Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

Native versus non native development
A comparison of React Native and Angular NativeScript to native mobile applications

Timothy Lawler Karvonen
Native versus non native – A comparison of React Native and Angular NativeScript to native mobile applications

Timothy Lawler Karvonen

2017-06-25

MID SWEDEN UNIVERSITY
Avdelningen för informationssystem och -teknologi (IST)

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
Supervisor: Martin Kjellqvist, martin.kjellqvist@miun.se
Author: Timothy Lawler Karvonen, tika1401@student.miun.se
Degree programme: Computer Engineering, 180 credits
Main field of study: Thesis Project DT099G, 15 hp
Semester, year: VT, 2017
Abstract

The traditional or the native way to develop mobile applications is to use Java for Android and Objective-c or Swift for iOS. The native way is favored by many since the code and the functionality is optimized for the platform. Another way to develop mobile applications is to do it the non-native way, with a programming language or technique not made for the platform. This approach has for long been frowned upon due the limited hardware access and performance loss. React Native and NativeScript offers mobile application development in a non-native way said full access to the native platforms API using JavaScript all from a single code base. The aim of this thesis has been to develop and compare four proof of concept applications of which two are developed natively for Android and iOS and the other are developed using the non-native React Native and NativeScript. The comparison is based on three aspects: accessing the device’s native hardware and APIs based on what the company Dewire requires from mobile applications, the performance difference on the respective platform and code reusability cross platform. There is no big difference between React Native and NativeScript when comparing native access and everything that was accessible on the native implementation was accessible on the non-native implementation. Based on the performance measurements, React Native falls behind NativeScript. NativeScript handles long lists better than React Native. Lastly a discussion is presented regarding code reusability when developing non-native applications along with some experienced best practices when doing so.

Keywords: Mobile Application, React Native, NativeScript, Android, iOS
Foreword

I would like to thank Dewire for providing me with the necessary means for my thesis and a special thanks to Johan Deckmar, my thesis supervisor at Dewire and Melenie Lindh for guiding us throughout the period. I would also like to thank my supervisor Martin Kjellqvist at Mid Sweden University for all the help with the report and the shown interest to my work.
# Table of Contents

Abstract.................................................................................................................iii
Foreword.......................................................................................................................iv
Terminology..................................................................................................................viii

1 Introduction..............................................................................................................1
  1.1 Overall aim.................................................................1
  1.2 Scope.............................................................................2
  1.3 Concrete and verifiable goals........................................2
  1.4 Problem statement....................................................2
  1.5 Outline.........................................................................2

2 Theory.....................................................................................................................4
  2.1 Application Sandbox....................................................4
  2.2 Android Manifest.......................................................4
  2.3 Broadcast receiver.....................................................4
  2.4 iOS Info.plist..............................................................4
  2.5 iOS Delegate..............................................................5
  2.6 iOS Protocol...............................................................5
  2.7 TypeScript.................................................................5
  2.8 React.............................................................................5
  2.9 Redux............................................................................6
  2.9.1 Action.......................................................................7
  2.9.2 Reducer....................................................................7
  2.9.3 Store........................................................................8
  2.9.4 Pure function.........................................................8
  2.10 React Native.............................................................9
  2.11 Angular.......................................................................9
  2.11.1 Component..........................................................9
  2.11.2 Components life cycle........................................10
  2.11.3 Templates.............................................................10
  2.11.4 Services...............................................................10
  2.12 NativeScript...........................................................10
  2.13 JavaScript module..................................................11
  2.14 Native module.........................................................11
  2.14.1 React Native.........................................................12
  2.14.2 NativeScript........................................................12
  2.15 Native UI component............................................12
  2.15.1 React Native.........................................................12
  2.16 Hot reload vs live reload............................................13

3 Methodology........................................................................................................14
  3.1 Theoretical study.......................................................14
  3.2 Answering the problem statement..............................14
Native versus non native – A comparison of React Native and Angular
NativeScript to native mobile applications
Timothy Lawler Karvonen

3.3 Proof of concept applications.................................................................14
3.4 Tests........................................................................................................15
3.4.1 Native Access.....................................................................................15
3.4.2 Performance measurement.................................................................15
3.5 Long list data..........................................................................................15
3.6 CLI and Live reload.................................................................................16
3.7 Tools and requirements...........................................................................16

4 Implementation.........................................................................................18
4.1 Application overview............................................................................18
4.2 Native Android.......................................................................................19
4.2.1 Setting up the UI................................................................................19
4.2.2 Animation............................................................................................20
4.2.3 Navigation............................................................................................20
4.2.4 Broadcast receiver..............................................................................21
4.3 Native iOS...............................................................................................21
4.3.1 Setting up the UI................................................................................21
4.3.2 Navigation............................................................................................22
4.3.3 Delegate...............................................................................................22
4.4 React Native.............................................................................................22
4.4.1 React Native CLI................................................................................22
4.4.2 Setting up the UI................................................................................23
4.4.3 Animation............................................................................................23
4.4.4 Redux...................................................................................................23
4.4.5 Navigation............................................................................................26
4.4.6 Platform Specific................................................................................26
4.4.7 Native Modules...................................................................................27
4.4.8 Native UI Component..........................................................................30
4.5 NativeScript with Angular......................................................................32
4.5.1 NativeScript CLI................................................................................32
4.5.2 Setting up the UI................................................................................33
4.5.3 Animation............................................................................................33
4.5.4 Navigation............................................................................................33
4.5.5 Platform specific................................................................................34
4.5.6 Native modules....................................................................................34
4.5.7 Native UI Components.........................................................................35
4.6 UI Testing.................................................................................................36

5 Results.....................................................................................................37
5.1 Native Access..........................................................................................37
5.2 Performance.............................................................................................38
5.2.1 Android...............................................................................................38
5.2.2 iOS.......................................................................................................45

6 Conclusion................................................................................................52
6.1 Code reusability cross platform.............................................................53
6.2 When is non native application development an option?......................54
6.3 Development time...................................................................................55
6.4 Live Reload.............................................................................................55
Native versus non native – A comparison of React Native and Angular
NativeScript to native mobile applications
Timothy Lawler Karvonen
2017-06-25

6.5 Best practices..........................................................56
6.5.1 React Native...........................................................56
6.5.2 NativeScript..........................................................57
6.6 Ethical aspects............................................................57
6.7 Alternative approaches..............................................57
6.8 Future work...............................................................58

References...........................................................................59

Appendix A: Redux flow for the React Native implementation..........68
## Terminology

### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI</td>
<td>User Interface.</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>ES X</td>
<td>ECMAScript version X</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>JS</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen  
2017-06-25

1 Introduction

The traditional or the native way to develop mobile applications is to use Java for Android and Objective-c or Swift for iOS. The native way is favored by many since the code and the functionality is optimized for the platform. Non native applications has for long been frowned upon due to the lack of support for hardware, loss in performance and that non native applications drop in user experience. Although as this might be true, developing applications non-natively comes with its perks. Non native applications are written in one language for all platforms. Since there is only one language for multiple platforms, non native development tends to be faster compared to native development. Unlike native development where most companies have separate development teams for different platforms non native applications usually only require one development team for all platforms. The company Dewire develops their applications exclusively in native language for big budget projects since their applications require the hardware support and the optimization that native development brings. As of that, Dewire has never seen non native applications and migrating away from the hardware support as an option in big-budget projects. Non native development has come a long way since Facebook announced React native which populates their JavaScript library React with native programming to give better hardware support. Google recently started to support NativeScript with their framework Angular to - as React Native - provide native programming and provide better hardware support. With the uprising of native programming in a non native fashion, this thesis compares React native and Angular NativeScript to native applications and explores the opportunities for if and when non native applications are suitable to replace native applications.

1.1 Overall aim

The overall purpose is to compare native applications to React native and Angular NativeScript applications in terms of how native-close both technologies are compared to full native. Primarily what will be investigated is how well they both support accessing hardware and the difference in performance when dealing with a big dataset in the form of a dynamic list and how well that list handles being animated. Another topic that will be analyzed is how well both frameworks operate when reusing code across platforms and which factors affect code reusability. Further, there are gaps in the documentation related to best practices when developing cross platform applications and best practices to follow, to ensure well performing clean applications. So this thesis also aims to investigate in best practices when developing cross platform mobile applications using React Native and Angular NativeScript.
1.2 Scope
The scope of this thesis is limited to certain features which the company Dewire sees as must have necessities in native application development. These features includes access to various hardware API’s and having an application which keeps up in performance even when faced with displaying a huge dataset in the form of a graphical list with the ability to often update delete or add new data to the dataset. These features are described in more detail in chapter 1.5. The applications developed will aim to be functional rather than visually appealing but the design principles of the respective platform will be included. The tests for the applications will be conducted separately for each target platform and will cover how the performance differs from non native to native.

1.3 Concrete and verifiable goals
- Develop one application with each technique(Android native, iOS native, React Native and Angular with NativeScript) which utilizes the platforms native API to access GPS, contacts, ongoing phone calls, carrier information, wifi information, access camera and read incoming text messages.
- Include one page in each application which implements a list containing lots of data.
- Perform UI tests on the list while measuring the device’s performance based on CPU load and memory allocation.

1.4 Problem statement
To determine if developing applications in React native and Angular NativeScript is a suitable replacement for developing applications in native language this thesis aims to answer the following questions:
1. Under what circumstances is code reusability cross platform possible?
2. When is non native application development an option?

To help answer and narrow the first question the following sub-questions are stated as following:
- Under what circumstances can graphical components be reused between Android and iOS?
- Under what circumstances can logic based components which interact with the user interface be reused between Android and iOS?
- How does the design principles between Android and iOS affect reusing code cross platform?

