Masaryk University
Faculty of Informatics

Analysis and experiments with NativeScript and React Native framework

Master's Thesis

Bc. Dominik Veselý

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Analysis and experiments with NativeScript and React Native framework

Master's Thesis

Bc. Dominik Veselý

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Dominik Veselý

Advisor: Mgr. Juraj Michálek

Acknowledgement

I would like to thank my supervisor Mgr. Juraj Michálek, and all the employees of Y Soft Corporation, who helped me with consultations, comments, and useful advice during the creation of this thesis.

Abstract

This thesis examines and compares two frameworks for cross-platform mobile development. These frameworks are NativeScript which is developed and maintained by Telerik and React Native which is developed and maintained by Facebook.

At the beginning, we will define the objectives we are going to investigate and compare and we will divide different approaches for creating cross-platform mobile applications. Later we will describe the common features of both frameworks. Then we will introduce each of our frameworks separately and finally, we will summarize the obtained data and compare both frameworks side-by-side.

During the examination of both frameworks, we will create a mobile application in each of them. The application will be a terminal for the Y Soft SafeQ. It should be able to scan bar codes to obtain server address to connect to. Then there needs to be a login screen where the user can sign in. And finally, there will be the page displaying a list of jobs that are ready to be printed with the ability to print each of such entries. Of course, both of our applications has to be cross-platform.

Keywords

NativeScript, React Native, Angular 2, React, cross-platform mobile development

Contents

List of Tables

List of Figures

1 Introduction

In this thesis, we are going to compare the two frameworks for mobile development which are NativeScript and React Native. The main aim of these frameworks is in cross-platform mobile development. That is creating a single application which will run on multiple devices with different operating systems.

Since mobile phones are matter of course nowadays, the importance of mobile applications is therefore huge. As of today, we have a multiple operating systems available for mobile phones where the biggest share has Android and iOS. The estimated market share is 86.8% for Android and 12.5% for iOS [1]. These add up to the sum of 99.3% market share. Therefore it is crucial to support these two mobile platforms with your application to cover almost entire smartphone share. To cover these two platforms we would have to develop our application twice, once for Android and once for iOS. This lead to the arrival of the cross-platform mobile development. Nowadays there are several approaches for creating a cross-platform mobile applications which will be described in section 2. It is important to choose the correct framework for your specific use case when creating a cross-platform application. NativeScript and React Native are two fairly young frameworks which should satisfy most of these use cases. In this thesis, we will try to find out how comprehensive and robust they actually are.

First, we will define the objectives that we are going to compare. We chose some general objectives that every solid mobile framework should satisfy. Also, since this thesis is realized in cooperation with Y Soft Corporation, a.s., we include some objectives specifically requested by them.

In the next chapter, we are going to divide the different approaches for creating a mobile application. We will briefly describe each of the approaches. We will try to point out the strongest and the weakest features of them and we will provide a representative framework for each of them.

Since both frameworks have several similar features, we will describe those in the third chapter. While these features are present in

both of the frameworks, the way they are implemented differs so we will still compare these features in later sections.

The fourth and fifth chapter will present the NativeScript and React Native framework in detail. We will introduce each of them with some basic information. Also, we will analyze the capabilities for each of our predefined objectives and we will describe how they are achieved in more detail.

In the final chapter, we will conduct the comparison of both frameworks side-by-side. We will try to summarize the findings for both frameworks for each objective and we will point out what is better or worse in either of the frameworks.

An important note is that we will use Android as the primary target platform and Windows as a primary development platform. That means that we will compare all the features and experiences during development from the Windows plus Android point of view.

For the development IDE[1], we chose Visual Studio Code [3]. It is free and open source development IDE from Microsoft and it has good support for both of our frameworks.

During this thesis, we will mention the npm several times. It is a packet manager for management of dependencies inside a JavaScript project. Packet managers are available in almost every programming language as it provides a good way for the management of other libraries your project depends on.

We will also shorten the writing of a word application to simply write the word app. It is generally used abbreviation [4] and it will save us some unnecessary writing while we will use this word a lot.

1.1 Objectives

First, we need to define objectives that we are going to rate and compare.

It is important to note that both examined frameworks allow us to use any native features and libraries of the target platforms so that almost everything should be achievable with them. Never the less

1. An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development [2].

we will consider a given feature as supported only if it is already accessible by the framework itself or supported by a 3rd party library.

With that said if we mark some feature as not supported by given framework, it does not mean this feature is not achievable within that framework, but it is only not yet supported by the vendor or by some 3rd party library.

Supported target platforms – We will define currently supported mobile platforms by our frameworks. Also, we will investigate plans for potentially supported platforms in the future. We will also cover the minimal OS versions for targeted platforms.

Framework complexity – This subject will cover the initial amount of time and knowledge required to be able to develop a standard app.

Result app performance – We will compare the resulting app performance on all targeted platforms. This is a very important aspect of user friendliness and usage of the app. Also, this is one of the core problems these frameworks should be solving.

Animations – Animations add a lot of smoothness and better app flow. This is one of the main reasons people choose to create native application (2.1) rather than cross-platform app created via WebView (2.2) because the performance penalty of this approach is even more observable when trying to implement smooth animations.

We will cover both ease of implementing animations as well as the performance and smoothness of animations in the final apps.

Custom fonts – One of the very important features that should be supported by the framework is support for using custom fonts. As Y Soft has its own custom font, the final mobile app should be able to use that font.

Community – Nowadays it is very important that frameworks or even programming languages have a solid community. It pretty accurately defines the state and liveness of given framework. The bigger community also predicts more add-ons and features created for that framework. We will try to find out the current popularity of both frameworks and its progress. It could be tricky to objectively measure the size and liveness of community and developers contributing to given framework. Luckily both frameworks use GitHub to version and manage its code so it will nicely serve us for our measurements. This topic also covers areas of Internet forums like the amount of asked questions or solved problems. Here we will mainly focus on

Stack Overflow, which is a special type of on-line forum and it is very popular nowadays.

The gathered data will be always from the same date (concretely 8.3.2017) for both frameworks to achieve the most accurate result.

On GitHub we will compare these statistics:

- Stars – This is probably the most significant indicator as it is a number of GitHub users who gave this framework a kind of „like“.
- Contributors – Number of users contributing to that framework with their code and ideas.
- Issues – Here we will compare the number of opened/closed issues. Naturally, the more popular the framework is the more issues will arise as no software is flawless. The decisive indicator will not be just the number of opened and closed issues but also the ratio between those two numbers. That ratio is an indicator of how big part of the opened issues is actually resolved.
- Contribution commits[2] – This shows the overall speed of development of the framework in a form of graph.
- Commits in last year – The detailed evolution of commits to the framework in the last year displayed in week intervals.

3D support – Since Y Soft company was interested in the integration of management of 3D printing into the mobile app it would be nice to be able to display such 3D model within our app. We will evaluate the ability to display 3D models possibly supplied in some standard 3D format like STL or Collada.

Barcode scanning – It is nowadays very common that some interactions with a mobile app are done through scanning a barcode with phone's camera. We will determine whether it is possible to scan QR codes[3] within given framework.

Build times – We will compare the speed of the compilation and synchronization of the whole app. We will measure the speeds of three build types:

2. On Git the commit means the smallest unit you can contribute with to the project. It is usually a small feature or fix that has been done.
3. QR code is a type of barcode.

- Compilation from scratch – This is the situation when we want to compile our application for the first time or when we want to rebuild the whole app. This type of compilation is the least often used.
- Successive compilation – This operation is performed when we want to update the whole app. It is used in a situation when we changed the native code or added some library that has native files. Also, we use it when we get back to continue developing our app.
- Livesync – This is the most frequent type and it is used to quickly transfer the files used in the JavaScript runtime. The way it works will be discussed in section 3.2. The speed of this synchronization is crucial as it determines the time developer has to wait between he made some change to the code and when the change is visible in the app. We will split this type into two subtypes that are synchronization of the view change (appearance) and synchronization of the code behind change (functionality).

The build times will be measured on our apps created in each framework. This way we can get more objective results as there will be some nontrivial amount of code already present.

Memory consumption – We will measure the memory consumption of the running app for each framework. The measured values will be in megabytes. We will provide the total memory consumed and allocated and free memory at the end of the measurement. We will present the average of the memory consumption and the standard deviation for each measured type.

