newly created component should be rendered when initialising the application, the component is registered with the *AppRegistry.registerComponent* function. This is shown in figure 2.7 with the ECMAScript 2015 (ES6) specification and without.

```
// ES6
AppRegistry.registerComponent('TestProject', () => TestProject);

// Conventional
AppRegistry.registerComponent('TestProject', function() {
  return TestProject;
});
```

Figure 2.7: Registering the component for the application

When the code is ran by the command in figure 2.3, the application will compile and the APK is installed on the device. The result from the code above is displayed in figure 2.8 and when a change is done, the developer can simply shake the device and reload the JavaScript in order to receive the updates in the application.

Figure 2.8: Result from code in section 2.4.3

2.5 Realm

Realm is a cross-platform database solution designed specifically for mobile applications. The database is simple to use, very lightweight and is said to be faster than SQLite or Core Data. Realm can be used in five different environments and one of these are React Native. Realm React Native is specifically designed for JavaScript and to enable reactive app development with live objects to instantaneously model, store and query data. Schemas are defined and are passed down to Realm during initialisation and contain the data model for an object. The object has a name and a set of properties which consists of a name, a type and an

optional field for the default value. A new object can be created by calling the *realm.create* method which takes the name of the schema to use and the data to store as parameters. The *realm.objects* method is called in order to retrieve objects and requires the name of the schema to obtain. Furthermore, the *filtered* method is used on the queried object for obtaining the desired objects. The creation of schemas can be seen in section 3.2.5, in figure 3.1.[29]

2.6 Survey evaluation

Evaluating a system through user experience is a frequently used approach in order to obtain information on how well the system performs from a users point of view. Usability evaluation focuses on efficiency and be evaluated by logging the system and obtaining data in form of errors and numbers of clicks while user experience provides information on how the user feels about the system. There are several approaches for evaluating of a system. Field studies can be performed in order to examine the usage of the system in real life where the user performed exploratory testing, meaning the user is allowed to explore the system without guidance and the conductor acts as an observer. Furthermore, questionnaires are a popular approach for system evaluation and can be used complementary to other methods.

A questionnaire is a fast and simple way of gathering data from users and often the users are more honest about the responses since they can be anonymous. Moreover, conducting the survey online results often in a wider spread and therefore more responses. By combining user performed exploratory testing with a survey containing questions about the system, the opinions of the user regarding the system in overall can be easily obtained. However, the demographic and the number of participants is an important part of the survey since it has an immense impact on the results and what conclusion which can be drawn. There are generally two different user groups which are used, either target group or expert evaluation. The target group consists of users likely to use the system when completed. These provide realistic opinions which reflects the thoughts of the end-users but are often more time consuming and expensive. Therefore, expert evaluation can be used in the process in order to obtain a critical view of the user experience early on and minimise the amount of negative review from the end-users later. Using expert evaluation often provides a harsher opinion of the system and needs to be taken into consideration while evaluating the responses from the survey. It is important for the conductor to analyse the users and examine their knowledge and other relative background since it can have an impact on the outcome and provide interesting conclusions. Furthermore, the number of participants is important since a low number can result in wrongfully drawn conclusions but as mentioned, a survey often aids in retrieving a higher grade of responses due to the simplicity and ability to be answered remotely.[30][34]

3 Method

The objectives of this paper are to evaluate the current state of support and the simplicity of developing React Native. Moreover, it will investigate the possibilities of creating an application in React Native which have the same look and feel as a native application and it will compare the performance of these two applications.

3.1 Pre-study

Developing a service requires knowledge. Whether it is the programming language, the environment or the overall work flow, the developer requires a solid ground when creating a new project. The pre-study of this paper strived to retrieve such knowledge in order to subsequently be able to create an application using React Native. As can be seen from the references in section 2.4, there are no published articles covering React Native and how the development is performed. The sources for understanding React Native were numerous blog posts, online tutorials and the documentation which provided enough support in order to create an application and try to replicate an existing android app.

In order to be able to perform the replication, an Android app had to be obtained. The reason for replicating an existing application is described in section 1.4 and the main argument is that it saves development time and shifts the focus to develop in React Native and be able to create an application with more functionality. There are many Android applications which are free and open-source (FOSS) and can be accessed by several websites. F-Droid is a repository for FOSS applications in form of a website and by examining the applications which had the appearance of containing enough functionality, Budget Watch[1] was chosen.

3.1.1 Budget Watch

Budget Watch is a FOSS application which helps the user to the manage their budgets and was created by Branden Archer. The user can create different budgets for groceries, clothing, etc., and record their day-to-day transactions. The transaction is assigned to different budgets and contain different kind of information, the user even have the possibility to capture a photo

[1] `https://f-droid.org/repository/browse/?fdfilter=Budget+watch&fdid=protect.budgetwatch`

of the receipt for an expense or revenue. This allows the user to obtain an overview of their expenses and revenues each month and aids them in following their budget.

Screenshots of the entire application can be seen in appendix A and the start scene of the application contains two items. The first item refers to budgets and consists of a purse icon, a text which says "Budgets" and a smaller text "Create and manage budgets" below. The corresponding item for transactions have an icon in form of a handshake, a text "Transactions" and the text "Enter transactions and revenues". When pressing the budgets item, the user is redirected to a scene containing a list of budgets, the date of the current month and the navigation bar has an icon for adding a new budget. The list items contains the name of the budget and below there is a progress bar which displays how much of the budgets maximum value that has been achieved so far. To the right of the progress bar this value is more accurate displayed with the current value and the maximum value. Pressing the icon in form of a plus sign redirects the user to a scene for adding a budget. This scene consists of an input field for the type of the budget with a placeholder "Groceries". Below, there is another input field which regards the maximum amount of expenses for that budget and only excepts digits. Lastly, there are two buttons. The first button cancels the adding of the budget and redirects back to the list while the other one saves the budget (if the two fields are not empty) and then redirects back to the list where the new budget is visible.

When the transaction item is pressed from the start scene, a scene for transactions is displayed. The scene is similar to budgets except there are two pages of transactions, one for expenses and one for revenues. Below the navigation bar there is a menu of two tabs which correlates to the two types of a transaction. When pressed, the application displays the page which consists of a list displaying the transaction of that type. The user can also switch between the two pages by swiping the finger left and right. The list items of transactions contains the name of the transaction, the value, the name of the budget assigned to the transaction, a receipt icon if the transaction have a captured image, and lastly the date of the transaction. The scene for adding a transaction contains:

- Name - An input field for the name of the transaction.
- Budget - A selectable list for the budget regarding the transaction.
- Account - An optional field for the account which was used for the transaction.
- Value - The value of the transaction.
- Note - An optional field if the user want to add a note.
- Date - The date for when the transaction occurred.
- Receipt - A button which opens the camera and allows the user to take a photo of the receipt.
- Buttons - Two buttons for cancelling or saving the transaction which redirects back to the list of transactions.

The user can also view the list of transaction by pressing a budget in the list of budgets. This redirects to the scene of transactions but only displays the transactions associated with that budget.

Lastly, the user can edit and remove budgets or transactions. The option to edit an item is displayed by pressing the item for a longer time. This action displays a popup that says "Edit" and when pressed, opens a scene for editing the item. The scene is the same as adding a budget or transaction but with the values of the pressed item in the input fields. In order to delete the item, there is an icon in the navigation bar in form of a trashcan which, when pressed, deletes the item and redirects to the previous scene.

3.2 Implementation

When the application for replication had been chosen and retrieved, the development could commence. Screenshots of Budget Watch, which can be found in appendix A, were printed and hung up in order to visualise the end product. By examining the XML files of the original Android application, values as padding, margin and other styling attributes could be obtained and were noted on the screenshots.

At first, the environment for developing React Native had to be set up. The documentation of React Native[28] covers this phase and has different sub-pages which explains in detail how to configure the environment for the two different platforms, with different operating systems on the developing computer and how to use a physical device or set up a virtual device. By following the steps which are presented in the "Getting started"-chapter, the different packages and engines which are required were installed and the environment was completed. Furthermore, before commencing the development of the application, a Github repository (described in 3.4) was created for the source code in order to track the versions of the code.

3.2.1 Navigation

Budget Watch have five main components: "Start", "Budgets", "Transactions", "AddBudget" and "AddTransaction". Firstly, the navigation and routing were set up for these five components, resulting in five empty scenes but with different titles in the navigation bar. The navigation is enabled by using the *Navigator* component which handles the transition between scenes in the application by providing route objects which are interpreted as scenes and initialised by the *renderScene* function. The *Navigation* component can have the node *navigationBar* attached, a component that will persist through all scene transitions and will handle the different layouts of the navigation bar through another component called *NavigationBarRouteMapper*. The *NavigationBarRouteMapper* tells the *Navigator* how to design the user interface of the *NavigationBar* on each scene. This means that every time a new scene is rendered from a route, the mapper is called and returns a layout of the navigation bar depending on the route which is to be rendered.[28][23]

3.2.2 Start scene

Secondly, the design of the start screen was added by implementing two views which contained the image and texts of the two buttons "Budgets" and "Transactions" and by examining the previously mentioned screenshots, the styling of the scene was easily added. In order to allow the user to navigate to the Budgets or Transactions scene by pressing the buttons on the start scene, the wrapper *TouchableNativeFeedback* was added around each button. When pressed down, the background colour of the view inside this wrapper can transition with a ripple effect and different events is called depending on the press from the user.[28] When the user presses the Budgets-button, the property *onPress* is used to call a function that pushes the *BudgetsComponent* to the navigator along with the title "Budgets" which is displayed in the navigation bar. This results in the *renderScene* method being called in the root component *BudgetWatch_ReactNative* in the index file which will render the provided *BudgetsComponent* and display it. The same functionality was also added regarding Transactions.

3.2.3 Budgets component

Consequently, the scene for Budgets was created. The dates of the current month was implemented by using the JavaScript *Date* object but had to be fixed in order to have the same format as in the Android application. Below the date, *BudgetsComponent* creates and returns a custom component named *BudgetList*. *BudgetList* applies the core component *ListView* in order to create a scrollable list of dynamic data. The data is used by the array

ListView.DataSource and each row is rendered with data from each item in the array. Since there were no database implemented, hard-coded data was used to populate the list. Each row of the list is rendered in the *renderRow* method which allows the developer to only specify the layout of every item in the list one time. Every item in the list of budgets have a progress bar which was implemented by importing the React component *ProgressBar*.

In order to allow the ability for the user to navigate back to the start scene or add a new budget, the user can press an icon in the navigation bar which redirects to the new scene. This functionality is implemented in *NavigationBarRouteMapper* which contains three methods: *LeftButton*, *RightButton* and *Title* where as all have route, navigator, index and navState as arguments. In order to provide with a back button, *LeftButton* returns an icon as long as the index is greater than 0. This will result in the icon not being rendered on the start scene and when pressed will pop the navigator and the previous scene is shown. The same principle applies for *RightButton* but since the icon in the right side of the navigation bar differs between scenes, the function compares the route and returns a view depending on the current route. The view contains a certain icon that, when pressed, pushes a new component to the navigator which renders the new scene.

3.2.4 Add Budget

Implementing a scene for adding a budget was straightforward since it simply consists of two input fields, one for name and one for the value, which are created using the component *TextInput*. The values of the fields are stored as states and updated every time the value in the field is edited. In order to mimic the buttons for cancelling or saving the budget, an external component[2] developed by James Ide were used. The Cancel-button pops the navigator and the Save-button should save the data. Furthermore Save-button verifies that the inputs are not empty and have to provide feedback to the user if the input is not valid. In the original application, this was done by the Android object Snackbar[3] which displays a message at the bottom of the screen. However, since a component for *Snackbar* does not exists for React Native, this was changed to the traditional *Toast* object. The message is created and displayed when the user tries to save a budget with empty values and provided feedback to what field is faulty.

3.2.5 Lists and database

In order to implement a local storage service for the application, Realm (described in section 2.5) was used. By creating the schemas as can be seen in figure 3.1 and the initial data for transactions and budgets in the index-file, the data could be passed down to components using props.

[2]https://github.com/ide/react-native-button
[3]https://developer.android.com/reference/android/support/design/widget/Snackbar.html

```
const BudgetSchema ={
  name: 'Budget',
  primaryKey: 'id',
  properties: {
    id: 'int',
    name: 'string',
    maxValue: 'int',
  }
};

const TransactionSchema ={
  name: 'Transaction',
  primaryKey: 'id',
  properties: {
    id: 'int',
    transactionType: 'int',
    name: 'string',
    budget: 'string',
    account: {type: 'string', optional: true},
    value: {type: 'float', default: 0},
    note: {type: 'string', optional: true},
    date: 'string',
    datems: 'int',
    receipt: {type: 'string', optional: true},
  }
};
```

Figure 3.1: Schemas for Realm

The data from the Budget-object was added to the *ListView.DataSource* and the Save-button for *AddBudget* calls a function which writes a new entry for Budgets in Realm and pops the navigator in order to return to the list of budgets. However, the current code would not work as intended since when a new item had been added to the database and the budget scene with the list is popped back to, the list would not update. This is due to a bug in the Android version of ListView since even though new data is used as state for the list component it does not re-render. This was solved by the code that can be seen in figure 3.2 and initially the property *onDidFocus* was added to Navigator, in the root component, that calls a method with the same name which takes a route as argument. The method is called with the new route of each scene after a transition or the initial mounting. By checking if the new route was Budgets, the component could be forced to re-render.

```
onDidFocus(route){
  if(route.name === "Budgets"){
    var data = realm.objects('Budget').sorted('name');
    return <Budgets navigator={navigator} realm={realm} data={data} />
  }
  else if(route.name === "Transactions"){
    var data = realm.objects('Transaction');
    var budgetName = route.passProps.budgetName;
    return <Transactions
      navigator={navigator}
      realm={realm}
      data={data}
      budgetName={budgetName}/>
  }
},

render() {
  this.createInitialItemsForDatabase();
  return (
    <Navigator
      ref={(nav) => { navigator = nav }}
      style={{flex:1}}
      initialRoute={{ name: 'Application1', component: Main}}
      renderScene={this.renderScene}
      onDidFocus={this.onDidFocus}
      realm = {realm}
      navigationBar={
        <Navigator.NavigationBar
          style= {styles.navigationBar}
          routeMapper={NavigationBarRouteMapper(realm)}
        />
      }
    />
  )
}
```

Figure 3.2: Forcing re-rendering of scenes containing lists

However this was not enough due to the previously mentioned bug and required the *BudgetList* component to implement the *componentWillUpdate* method. When new props have been received or a new state has been set, this method is invoked before the rendering. By setting the new state of the DataSource to the newly received data, the list is forced to be updated. However, before a new state has been set, there need to be a check if the old array of data differs from the new array. Otherwise the component will be stuck in an endless loop as the state is being set, then rendered with new data which will invoke *componentWillUpdate* that consequently will set a new state and so forth. The *componentWillUpdate* method can be seen in figure 3.3.

```
componentWillUpdate (nextProps, nextState) {
  if (this.state.dataSource._cachedRowCount !== this.props.data.length)) {
    this.setState({
      data: this.props.data,
      dataSource: this.state.dataSource.cloneWithRows(this.props.data)
    })
  }
}
```

Figure 3.3: Forcing the data for lists to update

3.2.6 List for transactions

When the basics of Budgets had been implemented, the same functionality was added for Transactions. The two halves of the application are similar too some extent but Transactions have some auxiliary functionalities. Firstly the *TransactionsComponent* consists of two parts, one for expenses and one for revenues, where the user can navigate between the two scenes by either sliding the finger on the screen or pressing the tabs below the navigation bar. *ViewPagerAndroid* is a container component which enables the user to switch left and right

between child views. The tabs were implemented using the external component *react-native-android-tablayout*[4] created by Albert Brand. In order for the two components to communicate and display the correct current view at the same time, a state containing the index of the current view was created. Furthermore, the user should be able to add a transaction by pressing the icon in the top right corner. This was implemented the same way as in Budgets except there are two different varieties of adding a transaction, either adding an expense or adding a revenue. When viewing the Expense-page the user add an expense and while viewing the Revenue-page, a revenue can be added. This enforced the navigator to communicate with the *TransactionsComponent* since the creation of the *AddTransactionComponent* is done in *NavigationBarRouteMapper* while the index of the current page and what type of transaction that should be created is within *TransactionsComponent*. Parent-child communication is simple in React Native due to props but there are no viable way of communicating between other components. However, by adding a new schema to Realm called *AppData*, a property called *currentTrans* was added which contains the index of the current type of transaction (Expense or Revenue) and is updated simultaneously as the state of the current view in *TransactionsComponent*. This property can be accessed by the Navigator which is able to render the component with correct data. Lastly, with a working list and database object of transactions, calculating the sum of transactions for each budget was implemented. By selecting all transactions from Realm at the rendering of each row for *BudgetList* and calculating the sum of all expenses subtracted by the sum of revenues, the result is provided to the progress bar and the text.