1.5 Outline
The project report follows the following outline: Chapter 2 – Theory, describes some technical parts in a theoretical way to give the reader a better understanding when reading the report. Chapter 3 – Methodology, describes the suggested approach for answering the problem statements, building the system and testing
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

the system. Chapter 4 – Implementation, describes how the system is built to help answer the problem statements. Chapter 5 – Result, presents the result gathered from the implementation and the theoretical study along with the performance measurements. Chapter 6 – Discussion presents some analysis of the result, answering the problem statements, some reflections gathered throughout the implementation and in words, puts the system to test on some real scenarios based on the result.
2 Theory

2.1 Application Sandbox

The application sandbox is a design principle to limit and protect the system. By limiting the access to certain features in the system for software the security is increased against a potential harmful software. [1] In iOS, an application which wishes to access features outside the applications sandbox must express the reason for accessing the feature in order to use the resource. Features outside the iOS sandbox are for example hardware such as camera and microphone. Basically the application sandbox states which features the application may use.[2]

2.2 Android Manifest

In Android, an application only has access to certain features by default. An application may be granted access to use features or access data beyond the default access by requesting permission to use that feature. [3] Features which uses less sensitive and less intrusive may not require any special permission for access. For access to sensitive system data or system features a permission must be specified and some may require user interaction before being granted access. [4] A permission in Android is requested by specifying the feature in the applications manifest file. [3]

2.3 Broadcast receiver

Android applications uses broadcast messages [7] to communicate with other applications or the actual Android system. By default, the Android system has some broadcast messages which are sent when certain events occur, for example when plugging in a charger or a text message is received. Applications are able to send custom events to notify the system or any other interested application that something occurred, for example data has been finished downloading. Applications may ‘listen’ for specific broadcasts by registering a broadcast receiver to listen for that specific broadcast. Once the broadcast has been registered, the Android system will automatically send the broadcasts to the active broadcast receivers.

2.4 iOS Info.plist

The iOS information property list, or info.plist is a key value file which contains necessary configuration and information about the application required to run. The Info.plist may contain features outside the default application sandbox which are required to run the application. [6]
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen

2.5 iOS Delegate

A delegate is used when delegating a certain task from one object to another object. The delegate is used to receive messages or events from the other object when a task has been completed or an event is dispatched. The delegate may act on behalf of the other object as illustrated in Illustration 1. Here the user has interacted with the window and wants to close it. The delegate is set up to respond to the ‘windowShouldClose’ event which responds with a ‘No’ and therefore controls the window’s behavior.[8]

2.6 iOS Protocol

A protocol in iOS is a set of methods or properties for which a class must or may implement. By making a class conform to a protocol, it is safe to assume that the class does in fact implement a method required by another class.[9] An iOS protocol is similar to a Java interface.

2.7 TypeScript

TypeScript [10] is an optionally typed programming language compatible with any JavaScript engine which supports at least ES3. TypeScript is compiled to and fully operational with JavaScript. Meaning TypeScript may communicate with JavaScript and JavaScript may communicate with TypeScript.

2.8 React

React [11] is a declarative component based library for JavaScript made for building user interfaces. Components in React hold their own state and data and are small building blocks for a potential complex user interface written entirely in JavaScript. The actual view of a component is created in the render method of the component and is written in either JavaScript or in JSX.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

JSX uses a XML-like syntax shown in Illustration 2. A component called ‘HelloMessage’ is created which only include the render method. Inside the render method a div-element is returned containing the text ‘Hello <passed property>’. As seen in the bottom of Illustration 2, a HelloMessage element is created passing along “Jane” as property. The component will return the text “Hello Jane” inside a div element as output. [11]

A stateful react component is a component which maintain its own state. As seen in Illustration 2, the component accessed the passed data from ‘this.props’. A stateful React component is a component which holds the required data in the components state object, accessible from ‘this.state’. In order for the component shown in Illustration 2 to update, the ‘owner’ of the component must update the data passed to the component. In stateful components, whenever the state’s data is updated, the component is updated. [11]

2.9 Redux
Redux[12] is a helper for managing a JavaScript application’s state and data to create applications which has a consistent behavior. The idea behind Redux is that the whole applications state and data is supposed to be stored in a single container called the store. The only way to modify the data in the store should be by emitting an action. The action should only be able to transform the data in the store by using pure JavaScript functions called reducers.
2.9.1 Action

As seen in Illustration 3, the action[13] is the first step for a component to pass data to the store. Actions are defined as plain JavaScript objects which must at least contain a type property, commonly defined as string constants. The action may contain more payload depending on the type of action to perform against the store. For example: if an action is to be defined to add a todo item to a list the action may look as illustrated in Illustration 4. An action is emitted to the store using the ‘store.dispatch()’ function.

```javascript
const ADD_TODO = 'ADD_TODO'

{
    type: ADD_TODO,
    text: 'Build my first Redux app'
}

Illustration 4: Example action to add a todo [13]
```

2.9.2 Reducer

“Actions describe the fact that something happened, but don't specify how the application's state changes in response. This is the job of reducers.” [14]

The reducer[14] contains the actual logic of updating a state in the store. The reducer expects to receive the current state and an action of what to perform on the state and then returns the new updated state. Redux has one important rule for the reducer: The reducer must be pure. Meaning that the reducer may not:
perform any mutation on the state, do anything other than what it is supposed to and the reducer may not call external impure functions.

Illustration 5: Example reducer for adding todo item

Illustration 5 Shows an example of a pure function in the reducer for adding a todo. The reducer as mentioned takes two arguments: the current state and the action. The illustrated example is a pure function since the rules for the reducer are followed.

2.9.3 Store

The Redux store[15] is what brings the action and reducer together. The store is responsible for:

- holding the applications entire state
- Allow components to read from the state
- Allow for the state to be updated through dispatching actions
- Register and unregister listeners

An application may only contain one store, but may contain several actions and reducers.

2.9.4 Pure function

Pure functions in sense of Redux are functions which does not directly modify the current state. The function must return a new instance of the state which represents the next state without using mutating functions. [16] The reducers has to be pure functions since the implementation of Redux checks if the state has been updated by comparing if the previous state is the same object as the new state. Without the shallow comparison comparison, Redux would have to do a so called deep compare of the state, which would be much more expensive than comparing the identity of the states. [17]
2.10 React Native

React Native[18], based on Facebook’s JavaScript library React, is a framework used for building mobile applications for Android and iOS with native rendering. Unlike React, React Native is targeted against mobile devices instead of browsers. Since React Native is based on React, web developers who are used to React are able to develop mobile applications with their knowledge. React Native uses the same XML like syntax, JSX as React does to render a view but uses a native bridge to render the view as actual native components. React Native also allows access to the native platforms API to access hardware such as the camera or the GPS.

In React Native, the UI is run on one thread and logic is executed on a separated thread called the JavaScript thread. The UI thread is responsible for rendering the active layout and respond to user interaction where as the JavaScript is responsible for pushing new layouts and process user interaction and all the logic which follows. [19]

2.11 Angular

Angular[20] is a client application development framework which uses HTML and one of TypeScript and JavaScript. The client is built using templates along with components which control the template and services to handle heavier application logic.

2.11.1 Component

An Angular component[21] is in broad terms the controller class for a view on the screen. The component controls the behavior of the view by responding to user interaction or life cycle hooks.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen 2017-06-25

2.11.2 Components life cycle

The Angular framework automatically handles creating, destroying and updating components in the application. These events are called the component life cycle which the Angular framework exposes hooks for. A component may implement the hook to respond to the event. Each life cycle hook is prefixed with the ‘ng’ keyword like the ‘ngOnInit’ hook, seen in Illustration 6, which is emitted once the component has been created by Angular. [22]

Illustration 6: Angular component’s lifecycle hooks [22]

2.11.3 Templates

A components visible view is defined using templates[23] which instructs Angular how the component is to be rendered to the user. A template is written in a HTML-like syntax where typical HTML elements like h2 and ul are mixed with Angular’s template syntax.

2.11.4 Services

“Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well..” [24]

Services[24] in Angular are objects which performs a specific task, could be heavier logic such as network calls or simpler tasks like logging a value. Angular has no specific definition of what a service exactly is but describes a service as something which performs a task when needed which may be reused at different times. Services exists in Angular since the design principles of Angular states that components should stay clear of fetching data and such heavy operations. Components should use services for such operations.

2.12 NativeScript

NativeScript[25] is used to build cross platform Android and iOS applications with native rendering like React Native. NativeScript may be used together with Angular and TypeScript to build the UI and NativeScript promises 100% access
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen 2017-06-25

to the native platforms API directly through JavaScript. NativeScript together with Angular offers reusing any pre existing Angular web code in the mobile application to enable faster development. [26] NativeScript offer live reload, meaning the application will automatically refresh when changes is detected without the need to recompile the whole application. [27]

2.13 JavaScript module

A JavaScript module is a piece of code, separated from where it is supposed to be used. [28] ECMAScript 6(ES6) provides a JavaScript syntax for exporting modules. According to ES6, a module may export in two ways: named export and default export. When exporting a module the first way, by named export, the module may contain several exports and is accomplished by the keyword exports. These exports are called named exports since they are identified by the name of the exported object when importing. Importing a named export is done by either importing the specific exports name or importing all exports at once by the ‘*’ wildcard.[29] Illustration 7 shows an example of a module with two named exports and how to specify to import those.

```javascript
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from 'lib';

Illustration 7: Example of named export[30]
```

The default exported module[31] is commonly used when dealing with JavaScript classes where one class is one module, as a JavaScript module is only allowed to contain one default export. Illustration 8 Shows an example of a module having a class default exported and how to import the module.

```javascript
//------ MyClass.js ------
export default class {} // no semicolon!

//------ main2.js ------
import MyClass from 'MyClass';

Illustration 8: Example of a default export module [31]
```

2.14 Native module

Native Modules in React Native and NativeScript is a JavaScript module which uses the frameworks method of accessing the native platform’s API.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen

2.14.1 React Native

In React Native, native modules are written in the native language for the platform. Java for Android and Objective-c or Swift for iOS. [32][33] React Native has defined a base class for native modules: Android extends ReactContextBaseJava[34] and iOS implements RCTBridgeModule[35].