1.2 Application to create

We will create a cross-platform app in each of our frameworks. Both apps will have a similar functionality. They will have to have these main capabilities:

- An ability to scan a QR code with the endpoint URL address, assuming the barcode scanning is available in the framework.

- Login screen where the users can sign in with their credentials.
- Dashboard where the logged user can see an overview of print jobs prepared for printing, which is called job list. Also when the print job is able to be printed we will provide a button through which the given job can be sent to the printer.
- The app has to communicate with the endpoint via RESTful[4] API[5].

Both apps will also contain a test environment for the performance testing.

4. Representational state transfer (REST) or RESTful Web services are one way of providing interoperability between computer systems on the Internet [5].
5. An application programming interface (API) is a set of clearly defined methods of communication between various software components [6].

2 Different approaches for creating mobile application

Here we will describe most common ways of creating mobile apps while we will mostly focus on the cross-platform ways of doing this.

2.1 Native application

First and the eldest method is to create single app multiple times for each target platform separately. Therefore it is not a cross-platform development, but it is a most narrow way to do that.

It has numerous advantages as you can use a 100% of each platform's features and performance. Therefore these applications will also be the fastest and smoothest ones.

On the other hand, in order to target n platforms, we have to create n applications. That is the reason why cross-platform mobile frameworks came into the scene.

2.2 Cross-platform though WebView

This is the simplest approach in achieving a cross-platform mobile app. These apps work in a way that on the start of the app there is shown a WebView[1] which is then present during the whole run of the app and everything the application does is shown within this WebView.

The biggest advantage of this approach is that it is very simple to create such app as you are in fact creating a web page that is shown across whole display. Therefore you are using commonly known web-based technologies like HTML[2], CSS[3] and JavaScript. These apps only try to mimic native platforms (buttons, labels, etc.), which is often not enough.

Problems of these apps comprise mostly performance as these apps render their HTML content into the WebView which then needs

1. It is like a window of a web browser, but there is no URL bar or navigation buttons.
2. Hypertext Markup Language
3. Cascading Style Sheets

to be processed again by the mobile's native environment. Also, there are other technical limitations mainly that the app runs in a single thread so the whole app might get stuck with GUI[4] as well when it performs some intensive „background task".

Example frameworks based on a WebView approach are Apache Cordova, Ionic, AppGyver Supersonic, TouchstoneJS.

2.3 Cross-platform though cross-compilation

These applications are written in one common language and then the code is compiled and native applications for each targeted platform are produced.

The advantage of this approach is that final apps are truly native in a way that all elements displayed on the screen are actually elements of given target platform rather than their mimics as described in 2.2. Final apps are also really performant as these apps contain an encapsulated runtime for the common code in which the framework works.

The main disadvantage is the requirement of the recompilation of the whole app when some change is done. Also when writing an app in this kind of framework it is generally needed to build different UI for each platform and only the code behind (manipulating data etc.) is shared.

An example of this approach is Xamarin where the common code is C#.

2.4 Cross-platform though interpreted runtime

Finally, there is this newest approach for creating cross-platform mobile applications which is said to be the future of mobile application development. It works bit similar to the cross-compilation way described in 2.3. The running app has some native core provided by the framework and another runtime of some interpreted program-

4. The graphical user interface (GUI), is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators [7].

ming language[5] which is common for all target platforms where the concrete app is developed. As React Native's documentation nicely describes „This framework uses the same fundamental UI building blocks as regular iOS and Android apps. You just put those building blocks together using JavaScript." [8].

Advantages are that all UI components (buttons, labels, etc.) are actual components of each target platform so the UI is completely native on each platform. Also, these frameworks provide ways to call arbitrary native features which give developers ability to access all features of given target platform. Up to here, this is very similar to the cross-compiled type of apps described in 2.3, but these frameworks offer more. Specifically they try to merge the common functionality[6] into common API but also provide ways to work with features specific to various platforms. Which means we only need to make one UI for all target platforms. Also for the most changes made in the app during the development, there is no need to recompile the whole app but just transfer the changed code which can by swapped at runtime thanks to the interpreted nature of chosen programming language.

The disadvantage is that this approach is still fairly new and so these frameworks are quite young and under large development.

Examples of such frameworks are NativeScript, React Native or Titanium.

5. Interpreted programming language does not need to be compiled before it's run. The „interpreter" which runs the code reads the code „as is", parses it and then executes it.
6. Functionality that is provided by all targeted platforms.

3 Common features of both frameworks

Both of our frameworks have a lot in common but also there are a lot of things they differ in. Here we will cover the common features so we won't have to explicitly mention them with each framework later.

Mainly they are trying to solve the same problem to create cross-platform mobile apps while writing only one code (at least most of the time). They do so by choosing a common programming language for writing the app code that is able to run on all supported platforms. This language happens to be JavaScript[1] for both frameworks as there is a JavaScript runtime available for all targeted platforms.

3.1 Framework architecture

The basic architecture is the same for both frameworks. A visualization of the architecture is displayed in figure 3.1. As briefly described in 2.4, we have two types of code in these frameworks. There is the native code which is unique for each target platform that wraps the whole application. Then there is the shared JavaScript code.

3.1.1 Native code

This code is Java for Android and Objective-C for iOS. Our application is compiled as a standard native application where on the native side there is mainly the framework's core. It provides the basic functionality for running the app. It creates a runtime for the JavaScript code (described in 3.1.2) to run. This basically creates a container which encapsulates the runtime that runs (interprets) and manages the JavaScript code. It also works as a sort of inter-communicator which provides a binding between JavaScript code and the target platform's native code so that we can actually call native functionality from JavaScript code. This includes calling native framework's features, 3rd party native libraries, application's native code and native API as shown in the architecture in figure 3.1.

1. Since now we will refer to the common code as JavaScript.

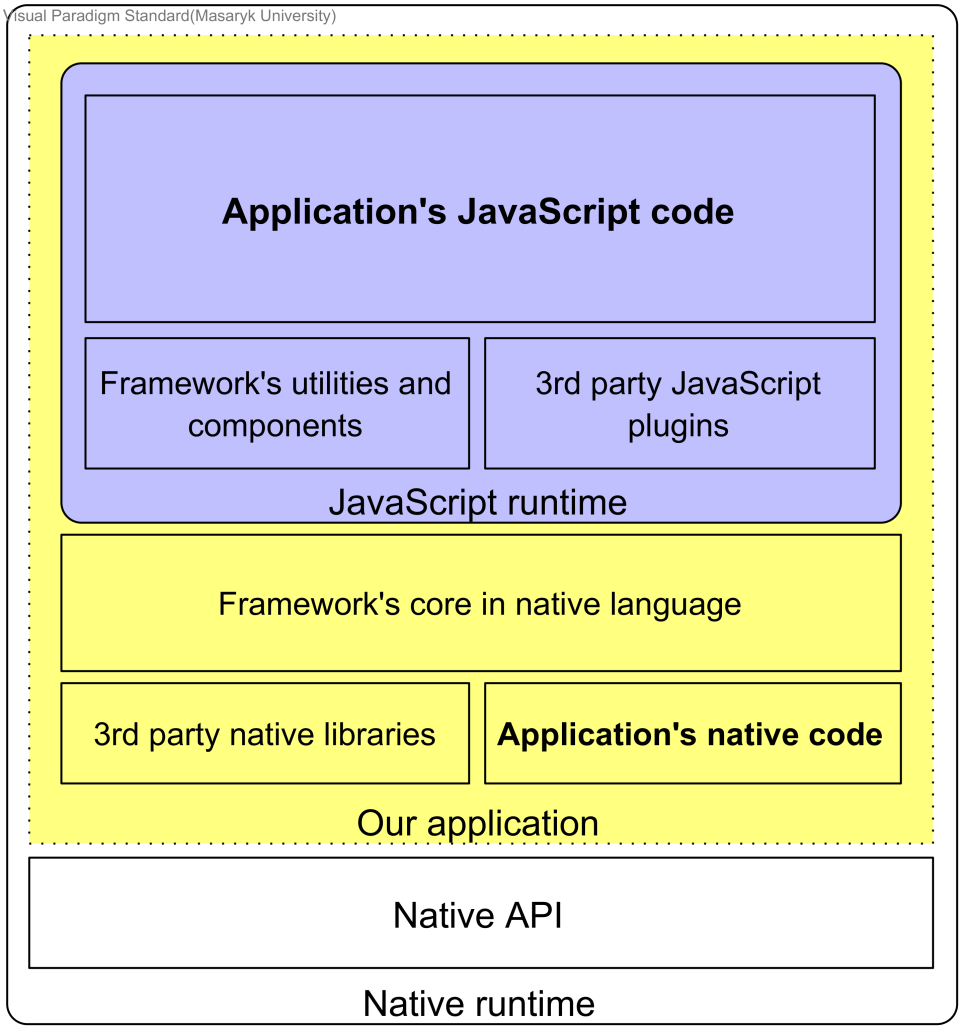

Figure 3.1: A visualization of the architecture of the frameworks

3.1.2 JavaScript code

This is the code that is shared between all target platforms and which is the one being interpreted in the runtime provided by the native core 3.1.1. This is the place where the developers write most of their app code. But there is not just the user code, there is also a big part of the framework's code since the components and a lot of functionality provided for developers by the framework are already written in JavaScript.