3.2.7 Add Transaction

The implementation for adding a transaction gained its core from adding a budget but contained more fields and additional information. First of all the user is required to choose which budget the transaction is connected to. This was applied by using the *Picker* component that retrieves the budgets from Realm and maps them into a list by using the *Array.prototype.map* method from JavaScript. Furthermore, a calendar component was required to be implemented in order for the user to choose the date when the transaction took place. The field of the form is a simple *Text* component that reads the current date from the state. When the text is pressed, *DatePickerAndroid* is used to open a standard Android date picker dialogue. The default date for the date picker is the date from the state and when a new date is chosen, the state is updated.

3.2.8 Edit or remove items

When the application was able to display and add both budgets and transactions, the ability to remove or edit items was implemented. This was done by adding a new property *onLongPress* for the *TouchableHighlight* that wraps each item in the lists. A modal was created and added to the Budgets scene with the visibility depending on a state that by default was false. When the user performs a long press on an item, the visibility state is set to true and the modal is displayed. The modal was styled in order to look like the *ContextMenu* which Android supports. The modal contains a text "Edit" and when pressed, opens a version of *AddBudget*. What differs from adding a new budget and editing an existing one is that there are preset values for the input fields which can be edited. The *onLongPress* passes down the *rowData* which contains the data for the selected budget to the modal. When the user presses edit, this data is once again passed down as props to the newly created *AddBudget* component and is added to the navigator, rendered in the index file and displayed. The state properties in *AddBudget* are given the values from the data and are passed down. If there are no data in the props, they are set to empty values since this implies that a new budget is to be added and not edited.

[4] https://github.com/AlbertBrand/react-native-android-tablayout

Furthermore, a new case was added for the *NavigationBarRouteMapper* in *RightButton* where an icon for deletion is displayed when the name of the route is "Edit Budget". The functionality of the icon is to delete the current budget that is being edited. However, as earlier mentioned, the communication between the scenes and the navigator is difficult and a new property *currentEditBudget* was added to the *AppData* object in Realm containing the id of the budget which is edited. The id is set from the modal when the scene for editing is rendered and with the use of the id, the navigator can easily remove the budget from Realm. The same functionality was applied to transactions where the only difference were that the name of the scene had to be either "Edit Expense" or "Edit Revenue", depending on the transaction type of the pressed transaction. Nevertheless, once again problems occurred at the re-rendering of the *ListView*. Since the formerly implemented verification for updating the list only compared the length of the original and the new data, an additional statement was implemented. Due to the bug in *ListView*, the list had been provided with the correct data and *DataSource* contained the updated values but did not re-render the list. Due to this, another statement which compared the two different datasets was not possible and yet another property had to be assigned to *AppData* in Realm. This property is a boolean named *shouldUpdate* and is set to true when an item is being edited. The verification for updating the list was given a statement which compared the lengths of the data but also examined the value of *shouldUpdate* and forced the update if it was true.

3.2.9 Hardware - back button and camera

On an Android application, the hardware back button[5] is by default the navigation for stepping back to the previous scene except on the first screen where it closes the application. In React Native, the back button will always close the entire application unless it is overwritten. This was done by using the API and adding an *eventListener* to *BackAndroid* which detects hardware back button presses and allows us to invoke it programmatically with our own functionality.

```
BackAndroid.addEventListener('hardwareBackPress', () => {
  if (navigator && navigator.getCurrentRoutes().length > 1) {
    navigator.pop();
    return true;
  }
  return false;
});
```

Figure 3.4: Implementation of navigation for physical back button

The only feature which remained was the ability to take a photo of a receipt regarding a transaction. React Native did not support a component for the camera at that time but Lochlan Wansbrough, who is the creator of many components for React Native, has developed a camera component[6]. The camera should be displayed when the Capture-button or Update-button is pressed while adding or editing a transaction and should canvas the entire screen, even the navigation bar. Since hiding the navigation bar in React Native for a specific scene is not possible, a modal was created which covered the entire screen and wrapped the camera component. When one of the formerly mentioned buttons were pressed, the modal was set to visible. The documentation for the camera only covers iOS and in order to implement a visual button which could capture a photo, an external view had to be implemented and arranged on top of the camera screen with absolute positioning. The implementation and rendering of the camera can be seen in figure 3.5. When an image was captured, the path to the image was added to the state of the transaction and subsequently saved to Realm. When the user

[5]https://developer.android.com/design/patterns/navigation.html
[6]https://github.com/lwansbrough/react-native-camera

navigates to display the receipt, a component containing an image with the source as the URI to the image is rendered and displayed.

```
<Modal
  animationType={'none'}
  transparent={false}
  visible={this.state.modalVisible}
  onRequestClose={() => {this._setModalVisible(false)}}>
  <View style={cameraStyles.container}>
    <Camera
      ref={(cam) => {
        this.camera = cam;
      }}
      style={cameraStyles.preview}
      aspect={Camera.constants.Aspect.fill}>
    </Camera>

    // Need to be outside of Camera component
    <TouchableHighlight
      style={cameraStyles.actionButton}
      onPress={this.takePicture.bind(this)}
      underlayColor="#d6d6d6">
      <View style={cameraStyles.buttonContainer}>
        <Image
          style={cameraStyles.cameraButton}
          source={require('./images/camera-icon.png')} />
      </View>
    </TouchableHighlight>
  </View>
</Modal>

// Function for capturing photo
takePicture() {
  var navigator = this.props.navigator;
  var thisComponent = this;
  this.camera.capture()
    .then(function(data){
      thisComponent.setState({inputReceipt: data.path});
      thisComponent._setModalVisible(false);
    })
    .catch(err => console.error(err));
}
```

Figure 3.5: Implementation of camera

The Android application had to be modified a bit in order to correspond to the guidelines of an Android application. The changes were minor changes in the layout but the functionality to save the current budgets and transactions were removed since this feature would be too hard and would not fit within the scope of this thesis. The code for the final Android application can be viewed and downloaded from a Github repository[7] and the final application in React Native is located in another repository[8].

3.3 Evaluation

In order to answer the two research questions listed in section 1.2 the evaluation was conducted into two parts: the feasibility of replication and the performance of the created application. The evaluation of developing an Android application in React Native and the possibility to duplicate the Budget Watch application was performed in two different processes. The development evaluation was observed by the developer himself and was based on the overall expression during the development phase and whether or not the targeted application could be reproduced using the React Native framework. This assessment would therefore not produce any visual results but a final discussion. If the replication of all functionality could not be completed before the deadline, the android application would have its redundant functionality removed in order to obtain two applications with identical functionalities.

[7] https://github.com/willedanielsson/BudgetWatch_android
[8] https://github.com/willedanielsson/BudgetWatch_ReactNative

3.3.1 Replication

Furthermore the final product of the development was tested by employees at Valtech. This assortment provides users which have knowledge in how an Android application should work and are qualified to be able to distinguish any faults in the React Native application. One could allow people of different knowledge assess the application which would provide a more reality-based result but by choosing experts of the area, a more professional result in acquired. The users were given a mobile device which had the original Budget Watch application and the replica installed and were given the question "Which one is created in React Native?". The users had the ability to use both applications freely and perform any actions they desired in order to produce an unbiased result by not telling the user what to do. The applications were named Application1 and Application2 and the user were given a simple form in order to record their answer. The form asked if the user knew which application that were of native origin or if the user could not answer. Furthermore, if the user had given an answer to one of the applications, a follow-up question were given which asked the certainty of the answer. The last question asked if the user would not mind using a React Native application or if the experience was bad enough to discourage future usage. This user test produced a result which provided numbers of how many users which could notice the differences in the applications and therefore how well the user experience of an application created in React Native was compared to a native application.

3.3.2 Performance

By using the profiling tool Trepn Profiler which is described below in section 3.4, the two applications could have different aspects of performance tested. The tests were done by using the same mobile device in order to establish that the results of the two applications did not vary due to the device. Furthermore, since the application does not require any internet connection, the mobile device was put in *flight-mode* which would remove any results being influenced by the traffic at the given time. In order to furthermore minimise surrounding influences, the test were performed every other time. First a test were done using the Android application and after that the React Native application were tested and so forth. Lastly, before every tests, the application had its non-initial data removed and was terminated.

There are many parts of an application which can be analysed when measuring performance but this paper focused on the following data points:

- GPU frequency which displays a value in Megahertz (MHz) and is a measurement of the speed at which the GPU operates (GPU is the unit which performs the graphical calculations and handled signal from the phone to the screen and contrariwise).
- CPU load which provides a percentile value of how much of the CPU (which is the unit that executes programs) is used by the application.
- Memory usage which returns how many Megabytes (MB) is required by the application.
- Battery Power which provides a value in milliwatts (mW) of how much power is drained by the profiled application.

Furthermore, there are three cases of the application which was tested for performance. The first test was performed when the application was started and as idle for 30 seconds. Thereafter there was two user cases done while measuring the performance where the application had already been started and the user navigated through it.

The first case was when the user navigated to Budgets and created a new category. The category received the type value "Test" and the value 500. When the category was created, the user was redirected to the lists of budgets. Thereafter the user edited the budget and

removed it. When the budget was removed and the list of budgets was updated, the test was done and the profiling was stopped. The screenshots with the values from the user road map of the first user case can be seen in figure 3.6

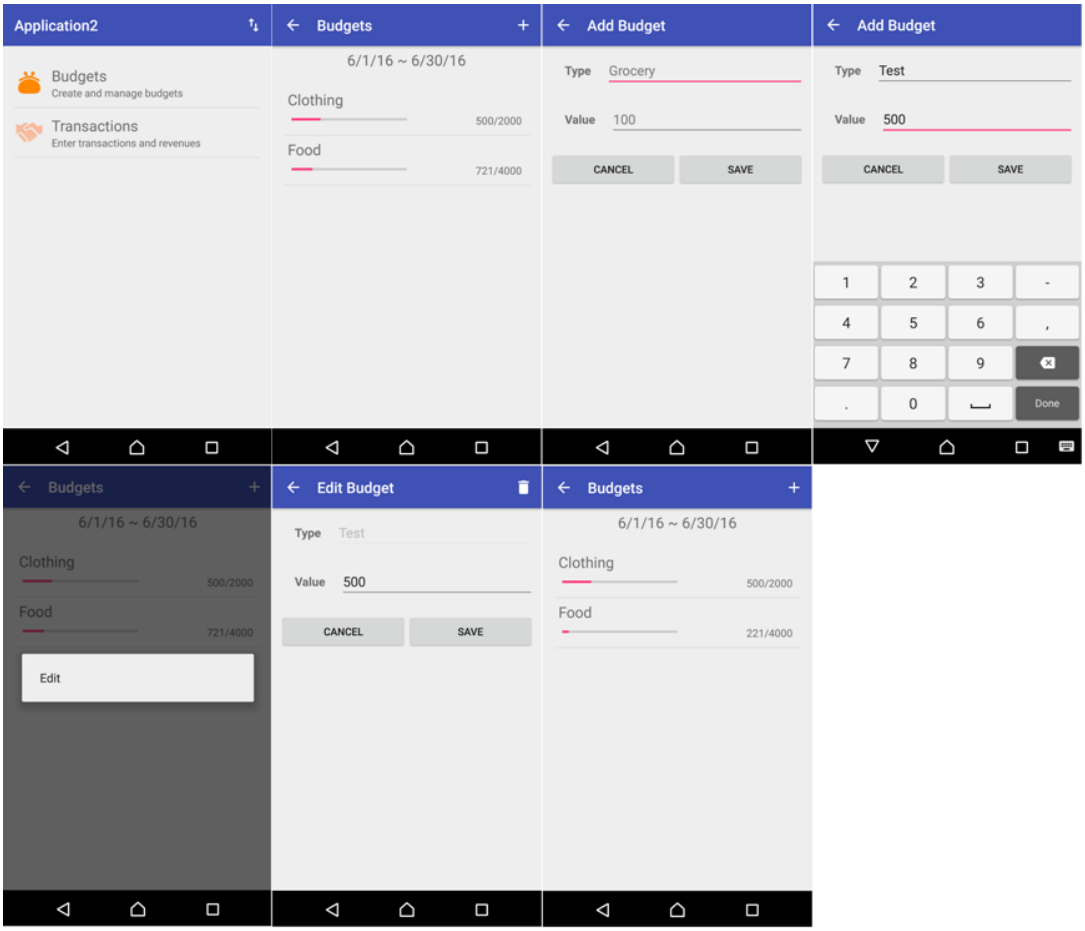


Figure 3.6: Road map of first user case

The second user case was when the user navigated to Transactions and created a new expense. Thenceforth the user typed in details for the new expense and took a photo which was attached as a receipt. The user consequently saved the transaction and the test was done as the profiling was stopped. The screenshots with the values from this user road map can be seen in figure 3.7

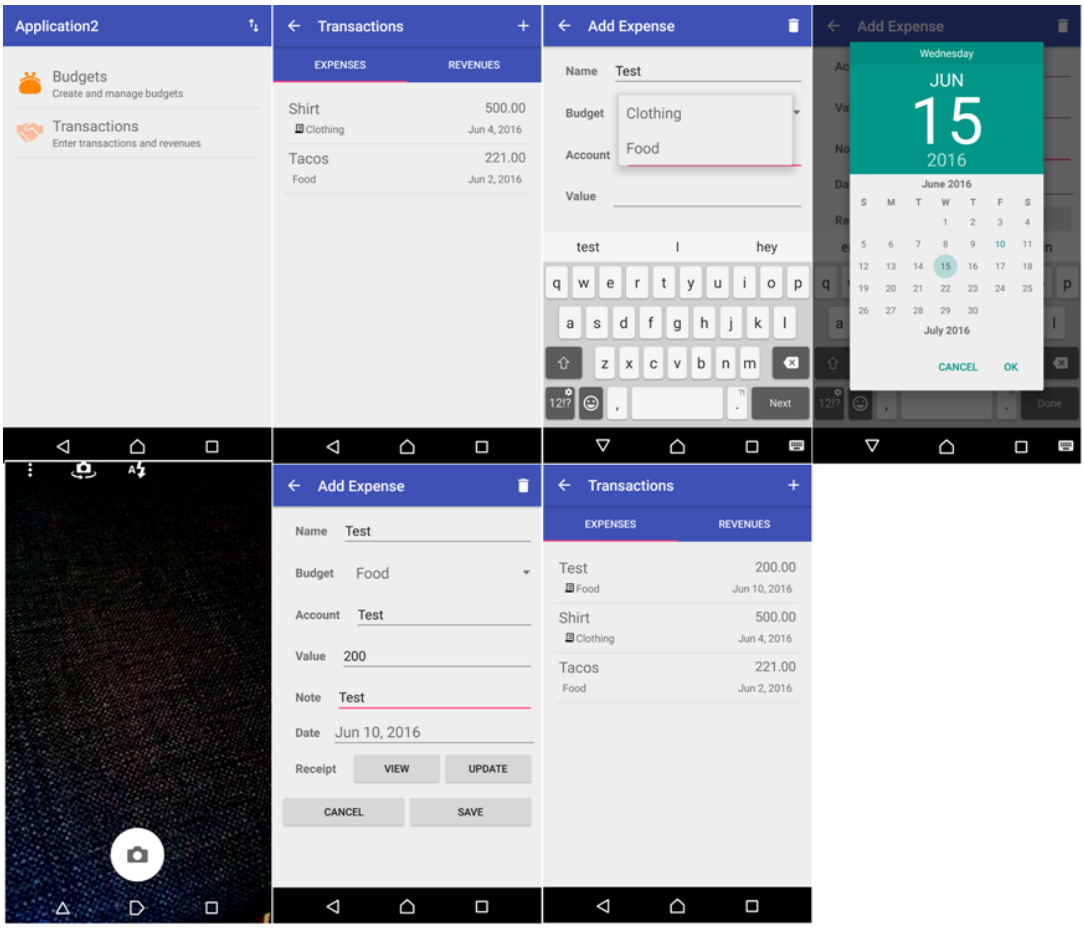


Figure 3.7: Road map of second user case

3.4 Tools

3.4.1 Trepn Profiler

Trepn Profiler[9] is a power and performance profiling application for Android devices and with additional features for devices with the Qualcomm Snapdragon processor since the profiler is a product of Qualcomm. Trepn offers customisable overlays which can display data in real-time and can measure battery power, CPU and GPU frequency and utilisation, network usage and different states of the device as GPS or screen. The collected data can be saved as .csv- or .db-format which is helpful for analysing the information. In section 1.5 previously written papers which are related is presented and in one article written by Arnesson[2], Trepn was used. Arnesson used Trepn as a tool for gathering the data when he compared different cross-platform tools by measuring performance and from his experience the tool was reliable and useful. Furthermore, Bakker performed an evaluation of different profilers for Android where he concluded that Trepn is the best for measuring the actual usage of components.[4]

3.4.2 Sony Experia Z1

The mobile device which was used throughout the whole project was a Sony Experia Z1 from 2015 with Android 5.1.1 installed. The device has a Qualcomm MSM8974 Snapdragon

[9]https://developer.qualcomm.com/software/trepn-power-profiler

800 processor (which enables additional features with Trepn Profiler), a Quad-core 2.2 GHz CPU and the GPU is an Adreno 330. The battery is a non-removable Li-Ion 3000 mAh battery and the screen in 5 inches in diameter with 1080 x 1920 pixels, which results in a 441 ppi pixel density. The device was provided by Valtech and used for development, performance analysing and during the user tests.