For Android, the native module which extends the base class must override the method ‘getName’ to set a name for the class to import as a JavaScript module. If the module is to expose any method within the imported JavaScript class, the method must be annotated with ‘@ReactMethod’ which makes the method callable from JavaScript. For the module to be exposed to JavaScript as a JavaScript module, the module must be registered as a native module using the method ‘createNativeModule’ from the base class ‘ReactPackage’. The last step of creating a native Android module is to provide the registered native module to the application. The ‘MainApplication.java’ should include a method called ‘getPackages’ when the project was created in which the module is provided to the application.[32]

An iOS native module as mentioned must implement the ‘RCTBridgeModule’ protocol in an Objective-c class. The class must include the ‘RCT_EXPORT_MODULE’ which may take one argument, the name of which the imported JavaScript module is to have. For exposing a method to be called from within JavaScript, the method must be included in the ‘RCT_EXPORT_METHOD’ macro. The macro takes a few arguments of which the first argument must be the name of the method.[33]

2.14.2 NativeScript

In NativeScript, native modules are created in JavaScript since the native API’s are entirely exposed to JavaScript. This is enabled due to the NativeScript runtime converts JavaScript types to native types and native types to JavaScript when necessary. [36][37] A native module in NativeScript is written like a JavaScript module with either export type.

2.15 Native UI component

A native UI component is a component, written in the native language for the platform, exposed and used in JavaScript. [38]

2.15.1 React Native

To create a native UI component for Android the ‘ViewManager’ super class is used. ‘ViewManager’ enforces two methods: ‘getName’ and ‘createViewInstance’. ‘getName’ as for native module is to give the module a name when importing to JavaScript. ‘createViewInstance’ is what actually creates and returns the view. This method may return any of the subclasses to ‘ViewGroup’. Like registering native modules, the native UI component must be registered to be exposed to JavaScript. The component is registered using the method ‘create-
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

ViewManagers’ from the ‘ReactPackage’ base class.[38] The package is then provided to the application the same way as with native modules.

Native iOS UI components are created by, in an Objective-c file inheriting the ‘RCTViewManager’ base class and include the ‘RCT_EXPORT_MODULE’ macro which takes an optional argument with the modules name. Unlike creating native modules, instead of creating a method to expose, an UIView holding the actual native view is created. [39]

2.16 Hot reload vs live reload

There is a fine line between the difference of hot reload and live reload. Hot reload means that only the affected changes are reloaded without affecting the applications state. Live reloading reloads the application when a change is detected, including the applications state. [40]
3 **Methodology**

Chapter 3 Methodology presents the suggested method with will be used to fulfill the goals presented in chapter 1.5 and the method which will be used to answer the problem statements presented in chapter 1.6.

3.1 **Theoretical study**

To get a better understanding of how native Android, IOS, React Native and NativeScript works, a theoretical study will be made throughout the project. The study, consisting of mainly the respective framework’s documentation will enable implementing one proof of concept applications per respective framework. The theoretical study will also contribute towards fulfilling the project goals presented in chapter 1.5 and to assist answering the problem statements presented in chapter 1.6.

Some sources may seem a bit odd in the report, for example GitHub issues, and Stack Overflow questions and since React Native and especially NativeScript has a small community these sources are needed. The alternative sources has and will be analyzed to make sure that the information is of relevance and that the information is of quality.

3.2 **Answering the problem statement**

The theoretical study presented in chapter 3.1 in combination with the proof of concept applications will be the foundation of how the problem statements presented in chapter 1.6 will be answered. An UI test, presented in chapter 3.4.2 will be conducted to evaluate how well the React Native and NativeScript implementation performs versus the native implementation for the respective platform. The result of the UI tests will be used to answer the latter problem statement presented.

3.3 **Proof of concept applications**

The focus of this thesis is to assess how native close React Native and NativeScript can become. Therefore Four proof of concept applications: one native Android, one native iOS, one React Native and one NativeScript application must be developed which will fulfill the goals presented in chapter 1.5. These proof of concept applications will provide: wifi access, phone call state, carrier information, incoming text messages(Android only as iOS does not allow this), access to the contact book and provide access to the native platforms hardware API's: GPS and camera. The applications will also implement an animated graphical list populated with a huge dataset with the ability to add, edit, and remove entries. The size of the dataset for the list will be determined by trying incrementing lengths until a fast paced scrolling yields a noticeable drop in fram-
erate. For the respective platform, the proof of concept applications will be as identical as possible to the appearance and to the functionality and will follow the respective platforms design principles.

3.4 **Tests**

To answer the latter problem statement regarding when non native application development is an option a series of test cases will be set up. These tests covers if and which native APIs is accessible through React Native and NativeScript and how well the non native implementation performs versus a native implementation. The test results will only be compared within each platform to give a fair assessment since hardware access and performance may differ device to device and platform to platform.

3.4.1 **Native Access**

One of the primary reasons for building the proof of concept applications is to test whether the native APIs are accessible and how easily one can access them. The company Dewire has come up with which native access that is to be tested for from what they require or may require from mobile applications.

Testing based on native access will be done by attempting to implement the native accessible features that the company Dewire has requested. The test will consist of if the feature is accessible and implementable and how the feature is implemented.

3.4.2 **Performance measurement**

A well performing UI is essential for an applications user experience and thus the performance tests will cover navigating the UI and rendering a good deal of views in a long list which features user interaction and animations. The tests will be performed manually and is split into two series of testing: one consisting of scrolling, editing, adding and removing from the long list and one consisting of navigating to and from layouts.

The list will be populated with a big enough dataset such that the native implementation yields a noticeable lower render rate when scrolling the list faster than expected from a real user. The list data will be the same for each platform and the size will be determined by increasing the length of the list until the native Android implementation gives this expected behavior.

3.5 **Long list data**

The list data will be in JSON array format and is generated through an online JSON generator[41]. Each entry will contain the following properties: ID: String, index: Integer, first name: String and last name: String. Once the required number of entries in this array has been achieved, the same list will be used on the following applications.
3.6 **CLI and Live reload**

Both React Native and NativeScript offer a command line interface (CLI) to speed up the production [42] [43]. Both the React Native and NativeScript CLI is built on NodeJS which means that node has to be installed to use the CLI’s. The React Native CLI is installed globally using NPM with the command ‘npm install -g react-native-cli’ [42] and the NativeScript CLI is also installed globally using NPM with the command ‘npm install -g nativescript’ [43]. The CLI’s will be used to create a starting project for both React Native and NativeScript.

Both React Native and NativeScript claims that their CLI includes deploying the application to a device with live reload. The CLI deploy would mean that the application could be deployed directly from the terminal without using the platform’s IDE. The live reload would mean that the application is updated as soon as a change has been made, saving time by eliminating a new manual deploy. The live reload will be used if possible throughout the React Native and NativeScript implementation.

3.7 **Tools and requirements**

The following tools and softwares are used in order to reach the result:

Native Android:

- Android SDK
- Device running Android version 6.0.1

Native iOS

- Computer running MacOS
- XCode IDE
- Device running iOS version 10.2.1

React Native:

- Everything noted under native Android and native iOS
- React Native CLI
- NodeJS
- React Native version 0.43
- React version 16.0
NativeScript

- Everything noted under native Android and native iOS
- NativeScript CLI
- NodeJS
- NativeScript-angular version 2.5
- Angular version 4.0

Testing

- Android studio’s android monitor
- Xcode’s instruments software
4 Implementation

Chapter 4, implementation is written in such way that the reader with basic knowledge of mobile application development, React, Angular and JavaScript could recreate the same result. The implementation is developed and is only guaranteeing the result on a macOS with a One Plus One running Android 6.0.2 and an iPhone 6S running iOS 10.2.1. The covered development environments are: Android Studio version 2.3 and XCode version 8.2.1.

4.1 Application overview

All of the proof of concept applications should follow the same layout throughout the implementation so the base layout seen in Illustration 9 was planned. The first page when initially opening the application is start which will hold the information which does not require user interaction in order to be displayed. GPS data, wifi information, on going calls, carrier information and incoming text messages are to be displayed here. Start also serves as the primary page for navigation and will provide the opportunity to navigate to every other page in the application. Long list will include the animated graphical list and provides the ability to add new entries to the list, remove entries from the list and edit the data present in a row. Contacts will provide a method for accessing and displaying the contacts currently on the device. Camera record and camera photo will
both provide access to the device’s camera to either record video or capture an image and save to the device’s gallery.

### 4.2 Native Android

The Android application will be using features outside the applications sandbox and thus a few permissions must be specified in the applications manifest file. The manifest file is set up to request permission to use: fine location in order to access GPS, access to network state to be able to access the current state of the wifi, read phone state to listen for ongoing calls, receive sms and read sms to listen for incoming text messages and read the content of the message and finally: camera, record video and write to external storage to be able to use the camera, record video and store the capture to the device’s external storage space.

#### 4.2.1 Setting up the UI

In Android a UI is defined by a view group which is basically a container to hold views[44]. This view group is defined either through code or by XML files, the latter of which this implementation is made using.

As seen in Illustration 10, to build the layout for the long list page first a constraint layout, which inherits view group[45] is placed to fill the width and height available. A list view is placed within the constraint layout to hold the row views populated with the long list data. By adding the line “android:id="@+id/longListList” to the list view element, a new resource will be created in the ‘R.java’ class and enables referencing the list view in the application[46]. For the functionality of adding new entries to the list a button should be placed above the list view seen from the z-axis in the bottom right
corner. So a relative layout, which inherits view group[47] is placed within the constraint layout, below the list view. The gravity of the relative layout is set to end, meaning that views placed within is aligned to the end or to the right of the layout. A button is placed inside the relative layout with ‘alignParentBottom’ set to true. This will make it so that the layout places the button aligned to the right, and the align parent bottom makes it so that it is also aligned to the bottom of the parent layout.