JavaScript code communicates with the framework's native core during whole life-cycle of the application. JavaScript calls get translated and performed in the native environment and the result is then transformed back and returned to the JavaScript environment.

3.2 Code compilation and code synchronization

Our frameworks also try to cope with problems of source code compilation. When developing in native languages or for example in the framework like Xamarin, after any changes are done it is required to recompile the whole app, deploy the compiled app to a device which can be real device or emulator[2] and finally reinstall the app for changes to take place. This is really long and tedious process. Currently, there is a big improvement with the so-called incremental building[3] which speeds whole deployment process by a lot but there is still a notable delay.

Both of our frameworks work the way that they have two types of code as described in 3.1 where only the native code needs to be compiled when deploying our app to a device. Luckily this native code barely changes as we write mostly JavaScript code which both of our frameworks take advantage of by compiling whole application only for the first time and any subsequent changes made only within JavaScript code can be then synchronized live to the running app without the need of complete recompilation.

2. An emulator is a program running on the computer which emulates some concrete mobile device (eg. Google Nexus 5X) with desired OS running on it. This removes the necessity to own the device when developing the app for it.

3. This process speeds up the compile time by recompiling only changed parts of the code.

Because the JavaScript is interpreted language[4] we don't even have to compile it before transferring it to a device. It is fair to note that JavaScript codes of both frameworks need to be preprocessed before being transferred to a mobile device as they contain quite a lot of enhancements like added type control to the originally untyped JavaScript (which will be explained in 4.4) or by adding a support for the newest JavaScript features that are not yet supported by the JavaScript runtimes on the target platforms. But this JavaScript preprocessing is still much faster than the compilation of the native code.

Also since the native core of the app is unchanged the running app mostly does not even have to be reinstalled nor restarted on the device and the changes are applied „on the fly" that we call „livesync". Plus both frameworks support automatic checking of the changed code and perform updates of the changed code automatically.

All the features mentioned above greatly reduces the deployment and synchronization time between the change is done and when it is visible in the running app and thus make the development process much faster and smoother.

3.3 Access to native features

As briefly mentioned in 1.1 and also as described in the architecture 3.1 both frameworks provide access to the arbitrary native feature. It allows us to call native API and native libraries from the JavaScript so that almost everything should be achievable. This leverages developers to use any feature or library of given platform so they are not constrained to features only provided by the framework itself.

Basically both our frameworks works the way that they implemented or chose some native components for each target platform that provide the same or similar functionality and linked them to be used from JavaScript. In JavaScript, we then simply use for example Button component and we work with it independent on the target platform. Our frameworks then do the job of binding this JavaScript Button wrapper to the corresponding native Button component implementation.

4. Interpreted programming language does not need to be compiled before running it as it's being interpreted during runtime.

The same way we can define our custom components. We provide implementations for each target platform in its native code and then create a JavaScript wrapper that will represent them. It could be quite difficult to provide some custom functionality to our app depending on the complexity of the task but it has huge potential as the developers will never get actually stuck on fact that some feature is not provided by the framework.

3.4 Usage of second level framework

Nowadays it is very popular to use some framework when programming in some programming language. A framework is generally some set of tools and basic functionality commonly required when building an app. Then simply provide some base ground for more convenient development.

Since our frameworks provide the core functionality to run the common code on multiple platforms it does not provide some high-level JavaScript features. Nevertheless, our frameworks bring a second level JavaScript frameworks on top of themselves. Each of them ships with different second level frameworks. So the choice between our frameworks could then be based on preferences between these second level frameworks.

3.5 Usage of arbitrary JavaScript library

Generally, since both our frameworks are JavaScript frameworks developers are free to use any npm package. The decisive thing is it might not work in the mobile JavaScript runtime which is in some ways different than runtime in a web browser. On a mobile there is no **DOM**[5] representation of a page nor notoriously known global variables like `window` or `document`. So for example, famous JavaScript library jQuery won't work as it is dependent on the DOM.

5. Document Object Model

3.6 Open-source

Both of our frameworks are open-source. This means that even though both of them have been created and maintained by big companies, the source code of them is freely available and the community can still contribute to these frameworks. This is a very popular trend nowadays that even such big companies make their software as open-source. The inspiration this comes from big corporations like Apache Software Foundation or Linux Foundation which are inherently non-profit corporations and still they are able to produce quality software. Another good aspect of the open-source software is that it is free.

4 NativeScript

4.1 About

NativeScript is a cross-platform framework for building native mobile apps using JavaScript, CSS, and XML or HTML. This framework is backed and maintained by Telerik [9]. The beginning of this framework dates back to the March of 2014. As a second level framework there is Angular 2 which is described in 4.5.

The versions of the framework and other packages used to create the final app and for the evaluation of the capabilities are displayed in table 4.1.

4.2 Target platform support

With NativeScript we can build apps for iOS and Android while there is a Windows support in planning.

4.2.1 Android

It is possible to develop and produce an Android app on almost any desktop platform including Windows, Linux, and Mac. To start working with NativeScript we need to install NativeScript CLI[1] [10] and Android Studio [11].

Created app requires Android API level 17[2] to run [12].

4.2.2 iOS

It is possible to develop an iOS app on Mac only as it requires to have Xcode installed which is available only for Mac [13].

Created app requires iOS8 or higher to run [14].

1. Command Line Interface
2. Android API level 17 matches the Android version 4.2.

Package	Version	Description
nativescript	2.5.2	the command line interface for management of the NativeScript project
tns-core-modules	2.5.1	basic features provided by the framework
@angular/*	2.4.3	standard Angular 2 with series of its modules
nativescript-angular	1.4.0	extension providing functionality of Angular 2 inside NativeScript

Table 4.1: The versions of npm packages used in our NativeScript app.

4.2.3 Windows

The Windows Universal platform[3] support is being promised as „coming soon“ but is currently stuck [15]. The reason is they tried several approaches to implement Windows Universal support with reasonable effort and complexity. This attempt failed as there would be needed serious intervention to the framework's core and a development of several extra tools. So the current statement is that Windows support is not worth the effort at the moment as there are more important things to improve in NativeScript. Anyways the Windows Universal platform support is currently planned for summer 2017.

4.3 Developer options

When building an app with NativeScript you can choose whether to write code in pure JavaScript or use TypeScript (described in 4.4). Even though TypeScript provides a lot of useful error checking and advanced code completion some might still prefer to write their app in pure JavaScript and NativeScript provides an option to do that.