3.4.3 GitHub

GitHub is a service for hosting Git repositories which is web-based and is the largest host of source code in the world with more than 15 million users. Git itself is a command-line tool for version control which was developed in 2005 by Linus Torvalds, the creator of Linux. With Git, the developer can track the changes done to the code by frequently committing the project. When a commit is done, Git takes a snapshot of the code and stores the differences made. The result of this is a stream of versions of the project which the developer can roll back too and provides security as the developer does not have to worry about breaking the code. Furthermore, branches can be created and merged which allows different features of the project being developed separately and combined when completed.[13][6]

3.4.4 Sublime Text 3

The editor used during the development of the application was Sublime Text[10] which is a cross-platform text editor for code and markup. Version 3 was released in 2013 and by default Sublime supports many languages and snippets for framework but can be extended with numerous plug-ins. The packages is installed by Package-Control which is a third-party package manager for Sublime Text. With the use of the manager, the package Babel[11] was installed which adds support for language definitions for ES6+ JavaScript with React JSX syntax extensions.

3.5 Analysis

The answers from the user test which is described in section 3.3.1 were summarised and plotted as graphs and there were three different results which could be gathered from their tests. The first one displayed how many users could distinguish the real Android application. That is how many answered the correct native application, how many guessed wrong and how many user which did not know. The second result showed the certainty of the users which were able to distinguish the native application. The third summarised the user experience as the user answered whether or not he or she enjoyed the React Native application and if they would not mind using a React Native application. Together with comments on their choice, solid results from the evaluation of replication could be established and is presented in section 4.1.

The performance tests were done separately from each other and each test was done three times in order to obtain the mean for a more just and complete result. Firstly, all tests were done on the android application and consequently the React Native application was tested. The sampled data were saved as a csv-file which is a comma-separated value file and can is easily imported to Excel where the data can be combined, categorised, calculated to a mean value but also create graphs from the original data. The resulting data is displayed and explained in section 4.2.

[10] https://www.sublimetext.com/3
[11] https://packagecontrol.io/packages/Babel%20Snippets

4 Results

This chapter displays the results from the two tests which were performed, the replication test and the performance test. Section 3.3.1 describes the replication test more thoroughly and the method used for the performance test is described in section 3.3.2.

4.1 Replication

The first research question sought out to examine if it is possible to create an application in React Native that is indistinguishable from a native Android application. Consultants from Valtech were given an opportunity to test the two created applications and had no restrictions of what they were allowed to do. The applications were reset after each test and terminated which would provide the same experience for all users. When the user had a verdict, the user filled out a simple form which contained questions regarding their experiences and of course, a question about which application was React Native.

Firstly, the users were questioned if they could distinguish the application which was created in React Native. The applications were named Application1 and Application2 where the first one was the React Native application. As can be seen from figure 4.1, no one guessed wrong and chose Application2 as the React Native application. Out of the 25 consultants, 18 users chose the correct application that was created in React Native and 7 users did not know and therefore didn't answer. From the comments that the users were allowed to leave, the most common reason for being able to distinguish the correct application was the transition between scenes. The ones that had chosen Application1 said that the transitions did not resemble a native animation while the ones who didn't know the correct answer said that there was a difference in animations but that they could not say which animation was native. Furthermore, a few users also noticed a delay in the start-up time for the applications and therefore guessed that the slower application was the React Native application.

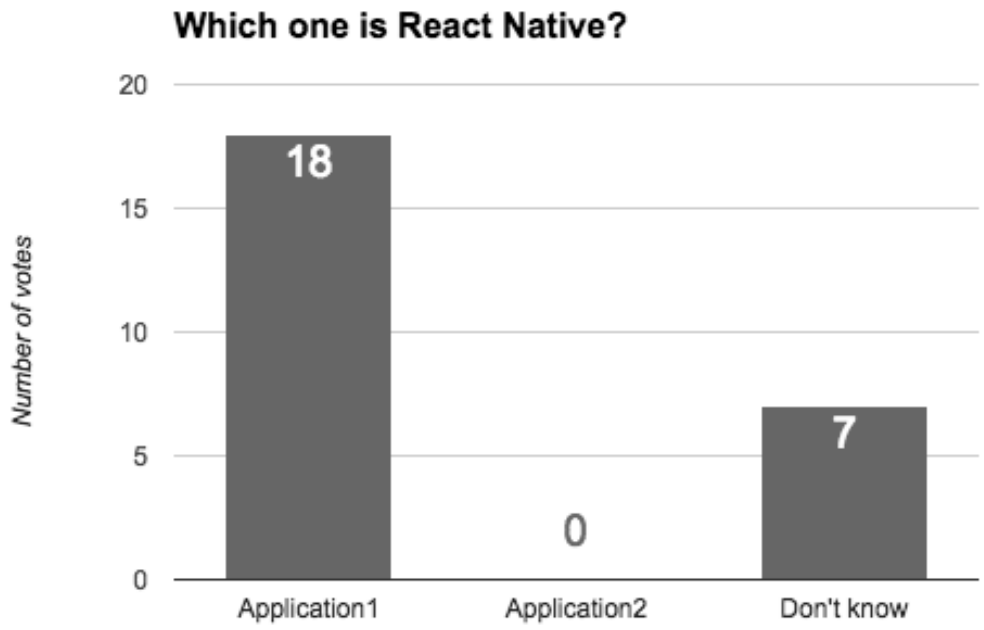


Figure 4.1: User responses where Application1 was the React Native application

When the user had chosen an answer on the first question, the ones who answered one of the applications were provided with an additional question. The question asked how certain they were of their choice. This question was necessary in order to determine the certainty of the users answer in order to be able to analyse the previous answer and see how clear the differences were. The results from the 18 who had chosen the correct answer is displayed below in figure 4.2. It is clear that the users who could distinguish the React Native application were certain of their answer since no one described their certainty as a "Motivated guess". Furthermore, 12 users knew that Application1 was the React Native application while the remaining 6 were pretty sure that it was not the Android application. The ones who answered "Positive" said that they knew how the animations should look since they had experiences in developing and using Android applications and that they did not recognise the transitions of Application1. The users who said that they were "Pretty sure" had similar motivations and some thought that since the start-up time of Application1 was slower, it probably was React Native but they could not be completely sure.

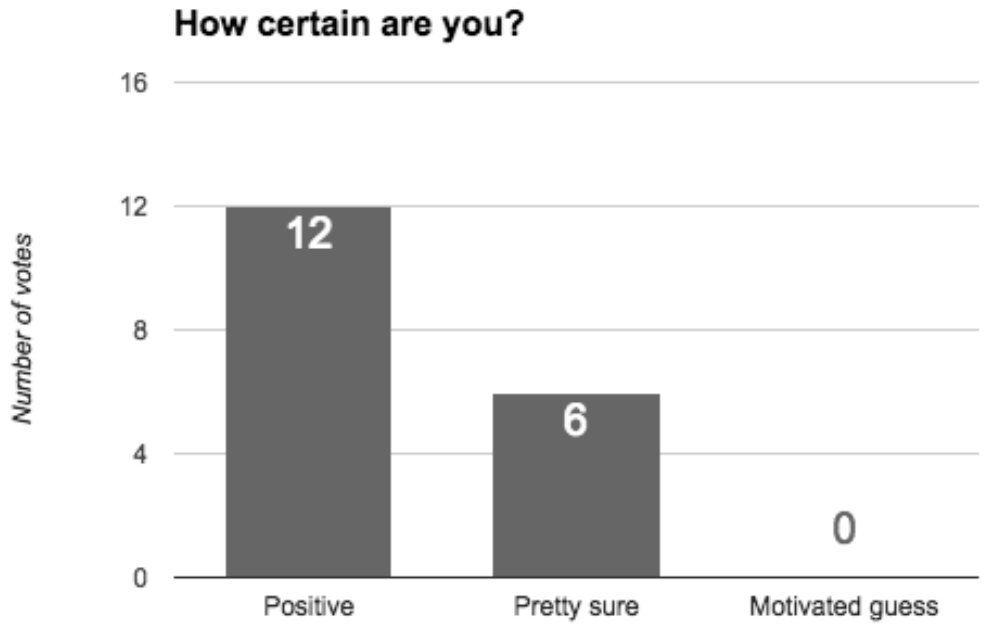


Figure 4.2: The certainty of the people who chose Application1 in the previous question

Lastly, the final question of the form asked the user if the faults of the React Native application was sufficiently severe for them in order to not use the application at all. Even though there might be some flaws and dissimilarities between the two applications, it is important to know if a React Native application can still be good enough for user to use it. The question itself was a simple yes or no question and the results can be seen in figure 4.3. As can be seen, all except one user did not have any problems using the application which was created in React Native. The 7 consultants who did not know which application was created in React Native thought it could be native but were also told which application was React Native and despite this, they did not have any problems with the React Native application. The one user who would not want to use a React Native application said that the animations felt slow compared to an Android application which made him restive. However, the user also pointed out that he is a perfectionist and if the transition problem would be solved, he would not mind a React Native application.

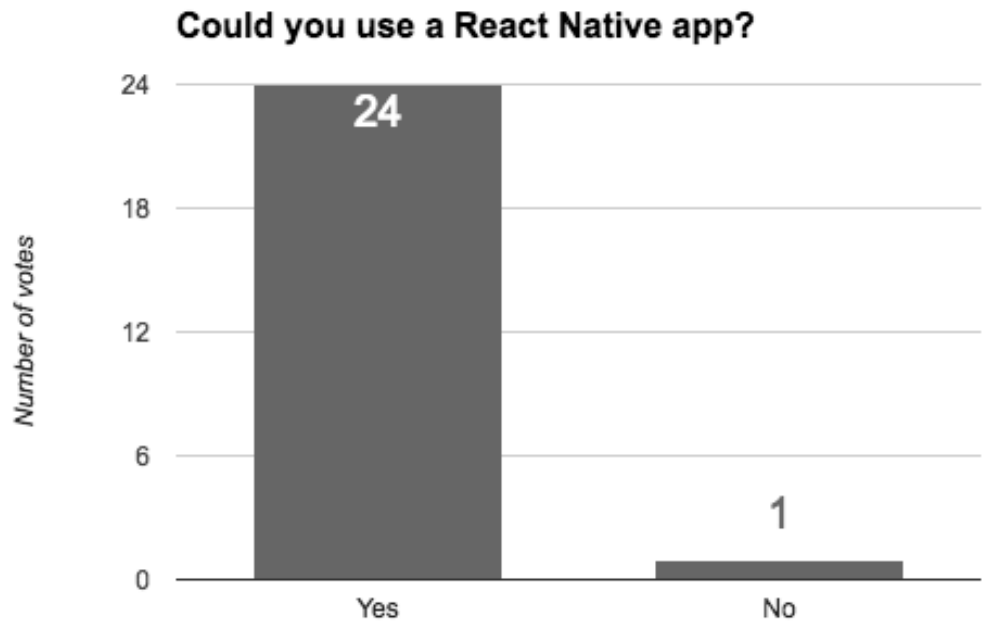


Figure 4.3: Results of how many users which still would use a React Native application

4.2 Performance

The aim of the second research question was to investigate if an application created in React Native have the same performance as a native Android application and, if not, how much differs. As described in section 3.3.2, three tests were conducted where the first one examined the performance of the two applications when they were idle, the second test measured the performance when a budget was created and removed, and the third test studied the efficiency of the applications as a transaction was created. The road map of the two latter tests can be seen in figure 3.6 and 3.7. Furthermore there were four aspects of performance which were measured: GPU frequency, CPU load, memory usage and power consumption.

4.2.1 First test

The first tests when the applications were idle was carried out for 30 seconds and three runs for each application was captured. The average of the three sets of data were calculated and are presented below in form of graphs. Firstly as can be seen in figure 4.4, the GPU frequency dropped significantly when the application was started and displayed for the user. When the application was switched out and run in the background, the frequency incremented and the frequency of the Android application rose to roughly 340 MHz before declining to 320 MHz along with the React Native application. This is the frequency measured throughout the rest of the test, except a small dip for the Android application after about 4.5 seconds. The average GPU frequency of the Android application was 315.78 MHz and respectively 315.58 MHz for the React Native application.

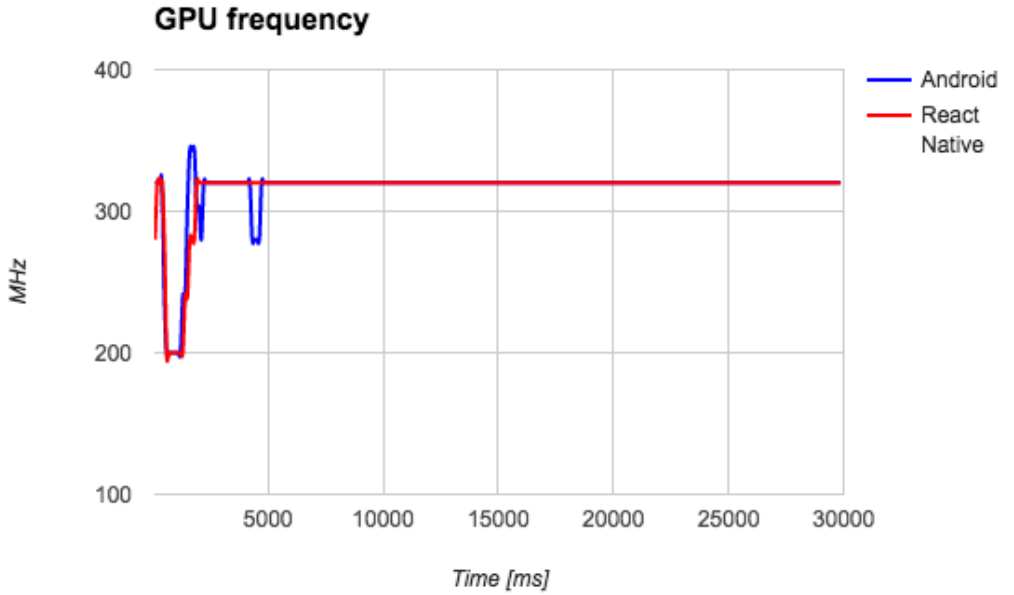


Figure 4.4: GPU frequency of idle applications

The graph of the CPU load in figure 4.5 is somewhat more difficult to read due to the jagged results but it can be noted that both applications had to a high degree the same CPU load expect the two distinctive spikes after 7.6 and 7.8 seconds for the React Native application. The average value for Android was 57.60 % and 57.89 % for the React Native application.

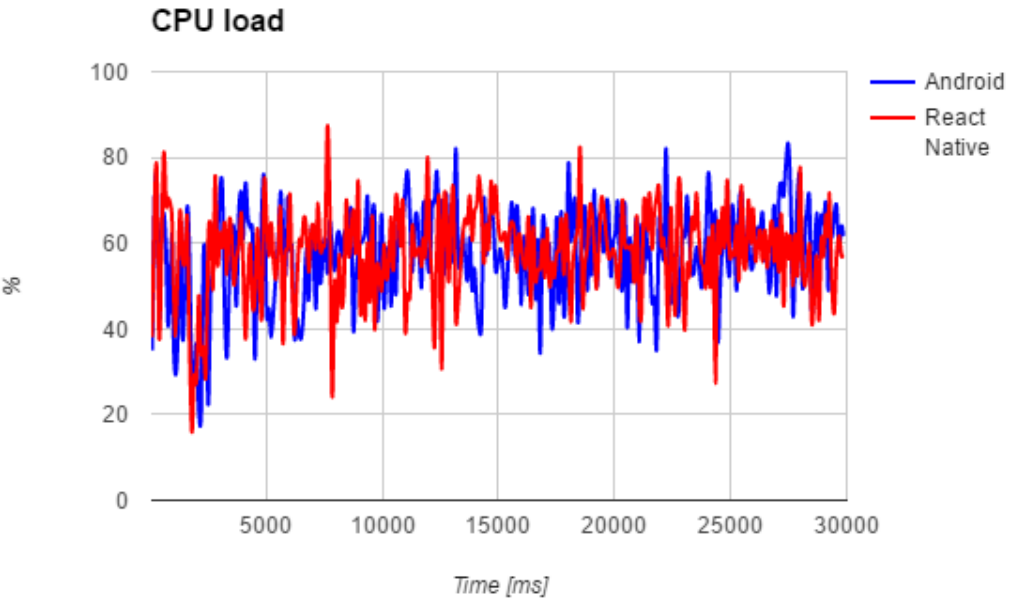


Figure 4.5: CPU load of idle applications

Furthermore, the memory usage of the applications had the same pattern with an increase of usage at the initialisation of the applications but with a steady decline afterwards. The clear difference is the somewhat lower usage of the Android application throughout which required a memory of 1607 Megabytes while the React Native application had a memory usage of 1616 MB in average.