4.2.2 Animation

To create an animated effect for when deleting a row in the long list, the ‘ObjectAnimator’ - which animates an object’s properties [48]- is used. The delete event is handled in the list’s adapter when by assigning an onClick event listener to the delete button when setting up the view for the row. Two ObjectAni-mators are set up: one to alter the row’s x position from the current position to the width of the device’s screen width, and one to alter the row’s alpha value from the current alpha value to zero. Once the delete button is pressed, both animations are started which gives a fading slide off animation. Since the rows in the list are reused, the animation has to be reset once the animation is finished. This is achieved by setting up animation listeners on the animations to listen for the ‘onAnimationEnd’ function and by calling animation.reverse(). [49]

4.2.3 Navigation

The navigation is set up creating a new activity for each new page presented in Illustration 9 and using intents to launch the new activity, removing the present activity when pushing the new activity. [50]

Illustration 11: Launching a new activity using intent

Four buttons are added the the start activity to provide a way for navigating to the rest of the activities which all have onClick listeners set up to launch a new intent once pressed. Illustration 11 Shows navigating to the contacts page by starting a new activity with an intent created of the contacts activity.
4.2.4 Broadcast receiver

To be able to listen for incoming text messages a broadcast receiver is used. A class which extends ‘BroadcastReceiver’ is created which overrides the function ‘onReceive’. OnReceive provides the text message received in the provided argument of type ‘Intent’. The class is used on the ‘start’ page by calling ‘registerListener’ passing along the class created above as first argument and the second argument – Which filter the BroadcastReceiver is to receive events from – is specified to receive events from ‘SMS_RECEIVED_ACTION’ to receive events about incoming text messages. [51]

4.3 Native iOS

4.3.1 Setting up the UI

The long list on the iOS application is set up by first creating a custom class for a cell template which is to be applied on each row in the list. As the list is to be of the type UITableView, the class must extend UITableViewCell. The cell contains two text labels and one button which are stored as references in the class. A function to handle pressing the edit button is created which sets the text of the labels to “New first name” and “New last name”.

The layout in iOS is set up using the projects storyboard file. A new scene is created with a table view containing a prototype of a cell with two labels and one button for editing the data. The list cell is set to be of the custom class created above.

In the controller class - which extends UITableViewController - for the UITableView set up above needs to override the following functions from the UITableViewController class:

- numberOfSections – How many sections the list should contain – set to return 1.
- numberOfRowsInSection – How many rows each sections is to contain – set to the size of the long list data set.
- cellForRowAt – Which cell to show on a specific row – Set to return an instance of a reused cell at the index with ‘tableView-dequeueReusable-
Native versus non-native – A comparison of React Native and NativeScript to
native application development
Timothy Lawler Karvonen
2017-06-25

Cell[54] and set the data of the cell’s label to the data of the long list
data at the current index.

- canEditRowAt – If the current row should be editable – set to always re-
turn true.

- editingStyle – How the table view should behave at specific events – Set
to respond to delete events which should delete the item from the long
list data set and delete the row from the table view with a fade anima-
tion.

4.3.2 Navigation

Since the full layout is built using the storyboard, the navigation is handled by
segues[55]. As the ‘start’ page is to serve all navigation, segues are created to
each scene in the storyboard with an unique identifier. As seen in
Illustration 13, to navigate from the ‘start’ page to the long list page, the function “perform-
Segue” is used. The identifier of the segue is specified as argument.

4.3.3 Delegate

To be able to listen for location updates, call state changes, selected contacts
and processed captured photos, delegate is used. To assign a delegate to listen
for location updates, the ‘start’ page is set to conform to the ‘CLLocationMan-
gerDelegate’ protocol[56] which may receive delegated events from a location
manager object. The location manager is of the type CLLocationManager and
since using location service is beyond the applications sandbox before using the
location manager, permission has to be granted by the user[57]. The location
manager is configured to have an accuracy of 100 meters with an update every
10 meters and to have its delegate set to the parent class. The CLLocationMan-
gerDelegate provides the class with the function ‘didUpdateLocation’ which
gives the longitude and latitude coordinates of the device.

4.4 React Native

4.4.1 React Native CLI

The React Native CLI is used to provide a starting point for using React Native
to build mobile applications and the project is created with the command ‘react-
native init <project name>’. To deploy and run the application with live reload
on a device the command ‘react-native run-android’ for running on an Android
device and ‘react-native run-ios’ for running on an iOS device is used.[58]
4.4.2 Setting up the UI

Building the user interface in React Native is done similarly to how the UI is built for React web applications. The render method for the components is used to return a XML-like JSX view hierarchy. Unlike React for the web where div and p elements are used, React Native introduces new predefined components such as view and text.

Illustration 14 shows the render function for the long list implementation. The long list consists of a flat list which renders the list items asynchronously and renders the list as a list view for Android and a UI table for iOS.

4.4.3 Animation

The ‘Animated’ library from the react native core is used to animate properties on a view. By default, the animated library returns an animatable view component[59] which is used as the root view for each row in the list. The animation should transform the x coordinate from the current position to the width of the screen for a 300ms duration creating a slide off effect. Since the React Native flat list is interpreted as a list view and a UITableView which both uses recycling views, the view has to be reset to its original state once finished. The start function for the animation accepts a callback function which is executed once the animation is finished. Inside the callback, the manipulated values are reset to the original state and the list row is deleted from the data set.

4.4.4 Redux

The implementation uses Redux to favor managing the applications current data state. Redux makes it so that instead of having the components holding their own data state and struggle with communicating cross components, the data state is held in a single global accessible store which components read from. Appendix A shows how the applications data flow is set up with Redux and which reducers along with actions are set up.
The long list should include the feature to add new entries to the list, remove entries from the list and edit the data in a row. Redux calls a reducer by emitting actions. These actions could hold data depending on the action and is passed to the reducer via the action parameter. This implementation separates the actions to a application global accessible location so that the actions are always described the same way and could be reused in the exact same way in a different part of the application. Illustration 15 Shows how the actions for the long list is specified. The long list has three actions available: addItem, removeItem and editItem which satisfies the required conditions for the long list.
The reducer in Redux handles updating the applications state. As seen in Appendix A, the long list reducer responds to three actions: addItem, removeItem and editItem and Illustration 16 shows the implementation of the long list reducer. Since the reducer must implement pure functions for updating the state, add item responder uses JavaScript’s array.concat method to return a new array consisting of the previous state concatenated with a new array containing only one object[60]. Remove item uses the array.slice function[61] to return part of the array up until the passed index concatenated with the rest of the array not including the passed index. Edit item makes use of ES6’s spread operation[62] to return a new array by spreading or copying the value returned from slicing the current state from the beginning up until the specified entry(leaving the specified entry), adding a new JavaScript object with the updated data and then copying the values from again slicing the rest of the current state(once again leaving out the specified entry). This behavior ensures that the current state is never directly modified since all methods return a new array which ensures that the reducer is pure.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen 2017-06-25

Illustration 17 Shows how the long list is connected the the long list data state and that the actions available for the long list is bound to dispatch an event to the store once called upon. The data in ‘state.LongListReducer’ contains the state associated with the reducer shown in Illustration 16 and will update once the reducer updates the state.

4.4.5 Navigation

Navigation between the different pages in the application is handled by the module ‘React Native Router Flux’(RNRF). RNRF uses Redux to store all available routes in the store which makes it so that the routes only has to be defined once to be accessible globally within in application. The routes could, instead of using absolute route, have a state based route meaning that the routes would serve different purposes depending on the specified state for the route. [63]

Illustration 18: Defining navigable pages with React Router Router Flux

With RNRF, the pages are defined as scene components as seen in Illustration 18 where every page defined with a key property will be accessible on the key value throughout the application.

Illustration 19: Connecting RNRF to the store

As seen in Illustration 19, connecting the router to Redux is done by using Redux’s connect and render the scenes within the Redux component Provider, passing along the scenes defined in Illustration 18.[64]

4.4.6 Platform Specific

React Native provides two ways of specifying code for different platforms: the first is through the platform module. The platform module is part of React Native’s core library and gives the opportunity to check which platform the appli-
cation is currently running on by the platform.OS variable. Platform.OS is one of either ‘ios’ or ‘android’ and can be used in both style sheets to specify different attributes depending on platform and in component logic.[65] The second option is to use the platform-specific file extension which allows for a whole file to only be loaded when running on the specified platform. This is achieved by adding either ‘.ios’ or ‘.android’ before the file extension.[66]

4.4.7 Native Modules

Native modules in React Native is written in the native language for the platform, Objective-c or Swift for iOS and Java for Android in order to combine real native code with React Native. Since the modules are written separately from the application, the available function calls has to be exposed to JavaScript in order to use them. The modules may pass data to JavaScript by using either a callback function[67] promises[68] or by emitting events[69]. For this implementation, accessing: Wifi information, phone call state and carrier information are done with native modules. For Android two additional features uses native modules: listening for incoming text messages and accessing the contacts on the phone. Android specific uses a BroadcastReceiver registered to listen for incoming text messages.

A native module for Android is created in three steps: first the Java file with the logic is created, then the Java file must be registered as a native module and lastly the native module must be provided to the application.[70]

Illustration 21: Android native module Java file

To create a native module in Java with the purpose of accessing the carrier information the first step is to create a Java file to hold the logic. As seen in Illustration 21, the class called HardwareProvider is created to inherit ReactContextBaseJavaModule – The default inheritance for an Android native module – which has one requirement, that the inheriting class implements the method getName.[71] A method called getCarrierInfo is created which expects to receive a callback function as parameter. The @ReactMethod annotation is put before the method declaration to expose the method to JavaScript. The method gets the
data about the carrier name and country code and then calls the callback function to pass the data back.

Illustration 22: Registering the Android module
The next step is to register the Java file as a native module and expose the module to JavaScript. As seen in Illustration 22, the class HardwareProviderPackage is created and implements ReactPackage. The react package interface enforces three methods for the class: createNativeModules, createViewManagers and createJSMODULES. CreateNativeModule is used when creating a native module and expects the method to return a list containing the native modules to be provided to the application. CreateViewManager is covered in chapter 4.4.5. The last method will not be covered in this implementation. For getting the carrier information and since a new native module is to be created the method createNativeModule is constructed to add a new object of the HardwareProvider to a NativeModule list and return it. The other methods are set to return an empty list.[72]

Illustration 23: Providing the application with the native modules
The last step of creating a native module in React Native for Android is to provide the application with the native module registered in the previous step. Illustration 23 Shows the method getPackages which is located in the ‘MainApplication.java’ file. This method should includes all native modules available for the application including the HardwareProviderPackage shown in Illustration 22.