Also, there is an option whether to use that second layer framework which is Angular 2. Since NativeScript itself is older then the Angular 2 framework it means the support for it was added later on when

3. Windows universal means support for desktop and mobile/tablet applications.

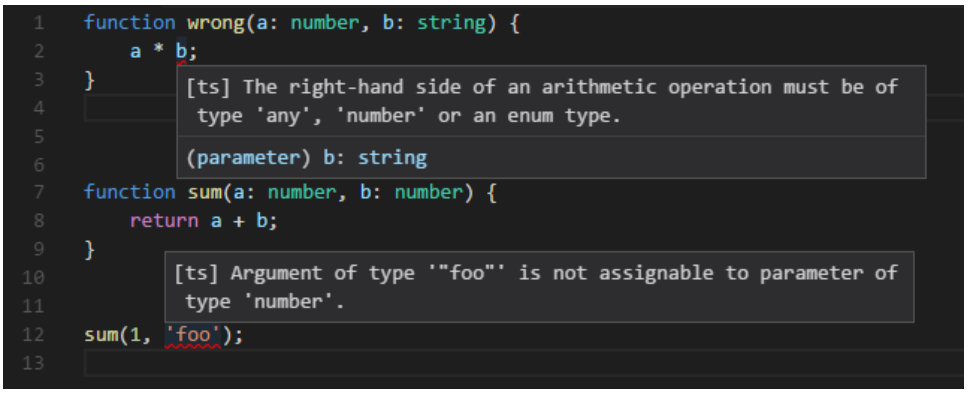

Figure 4.1: An example of TypeScript error checking integrated with the Visual Studio Code.

NativeScript was already released. This shows that NativeScript was able to adopt this framework on top of itself and work with it correctly.

4.4 TypeScript

TypeScript is an open-source programming language developed and maintained by Microsoft. It is built on top of the JavaScript language and it provides additional features to JavaScript namely type annotations and compile-time type checking, interfaces, enumerables, etc. TypeScript code gets compiled back to JavaScript which is then run.

More specifically TypeScript is a strict superset of ECMAScript 2015. It also provides new JavaScript features of event proposals in JavaScript that are currently not supported by all web browsers or other JavaScript interpreters but these features get compiled to some older standardized versions of JavaScript. We can specify a target version of the JavaScript that we want to compile our TypeScript code to. This way we can write code with the newest features and still be able to run the compiled version in older browsers.

When using TypeScript it is important to work with IDE supporting it as it then offers code completion and live error checking as shown in figure 4.1.

In connection with NativeScript, its runtime requires ECMAScript 2015 code which is default TypeScript target compilation version.

As a negative side of TypeScript could be seen the fact that as the project gets bigger the time it requires to run the type checking and code compilation gets equally larger. So this could become a bottleneck when developing a bigger app.

4.5 Angular 2

At the beginning, there was no second level JavaScript framework on top of the NativeScript, but later NativeScript developers got together with Angular developers and created Angular 2 which can be used as a web framework as well as mobile framework inside NativeScript.

Angular 2 is JavaScript framework build on TypeScript. It is a successor of AngularJS framework and it is one of the most famous frameworks for developing a frontend of a web app.

This framework has been integrated to work with NativeScript through the nativescript-angular module. In NativeScript this framework works for practically the same purposes as on the web. It mainly offers features like better structuring of pages of our app into Components, moving business logic into separated service classes, providing developer with all sort of features like routing, HTTP calls based on reactive style programming and it also leverages developers with proven design patterns like inversion of control which is achieved by the dependency injection. Plus since the framework works the same way in NativeScript as on the web, some model classes – those are classes encapsulating logic for the user logging in/out, obtaining user information etc. – can be shared between a web app and mobile app.

The main difference between a web and mobile Angular is that in mobile version you cannot use standard HTML tags like `<div>` (which may a lot of people assume otherwise). There is a completely different set of tags available and those are the native components or more correctly their NativeScript abstractions like `<Label>` for displaying text.

4.5.1 Problems of Angular 2 inside NativeScript

We noticed that the fact that user has the ability to choose whether to use Angular 2 framework or not has a negative effect as well. The

problem arises when there is described only the one way of working with some component in the NativeScript documentation. If it happens to be the case that there is described only the case without the usage of Angular, the developer then has to either go to the source code definition of the component to explore how it should be used or he has to search the Internet for some unofficial example of the usage.

Angular 2 provides data binding meaning that when a developer changes the value of some variable this change is automatically propagated into the view visible to a user. Angular uses so-called zones and only changes done within this zones are being automatically propagated which solves the performance issues from Angular 1. Unfortunately sometimes when we change some value the newly set value does not get automatically propagated as the change was not done within a zone. There is an easy fix for this problem as we can wrap the setting of our variable to the zone manually, but it is an extra effort we need to make and it could be a potential cause of problems.

Even though Angular 2 is integrated into NativeScript for a fairly long time, there are still some issues with this connection. One of the main problems is the fact that when we put into HTML markup some component that Angular does not know, it simply gets skipped. There is no error provided, the skipping of the unknown component is done without any notice. This is also confusing when there is only a misclick in the component name or maybe the desired component actually exists but it is not registered to be able to use it and then the developer is confused about why it is not being displayed while there is no error provided.

4.6 Styling

In NativeScript apps are being styled via CSS. This technique was originally designed to visually style web pages, but thanks to its advantages many other frameworks adopted it as a styling tool. NativeScript chose to style its apps through CSS as it has a lot of similar aspects of web development. In NativeScript not all CSS rules are supported. There is only a strict subset of rules that is achievable by native components as well as some extra rules specific to NativeScript not present in a pure CSS.

The main advantage of using CSS is that it is easy to use and well as it is known from a web development. Also using CSS opens options for usage of CSS preprocessors[4]. NativeScript offers to use SASS as a CSS preprocessor available as an npm package [16].

On the other hand, when styling web app, developers are leveraged with web browser features (DevTools) like inspecting page elements, applying styles without the need to reload the page, analyzing which styles are being applied and which are overwritten. None of which is available when styling an app in NativeScript. That leads to few problem like when some CSS rule is not being applied and the developer has no way of determining the core of the problem (CSS selector might be wrong, the rule might be overwritten or there could be a bug in the framework).

We encountered problems with CSS in our app when trying to apply the custom font to the whole app. The font got applied almost everywhere except for the buttons. After trying to specify the CSS path as specific as possible, then trying to enforce the font specification rule by the `!important` keyword, none of this solved the problem. Luckily there is also available the ability to style the elements through the `style` attribute through which the custom font gets applied. This leads to not so clean code as each time we add a button, we always need to add the manual font styling. Also if we wanted to then change the font of the app, we would not only have to change the one entry in CSS file, but also go through all HTML files and change all occurrences of manually applied font on all buttons of the app which is common disadvantage of styling elements through `style` attribute.

```
1 // Set custom values
2 $orange: #FF6600;
3 $brown: #FF6600;
4
5 // Import theme
6 @import 'nativescript-theme-core/scss/orange';
7
8 // Custom variables
9
```

4. CSS preprocessor is language extending pure CSS that makes it easier to style the application. It leverages CSS language with extra features like nested styling rules, reusable code snippets, variables etc. This language is always compiled to pure CSS that is then used to style the app.

```
10 // Import styles
11 @import 'nativescript-theme-core/scss/index';
12
13 // Custom CSS rules
14
15 .page, .form, .action-bar, label, button, .btn-primary,
       .input-field {
16     font-family: Ysoft-Regular;
17 }
18
19 button, label, stack-layout {
20     horizontal-align: center; // Custom rule
21 }
```

Listing 4.1: Styling of the NativeScript app with the SASS preprocessor.

There is a sample of using the SASS language and applying some basic styles global for the whole app in the code 4.1. We can see one example of NativeScript's custom CSS rule on the line 20.

4.7 Animations

In NativeScript we have an ability to animate elements that are being displayed to improve the attractiveness of our app.

It is possible to to animate various properties of visible elements:

- **Opacity** – Opacity means the level of visibility where the value is between 0 (not visible at all) and 1 (completely visible).
- **Background color** – We can create a fluent transition between two colors of a background.
- **Translate X** and **Translate Y** – By these two properties we can change the position of the element on the screen. By animating these properties we will achieve fluent movement of the element to the desired position on the screen.
- **Scale X** and **Scale Y** – By animating these two properties we can fluently „scale“ the element. This means we make it larger or smaller.
- **Rotate** – Rotates the component appropriately around its Z axes.

When animating and element we can setup multiple properties of the animation. Those properties are:

- **Duration** – Length of the animation
- **Delay** – Delay before the animation starts
- **Iterations** – Number of animations to be performed
- **Timing function** – Function that controls the process of the animation. We can choose from predefined functions – like popular **easing** functions – or we can define own function for example through cubic Bezier curve.
- others – There are several other properties like the direction of the animation, whether the animation should persist the final state after it finishes, etc.

Animations can be achieved by two approaches which are declarative and imperative.