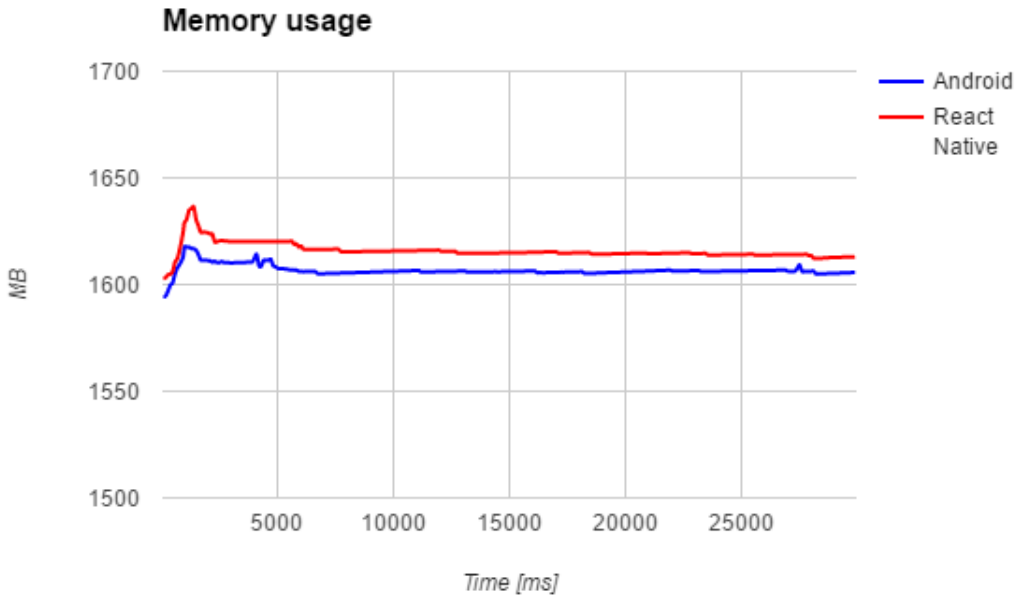


Figure 4.6: Memory usage of idle applications

Lastly, the power consumption also had the same pattern of the applications. The consumption was immense at the beginning when the applications were started but descended over time. Noteworthy are the accretions of the React Native application after 1 second and the Android application after 4.5 seconds. The average power consumption was 1362 milliwatts and 1350 milliwatts for the Android application and the React Native application.

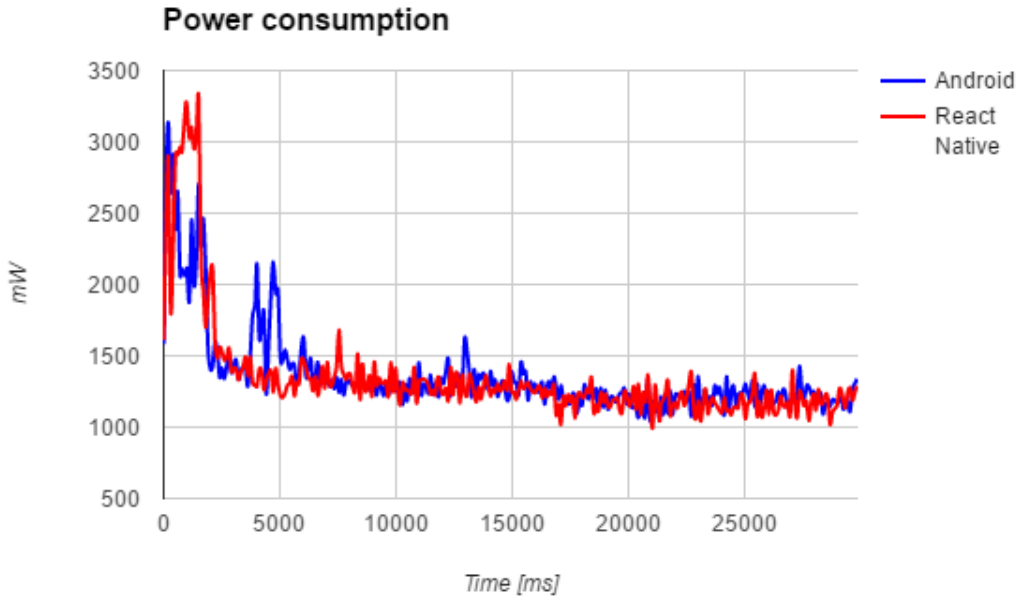


Figure 4.7: Power consumption of idle applications

4.2.2 Second test

The second test is described in section 3.3.2 and consists of the user adding a new budget and subsequently deleting it. In order to perform these actions, the tests took roughly less than 12 seconds for both applications to execute. The main actions performed during the test are presented below in figure 4.8.

Action	Time [s]
Navigate to Budgets	1.5
Pressing "Add budget" icon	3
Saving budget	7
Pressing "Edit budget" button	9
Deleting budget	11

Figure 4.8: Actions performed at approximate timestamps for second performance test

The GPU frequency of the applications when handling budgets are displayed below in figure 4.9. When the user navigated to the list of budgets, the frequency dropped to 200 MHz for the Android application and 240 MHz for the React Native application. The frequency once again dropped when the scene for adding a budget was rendered after approximately 3 seconds. Afterwards the Android application had a stable frequency at 320 MHz while the frequency of the React Native application had some drops. Subsequently, when the user saved a budget and was directed to the list of budgets the frequency once again dropped and continued to increase and decrease throughout the rest of the test as editing a budget and removing a budget was done, resulting in the previous scenes re-rendering with a few minor changes. The average frequency was 291 MHz for both the Android and the React Native application.

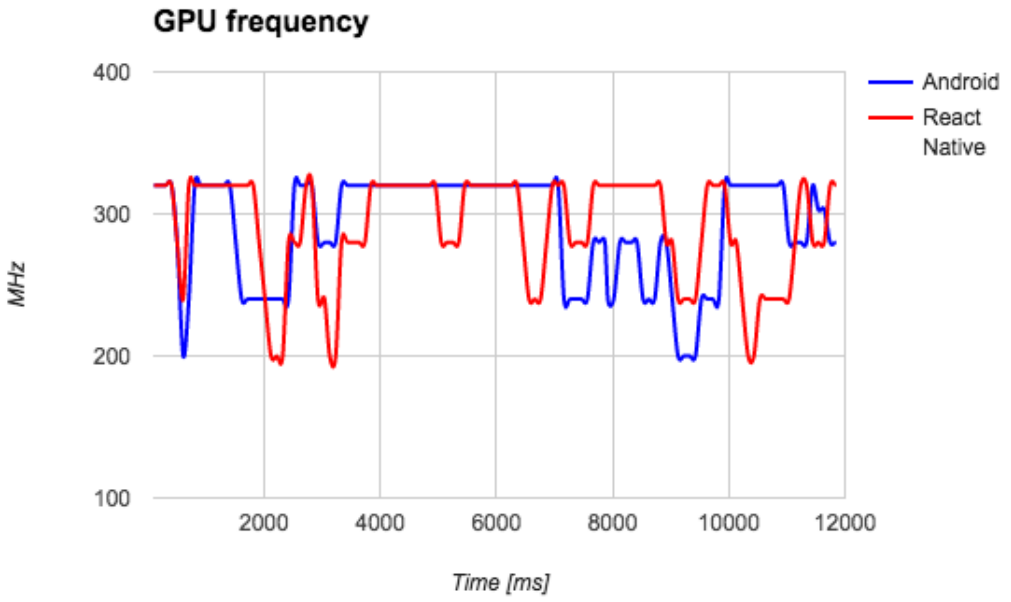


Figure 4.9: GPU frequency when handling budgets

The load of the CPU for the Android and the React Native application are presented in figure 4.10. As the earlier test, the graph is uneven but shows a high CPU load at the beginning of the test with an enormous drop after the list of budgets are displayed and the button for adding a new budget has not yet been pressed. Subsequently the load declined while the user was editing the values for the fields but increased when the user saved, edited and deleted that budget. Furthermore, the graph displays the CPU load of the Android application being slightly less than the React Native one which is also confirmed with the Android application having an average load of 36.14 % and 41.64 % for the application created in React Native.

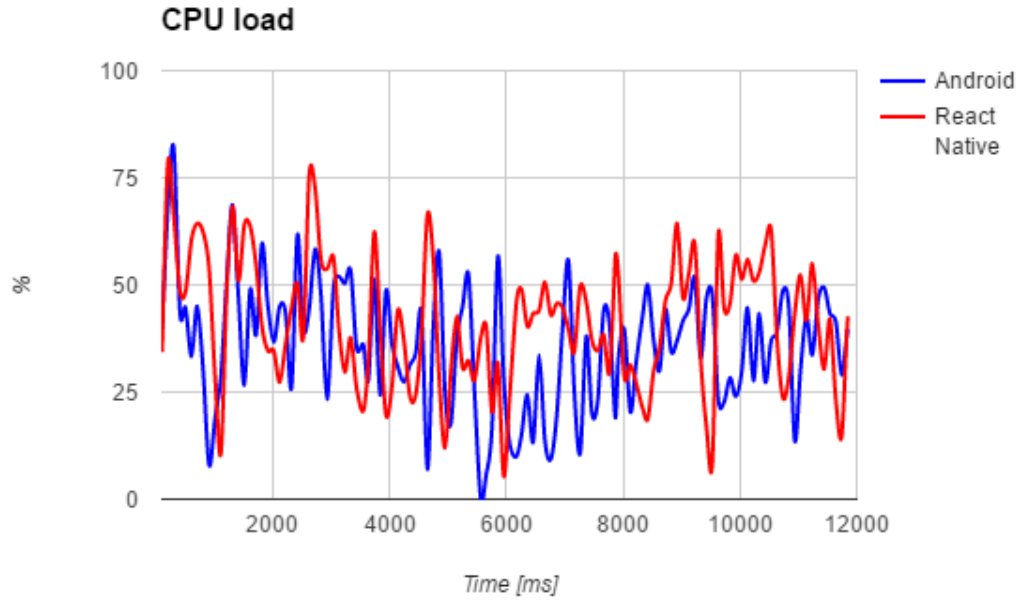


Figure 4.10: CPU load when handling budgets

The memory usage in the second test which can be seen below in figure 4.11 was similar to the first test as both applications had a similar usage throughout the whole evaluation. The most distinctive differences are the increase of the usage for the React Native application at the beginning of the test and the overall greater usage with an average memory usage of 1620 Megabytes for the Android application and 1672 Megabytes for the React Native one.

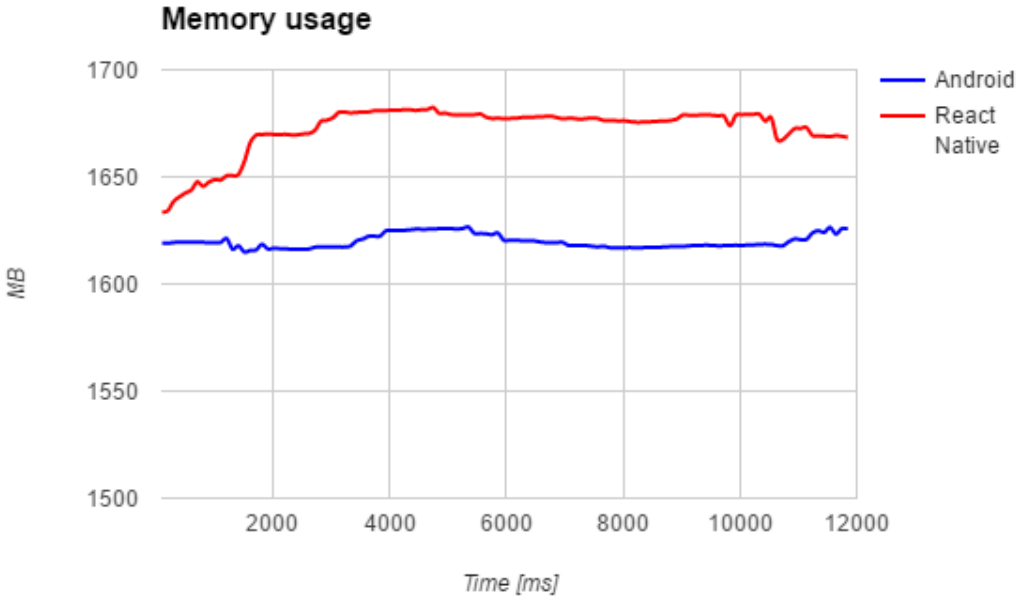


Figure 4.11: Memory usage of idle applications

The power consumption differed from the first test regarding the initial drainage since it had to be initialised in the first test while in the second test the application was already running when the measurements began. The graph in figure 4.12 displays increases of power consumption at key events and performed actions. Initially the drainage increased when the list of budgets were rendered and after 3 seconds when a budget was to be added. Subsequently, drops and inclines of consumption was registered and when the budgets asadded after 7 seconds the drainage of the Android application was very high when the new budget was saved and the list of budgets was displayed. Finally, a higher drainage can be viewed at the end of the test where the React Native application had a spike at 10.5 seconds as the scene for editing a budget was displayed and the Android application had a higher consumption after 11 seconds as the budget was deleted. To summarise, the average power consumption of the Android application was 1749 milliwatts while the React Native application had a consumption of 1803 milliwatts.

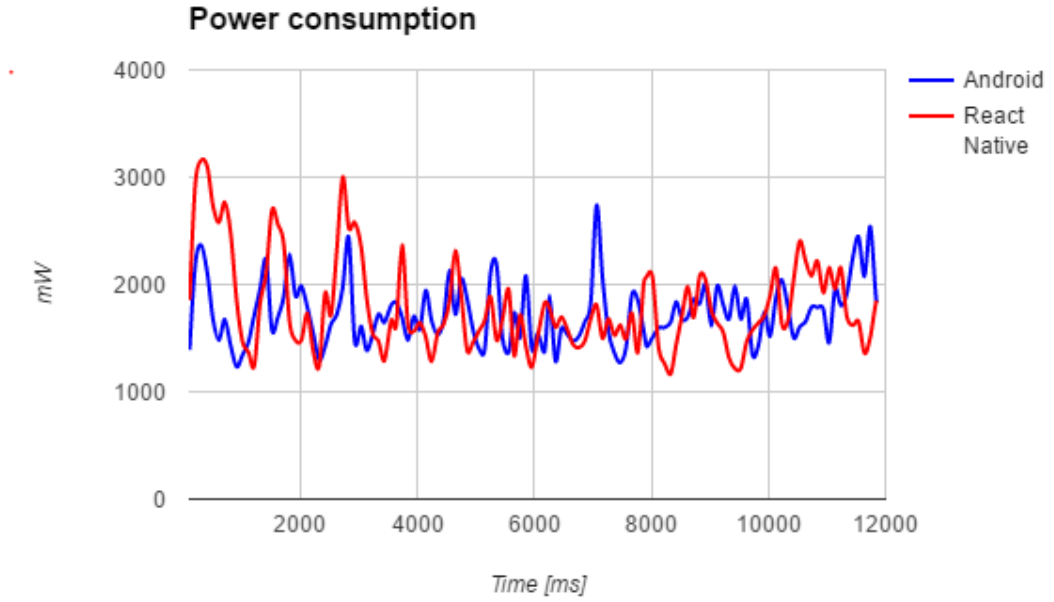


Figure 4.12: Power consumption of idle applications

4.2.3 Third test

The third and last test case is also described in section 3.3.2 and involved the user adding a new transaction with certain values, changing date, capturing a receipt and finally saving the transaction. This test however had a significant difference regarding the execution time of the two applications. The android application was able to perform all action under 23 seconds while the React Native application was slower and finished at roughly 30 seconds. The difference in execution time occurs after the calendar is opened and therefore, the results of the tests can not be displayed in the same graphs due to the differences and the timestamps for specific actions are different for the two applications as can be seen in figure 4.13.