A native module written in Swift for iOS is created in three steps: first the Swift file with the logic is created, then an Objective-C file is created to expose the methods from the Swift file to JavaScript and lastly an Objective-C header file is created to allow the Swift file to access Objective-C code.[73]
To create a native module in Swift for the purpose of accessing the carrier information and sim card information, the first step is to create the Swift file with the logic. Illustration 24 Shows the class CarrierInfo with the ‘@objc’ tag (which exposes the Swift class to Objective-C) and its one function – getCarrierInfo – which takes a single argument, RCTResponseSenderBlock, which is React native's way of allowing the method to use a callback function[67]. The getCarrierInfo method retrieves the necessary information from the CoreTelephony library and returns the data as an associative array by calling the callback function passed.

The next step is to create the Objective-C file that should expose the class constructed in Illustration 24 to JavaScript. As seen in Illustration 25, the swift class is exposed with ‘RCT_EXTERN_MODULE’ passing the class name and the type of the class. In this case: CarrierInfo and NSObject. The method getCarrierInfo in the class is exposed by ‘RCT_EXTERN_METHOD’ passing the defined name of the method and also what parameters the method expects. In this case: getCarrierInfo and a RCTResponseSenderBlock. The class and method is now exposed and is usable in JavaScript but one last step is required for the module to work.

The last step of creating an iOS native module for React Native is to expose some Objective-C libraries to Swift. As seen in Illustration 26, a new Objective-
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

2017-06-25

C header file is created which is to serve one purpose, import the required header files for the Swift to Objective-C bridging to work. The library RCT-BridgeModule.h is required for creating native modules in React Native and the second import – RCTViewManager.h – will be used later for creating native UI Components.

```javascript
var carrierModule = NativeModules.CarrierInfo;
carrierModule.getCarrierInfo((res) => {
   // Callback, do something with the data in res
});
```

Illustration 27: Calling the method getCarrierInfo passing along a callback function

Both the Android and iOS native module is used in JavaScript the same way. The module ‘NativeModules’ is imported from the react-native module which holds all exposed native modules. As seen in Illustration 27 a new object is created of the class CarrierInfo which is used to call the function getCarrierInfo, passing along a function to the callback parameter. When the native module returns data by calling the callback function, the function passed is triggered and the resulting data is accessible through ‘res’.

4.4.8 Native UI Component

Creating a native UI component in React Native is very similar to creating a native module. The process of creating a native UI component follows almost the same principles of creating a native module, create logic, expose to JavaScript and expose Objective-C for iOS and provide the module to the application for Android. The difference is mainly that a native UI component uses a view instead of logic and the module is registered using a different method. For this implementation, accessing the camera is done with a native UI component. IOS specific uses a native UI component with a delegate which responds to selecting contacts for accessing the contacts and to respond to processed images from the camera.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen

As seen in Illustration 28, the process of creating a native UI component for Android is separated into three required steps. For this implementation a fourth step is covered as well for maintainability.

- The only step not enforced is the Java file ‘view’ which holds the view of the component. This step could be incorporated in the next section. This file must include at least one method – getViewGroup – which is to return the root view for the component.[74]

- View manager which inherits ‘SimpleViewManager<ViewGroup>’ enforces two methods. The first method is getName, just like in native modules and the second method is createViewInstance. CreateViewInstance is responsible for creating the actual native view consisting of, in this case a view group. The view group is achieved by calling getViewGroup from the previous file.[74]

- The UI component must be registered through the ‘ReactPackage’ like when registering native modules. Unlike registering native modules, registering native UI components uses the method ‘createViewManager’ to create a new instance of the view manager from the previous step. [74]

- The component must be provided to the application by including the view manager in the list of react packages in the ‘MainApplication.java’ file.[74]
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

As seen in Illustration 29, the process of creating a native UI component for iOS in react native is split into two required steps. A third step is included in this implementation for maintainability.

- The only step not enforced is to separate the view from the view manager as this could be merged with the next step.

- A view manager which inherits RCTViewManager must be present to create the view for the component. The view manager overrides the method view which is to return a UIView retrieved from the previous step.

- The view must be exposed to JavaScript as a module through Objective-C by calling RCT_EXTERN_MODULE, passing along the view manager and the type for the view manager.

- Illustration 26 shows importing two libraries which of the latter library is required for creating UI components.

The native UI component is used in JavaScript the same way for Android and iOS, as seen in Illustration 30, the camera view component is imported by calling ‘requireNativeComponent’ and then used like a normal react component.

4.5 NativeScript with Angular

4.5.1 NativeScript CLI

The NativeScript CLI is used to provide a starting point for using NativeScript with Angular and the project is created with the command ‘tns create <project-Name> --template nativescript-template-ng-tutorial’[76]. To deploy and run the application with live reload on a device the command ‘tns run android’ for run-
ning on an Android device and ‘tns run ios’ for running on an iOS device is used.[27]

4.5.2 Setting up the UI?

Angular is used together with NativeScript to build the UI and to split the pages into components. Default NativeScript UI is built using XML files as the view, CSS files for styling and JavaScript or TypeScript files for logic. Together with Angular, the view is split into components, with XML or HTML files for building the view, Angular component files for the logic and CSS files for styling. The long list page is constructed using the NativeScript component ListView which renders as a list view for Android and a UI table for iOS.[77]

4.5.3 Animation

![Illustration 31: Animation in NativeScript to shrink and restore a view](image)

For animating the list view items, the NativeScript animate API is used. As seen in Illustration 31, the row which is to be deleted is animated by having the scale of the view shrunk down along with reaching zero opacity. Since the list view is recycling views, the view has to be restored to its original state once the row has been animated and deleted. NativeScript’s animate function returns a promise[78] so once the animation is complete, the item is removed from the list and the row is restored to its original scale and opacity.

4.5.4 Navigation

NativeScript has integrated native navigation e.g. the back button for Android and the back button on the navigation bar for iOS to go back to the previous scene when navigation between NativeScript pages. Pages are automatically created by the NativeScript framework either during runtime or when bootstrapping the application.[79] Since the application uses the Angular framework and is free to use any of the existing angular libraries, the angular router library (which is recommended by the NativeScript documentation[79]) is used to handle the navigation in the application.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen 2017-06-25

4.5.5 Platform specific

NativeScript allows, like React Native, platform specific file extension, ‘.android.’ and ‘.ios.’ and like React Native’s platform.OS, NativeScript offers platform.ios and platform.android which acts in a similar way. NativeScript also allows platform specific markup in the components view file with the ‘<ios>’ and ‘<android>’ block. UI element attributes can be platform specified by the ‘ios:attribute’ and ‘android:attribute’ syntax, “Every UI element attribute can be customized on a platform-by-platform basis!”[80]. Not only can css files be platform specified by using the platform specific file extension, but NativeScript also allows platform specific classes in the css files by adding either ‘.ios’ or ‘.android’ before the class name.[80]

4.5.6 Native modules

NativeScript allows accessing the platforms native API directly from JavaScript or TypeScript. This means that, unlike React Native which requires three steps to create a native module, a native module in NativeScript is created in one step. By a JavaScript module. As seen in Illustration 32, a native module in NativeScript is created in JavaScript and requires nothing in particular. For this implementation all native modules return a promise which allows for passing data back to where the module is used. Native modules are used in the NativeScript implementation to access the carrier information, the phone call state along with a delegate on iOS, listen for incoming text messages on Android using a BroadcastReceiver and accessing the contacts on Android.

Illustration 32: Native module in NativeScript

Illustration 33 shows accessing the carrier information from a iOS device by communicating with the native library “CTTelephonyNetworkInfo”. The module returns a promise passing along an object called carrier containing the carrier information once done getting the data.
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

2017-06-25

Illustration 34: Getting the carrier information on Android in NativeScript

Illustration 34 shows getting the carrier information from an Android device by requesting the system service “TELEPHONY_SERVICE”. The module returns a promise passing along an object called carrier with the carrier data if successful, otherwise the promise is rejected passing along the error message.

```
function getCarrierInformation() {
  return new Promise((resolve, reject) => {
    try{
      let telephoneManager = this.getSystemService(Context.TELEPHONY_SERVICE);
      let carrierName = telephoneManager.getSimOperatorName();
      let countryCode = telephoneManager.getSimCountryIso();
      let carrier = new this.Carrier(carrierName, countryCode);
      resolve(carrier);
    } catch (error){
      reject(error);
    }
  });
}
```

Illustration 35: Using the carrier module

Illustration 35 shows using the carrier module in JavaScript by importing the module created in Illustration 33 and Illustration 34, calling the ‘getCarrierInformation’ function and awaits a successful resolve or an error.

4.5.7 Native UI Components

For this implementation the native UI components are built using the presented method in 4.6.6 but instead of returning a promise the page is built dynamically using the native API. Native UI components are used to create the camera views and the iOS contact view along with a delegate for each which on iOS responds to processed images by the camera and selecting contacts from the contact picker.

As mentioned before, NativeScript automatically creates pages of their scenes and by requiring the module ‘ui/frame’ the current page is accessible through ‘ui/frame.topmost().currentPage’[81]. In iOS, building a UI in code uses the ‘UINavigationController’[82] and Android uses ViewGroup[44]. Both of these are accessible through the ‘ui/frame’ module through the respective ‘ui/frame.-topmost().ios.controller’ and ‘ui/frame.topmost().android.rootViewGroup’. Getting a hold of the root view is the only required step for dynamically building the view and further the required UI elements are added as sub views to the root view.
4.6 UI Testing

The UI tests are split into two groups: Android and iOS. The two platforms are tested separated from each other since comparing the two platforms to each other is out of the scope for this project and irrelevant to the result.

The test is to be conducted as following:

1. Navigate to the long list page.
2. Scroll down the list fast 50 times.
3. Edit and Delete 15 rows.
4. Scroll up the list fast 25 times to scroll over already scrolled rows.
5. Add a new item 10 times.
6. Scroll up the list fast 25 times.
7. Edit and delete repeated 25 times.