4.7.1 Declarative

This way we can define animations through CSS[5]. It is a very convenient way of defining animations as it's fairly easy since it uses the same syntax as animations on the web so that many developers will be familiar with it straight away. This way we can also animate multiple components via single CSS definition. On the other, hand we have to put all animation definitions in a single CSS rule otherwise the animation will not be performed. This means that we cannot create CSS animation which will for example „fade in" multiple elements and then modify the animation for each component (like changing the delay so that all the components would not „fade in" in the same time).

An example of the declarative definition of an animation is displayed in the code 4.2.

5. NativeScript uses CSS3 compatible animation definition.

```
@keyframes fade-in {
    from { opacity: 0; }
    to { opacity: 1; }
}

.fade-in {
    animation-name: fade-in;
    animation-duration: 1s;
    opacity: 0;
    animation-fill-mode: forwards;
}
```

Listing 4.2: Defining the „fade in“ animation through CSS.

4.7.2 Imperative

This is an alternative way of defining animations through JavaScript code. This method brings more control over animations as we can precisely control things like animation sequence[6], canceling a running animation, hooking to the animation competition, etc.

Most of the time developer will be satisfied with the declarative approach but it is nice to have an alternative with more settings.

An example of the imperative definition of an animation is displayed in the code 4.3.

```
this.fadeLabel.nativeElement.animate({
    opacity: 1,
    duration: 1000,
    curve: AnimationCurve.easeInOut
});
```

Listing 4.3: Defining the „fade in“ animation through JavaScript code.

4.8 Navigation

Because the routing module is a part of the Angular 2 framework NativeScript uses this module and extends it to add few mobile phone specific things to the navigation like animated transition between pages, etc.

6. The animation sequence is a subsequent animation of multiple elements.

When defining a routing module in our app we start with the definition of routes, these are the parts of our app we can navigate to. We specify the path string similar to the web browser URL bar which will be an identifier and we specify the component which should be rendered at given path. The example definition of routes is displayed in code 4.4.

```
1 const routes: Routes = [
2     {path: "", redirectTo: "/home", pathMatch: "full"},
3     {path: "home", component: IntroComponent},
4     {path: "login", component: LoginComponent},
5     {path: "dashboard", component: DashboardComponent},
6 ];
```

Listing 4.4: Definition of the routes

Then we can navigate through our app by either setting the navigation link on the UI component as displayed in code 4.5 on line 2 which is similar to defining the `href` attribute on the web, or we can manually navigate from JavaScript code as displayed in code 4.5 on lines 5-9.

```
1 // Navigation through component definition
2 <Button text="Go home" class="btn btn-primary" [
     nsRouterLink]="['']" transition="fade"></Button>
3
4 // Navigation through JavaScript code
5 this.router.navigate("", {
6   transition: {
7       name: "fade"
8     }
9 });
```

Listing 4.5: Using the routing module to navigate between pages

The definition of routes and then the navigation itself is really straight forward and easy. We can also perform some additional steps like clearing the navigation stack[7] so that user won't be able to return to this page (he is being navigated from) nor any previously visited pages.

One of the flaws noticed about this navigation is when navigating between the same page with different arguments (eg. browsing

7. It is natural on mobile devices when navigating forward the previous page is preserved so we can then return to this page with the back button.

among print jobs). The router uses the exact same component and only switches the data that are being shown (eg. print job data). This is a fast way to display new data but NativeScript does not perform the layout recalculation so we need to enforce the page to be recalculated by for example hiding the old content, switching the data and then showing the new content. This can be beautified with an animation but it needs to be done because when the layout was calculated with the old data all labels[8] were created with enough space for the old text. When the text gets changed „on the fly“ the previously calculated label width could not be wide enough for the newly assigned text which would then be trimmed with the ellipsis. To show this o some example if we would have a label with text: *Short text*, and later we would assign text to it like: *This is a longer text*, the result would look like: *This is...* Also when navigating to some other component, this new component needs to be first created internally by the framework. When navigating to some more complex component the creation time is notable and during this time whole app freezes. This is mostly visible when we have some animation being played when the navigation happens which results in the freeze of the animation.

4.9 Custom fonts

NativeScript comes well prepared for use of custom fonts. All that is required is to put font files (eg. in otf format) to the */fonts* folder of our project and then apply it through CSS like so: `font-family: Ysoft-Regular;` which also visible in the code 4.1:16.

The basic setup is really simple except there is that problem with changing the font of buttons as explained in 4.6.

4.10 Accessing native API

One of the huge advantages of NativeScript is that we can access arbitrary native API from within a JavaScript code. If we want to run a snippet of a native code all we need is to translate this snippet into JavaScript syntax and the framework will perform the task in a native

8. Labels are used to display text.

environment. The developer then does not have to leave the JavaScript code at all since he is only writing the native code in JavaScript syntax. This way we can fine grind our app to have some specific native features of each target platform without writing any native code.

There is an example of accessing the native API in the code 4.6. There is a function which arguments are normal JavaScript classes of the NativeScript framework. All NativeScript's JavaScript classes have two special properties that are `android` and `ios` where those properties carry the native implementation of given object but only when we are on such platform otherwise they are `null`. This way we can check whether we are on the desired platform as is visible on lines 2 and 5. Then we can call some method that is available only in the native implementation of the object, for example the `setHintTextColor` method available on Android's implementation. This method takes the color object but only the specific android implementation so we could not pass the `args.color` to this method since it is just a JavaScript object. Nevertheless, we can get the native implementation of the color with `args.color.android` which can be also passed to the native method. There is also a similar more complex example for iOS platform where we also create a new instance of some native object (on line 10) but the idea is identical.

```
1  function setHintColor(args: { view: TextField, color:
       Color }) {
2    if (args.view.android) {
3      args.view.android.setHintTextColor(args.color.android
         );
4    }
5    if (args.view.ios) {
6      let dictionary = new NSDictionary(
7        [args.color.ios],
8        [NSForegroundColorAttributeName]
9      );
10     args.view.ios.attributedPlaceholder =
         NSAttributedString.alloc().
         initWithStringAttributes(
11       args.view.hint, dictionary);
12   }
13 }
```

Listing 4.6: An example of accessing the native API for both Android and iOS [17].

This direct access to the native API has a huge potential and could be even the decisive factor when choosing the appropriate framework to use.

4.11 Error handling and debugging

In NativeScript we receive errors and warning during the runtime of the app with two ways. During developing an app we are connected to the device running our current version of the app and so we are automatically getting the feed of the console logging. These console loggings are mostly warnings and developer dumps[9]. An example of such logging is displayed in code 4.7.

```
1 W System.err: at com.tns.Runtime.callJSMethodNative(Native Method)
2 W System.err: at com.tns.Runtime.dispatchCallJSMethodNative(Runtime.
      java:865)
3 W System.err: at com.tns.Runtime.callJSMethodImpl(Runtime.java:730)
4 W System.err: at com.tns.Runtime.callJSMethod(Runtime.java:716)
5 W System.err: at com.tns.Runtime.callJSMethod(Runtime.java:697)
6 W System.err: at com.tns.Runtime.callJSMethod(Runtime.java:687)
```

Listing 4.7: Console loggin output with en error feedback

Another way errors are provided is right inside our app. When an error occurs, the developer is provided with the error screen displaying a detailed log of the exception. This approach is more clear then the console logging as it is much more clear and nicely formatted. Also, the developer does not have to be checking the console all the time. Examples of such error screens are displayed in figure 4.2.

The form of displaying the exception is one thing, but there also has to be some meaningful error message and meaningful stack trace for the exception to be useful so the developer can accurately locate and fix the error. With NativeScript we experienced both the good and the bad examples of exceptions. Examples of bad error messages and bad stack traces are in code 4.7 and on the right image of figure 4.2. An example of a good exception is presented on the left image of figure 4.2. Unfortunately, the bad errors were much more frequent than the good ones during our development.

9. Developers are able to use special functions to display some values in the console which helps them assemble the app logic correctly. Such values could be the current value of a variable, data received from some service, etc.