Action	Time Android[s]	Time React Native [s]
Navigate to transactions	1.5	1.5
Pressing "Add transaction" icon	2	2
Open calendar	13	13
Open camera	16	17.5
Take picture	19	23
Save transaction	21	27.5

Figure 4.13: Actions performed at approximate timestamps for third performance test

Figure 4.14 and 4.15 displays the GPU frequency of the applications during the third test. Both applications had the same initial measurements with drops of frequency in the beginning due to the rendering of scenes for transactions, adding a new transaction and after 13 seconds opening the calendar. The graph displaying the Android application shows how the frequency dropped to 200 MHz when the camera was opened after 16 seconds and once again when the image was captured. Furthermore, the GPU frequency descended when the new transaction was saved and redirected to the list of transactions. The React Native application have the same pattern however with the respective timestamps which can be seen in table 4.13. The average GPU frequency of the Android application was 289 MHz while the React Native application had a frequency of 294 MHz.

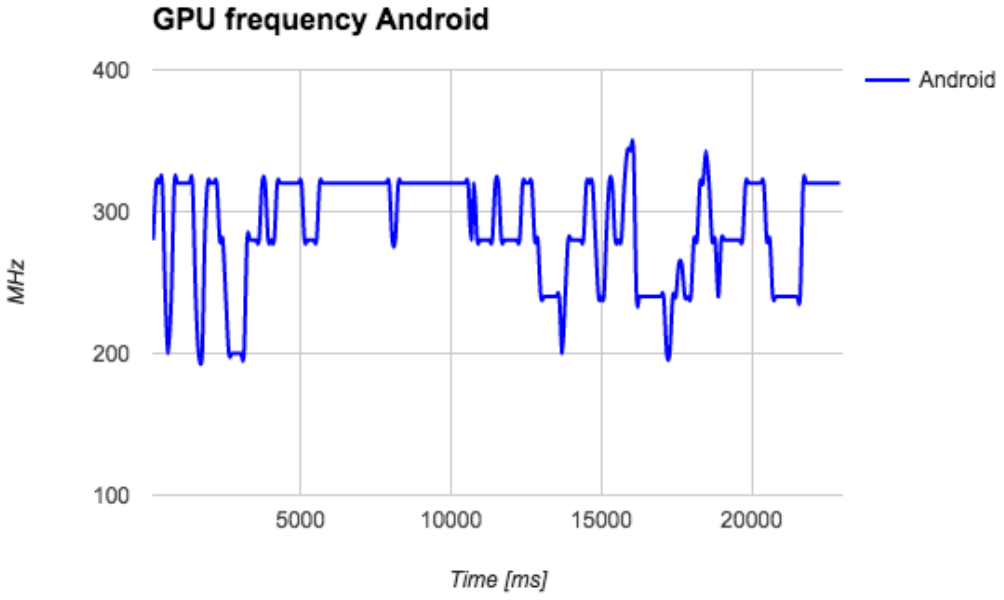


Figure 4.14: GPU frequency of Android application when handling transaction

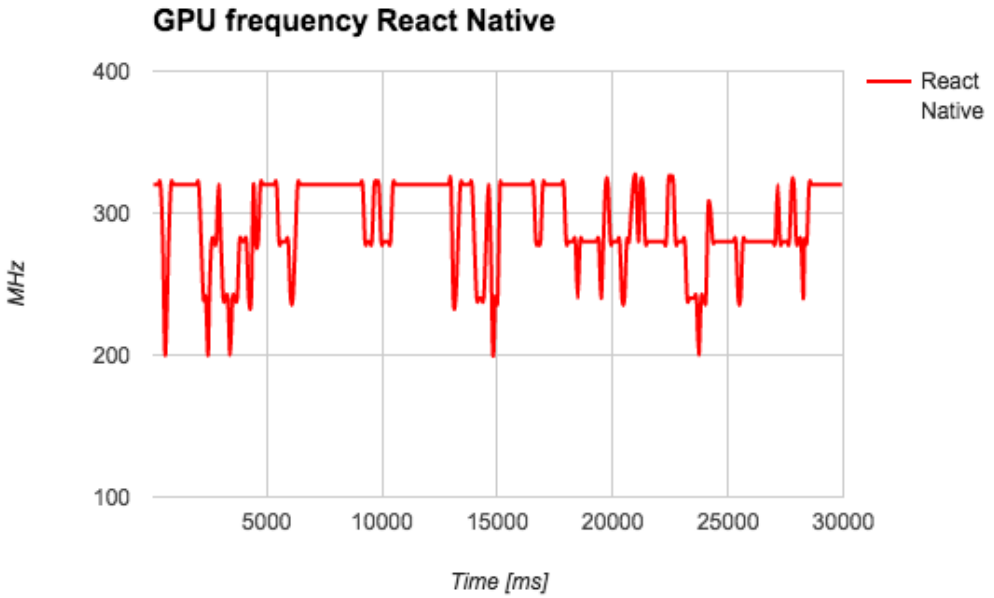


Figure 4.15: GPU frequency of React Native application when handling transaction

The CPU load of the Android applications can be seen in figure 4.16 and figure 4.17 displays the load for the React Native application. The comparison of the two graphs is difficult but there are two values which protrude. There was a sizable drop of CPU load before the navigation to transactions occurred. Furthermore there was a significant general increase of CPU load for the Android application after 16 seconds and after 20 seconds for the React Native application. Conclusively, the average CPU load of the Android application was 37.88 % while the React Native application had an average of 41.76 %.

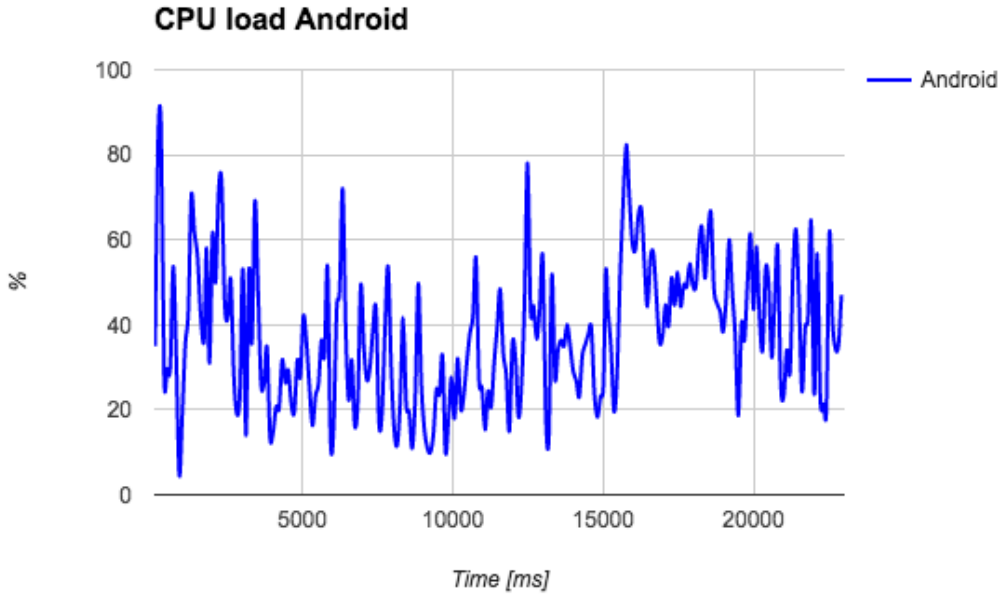


Figure 4.16: CPU load of Android application when handling transaction

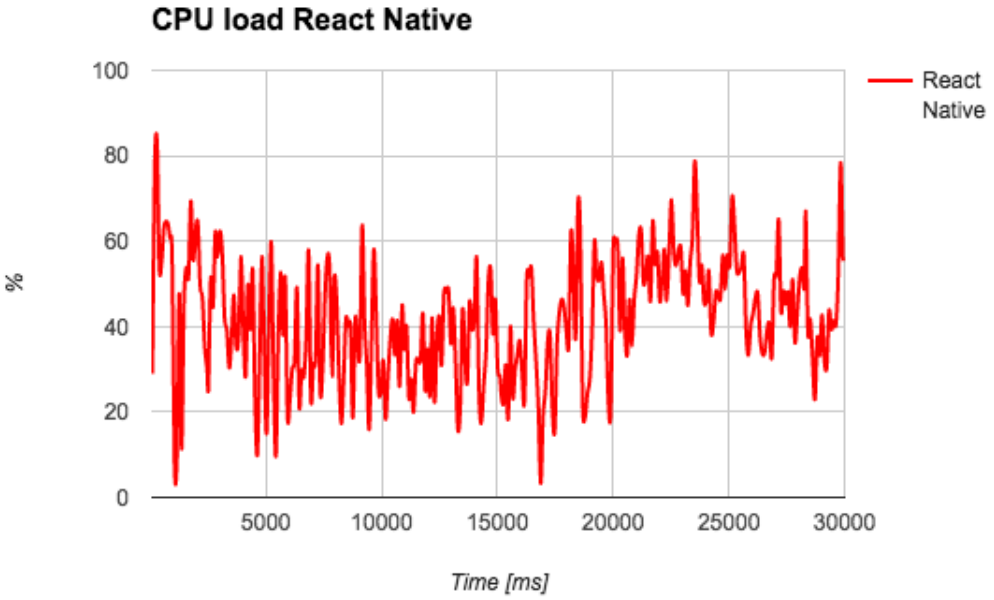


Figure 4.17: CPU load of React Native application when handling transaction

Figure 4.18 and 4.19 displays the memory usage of the two applications. The memory usage of the Android application slowly increased over time, starting at around 1560 MB and reaches a value of 1586 MB. However, there was a slightly more immense increase at the beginning of the test and after approximately 16 seconds there was a hefty temporary increase of usage which lasted for two seconds before dropping once again. The memory usage of the

React Native application also increased over time but not at the same level or steadiness. The usage began at 1600 MB and had reached nearly 1700 MB at the end of the test. Furthermore the memory usage increased drastically after two seconds and once again after 23 seconds. The average memory usage of the Android application was 1582 MB and 1655 MB for the application created in React Native.

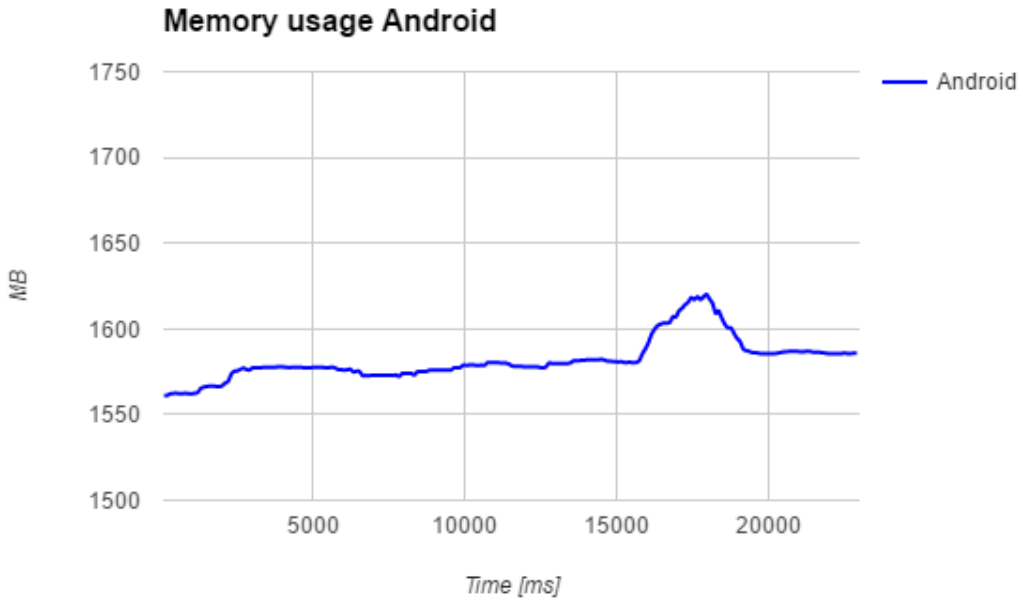


Figure 4.18: Memory usage of Android application when handling transaction

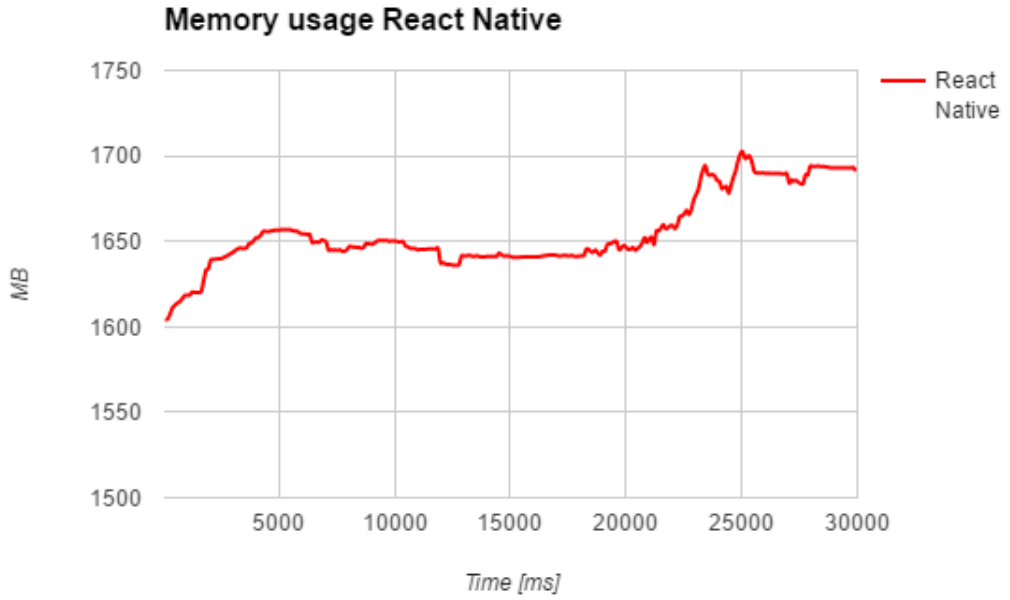


Figure 4.19: Memory usage of React Native application when handling transaction

Lastly, the power consumption of the Android and React Native application can be seen in figure 4.20 and 4.21 and clearly shows the pattern which is equivalent for both. The power consumption spiked at the beginning of the test when the application was switched to in order to perform the test. Subsequently, the drainage spiked and dropped throughout until the camera was opened and the consumption increased drastically after approximately 16 seconds for the Android application and 20 seconds for the React Native application. When the picture was taken, the power consumption decreased to the previous levels. The average drainage of the Android application was 1992 mW and 1972 mW for the React Native application.

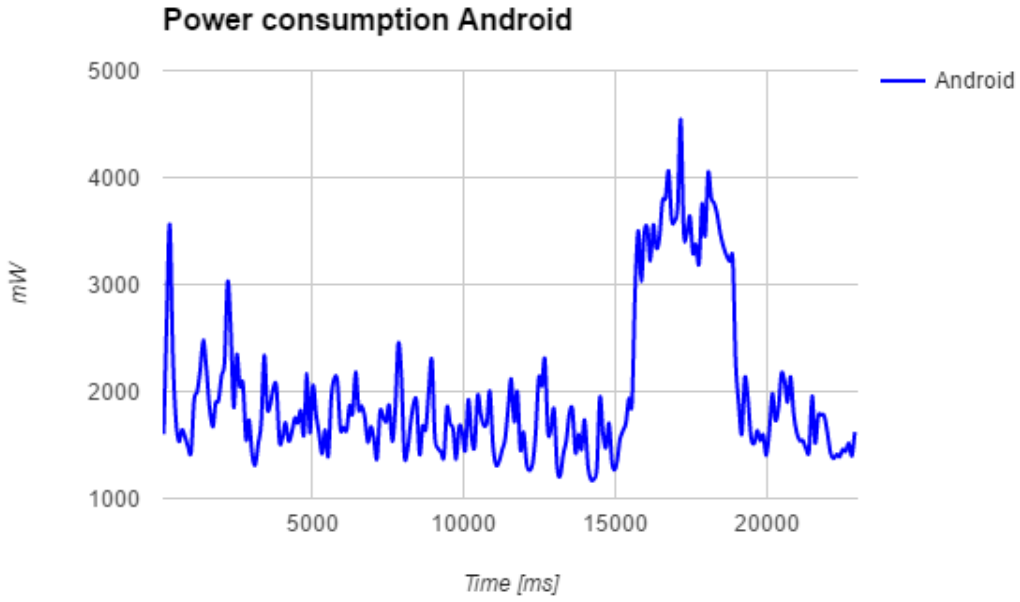


Figure 4.20: Power consumption of Android application when handling transaction

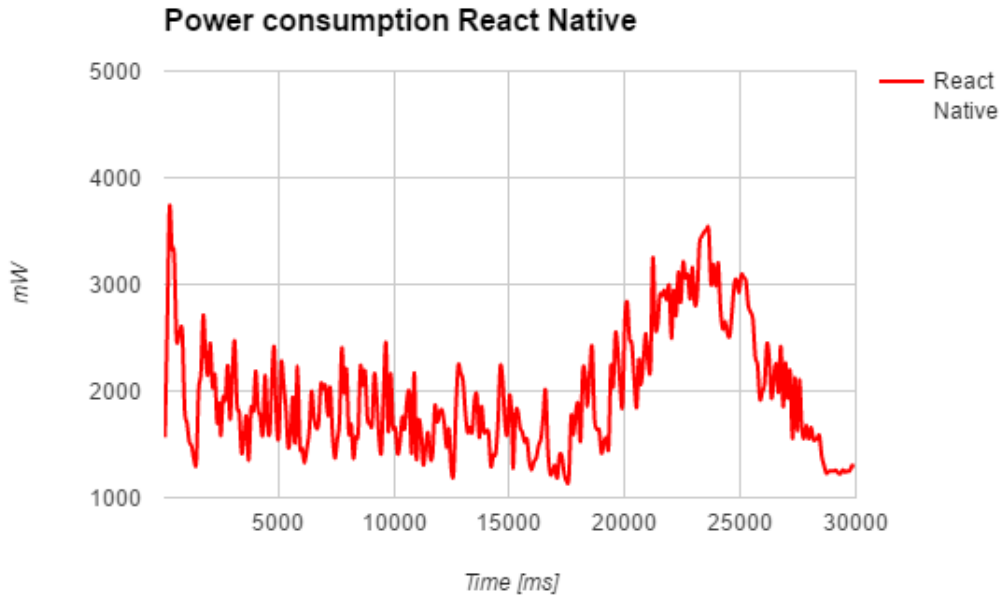


Figure 4.21: Power consumption of React Native application when handling transaction

5 Discussion

This chapter sums up and discusses the results which are earlier presented and explains the reason for the data which was gathered. Furthermore it reviews and explains the methods used in the thesis and explains the reason for the choices which were made. Lastly, it concludes and explains the development of the React Native application and how this process felt and the overall feeling it provided.