Navigation test:

- navigate from the start page to the long list page and back to the start page 10 times

The test scenarios are performed manually with a short break in between each task to let the CPU and the memory allocation to return to idle state. While performing the tasks the device is connected to: for Android, Android studio’s android monitor and for iOS, Xcode’s software Instruments to record the load of the CPU and memory allocation.
Results

Chapter 5 result presents the result of this thesis where chapter 5.1 along with 5.2 aims to answer how native close React Native and NativeScript is.

5.1 Native Access

<table>
<thead>
<tr>
<th>Feature</th>
<th>React Native Android</th>
<th>React Native iOS</th>
<th>NativeScript + Angular Android</th>
<th>NativeScript + Angular iOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS</td>
<td>JavaScript geo location</td>
<td>JavaScript geo location</td>
<td>NativeScript module</td>
<td>NativeScript module</td>
</tr>
<tr>
<td>Wifi information</td>
<td>React Native module</td>
<td>React Native module</td>
<td>NativeScript module</td>
<td>NativeScript module</td>
</tr>
<tr>
<td>Phone Call state</td>
<td>Native module</td>
<td>Native module + delegate</td>
<td>Native module</td>
<td>Native module + delegate</td>
</tr>
<tr>
<td>Carrier Information</td>
<td>Native module</td>
<td>Native module</td>
<td>Native module</td>
<td>Native Module</td>
</tr>
<tr>
<td>Incoming Text messages</td>
<td>Native module + broadcast receiver</td>
<td>-</td>
<td>Native module + broadcast receiver</td>
<td>-</td>
</tr>
<tr>
<td>Phone Contacts</td>
<td>Native module</td>
<td>Native UI component</td>
<td>Native module</td>
<td>Native UI component</td>
</tr>
<tr>
<td>Camera</td>
<td>Native UI component</td>
<td>Native UI component</td>
<td>Native UI component and NativeScript module</td>
<td>Native UI component and NativeScript module</td>
</tr>
<tr>
<td>Save images to gallery</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Live reload/ hot reload</td>
<td>Live and hot reload</td>
<td>Live and hot reload</td>
<td>Live Reload</td>
<td>Did not make it work</td>
</tr>
</tbody>
</table>

Table 1: Native Access shows how the native feature is accessed on each framework

Looking at table 1, there are a lot of similarities between React Native and NativeScript and also across the platforms when it comes to accessing the native API's. The native implementation is left out on purpose since if the API is accessible on non native, it is accessible on native. The broad overview of table 1 shows that every feature covered by this implementation works on both frameworks. Accessing incoming text messages on iOS is an exception since it is not
possible even with a native iOS implementation. Most native API’s are accessible through native modules since most of the times it is only about passing data. The camera and iOS contacts are accessible via a native component since they require adding visible components to the screen such as a camera preview and a contact modal.

5.2 Performance
Following are the results from the conducted UI tests. Because of the platforms automatic garbage collection, the amount of memory allocated is difficult to pin down. What is interesting about the memory allocation shows is how the memory is allocated when performing the specified task. The CPU load is presented in percent of how much of the CPU the application is using while performing the specified task.

5.2.1 Android

*Illustration 36: Memory allocation when scrolling long list*
Illustration 37: CPU usage when scrolling the long list

The memory allocation shown in Illustration 36 shows that the React Native and Native Script application behaves in a similar way when scrolling the list. For scrolling the long list the native application uses on average 30% CPU, the React Native application uses average 51% CPU and the NativeScript application uses on average 45% CPU.

Illustration 38: Memory allocation when editing and deleting in the long list
Illustration 39: CPU usage when editing and deleting in the long list

The memory allocation shown in Illustration 38 shows that when editing and deleting in the long list the memory allocation is almost constant in all three applications and the lowest allocating application is the NativeScript version. The CPU usage peaks a lot on the React Native implementation and is at most at around 65-70% usage. Based on the average CPU load, the native implementation is the lowest CPU using application for editing and deleting entries with 28% usage versus 29% for NativeScript. The most CPU using application is the React Native application with an average of 35%.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

Illustration 40: Memory allocation when scrolling up already scrolled space

Illustration 41: CPU usage when scrolling up already scrolled space

Illustration 40 and Illustration 41 shows the memory allocation and the CPU usage when after editing and deleting the list is scrolled up already scrolled over space. The graph in Illustration 41 shows a similar behavior as the scrolling down graph shown in Illustration 37. The average CPU load for the native implementation is on average 47%, the React Native application uses on average 46% and the NativeScript application uses on average 41%.
Illustration 42: Memory allocation when adding elements to the long list

Illustration 43: CPU usage when adding elements to the long list

Illustration 42 and Illustration 43 shows the memory allocation and the CPU load when adding new elements to the long list. As seen in Illustration 42, the memory allocation for the React Native application is constant and the lowest allocating application is the NativeScript application. Illustration 43 Shows that the CPU usage for the React Native application and the NativeScript application is very similar, both peaking at around 70% CPU usage where the native application hovers around 40-45%. On average the native application uses 34%
CPU, the React Native application uses on average 38% CPU and the NativeScript application uses on average 35% CPU.

**Illustration 44:** Memory allocation when navigation to and from the long list page

**Illustration 45:** CPU usage when navigating to and from the long list page
Illustration 44 and Illustration 45 shows the memory allocation and the CPU usage when navigating to and from the long list page. Illustration 44 shows that the native applications memory allocation is almost flat once the first navigation and both the React Native and the NativeScript application allocates memory at an almost constant rate. The CPU usage shown in Illustration 45 shows that the native application has an almost flat CPU load around 40% and the React Native application has a semi flat CPU usage once the first navigate has been made around 60% but goes up to around 70% towards the end. The NativeScript application has one initial CPU burst to around 55% on the first navigation but hovers around 35-45% after that. On an average the native application uses 36% CPU, the React Native application uses on average 58% CPU and the NativeScript uses on average 40% CPU.

5.2.2 iOS

Illustration 46: Memory allocation when scrolling the long list
The iOS device uses two cores, therefore the maximum CPU load for the iOS application is 200%.

Seen in Illustration 46, the memory allocation when scrolling behaves similar on the NativeScript and the native implementation. Illustration 47 shows a big difference in the three applications CPU load. The native application has an average of 27% CPU and peaks at 47%, the React Native application uses on average 85% CPU and peaks at 129% and the NativeScript uses on average 51% CPU and peaks at 86%.
Illustration 48: Memory allocation when editing and deleting in the long list

Illustration 49: CPU usage when editing and deleting in the long list

Illustration 48 and Illustration 49 shows the memory allocation and the CPU load when editing and deleting the data in the long list. As seen in Illustration 48 there is no big difference between the three applications memory wise. Illustration 49 Shows that the native application uses on average 10% CPU, the React Native application uses on average 66% CPU and the NativeScript application uses on average 24% CPU.
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

2017-06-25

Illustration 50: Memory allocation when scrolling up already scrolled space

Illustration 51: CPU usage when scrolling up already scrolled space

Illustration 50 and Illustration 51 shows the memory allocation and the CPU usage when scrolling up back over already scrolled space. As seen in Illustration 50 the memory allocation for the native application is flat throughout the full scroll where the React Native application starts off high, falls off in the middle and returns to high towards the end. The NativeScript application’s memory allocation is similar to the native application’s memory allocation, almost flat. Il-
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen

Illustration 51 Shows the CPU load for the three applications with big differences. The native application uses on average 35% CPU, the React Native application uses on average 110% CPU and the NativeScript application uses on average 71% CPU.

Illustration 52: Memory allocation when adding elements to the long list

Illustration 53: CPU usage when adding elements to the long list
Illustration 52 and Illustration 53 shows the memory allocation and the CPU usage when adding the elements to the long list. As seen in Illustration 52, the memory allocation for the native application and the NativeScript application is close to flat and the React Native application rises at an almost constant speed flattening out towards the end. Illustration 53 Shows the CPU usage where the native application uses on average 14% CPU, the React Native application uses on average 111% CPU and the NativeScript application uses on average 44% CPU.

Illustration 54: Memory allocation when navigating to and from the long list page
Illustration 55: CPU usage when navigating to and from the long list page

Illustration 54 shows the memory allocation when navigating to and from the long list page. The Native application and the React Native application is similar and almost flat and the NativeScript application allocates memory at a constant rate. Illustration 55 Shows the CPU usage when navigating to and from the long list page. The native application uses on an average 24% CPU, the React Native application uses on average 62% CPU and the NativeScript application uses on average 34% CPU.
6 Conclusion

My hypothesis was that the React Native application was going to be almost as well performing as the native application since Facebook with a lot of resources are the developers. As the result proves, React Native was the worst performing of all the applications. What is not shown in the result is how the feel is when using the application. The React Native application when using the long list has a long response time and is over all not usable as an application. The NativeScript application on the other hand performed much better than the React Native application in the result and again what does not show in the result is the feel when using the application. The response time is close to the native applications response time when pressing buttons and scrolling in the long list but the render time when scrolling is noticeably slower than the native application.

The biggest difference between React Native and NativeScript with Angular is that one uses React and one uses (optional) Angular for building the UI. Apart from that the biggest difference is how the two frameworks expose the native API. React Native relies on walking away from - in my case – VisualStudio code into Android Studio or XCode when a native module or UI component is a must. Whereas NativeScript exposes the native API directly in JavaScript meaning that there is no need for a second IDE. React Native prefer to have the native code separated from the JavaScript – file structure wise – and NativeScript keeps the native code together with the JavaScript.

When it comes to native access, there is little, if any, differences between the two frameworks. As seen in chapter 5.1, what is possible to access in native is accessible in React Native and what is accessible in React Native is accessible in NativeScript.

Over all based on the performance result, out of React Native and NativeScript, NativeScript performed much better than React Native in almost all categories. React Native when scrolling, editing and deleting and adding new data to the long list has a CPU load of over 100%. When using the application the React Native application is also noticeably slower with several seconds response time in the long list. Over all the React Native application is quite fast, for example navigating to any other page than the long list page.