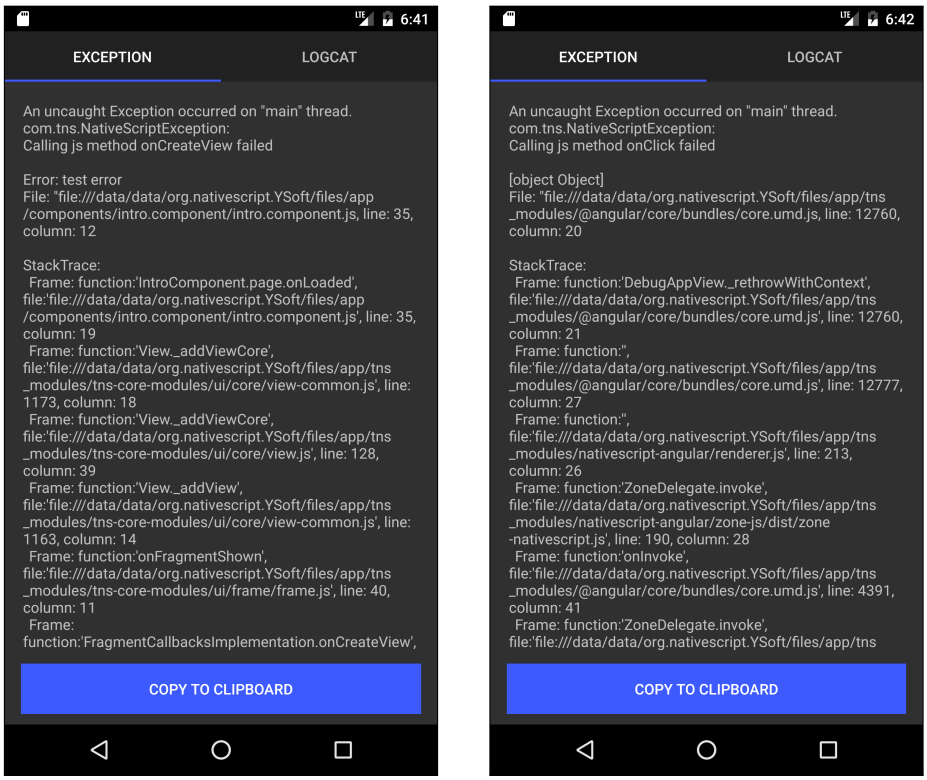

Figure 4.2: Error screens showing exceptions in a NativeScript app. There are exceptions with good stack trace (left) and with bad stack trace (right).

Luckily when writing the app in TypeScript (described in 4.4) it avoids a lot of runtime errors as it is able to catch a lot of them by the static analysis of the code. This mainly detects errors like importing JavaScript module that does not exist, using a method that does not exist on a particular object or applying operations with wrong argument types.

As for debugging the app, we are able to use Visual Studio Code which contains built-in debugger that can be attached to the running NativeScript app. With this, we can use all the features of modern debuggers like placing breakpoints and stepping through app instruction-by-instruction, viewing the current values of variables, etc.

Build type	$\overline{X}$	σ
Full compilation	02:17,03	00:02,00
Successive compilation	00:21,27	00:00,82
Livesync view	00:04,90	00:00,20
Livesync code	00:11,58	00:00,95

Table 4.2: The average time and standard deviation of compilation and synchronization speed in NativeScript.

4.12 Build times

All the various types of compilation have been presented in 1.1. The measured times of each build type in NativeScript are shown in table 4.2. We can see that the standard deviation is very small so all build times are consistent. We will talk more about the measured times in comparison chapter in section 6.3.

4.13 Community

Here we will mostly present the gathered data. Their meaning was described in the introduction in section 1.1.

GitHub statistics:

- Stars – 9620
- Contributors – 73
- Issues – 213 opened (2094 closed, solved percentage 90.77%)

On Stack Overflow there are 1620 questions with the **nativescript** tag.

The evolution of contribution commits and commits in the last year are displayed in figures 4.4 and 4.3.

4.13.1 3rd party libraries

NativeScript has its Plugin Marketplace currently containing only 17 plugins [20]. On the other hand, these are approved by Telerik and

Figure 4.3: Graph showing the contribution commits evolution in NativeScript GitHub repository [18].

Figure 4.4: Graph showing commits evolution in the last year in NativeScript GitHub repository [19].

they should be top quality (including code quality, documentation, and future support).

Telerik itself offers some premium UI components including Chart, ListView, SideDrawer, Calendar, DataForm [21] but only the SideDrawer and ListView are available for free.

On the npm portal, there are over 400 3rd party libraries created for NativeScript [22], some of which might be targeting only one platform, some of them might not work at all.

4.14 3D support

It is currently not easily possible to display 3D object within NativeScript. There no JavaScript wrapper for OpenGL but there is an opened issue about it on GitHub [23]. Plain OpenGL support would still require a lot of work to display a 3D object. There is a popular JavaScript library called three.js which makes it very easy to display 3D objects but it would not be possible to use this library in NativeScript as it is dependent on the DOM. Also, it renders all graphics into HTML 5 canvas element which is also not supported in NativeScript [24].

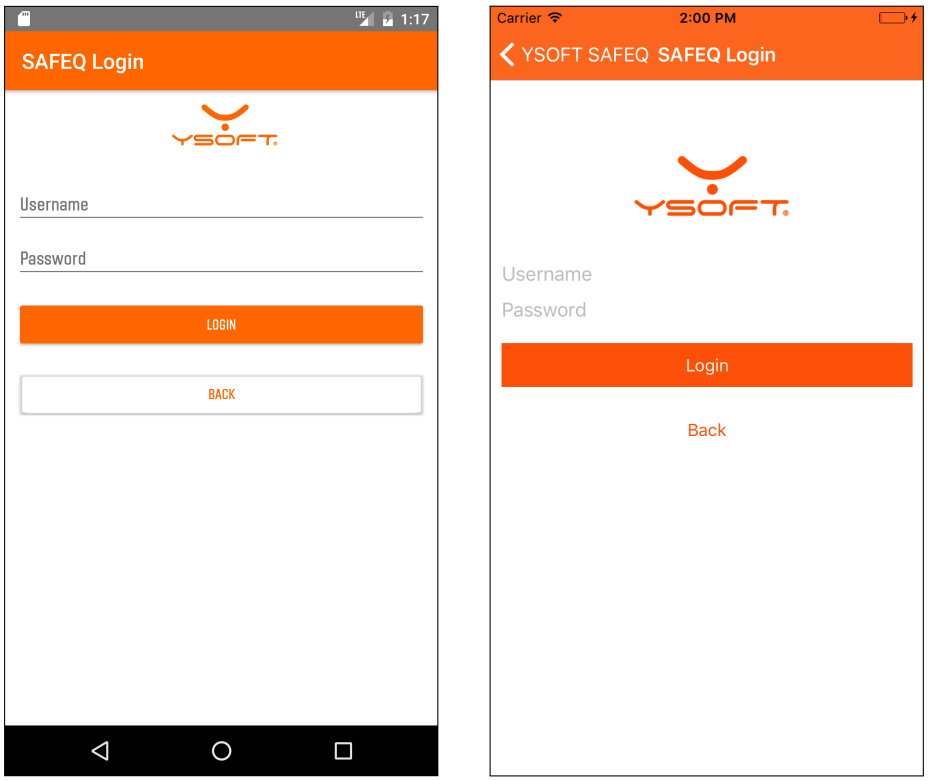

Figure 4.5: The login screen of the application created in NativeScript on Android (left) and iOS (right).

4.15 Created application

Here we show two example pages of the created application in NativeScript. There are always two versions of the image – one for Android and one for iOS. There is a login screen displayed in figure 4.5. Also, we have a job list screen displayed on figure 4.6.

During the development of the final app, we encountered several issues. There was the problem with enforcing the usage of our custom font as described in 4.9 and 4.6. There is that problem with layout recalculation as described in 4.8. We also had to deal with the problems of Angular 2 that are described in 4.5.1.