5.1 Results

The previous chapter displays the gathered data from both the user test for replication and the performance tests. The provided results are discussed below.

5.1.1 Replication test

The results from the replication test, described in section 3.3.1 where employees at Valtech tested two applications and were given the task to choose which application had been created with React Native, was successful. The results concluded that most of the users could distinguish the React Native app due to minor differences (mainly the transitions between scenes) but the users did not mind using the React Native application. The problem with the transitions was a result of the Android application's usage of Activities for the scenes which have a certain animation when the transition occurs. One can change the different transitions in React Native but no animation is the same as the native animation for a new Activity. The animation which was used for transitions in React Native was therefore implemented to look like the transitions between the Fragments in Android. However, the animation is not identical and the animation itself is a bit slower than in native Fragments and some of the users could identify this discrepancy as a non-native behaviour. Despite a result which clearly showed that the React Native application was not able to completely have the same look and feel as a native Android application, it was good enough for the users which still did not have any problems using a React Native application.

5.1.2 Performance test

As can be seen from every test in section 4.2, the React Native application does not have the same performance as the Android application. However, the difference is less than what could be expected and the React Native application had a lower average power consumption during the third test. The reason why the React Native application is not able to perform as good as a native application is quite obvious, a native approach is compared to an hybrid and React Native does not build in ART which is described in section 2.1.1. Since React Native applications run in an embedded instance of V8 (JavaScriptCore for iOS) the applications does not use the same runtime and therefore there will be differences in the performance since ART is created solely for Android while the V8 engine runs on different systems and is not optimised for Android. Even though V8 is fast, it can not achieve the same results as ART and therefore the results with slightly better values for Android is achieved.[25][1]

First test

The first test measured the performance of the applications as they were instantiated and how the applications were handled when ran in the background. All four data points displays the same result with the Android application having better performance and without any particular data that stands out. It is clear that the application required the most memory and power when the application is started, the GPU and CPU is utilised as most when the application is opened and when the application is switched away and ran in the background, the values decline.

Second test

The second test displays a more interesting result regarding user interactivity with the application since the test measured the performance as the user added and then deleted a budget. The GPU frequency in figure 4.9 shows how the GPU drops for both applications when the user interacts with the screen, for example when the user navigates to budgets or when the budget is saved. However, the declines of frequency occur more often for the React Native application as the drops also take place when the user provides input to the fields when adding a budget, drops that don't occur for the Android application. Moreover, the React Native application have decreases of GPU frequency with a value of 200 MHz more times and the greatest difference is when a budget is edited. The overall GPU result displays that the Android application has a more stable frequency.

The CPU load shows a similar result as the React Native application has most of the time a slightly higher load throughout the whole test. The CPU load have a distinctive drop after 1 second which is when the application had started and before the user has navigated to budgets. When the user adds a new budget, the CPU load increases and subsequently declines until the budget is saved where it once again increases.

The memory usage for the second test does not have any unique events. However, it can be seen that the usage is quite similar in the beginning but as the interactivity starts (the user navigates to budgets after 1.5 seconds) the memory usage drastically increases for the React Native application while the Android application remains at a similar size throughout the whole test.

The power consumption displays the result which best demonstrates the performance differences of the two applications. One can see that the three spikes in the beginning of the test when the applications is opened, the user navigates to budgets and when the button for adding a new budget is pressed, the drainage increases for both applications but the React Native application have higher values. However, after seven seconds, there is an increase of consumption that is higher for the Android application. This is when the application saves a budget and the lower consumption for React Native is due to the use of Realm which is

much faster and more efficient than SQLite which is the technique used for the Android application.[29]

Third test

The third and final test provided results which were similar to the previous test. This test, as stated and further explained in section 3.3.2, measured the performance as a user adds a transaction with all values provided and an image captured. The greatest difference was not the values from the data points but the time required to perform the actions. The Android application was faster and only took approximately 23 seconds while the React Native application finished at about 30 seconds. This was caused by some parts of the React Native application which were not able to be rendered and displayed as fast as the native application. As can be seen in figure 4.13, the user was able to perform all actions up until opening the calendar at similar times for both applications. However, the calendar component was the first part of the application which was a bit slower in the React Native application as it took about 4.5 seconds for the application to open the calendar, pick a new date and close the dialogue while the Android application performed the same tasks in approximately 3 seconds. Subsequently, the user pressed a button which opened the camera and then captured an image which was saved, the camera was closed, the user was redirected to the previous scene and then the transaction was saved. For the Android application, this sequence took approximately 5 seconds while the React Native application required 10 seconds in order to perform the same actions. The difference is protruding and the reason why it took twice the amount of time to do this was because the camera component used in the React Native application was a external component. If Facebook were to create their own, the time required for instantiating and using the camera might be much less.

The GPU frequency for the third test did not provide any further insight as it also confirmed that the frequency drops when scenes are rendered and that the overall frequency was higher for the React Native application.

The CPU load provided the same results and it can be seen in figure 4.16 and 4.17 that the CPU required the most load when the camera was instantiated after 16 seconds for the Android application and after 17.5 seconds for the React Native app. An interesting value is the CPU load for the Android application when the calendar is opened after 13 seconds which reaches a value of nearly 80 % whereas this event does not provide any unique values for the React Native application. The reason for this is the way the *Datepicker* was implemented in the React Native application. The component is created inside an invisible modal when the scene for adding a budget is rendered and when the user presses the text for changing the date, the modal is simply set to visible. This means that the CPU load required when the calendar is opened is greater for the Android application which at that point instantiates the *Datepicker*. The earlier instantiation of the date component is also a reason why the overall CPU load is higher at the rendering for the *AddTransaction* scene for the React Native app.

The memory usage displays a similar result as the other tests regarding the memory. Figure 4.19 displays how the usage increases when the camera captures an image and continues at a high rate. A spike can be seen after 23 seconds which is when the image is captured and the spike after 25 seconds is when the image itself is saved on the mobile device. However, the memory usage for the Android application did not follow the same pattern as can be seen in figure 4.18. The usage did not increase when the image was captured but when the camera was opened and subsequently the Android app was able to rapidly drop the usage as the image was captured.

The last data point which was measured was power consumption and shows how both applications required more power when the applications are instantiated and when scenes are rendered (for example when the scene for adding a transaction is displayed after about 2 seconds). The only difference, except the slightly higher values for the React Native app, is the drainage when the camera is opened. The Android application had an enormous spike

after 15 seconds which lasted about five seconds and reached a consumption value of 4540 mW while the React Native application had a longer time span of approximately 10 seconds but with far less consumption (highest value was 3504 mW). The reason why the React Native application had a lower drainage during the use of the camera was because the camera which was used for React Native was, as earlier mentioned, an external component and only had support for basic functionality. The camera that was used in React Native only had support for capturing the image while the Android camera had support for switching cameras, enabling flash, zooming and different settings. The usage of this camera therefore required more power to be used and caused the spike which can be seen in figure 4.20.

As earlier mentioned, the average power consumption in the third test was lower for React Native and was the only test and data point which displayed a favour for React Native. First of all, one could argue that the average values for the third test does not provide a fair view since the Android application was approximately seven seconds faster which results in the overall consumption being far more for the React Native application. However, the reason why the average consumption was lesser in the React Native application was due to the instantiation and usage of the camera which had less support and was less complicated than the camera used in Android. The external camera component which was used did not have support for more features than the basics and due to the limited knowledge in Android, the original application could not be altered in order to use a camera with less functionality.

5.2 Method

Section 3.2 describes how the React Native application was developed and contains pieces of code for demonstrating problems and the fixes which occurred throughout the development phase. This walkthrough together with the repository for both applications on Github should aid in replicating the applications and the tests. However, React Native is a new framework which is continuously updated and patched. This can result in parts of the application being deprecated in a near future and bugs which was encountered (as the problem with updating a list) might also be fixed. Furthermore, some components used in the application are external and since Facebook uses the open-source community to help them improve the framework, these components might be added to the framework in the future. If this were to be done, the components themselves might contain more functionalities but above all have better performance. Moreover, the results from the replication test were primarily because of the animations between the transitions. Since React Native is under development, this part of the framework might be fixed in order to provide animations which have the same look and feel as a native application. Due to these possibilities mentioned above, the results might differ in future even if the same applications are tested.

5.2.1 Using an existing Android application

First of all, in order to compare two applications, an already existing Android application was obtained. The main reason for this was to save development time and focus the development on React Native. By having a finished Android application, most of the development time could focus on creating a React Native application and allowed for a more complete application to be developed. However, since the Android application had predetermined functionalities and features, it was hard to replicate the whole application. For example, the original application allowed the user to save the current budgets and transaction in a csv-file and later on import it. Since this feature would not provide a more genuine use experience and would be time consuming to develop, the feature was removed from the Android application.

Furthermore, due to the fact that the Android app was not developed from scratch, many parts of the original application did not consist with how an Android application should be designed. This also had to be fixed in the Android app which took time from React Native

development. However, when deciding to use an existing Android application, it was clear that the application had to be open-source since faults like the ones mentioned could appear. With the source code available and editable, it was easy to change the application in order to create two similar applications without removing any features which is imperative for the original application. If the Android application was not open-source, the created React Native application would have been forced to contain features and design choices which are discouraged from the official Android development documentation.

Furthermore, since the Android application was not developed from scratch, the Android app was not designed and created in order to fit a React Native application. One could develop the Android application in order to look and feel like a React Native application which would provide a tendentious result. However by choosing an already existing application, the thesis can more accurately try to develop a React Native application which would look and feel like an Android app.

5.2.2 iOS

A big part of React Native which is not evaluated or examined in this thesis is the ability to create both an Android and an iOS application. The ability to create two applications from more or less the same code is one of React Native's best features but this thesis only focused on the Android half. This might provide an unfair result since even though the React Native applications did not have the same performance as the Android application, most of the code of the application could be used for yet another application. However, this thesis did not cover this aspect due to two main reasons. First of all, it would be time consuming to create yet another native iOS application and then create a React Native application for that application. Furthermore, there are many popular application which looks the same for both Android and iOS but it would be hard, or even impossible, to find two applications for these operative systems which are open source. This would require this project to create an Android and iOS application from scratch which would collide with the above mentioned choices of method. Lastly, by only focusing on the Android part of React Native, a more criticised view was used towards the framework. If the framework is easy to use and can create an application for Android which is as good as a native application, the ability to also compile the code with a few changes to iOS would be viewed as a huge bonus.

5.2.3 Literature

Lastly, as can be seen in the theory and methodology (especially regarding the theory of React Native in section 2.4) there is a lack of scientific literature used. The reason for this is simple, there is only one book written by Bonnie Eisenman[8] which contains relevant information about the framework and a general lack of scientific reports about React Native. Since React Native is a new framework, writers have not yet been able to write complete books which have resulted in the available sources for learning about React Native is either by reading the documentation or blog posts as James Long's post[20] regarding the bridge. However, since the procedure for developing and the theory about React Native had to be gained, these sources were the ones who had to be used in order for this thesis to be carried out. The few published books and scientific reports which existed was primarily used and when these did not provide enough information, the documentation and blog posts were accessed and used.

When using blog posts or similar online sources, it is important to make sure that the source is as reliable as possible. This was done by firstly researching the author and what experiences he or she had. An author which has experience with React Native or is one of the developers of the framework results in a more reliable post. However, it can not be assured since the person could still have misinterpreted the content or faking their credentials. Subsequently, the source of the post was investigated. By looking into the website where the post was published, it was possible to find out if the source was more or less reliable. By

only choosing posts which were posted on popular tech-blogs or similar, the reliability of the source increased. Finally, when a reliable source had been found, one had to find out if the information correlates to the documentation of React Native. If there were discrepancies, the source was not used since it may contain other false information. However, even though all these precautions were made, the sources still may not be completely true which one have to keep in mind while reading the articles.

5.2.4 Replication

In order to test how well React Native could replicate an Android application, people were allowed to test the two applications and try to guess which one was a React Native application. The users for this test were employees at Valtech, consultants which have experience in Android development and applications in general. In order to retrieve a result which would reflect the reality of applications in the market, using ordinary and random people should be used. However this thesis, as earlier stated, have a criticised view towards React Native and therefore chose to use IT consultants. These people know what to look for and can distinguish parts of an application which behaves or looks faulty, parts of an application which regular user would not notice or mind at all. By allowing professionals to evaluate the applications, the results which were provided are insights from experienced users and very demeritorious. Since the React Native application did surprisingly well on the replication test, the framework was approved by experienced critics and flaws that otherwise would not been noticed, still passed the test.

Moreover, the applications which were used for the replication test were not extensive and did not contain much functionality. Budget Watch is a small application and therefore the users could not test a lot of features. However, it was not the features of the application that were supposed to be tested but how the application looked and felt. Despite the size of the app, it contained a lot of components which are often used in Android applications. The reason why the chosen application did not have many features was because it was hard to know the difficulty of developing a React Native application and it was unknown whether or not there would be time for creating a more advanced app.

5.2.5 Performance

The thesis also measured the performance of the React Native application by using Trepn Profiler to record the data for different data points as CPU load, power usage etc. However, in order to record these results in different scenarios, the tests had to be performed manually. Even though many tests were recorded in order to obtain the same interactivity and execution time for both applications, a more accurate result could have been achieved by having automatic tests perform the desired actions for the application instead of having a user click and type values. This would remove the human factor for the tests which could have a part in the results, especially in the third test where the time to perform these action differed a lot between the two applications. However, the main reason why automatic tests were not used was due to the fact that this would require the applications the be in debug-mode and not release-mode. Debug-mode offers more development features but the application will not perform as good as when put in release-mode. Even though both applications would be in debug, it is unclear how much this affects the applications since the React Native application is sometimes incredibly slow. Furthermore, not having the applications in release-mode would not reflect a real market scenario and how actual users would interact with the applications. In order to receive values which was not affected by human errors, the tests were performed multiple times until three faultless cases where recorded which had the optimal lapse of time.

Furthermore, the performance was measured by focusing on GPU frequency, CPU load, memory usage and power consumption. However, even if GPU frequency covers the graph-

ical performance, it would be very interesting to also measure the FPS (frames per second) of the applications since it would show how well the user interface is displayed and re-rendered for the user. However in order to measure this, the measurement had to be added to the code and the applications had to be in debug-mode which as earlier stated was not an option. Finally, as earlier discussed, if a bigger application with more functionalities and features would have been used the differences in performance measurements would probably have been bigger since the test cases would cover more actions.

Trepn Profiler, described in section 3.4.1, was used in order to obtain the performance from different components of the mobile device. There are some concerns when using applications for profiling since the values which are obtained are estimations based on measurements of the hardware and other applications. In order to retrieve the most accurate result, all other applications were terminated and in order to remove any other interference, the mobile device was put into flight-mode which was possible since the application itself did not require any internet connection. In order to furthermore retrieve a valid result, the same mobile device was used for all tests and the two different applications were tested every other turn.