Looking at platform specification options, NativeScript is way ahead of React Native. React Native offers two ways of specifying code for a specific platform – platform specific file extension and platform.OS – where NativeScript offers the equivalent of React Native along with several different more options. Platform specific styling is more natural in NativeScript than it is in React Native since NativeScript offers platform specific class names and platform specific attributes.
The live reload on the two frameworks is a bit different. Personally I never got live reload on NativeScript iOS to work. React Native’s live reload feels faster than NativeScript’s live reload but it is difficult to say. Nevertheless both frameworks offer live reload for a faster development process.

Looking at the documentation, React Native’s documentation is easier to search in and over all better than NativeScript’s documentation. The community is also a lot bigger for React Native with 6,822 subscribers on their forum on Reddit (2017-05-24) compared to NativeScript’s 354 subscribers on their forum on Reddit (2017-05-24). So getting help when stuck is easier on React Native than NativeScript.

Angular has a reputation of having a steeper learning curve than React which is to be taken into consideration if choosing to develop in either of these. Personally if considering building an application requiring a lot of native access then I would prefer NativeScript due to the fact that NativeScript exposes the native API’s directly to JavaScript and over all I liked that the application is developed with Angular which makes separation a lot easier.

6.1 Code reusability cross platform

Both React Native and NativeScript is made to encourage one code base for multiple platforms and does it well. Both frameworks allow for ways of specifying platform specific code and platform specific styling. Following, the presented sub queries to the first problem statement presented in chapter 1.6 are answered.

Under what circumstances can graphical components be reused between iPhone and Android?

The default graphical components given by the respective framework are made to work cross platform. Only one thing arises problems when trying to use the same UI component cross platform. Which is when it is required to create a custom native implementation of the UI component. Which then requires one implementation of the component per platform. But as long as the component uses the same name, only one instance of the component is required to be defined in the view.

On the contrary, for each component given by the frameworks, there must exist a platform specific representation of the component. The question must be defined further to answer and is rephrased as following:

- Under what circumstance can graphical components provided by the framework be reused between iPhone and Android?

Given the above question there is nothing that contradicts reusing the graphical component cross the platforms.

52
Under what circumstance can logic based components which interact with the user interface be reused between iPhone and Android?

Components which include logic which interacts with the user interface would most of the times manipulate properties on a specific UI component, for example having a button trigger an animation on another component. Both frameworks use serialization to pass the components properties to the native interpretation so that the property is translated correctly. This question is answered with a similar answer to the previous question. As long as the components given by the framework are used there is no problem with reusing them. Only when creating custom components problems arise, but as long as the properties are passed to the component the correct way and are named the same there is no problem defining only one component.

How does the design principles between iPhone and Android affect code reuse cross platform?

The design principles do only affect the styling of the components. The base view is the same for the different platforms but the styling is supposed to be different when using an Android device or an iOS device. This is solved by both frameworks by making use of either platform specific extension syntax for the style sheets or by including a different style based on the platform. So the design principles and the overall design of the application will be troublesome to reuse cross platform.

6.2 When is non native application development an option?

For answering when a non native application development is an option, a few scenarios are set up. The scenarios include: Developing a game, developing a social media application and developing a business application. These are common scenarios for developers and the technique included applies to many other scenarios as well.

Developing a game

If a game was to be developed on either of these frameworks. No matter the type of game, 2D or 3D, performance is one of the highest priorities which makes the native implementation the only available option. The native implementation holds the lowest CPU load and memory allocation on almost all of the test cases which makes the native implementation almost the only contestant for developing a game in.

Developing a social media application

The type of the social media platform is what determines if a native implementation is the better choice. For example, if the intention was to create Snap Chat, which relies heavily on customizing the device’s camera API then maybe a na-
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

tive implementation would suite the needs better. A React Native or NativeScript application would definitely get the job done but there would be no need for a non native application since 90% of the application would be in the native platforms code. If on the other hand, the social media to create was Twitter, which relies more on passing text over the network than it relies on customizing the camera API, then a React Native or NativeScript application is definitely an option.

Business

Assume an internal business application for displaying last years booking and managing staff’s schedule which would rely mostly on fetching and pushing data around and displaying lots of data at the same time. Network connection is out of this thesis scope so there is nothing to say about that. On the other hand, displaying lots of data at the same time resembles the long list implementation and according to the performance tests made the React Native implementation does not hold up. Although the NativeScript application did actually perform well enough to be a contestant along the native application. If the business already has a web site developed in Angular, then the NativeScript with Angular application is a definite choice since much of the already existing code can be reused and there is no need for the developers to learn both the android and the iOS framework.

6.3 Development time

There was a noticeable difference in development time for the proof of concept applications. My prior knowledge was basic knowledge in Android and React, which meant that: iOS, Redux, Angular and NativeScript was entirely new to me. Both native applications took about one and a half week each to complete, resulting in about three weeks to get the same applications up and running on native Android and native iOS. React took about one and a half week, mostly due to redux having a steep learning curve, to get an Android and an iOS application up and running. The Angular NativeScript application took about one week to complete to get an Android and an iOS application up and running.

The resulting development time is entirely personal and without Redux, the react application would probably have taken about one week as well to complete. This gives about one third development time of native development when doing the applications the non native way.

6.4 Live Reload

The live reload was a nice feature to use when developing the applications. It provided faster development since there was almost no down time between each build. Looking at table 1 in chapter 5.1, the result shows that the only time the live reload did not work was on NativeScript iOS. Since the community for NativeScript is so small and it’s a small chance that some one has encountered some error before there is nothing much to say why I never made it work. My
speculation is that: Apple is very restrictive of how an application is allowed to be deployed on an iPhone and with no hassle the application is deployed through XCode but since NativeScript’s live reload relies on having the application deployed through NativeScript’s CLI there must be something that is not approved correctly by Apple. The weird thing is that, React Native’s live reload also relies on deploying the application from the CLI and that works fine on iOS.

6.5 Best practices

There are severe limitations when it comes to best practices for developing applications in React Native and NativeScript. There is even lesser best practices for using native modules and UI components since it’s not at common to do so when developing in these frameworks. Following are some best practices that was discovered by experience when developing the applications and the outline for this chapter is as following: First some general best practices for the two frameworks and then one section React Native specific and one section NativeScript specific.

Specifying size for native UI components. When creating native UI components in both React Native and NativeScript it is important to specify a certain size for the element as the element has no initial size when created. The element may be resized after creation though.

6.5.1 React Native

Func.bind(this) in the constructor instead of inline. Scopes in JavaScript can cause a lot of head ache if not handled properly. When developing a React application everyone will eventually have to use the correct scope and is faced with two options: call the function and bind this inline like ‘function().bind(this)’ or create a constructor, call super in the constructor and then assign the function to the class’ along with the same ‘function().bind(this).

[83] The latter is a bit more time consuming so naturally the developer choses the first alternative saving a few seconds. This works fine in many cases and goes by un-noticed. I did this too in the long list implementation to start off with. What I did not realize was that .bind returns a copy of the function[84]. The list was incredibly slow with over six seconds response time after the list had been scrolled a bit. I changed and put ‘this.function = this.function.bind(this)’ in the constructor and the response time went from >6 seconds down to <1 second.

Platform specific Styling. Unlike React for the web, React Native has not ‘className’ attribute. Instead the styling is – by recommendations from the React Native documentation – defined as plain old JavaScript objects. React Native does not offer any way of specifying a specific class or style as iOS specific or Android specific. The way I solved this was by using React Native’s platform specific file extension to create two files – style.ios.js and style.android.js – and export a style objects in them. Then it is just to use ‘var style =
require('path/style') and the framework will automatically include the correct style for the platform.

Break off as much reusable logic as possible. This says it self and is a no brainer for an accustomed developer but it still deserves its own shout out. React Native uses JavaScript for the logic. React for the web uses JavaScript for the logic. Break away as much of the logic e.g. data fetching, data models etc since this can be reused both on React for the web and React Native.

6.5.2 NativeScript

Make use of the fact that NativeScript supports Angular. Angular is not only great at building a good user experience in terms of UI, but Angular offers so much more than just UI. There is no limit to how much of the Angular library you use but the more you use it, the more you can reuse on the web. For example css style sheets, services, data models, navigation and animation.

6.6 Ethical aspects

There are some ethical aspects to consider from this thesis work. The first is the whole idea of going against what the developer of the platform (Android and iOS) want from the developers. If more people went the non-native approach, there would be a substantial drop in developers visiting the official native documentation. This could lead to, at lowest, a decrease in quality of the official documentation and those who actually stay true to native might find it difficult to find what they are looking for.

The most obvious ethical aspect of this thesis work is the fact that non-native development requires if any native knowledge. As long as the developers stay away from native modules and native UI components, they only need web development knowledge. This means that the demand for native mobile developers will decrease if more and more adapt the non-native approach. On the contrary, the demand for web developers with a broader competence level will be sought after.

6.7 Alternative approaches

There are a lot of ways the implementation could have been done otherwise. For example, there are several libraries out just for navigation, Redux is just one of many state managers for React, the native implementation might not be the most optional. All of which may affect the result either positive or negative. Out of what was experienced during the development, the following alternative approaches which may or may not have an affect on the result were thought of:

- In the NativeScript implementation, the camera view is made not made as a native UI component as the React Native implementation is. The camera view is built dynamically by building the camera view on the current page instead of placing a camera component on the scene. This
could probably be made into a ‘real’ UI component by using Angular’
directive and thus be reused as a component in the html files.

- The long list implementation for React Native uses flatlist which is no-
ticeably slower than the NativeScript implementation. Both implementa-
tions renders a List View for android and an UI table for iOS but the re-
act native implementation(specifically on Android) is still noticeable
slower. Why I do not know. And a different approach could be to test
different public modules for listviews to see if there is any difference.
This is also dedicated to future work. On a side note: In a real applica-
tion there would almost never be 25 000 consecutive rows in a list view.
Some pagination would be made.

6.8 Future work

During the development, a few questions appeared that would be interesting to
investigate. Some of which are as following:

- React with NativeScript – I found a source(which is lost now) saying
  that purely architectural, NativeScript would work with React.