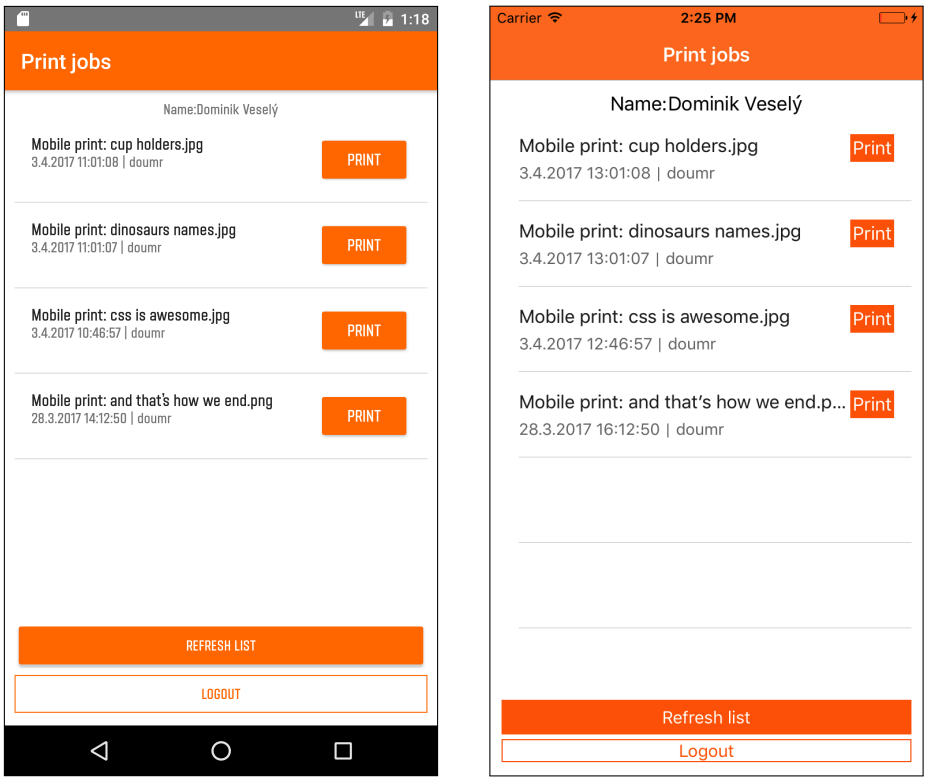

Figure 4.6: The job list screen of the application created in NativeScript on Android (left) and iOS (right).

4.15.1 Memory consumption

Measured memory consumption of the running app is displayed in table 4.3. The standard deviation is negligible so the results can be considered consistent.

4.16 Summary

The NativeScript framework seems to be a solid tool for developing a cross-platform mobile application. Although we encountered several issues during the development of our app which were summarized in 4.15, we never got actually completely stuck. The Telerik company maintains this project since its beginning in 2013 and it just released

Consumed memory type	$\overline{X}$	σ
Total	39,96	0,01
Allocated	24,10	0,06
Free	15,86	0,06

Table 4.3: The recorded memory consumption of our NativeScript app.

version 3 of the NativeScript which promises mainly the performance upgrade. Also, NativeScript has a well-worked tutorial of creating an actual mobile app during which the developer is gradually acquainted with all the basic essentials of the framework.

Pros

- General advantages of the Angular 2 framework.
- Usage of TypeScript with all its features.
- Large number of layout components which help to position the elements in many ways across the screen.
- Good learning curve of the NativeScript an Angular 2 framework.
- Direct access to the arbitrary native API (as described in 4.10).
- Strong emphasis on both target platforms. Android and iOS are considered equally important in NativeScript.
- Good support for the development on all platforms by using the Visual Studio Code available for Windows, Mac, and Linux.

Cons

- Limited options of code synchronization of the changed code with running app[10].

10. There is available only the option that automatically transfers the changed files. When we need to make a change that is involving multiple files the first updated file gets immediately transferred and the app often crushes due to the code inconsistency.

- A lot of errors have insufficient error messages or stack traces.
- There are still quite a lot of silent errors.
- Some of the 3rd party libraries are not so well integrated with NativeScript[11].
- API documentation often misses some deeper description of properties[12].

11. For example, the barcode scanner when invoked it jumps out of the NativeScript app into a completely separate screen and when it finishes it gets back to the NativeScript app.
12. Developers are mostly provided only with an example usage of some component.

5 React Native

5.1 About

React Native is a cross-platform framework for building native mobile apps using JavaScript and React. This framework is backed and maintained by Facebook [8]. The beginning of this framework dates back to the January of 2015. As a second level framework is used React which will be described in 5.3.

The versions of the framework and other packages used to create the final app and for the evaluation of the capabilities are displayed in table 5.1.

5.2 Target platform support

With React Native we can build native apps for iOS and Android with incoming official Windows support.

5.2.1 Android

Originally it was not possible to target Android platform with React Native. The support for the Android platform was added in September of 2015. It is possible to develop and produce an Android app on any desktop platform (Windows, Linux, Mac). To start working with React Native we need to install React Native command line interface [10] and Android Studio [11].

Package	Version	Description
react-native-cli	2.0.1	the command line interface for management of the React Native project
react-native	0.42.0	basic features provided by the framework
react	15.4.2	standard React framework

Table 5.1: The versions of npm packages used in our React Native app.

Figure 5.1: Part of React Native Getting Started guide showing that iOS is not supported on non-Mac devices [26].

Created app requires Android API level 16[1] to run [25].

5.2.2 iOS

Same as with NativeScript it is possible to develop an iOS app on Mac only as it requires to have Xcode installed which is available only for Mac [26]. Part of the React Native's documentation showing the inability to develop iOS apps on non-Mac devices is shown in figure 5.1.

Created app requires iOS8 or higher to run [25].

5.2.3 Windows

The incoming Windows support is currently at alpha stage. There is a separated repository on GitHub supporting React Native for Windows that is developed by Microsoft [27] but it is not officially supported yet.

5.3 React

React is JavaScript framework for building user interfaces. It is created and developed by Facebook. It was originally developed for the web but it is capable of being used in React Native to support the building of mobile apps as well.

1. Android API level 16 matches the Android version 4.1.

It solves some basic tasks of a front-end development. Mainly it splits the complex pages into components. Basically everything in React is a component whether it is used as a component representing a whole page or a component containing just a single line of text. This way the whole app is a tree of components. This works well together with another tactic React uses which is one-way data flow. This means that whenever there is a change of the data in some part of the component tree the update is propagated down to the subcomponents of where the change occurred. This way we can achieve higher performance and more control over the behavior of our app. While we talk about data React also makes a big difference between the data that has been provided to the component by its parent component (those are called props and the component itself cannot modify these props) and the internal data of the component determining its state. Another feature React is famous for is JSX. It is a syntax extension to JavaScript and it combines the HTML markup with the JavaScript language. This way we create the UI markup of our app inside the JavaScript code while the usage is very simple and straight forward. Therefore we don't have to have a separate files for creating the markup of the UI and for the code behind. The usage of JSX can be seen in the sample codes in later sections of this chapter (eg. in code 5.2).

React propagates a different style of writing and composing application logic so it is important to follow its pattern a build the app „React vise" [28]. For example, it is very important, where we keep the state of our app, things that we keep in the state or where we place some business logic (like logging a user in/out).

5.4 Styling

In React Native we style elements by applying styles written as JavaScript objects. Styles are being applied per element. Style names are mostly similar to CSS, they are just written in camel-case [29]. This framework is a bit more strict with styling as it claims the pure CSS to not be the cleanest solution in certain cases [30].

This way if we needed to style all buttons of our app the same way we could apply the same set of rules on each occurrence of a button in our app. But the preferred way is to create a new component containing

only the button styled to our needs and then using this component across our app [30]. This goes in hand with React's approach for grouping things into components. Therefore styles for the button should lie next to the definition of the button and not be together with other styles.

```
StyleSheet.create({
    container: {
        margin: 10
    },
    toolbar: {
        backgroundColor: '#a9a9a9',
        height: 56,
    },
    logo: {
        alignSelf: 'center'
    },
    centering: {
        alignItems: 'center',
        justifyContent: 'center',
        padding: 8
    },
});
```

Listing 5.1: Styling example

5.5 Animations

Animations in React Native are supported through so-called „Animation API". We can animate component properties like opacity, translation, scale, rotation. It comes with three predefined animation effects:

- **spring** – effect of bounciness at the end
- **decay** – animation starts with initial velocity and the speed decays over time
- **timing** – animation over given period of time with *easing* function (linear, quadratic, etc.)

This API is very versatile and can be used many ways. It also states to be very performant and highly optimized [31]. We provide an

example of the definition of an animation in code A.1 in the appendix section. It works the way that we supply some variable – single number (in the code on line 4) or vector of length 2 – that is later animated in a way that the API smoothly changes the value of given variable (in the code on line 13). This way we can assign this „animated value“ to some component's property (size, opacity, position) which is then appropriately animated (in the code on line 20).