5.3 Development

The main purpose of this thesis and a part of the first research question was to evaluate the development in React Native and the available support for the framework. First of all, the documentation for React Native is outstanding. Setting up a new project can often be a gruesome process and be time consuming since problems in the environment often occurs. However, due to the well written documentation, the initialisation of React Native was easy and by providing the documentation with the target operating system and the development OS, a custom step-by-step page is returned and within minutes the development can begin. Furthermore, the documentation also contains how the different components are used, how they can be tinkered with but also code examples for how to use them accordingly. The only part of the documentation which was difficult to comprehend, and where a blog[23] had to be used in order to understand, was how navigation and routing works.

Due to the well written documentation and community, React Native was fairly easy to understand and use. Since the framework is built upon React which has become popular, one does not need to only focus on React Native but can get answers based on React instead. Since the services which are created in React or React Native are built by using components and uses declarative programming, specific problems are easy to solve and get answers to. For example, if one have problems with how the lists in React Native works, it is possible to investigate that specific and isolated component. The surrounding environment does not need to be explained (or at least should not be needed to be explained) since the problem lies within that component. By having a structure of an application based on components which are separated and only uses data sent as props, the classic divide and conquer paradigm is obtained which results in a problem which is often more easily explained.

One aspect in the development of the React Native application was the small amount of code which was needed. As can be seen in the repositories for the Android application[1] and the React Native application[2], less then half of the amount of code was used for the latter. This together with the simplicity of developing in React Native was the reason why the development time was far less than expected. This allowed for more time being put into experimenting with the different components and re-factoring the code. One reason why the amount of code was higher for the Android application was because of the structure of the application with Java which handles functionality while also having XML-files for the structure. This is for the most part put together in React Native and also contributes to having all code for a component in the same file, resulting in far less files and more control.

[1] `https://github.com/willedanielsson/BudgetWatch_android`
[2] `https://github.com/willedanielsson/BudgetWatch_ReactNative`

However, there are some faults in React Native and the most sever concern is the bugs in the framework. React Native is as earlier stated a new framework and one can not anticipate it to work perfectly. Bugs in the components of the framework are mostly not vital for the component to work and is often easily fixed. Furthermore, since there are a lot of components which have to be retrieved externally since they are not supported in the framework, they are more likely to contain bugs and most of them only have documentation and more features for iOS since it was released earlier. However, the largest and most severe bug was not from an external but from the List-component in the framework. This bug is described in section 3.2.5 and the problem was that the list did not update even though new data was passed down to the list as props. This bug was frustrating and is a big fault in the framework since lists are very often used in applications and having a component which does not update when new data is set in the state, which is one of the main features in React and React Native, is ludicrous.

Finally, one part of the React Native framework which is not brought up in full in this thesis is the ability to create own native modules. Sometimes the application need to access the platform API but React Native does not have a corresponding module yet. In this case, the framework is built so it is possible for developers to write native code and access the full power of the platform. This is the reason why external modules exists and were used in the Budget Watch application since modules which is not supported in the framework are created by the community and are easily added to a project. By allowing the community to add their own modules to be used with the framework, Facebook has made it possible for the framework itself to be expanded and improved not by only their employees but also developers who uses React Native.

6 Conclusion

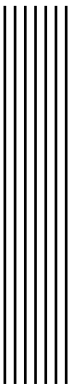
React Native is a new and interesting framework which have been praised by developers since it might be the technique which they have wished for and might change the way we create applications. This thesis have evaluated the framework by focusing on the Android compilation and compared the user experience and the performance of two applications and to a small extent the development process when developing in React Native.

Developing a React Native application was surprisingly easy and the process was fast which resulted in the application being finished earlier than expected. The documentation itself provides a lot of examples on how to add certain functionality and together with a community which produces a lot of guides and components to be used, the development is made easy even though the framework is merely a year old. The simplicity in creating a React Native application did not have any negative affect on the user experience of the application as a few users could not distinguish the React Native app from the Android application. Furthermore, the difference which the others noticed was the animation between transitions which hopefully is fixed and supported in the near future. Lastly, almost everyone of the participants of the test said that they had no problems using a React Native application which is a very good verdict and the goal to have the same look and feel as a native application have been accomplished.

Lastly, the two applications were performance tested in regards to GPU frequency, CPU load, memory usage and power consumption. Three different cases were tested and all measurements concluded in the same result, a React Native application does not have as good performance as a native Android application. However, the differences in the tests were small and the React Native application was able to challenge the Android application in a notable way.

This thesis have evaluated how well an application created in React Native compares to a native Android application and there are some work which would be interesting to investigate in the future. First and foremost, since this thesis did not even regard iOS, a similar thesis which would compare React Native to iOS would be interesting. We have seen that the framework can create an application which can perform and feel like a native Android app but not how well it performs regarding the other operating system is yet unknown. Furthermore, it would be interesting to investigate how much of the code that can be shared between an iOS and an Android application created in React Native. Since the framework is able to

construct two applications with a lot of shared code, it is intriguing to know how much of the code that has to be platform-specific due to native components.



Bibliography

[1] *Android source website | ART and Dalvik*. https://source.android.com/devices/tech/dalvik/index.html. Accessed: 2016-04-19.

[2] Andreas Arnesson. "Codename one and PhoneGap, a performance comparison". MA thesis. Blekinge Institute of Technology, Department of Software Engineering, June 2015.

[3] Ugaitz Moreno Arocena. "Energy Consumption Studies for 3G Traffic Consolidation on Android using WiFi and Bluetooth". MA thesis. Linköping university, Department of Computer and Information Science, Jan. 2014.

[4] Alexander Bakker. *Comparing Energy Profilers for Android*. University of Twente. 2016.

[5] V. Balasubramanee et al. "Twitter bootstrap and AngularJS: Frontend frameworks to expedite science gateway development". In: *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. Sept. 2013, pp. 1–1. DOI: 10.1109/CLUSTER.2013.6702640.

[6] Scott Chacon. *Pro Git*. 1st. Berkely, CA, USA: Apress, 2009. ISBN: 9781430218333.

[7] A. Cockburn et al. "WebView: A Graphical Aid for Revisiting Web Pages". In: *OZCHI'99 Australian Conference on Human Computer Interaction* (Nov. 1999).

[8] Bonnie Eisenman. *Learning React Native*. O'Reilly Media, Inc., Dec. 2015. ISBN: 9781491929049.

[9] Bonnie Eisenman. *Writing Cross-Platform Apps with React Native*. http://www.infoq.com/articles/react-native-introduction. Accessed: 2016-04-25. Feb. 2016.

[10] Artemij Fedosejev. *React.js Essentials*. Packt Publishing Ltd., 2015. ISBN: 978-1-78355-162-0.

[11] Ben Frain. *Responsive Web Design with HTML5 and CSS3*. Packt Publishing Ltd, 1 jan. 2012.

[12] Cory Gackenheimer. *Introduction to React*. Apress, 2015. Chap. What Is React?, pp. 1–20. ISBN: 978-1-4842-1245-5. DOI: 10.1007/978-1-4842-1245-5_1.

[13] Georgios Gousios et al. "Lean GHTorrent: GitHub Data on Demand". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. ACM, 2014, pp. 384–387. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597126.

[14] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, Inc., 1987. ISBN: 0-13-821844-7.

[15] Erik Johansson and Tobias Andersson. *A closer look and comparison of cross-platform development environment for smartphones*. June 2014.

[16] Anmol Khandeparkar, Rashmi Gupta, and B. Sindhya. "An Introduction to Hybrid Platform Mobile Application Development". In: *International Journal of Computer Applications* 118.15 (2015).

[17] Benjamin LaGrone. *Html5 and Css3 Responsive Web Design Cookbook*. Packt Publishing Ltd, 23 may 2013.

[18] Arnaud Le Hors et al. *Document Object Model (DOM) Level 3 Core Specification*. World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407. Apr. 2004.

[19] Seung-Ho Lim. "Experimental Comparison of Hybrid and Native Applications for Mobile Systems". In: *International Journal of Multimedia and Ubiquitous Engineering* 10.3 (2015), pp. 1–12.

[20] James Long. *BRIDGING IN REACT NATIVE: An in-depth look into React Native's core*. http://jlongster.com/First-Impressions-using-React-Native. Accessed: 2016-04-26. Oct. 2015.

[21] L. Ma, L. Gu, and J. Wang. "Research and Development of Mobile Application for Android Platform". In: *International Journal of Multimedia and Ubiquitous Engineering* 9.4 (2014), pp. 187–198.

[22] Angel Torres Moreira, Mónica Aguilar-Igartua, and Silvia Puglisi. "Design and implementation of an Android application to anonymously analyse locations of the citizens in Barcelona". In: *CoRR* abs/1507.04585 (2015).

[23] Dotan Nahum. *Routing and Navigation in React Native*. http://blog.paracode.com/2016/01/05/routing-and-navigation-in-react-native/. Accessed: 2016-04-28. Jan. 2016.

[24] Johan Nordström and Thommie Jönsson. *Orderhantering via Android*. Kristianstad University, School of Health and Society. 2012.

[25] Tom Occhino. *React Native: Bringing modern web techniques to mobile*. https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/. Accessed: 2016-04-23.

[26] Tom Occhino. *React.js Conf 2015 Keynote - Introducing React Native*. Facebook Developers - YouTube. 2015. URL: https://www.youtube.com/watch?v=KVZ-P-ZI6W4.

[27] *React Documentation*. http://facebook.github.io/react/docs. Accessed: 2016-04-21.

[28] *React Native Documentation*. https://facebook.github.io/react-native/. Accessed: 2016-04-23.

[29] *Realm official website*. https://realm.io/. Accessed: 2016-05-06.

[30] Virpi Roto, Marianna Obrist, and Kaisa Väänänen-vainio-mattila. *User Experience Evaluation Methods in Academic and Industrial Contexts*.

[31] Rajinder Singh. "An Overview of Android Operating System and Its Security Features". In: *Journal of Engineering Research and Applications* 4.2 (2014), pp. 519–521.

[32] Benny Skogberg. *Android Application Development*. School of Technology, Malmö University. 2010.

[33] Andreas Sommer. *Comparison and evaluation of cross-platform frameworks for the development of mobile business applications*. Technische Universität München. 2012.

[34] Arnold P. O. S. Vermeeren et al. "User Experience Evaluation Methods: Current State and Development Needs". In: *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*. NordiCHI '10. ACM, 2010, pp. 521–530. ISBN: 978-1-60558-934-3. DOI: 10.1145/1868914.1868973. URL: http://doi.acm.org/10.1145/1868914.1868973.

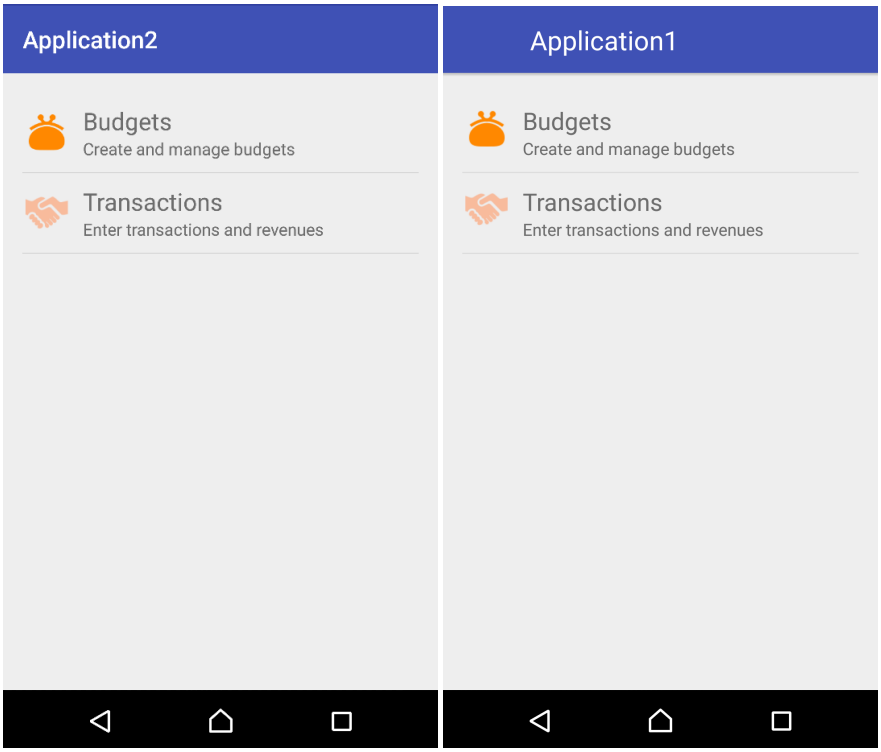
[35] Marko Vitas. "ART vs Dalvik-introducing the new Android runtime in KitKat". In: *URl: https://www. infinum. co/the-capsized-eight/articles/art-vs-dalvikintroducing-the-new-android-runtime-in-kit-kat (cons. 2-2015)* (2013).

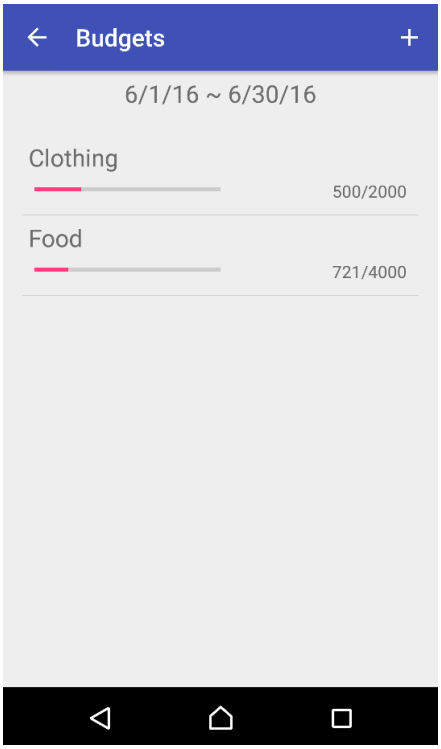
[36] Tom Williams. *RTC: Android Poised to Move from Phones and Tablets to Wider Embedded Applications*. http://www.rtcmagazine.com/articles/view/102484. Accessed: 2016-04-15.

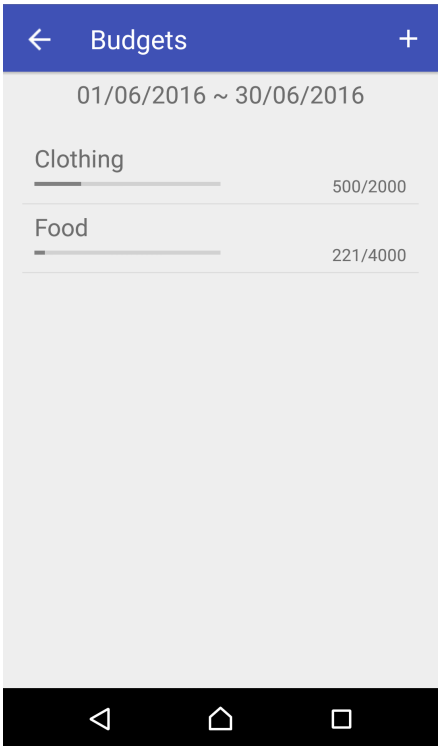
[37] Daniel Witte and Philipp von Weitershausen. *React Native for Android: How we built the first cross-platform React Native app*. https://code.facebook.com/posts/1189117404435352/react-native-for-android-how-we-built-the-first-cross-platform-react-native-app/. Accessed: 2016-04-24.

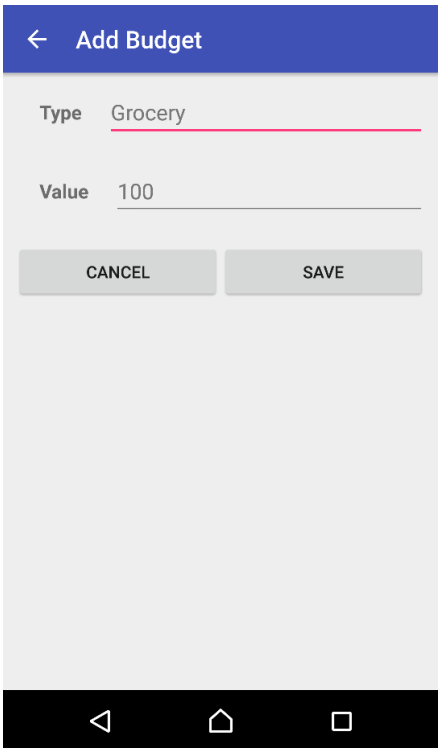
[38] L. Zhang et al. "Accurate online power estimation and automatic battery behavior based power model generation for smartphones". In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*. Oct. 2010, pp. 105–114.