- Investigate further why the long list implementation on React Native is
  slower than the long list implementation on NativeScript. Both uses the
  same component once rendered.

- Do performance tests on other than only the long list since React Native
  was so far behind in the performance on it.

- Investigate further why the live reload was unable to work on Native-
  Script iOS.

These questions are dedicated to potential future work as it is not in this thesis’s
scope but would be interesting and perhaps useful to get an answer to.

The UI tests was supposed to measure the current frames per seconds while per-
forming the tasks. Unfortunately the FPS tests was forgotten so performing FPS
tests while re-enacting the test cases presented in this report is dedicated to fu-
ture work.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen

References

https://developer.apple.com/app-sandboxing/
Retrieved 2017-06-01

Retrieved 2017-06-01

Retrieved 2017-05-15

Retrieved 2017-05-15

Retrieved 2017-06-01

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html
Retrieved 2017-06-01

Retrieved 2017-05-29

Retrieved 2017-05-31

Retrieved 2017-05-31
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

[10] TypeScript, “TypeScript – JavaScript that scales”
https://www.typescriptlang.org/
Retrieved 2017-05-30

https://facebook.github.io/react/
Retrieved 2017-05-31

[12] Redux, “Redux”
http://redux.js.org/
Retrieved 2017-05-31

http://redux.js.org/docs/basics/Actions.html
Retrieved 2017-05-31

[14] Redux, “Reducers”
http://redux.js.org/docs/basics/Reducers.html
Retrieved 2017-05-31

http://redux.js.org/docs/basics/Store.html
Retrieved 2017-05-31

[16] Redux, “Changes are made with pure functions”
http://redux.js.org/docs/introduction/ThreePrinciples.html#changes-are-made-with-pure-functions
Retrieved 2017-05-31

https://medium.freecodecamp.com/why-redux-needs-reducers-to-be-pure-functions-d438e58ae468
Retrieved 2017-05-31

Retrieved 2017-06-01

[19] React Native, “JS frame rate(JavaScript thread)
https://facebook.github.io/react-native/docs/performance.html#js-frame-rate-javascript-thread
Retrieved 2017-05-31
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

https://angular.io/docs/ts/latest/guide/architecture.html
Retrieved 2017-06-01

https://angular.io/docs/ts/latest/guide/architecture.html#!#components
Retrieved 2017-06-01

[22] Angular, “Lifecycle hooks”
https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html#!#hooks-overview
Retrieved 2017-06-01

[23] Angular, “Templates”
https://angular.io/docs/ts/latest/guide/architecture.html#!#templates
Retrieved 2017-06-01

https://angular.io/docs/ts/latest/guide/architecture.html#!#services
Retrieved 2017-06-01

https://docs.nativescript.org/
Retrieved 2017-05-31

[26] NativeScript, “NativeScript with Angular”
https://docs.nativescript.org/angular/start/introduction.html
Retrieved 2017-05-31

[27] NativeScript, “Running apps”
https://docs.nativescript.org/angular/tutorial/ng-chapter-1.html#12-running-apps
Retrieved 2017-05-31

Retrieved 2017-05-30

Retrieved 2017-05-30

Retrieved 2017-05-30
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

2017-06-25

[31] ExploringJS, “Default exports”
Retrieved 2017-05-31

[32] React Native, “Native Modules - Android”
Retrieved 2017-05-31

[33] React Native, “Native Modules – iOS”
https://facebook.github.io/react-native/docs/native-modules-ios.html
Retrieved 2017-05-31

[34] React Native, “The Toast Module”
https://facebook.github.io/react-native/docs/native-modules-android.html#the-toast-module
Retrieved 2017-05-31

[35] React Native, “iOS Calendar Module Example”
https://facebook.github.io/react-native/docs/native-modules-ios.html
Retrieved 2017-05-31

[36] NativeScript, “Data Conversion”
https://docs.nativescript.org/angular/runtimes/android/marshalling/overview.html
Retrieved 2017-05-31

[37] NativeScript, “Marshalling”
https://docs.nativescript.org/angular/runtimes/ios/marshalling/Marshalling-Overview.html
Retrieved 2017-05-31

[38] React Native, “Native UI Components - Android”
https://facebook.github.io/react-native/docs/native-components-android.html
Retrieved 2017-05-31

[39] React Native, “Native UI Components – iOS”
https://facebook.github.io/react-native/docs/native-components-ios.html
Retrieved 2017-05-31

[40] Stack Overflow, “What is the difference between Hot Reloading and Live Reloading in React Native?”
Retrieved 2017-03-31
Note: Nader Dabit’s answer, edited Jan 2.
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

[41] JSON generator, “JSON generator – tool for generating random data”
http://www.json-generator.com/
Retrieved 2017-05-17

[42] React Native, “The React Native CLI”
https://facebook.github.io/react-native/docs/getting-started.html#the-react-native-cli
Retrieved 2017-05-25

[43] NativeScript, “Install the NativeScript CLI”
http://docs.nativescript.org/start/quick-setup.html#step-2-install-the-nativescript-cli
Retrieved 2017-05-25

Retrieved 2017-05-16

Retrieved 2017-05-17

Retrieved 2017-05-17

[47] Android Developer, “RelativeLayout”
Retrieved 2017-05-17

Retrieved 2017-05-29

Retrieved 2017-05-29

Retrieved 2017-05-16
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen

2017-06-25

[51] Android Developer, “SMS_RECEIVED_ACTION”
Retrieved 2017-05-29

[52] Apple Developer, “UITableViewCell”
Retrieved 2017-05-29

[53] Apple Developer, “UITableViewController”
Retrieved 2017-05-29

[54] Apple Developer, “dequeueReusableCell(withIdentifier:)”
Retrieved 2017-05-29

https://developer.apple.com/library/content/featuredarticles/ViewContro
erPGforiPhoneOS/UsingSegues.html
Retrieved 2017-05-29

[56] Apple Developer, “CLLocationManagerDelegate”
legate
Retrieved 2017-05-29

[57] Apple Developer “CLLocationManager”
Retrieved 2017-05-29

[58] React Native, “Testing your React Native Installation”
https://facebook.github.io/react-native/docs/getting-started.html#testing-your-react-native-installation
Retrieved 2017-05-25

[59] React Native “Animations”
https://facebook.github.io/react-native/docs/animations.html
Retrieved 2017-05-30

[60] Mozilla Developer Network, “Array.prototype.concat()”
https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?
v=control
Retrieved 2017-05-17
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen
2017-06-25

[61] Mozilla Developer Network, “Array.prototype.slice()”
Retrieved 2017-05-17

Retrieved 2017-05-17

[63] GitHub, React Native Router Flux”
https://github.com/aksonov/react-native-router-flux
Retrieved 2017-05-17

[64] GitHub, “Redux/Flux”
https://github.com/aksonov/react-native-router-flux/blob/master/docs/REDUX_FLUX.md
Retrieved 2017-05-18

[65] React Native, “Platform Module”
https://facebook.github.io/react-native/docs/platform-specific-code.html#platform-module
Retrieved 2017-05-18

[66] React Native, “Platform-specific extensions”
https://facebook.github.io/react-native/docs/platform-specific-code.html#platform-specific-extensions
Retrieved 2017-05-18

[67] React Native, “Callback”
https://facebook.github.io/react-native/docs/native-modules-ios.html#callbacks
Retrieved 2017-05-18

[68] React Native, “Promises”
https://facebook.github.io/react-native/docs/native-modules-ios.html#promises
Retrieved 2017-05-18

[69] React Native, “Sending Events to JavaScript”
https://facebook.github.io/react-native/docs/native-modules-ios.html#sending-events-to-javascript
Retrieved 2017-05-18
Native versus non native – A comparison of React Native and NativeScript to native application development
Timothy Lawler Karvonen 2017-06-25

[70] React Native, “Native Modules Android”

[71] React Native, “The Toast Module”
https://facebook.github.io/react-native/docs/native-modules-android.html#the-toast-module
Retrieved 2017-05-18

[72] React Native, “Register the Module”
https://facebook.github.io/react-native/docs/native-modules-android.html#register-the-module
Retrieved 2017-05-18

[73] GitHub gist, “React Native – Swift Native Modules”
https://gist.github.com/robertjpayne/855f6db15d5ceca12f6c5
Retrieved 2017-05-18
Note: zxcpoiu’s comment on 11th April 2016

[74] React Native, “Native Components Android”
https://facebook.github.io/react-native/docs/native-components-android.html
Retrieved 2017-05-18

[75] React Native, “Properties”
https://facebook.github.io/react-native/docs/native-components-ios.html#properties
Retrieved 2017-05-19

[76] NativeScript “Creating Apps”
http://docs.nativescript.org/angular/tutorial/ng-chapter-1#11-creating-apps
Retrieved 2017-05-25

[77] NativeScript, “Listview”
https://docs.nativescript.org/angular/code-samples/ui/listview.html
Retrieved 2017-05-19

[78] NativeScript, “Using NativeScript Animations”
https://www.nativescript.org/blog/using-nativescript-animations
Retrieved 2017-05-23

[79] NativeScript, “Pages”
https://docs.nativescript.org/angular/core-concepts/angular-navigation.html#pages
Retrieved 2017-05-22
Native versus non native – A comparison of React Native and NativeScript to native application development

Timothy Lawler Karvonen 2017-06-25

[80] Developer Telerik, “Platform-Specific Development with NativeScript”
Retrieved 2017-05-19

[81] Stack Overflow, “NativeScript get current page”
https://stackoverflow.com/questions/36074420/nativescript-get-current-page
Retrieved 2017-05-22

[82] Apple Developer, “UIViewController”
Retrieved 2017-05-22

[83] React, “Handling Events”
https://facebook.github.io/react/docs/handling-events.html
Retrieved 2017-05-24

[84] MDN, “Function.prototype.bind()”
Retrieved 2017-05-24
Appendix A: Redux flow for the React Native implementation

The following appendix illustrates how the flow for Redux works in the React Native application and is used to give further information in chapter 4.4.4.