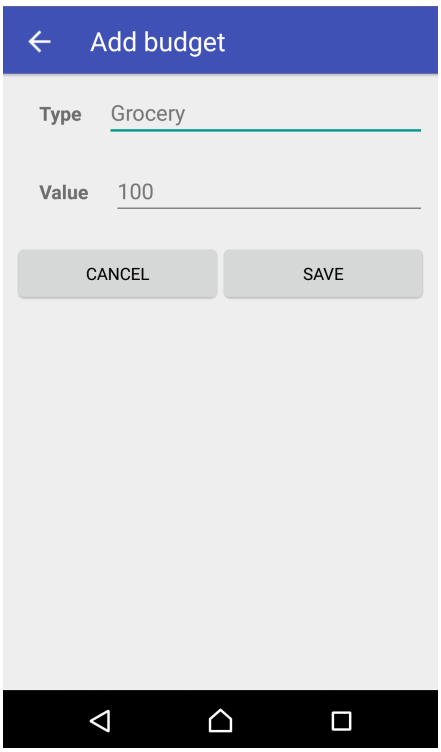
A Budget Watch screenshots

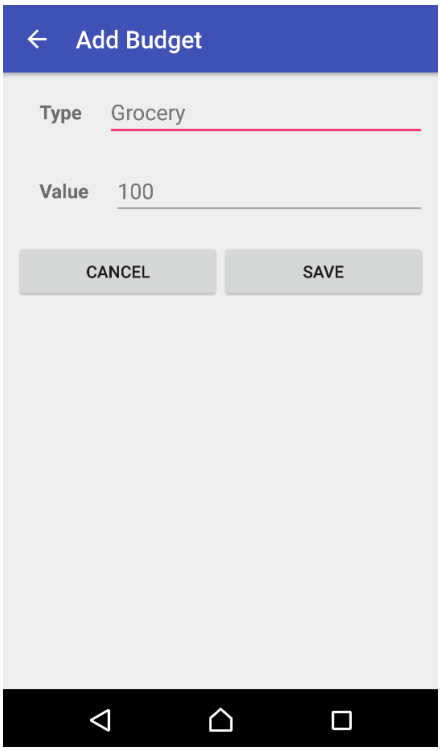


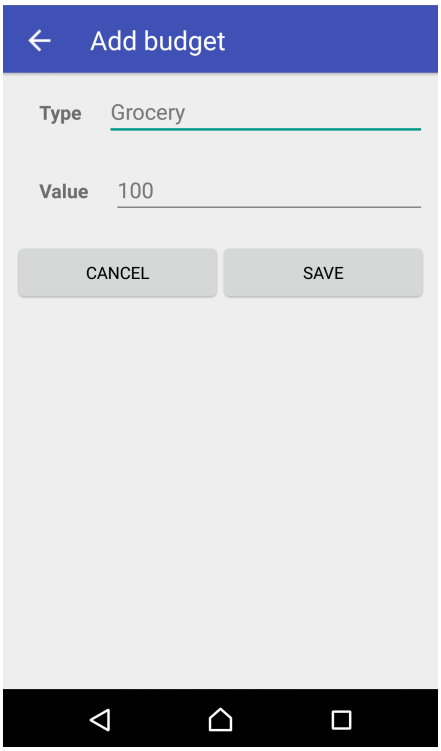

Budgets
6/1/16 ~ 6/30/16
Clothing
500/2000
Food
721/4000

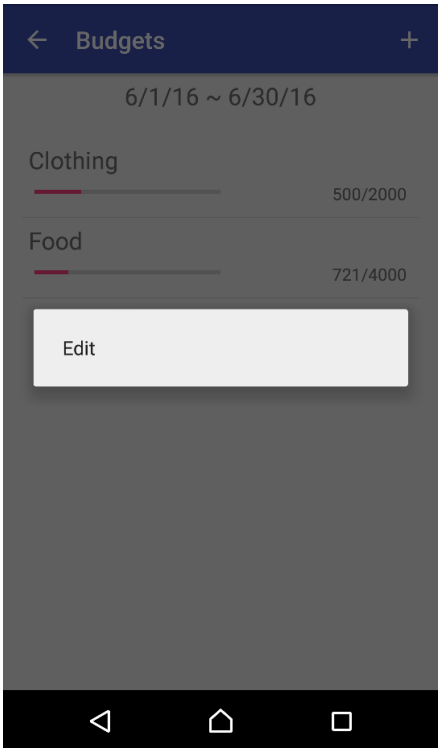

Budgets
01/06/2016 ~ 30/06/2016
Clothing
500/2000
Food
221/4000

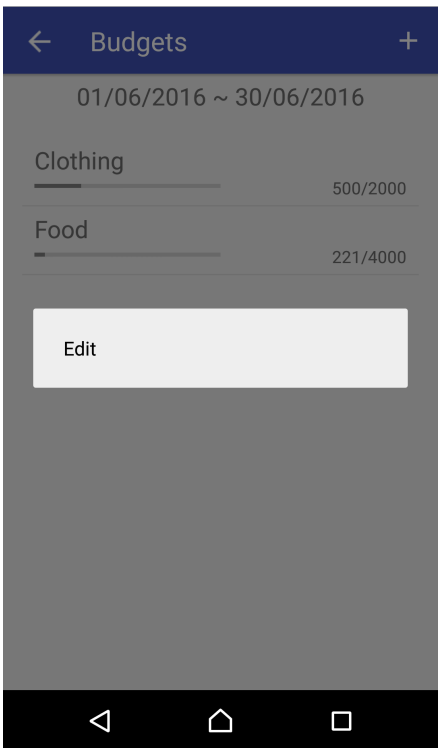

Add Budget
Type
Grocery
Value
100
CANCEL
SAVE

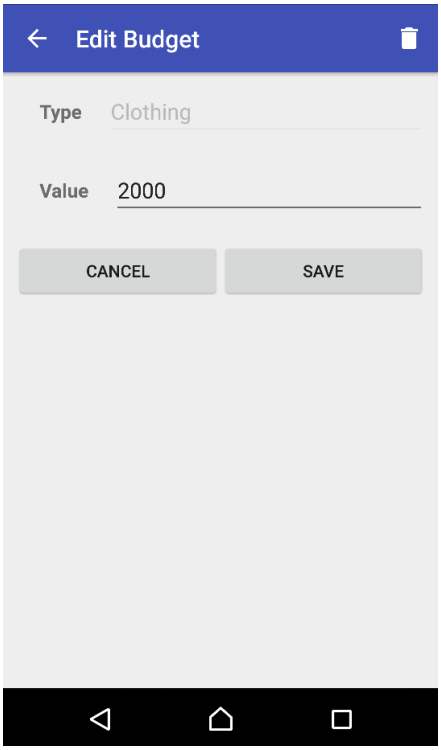

Add budget
Type
Grocery
Value
100
CANCEL
SAVE

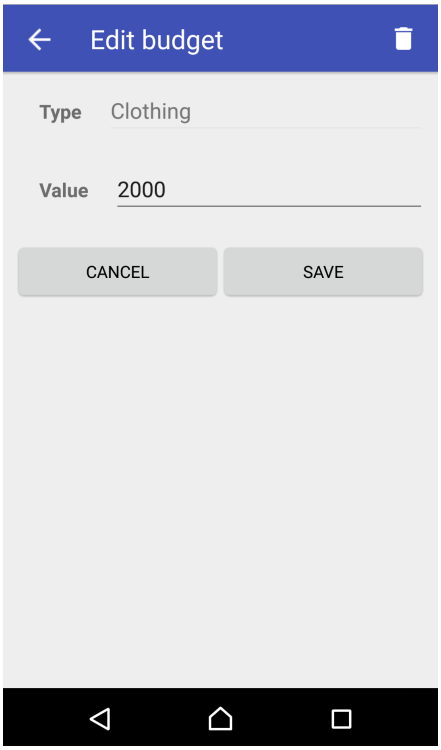

Add Budget
Type
Grocery
Value
100
CANCEL
SAVE

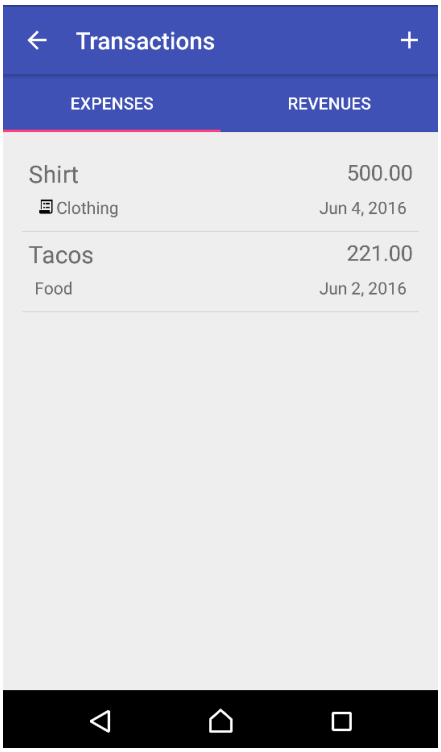

Add budget
Type
Grocery
Value
100
CANCEL
SAVE


Budgets
6/1/16 ~ 6/30/16
Clothing
500/2000
Food
721/4000
Edit

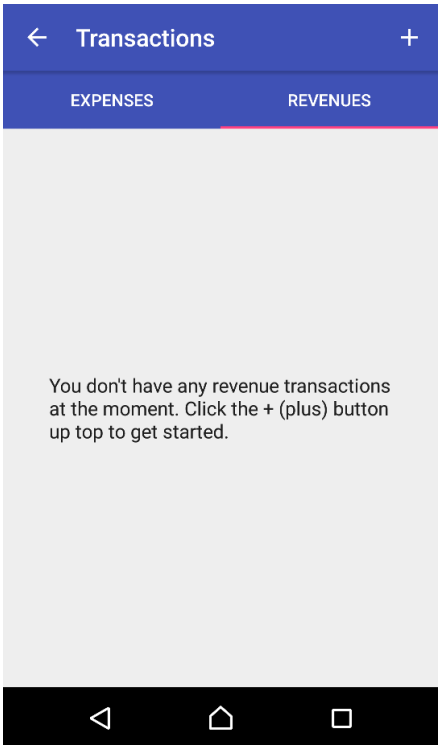

Budgets
01/06/2016 ~ 30/06/2016
Clothing
500/2000
Food
221/4000
Edit

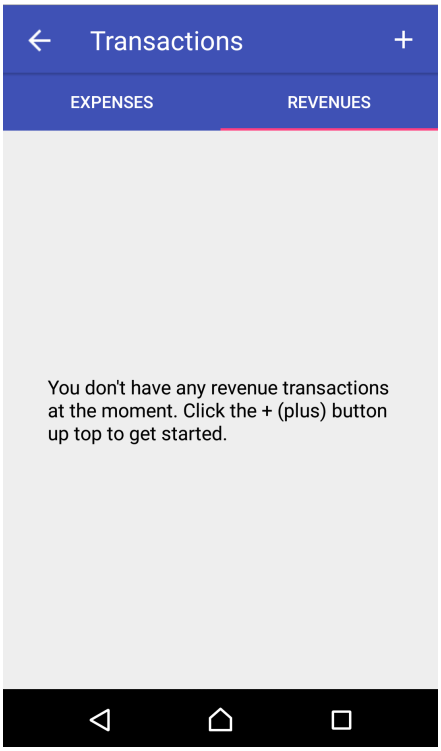

Edit Budget
Type
Clothing
Value
2000
CANCEL
SAVE

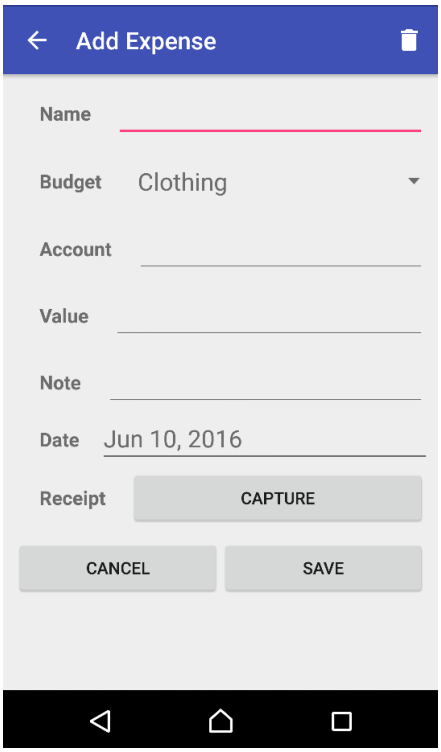

Edit budget
Type
Clothing
Value
2000
CANCEL
SAVE

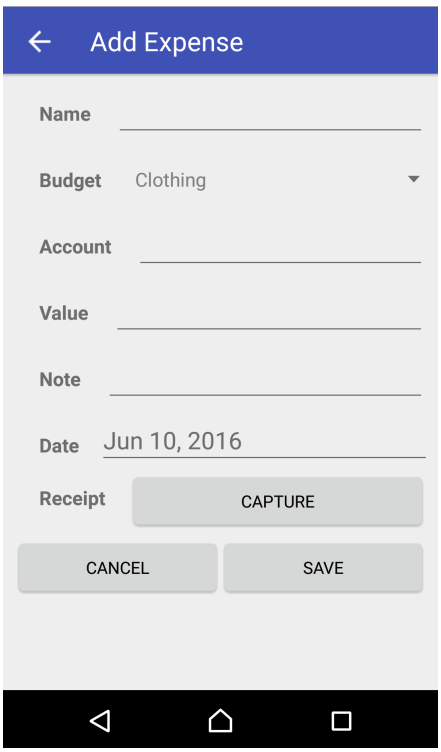

Transactions
EXPENSES
REVENUES
Shirt
500.00
Clothing
Jun 4, 2016
Tacos
221.00
Food
Jun 2, 2016

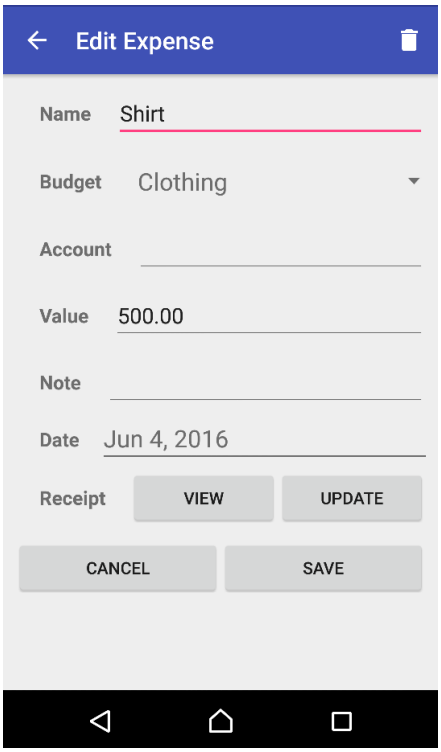

Transactions
EXPENSES
REVENUES
Shirt
500
Clothing
Jun 04, 2016
Tacos
221
Food
Jun 02, 2016

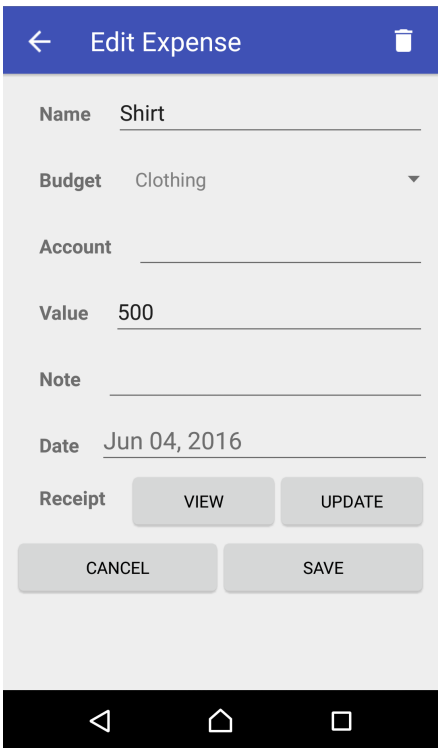

Transactions
EXPENSES
REVENUES
You don't have any revenue transactions at the moment. Click the + (plus) button up top to get started.


Transactions
EXPENSES
REVENUES
You don't have any revenue transactions at the moment. Click the + (plus) button up top to get started.


Add Expense
Name
Budget
Clothing
Account
Value
Note
Date
Jun 10, 2016
Receipt
CAPTURE
CANCEL
SAVE


Add Expense
Name
Budget
Clothing
Account
Value
Note
Date
Jun 10, 2016
Receipt
CAPTURE
CANCEL
SAVE


Edit Expense
Name
Shirt
Budget
Clothing
Account
Value
500.00
Note
Date
Jun 4, 2016
Receipt
VIEW
UPDATE
CANCEL
SAVE


Edit Expense
Name
Shirt
Budget
Clothing
Account
Value
500
Note
Date
Jun 04, 2016
Receipt
VIEW
UPDATE
CANCEL
SAVE