As we can see, to achieve an animation we need to write quite a lot of code. On the other hand, this approach is highly customizable and offers an opportunity for a variety of animation effects.

5.6 Navigation

It is a bit more complicated with navigation in React Native. This is one of the places where there are bigger relics of the iOS-only platform support and they recommend to use iOS specific NavigatorIOS when building an iOS-only app [32].

There is a common navigation API for both Android and iOS called Navigator but its usability is very inconvenient. It is fairly simple but heavy to use and also pretty confused because it works on imperative rather than declarative approach. React Native knows about that and they are trying to build better common navigation supporting both platforms while preserving the ease of use. There was an experiment with the NavigationExperimental API which brought new ideas but is currently deprecated [33].

An example of the Navigator usage is in code A.2 in the appendix section. It is the most basic example and it is already pretty complicated. As we see on the lines 6-15 we need to use the `switch` to decide which screen to even show. Not to mention adding the functionality of transitions between these pages, it would require a lot of additional boilerplate code.

Nevertheless, the community took part and created several libraries for the navigation. Namely, there is react-native-router-flux [34] which is the most popular one and which we used in our React Native app. From this point, we will focus on this third-party created navigator.

We start with the definition of routes as the scenes[2] we want to navigate to. It is built with JSX (which was described in 5.3) just as any other React component. For each scene we must specify the `key` prop that is the text string by which we identify this scene, `component` prop which is a React component to be displayed at this route and then optionally several more props like `title` which defines the text displayed at navigation bar on top of the screen, type of the animation when navigating to that particular scene, etc. An example of routes definition is displayed in code 5.2.

```
export default class App extends Component {
    render() {
        return (
            <Router>
                <Scene key="modal" component={Modal} >
                    <Scene key="root">
                        <Scene key="home" hideNavBar={
                            true} direction="vertical">
                            <Scene key="intro" component
                                ={Intro} title="Y Soft" />
                            <Scene key="scan" component={
                                BarcodeScanner} animation=
                                "fade" />
                            <Scene key="login" component
                                ={Login} title="Login" />
                        </Scene>
                        <Scene key="logged">
                            <Scene key="dashboard"
                                component={Dashboard}
                                title="Dashboard" />
                        </Scene>
                    </Scene>
                </Scene>
            </Router>
        );
    }
}
```

Listing 5.2: Definition of scenes using third-party react-native-router-flux navigator

Then we can navigate simply by calling functions provided by the library. The library will create navigation functions named after pro-

2. Scenes are the various pages we can navigate to in our app.

vided scene keys. This makes it easy to hook this navigation functions to the callback functions (eg. to `onPress` callback of the button). Or we can call these functions anywhere in our code and also supply additional parameters which will be props that will be passed to the React component we navigate to. Examples of both mentioned navigation types are displayed in code 5.3.

```
1 // Navigation anywhere in the code
2 function navigateToIntro() {
3   Actions.intro(/* additional props... */);
4 }
5
6 // Navigation by hooking to the press callback
7 <YButton onPress={Actions.scan}>Go scan barcode</YButton>
```

Listing 5.3: Using navigation with third-party react-native-router-flux navigator

In comparison with the built in Navigator, this approach is much much clearer and easier to use.

5.7 Custom fonts

To add a custom font in React Native we have to do different steps for adding the font to the Android project and iOS project. The steps required to add the font for both target platforms will be described in its separate sections.

Applying the font is the same as applying any other styling to the components through `{fontFamily: 'YsoftRegular'}`.

5.7.1 Android

For Android, we need to put the font files (eg. in the otf format) within the android folder of our project. More precisely we need to put it to the *android/app/src/main/assets/fonts/* folder.

5.7.2 iOS

For iOS we need to add the font through the Xcode IDE. The steps required to add the font are following:

- import the font files to the project
- include the font as a Target Membership
- select the font files to be bundled with our app
- include all added fonts to the Info.plist file of the Xcode project

5.8 Error handling and debugging

All errors and warnings in React Native app are shown inside the running app. This is a good way to do it as it does not require developer to seek for errors and warnings on various places.

When an error or some non-standard action occurs within our app, developer is provided with the error or warning screen depending on the type of the problem. Error is show as a full screen report with the error message and stack trace. React Native even provides a way to get quickly to the problematic code by clicking on particular line in the stack trace which opens up development IDE with the file and line where the error occurred. An example of such error screens is displayed on figure 5.2.

Warning in React Native is an event developer should know about but is not necessary to be fixed for the app to run. Warnings are initially displayed collapsed so that they don't take up the whole screen. When the developer is interested in this warning it can by expanded by simply taping on it. Example of collapsed and expanded warning is displayed on figure 5.3.

As already mentioned in NativeScript's error handling section 4.11 the content of the error message and stack trace is crucial for the developer to be able to fix the issue. In React Native we experienced both the good and the bad types of this error messages. An example of bad error message and stack trace is in the right image of figure 5.2. An example of a good error message is in the left image of figure 5.2. Luckily the good error messages dominated the bad ones during our development.

To prevent errors React Native provides Flow [35]. It is an open source static type checker from Facebook. It is quite similar to TypeScript 4.4 but it does not integrate so well with Visual Studio Code.

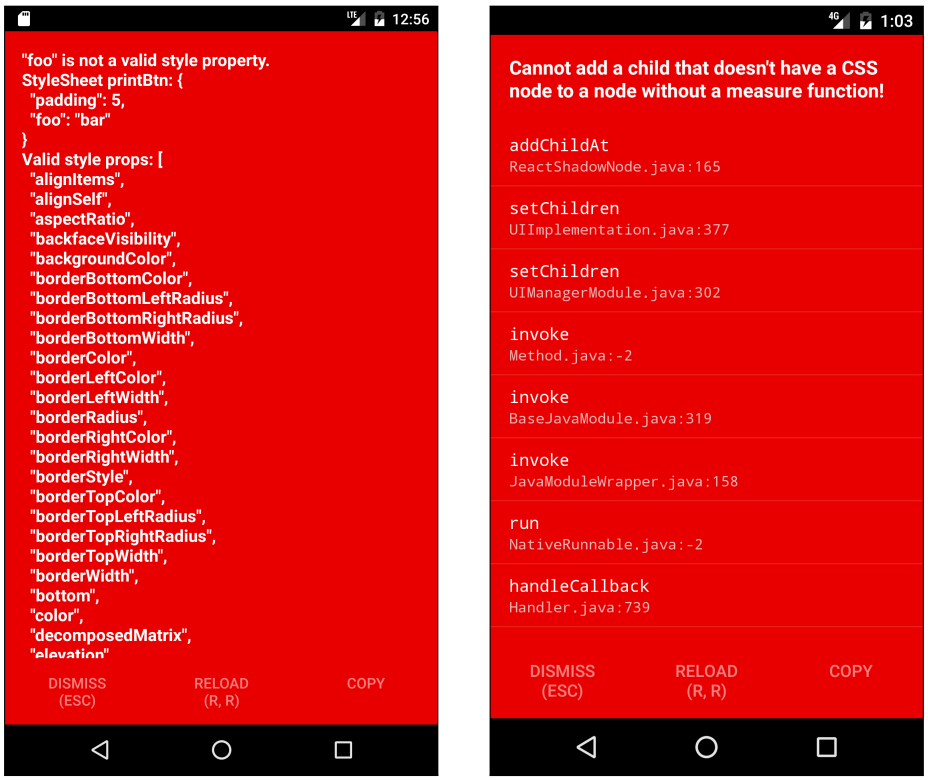

Figure 5.2: Error screens in React Native app. Error with good message is on the left. Error with bad message is on the right.

To access the JavaScript dumps we can connect to the running React Native app with one simple command. This command is `react-native log-android` for Android and `react-native log-ios` for iOS. This way we can access the console dumps from the app.

The more powerful way to log the console dumps and to debug the app is to connect the running app to the web browser supporting Chrome Developer Tools. We need to enable Remote JavaScript debugging in our React Native app and then connect to the running app from the web browser. Then we can see the console log of the app and we can use all the standard debugging features like placing breakpoints and stepping through app instruction-by-instruction, viewing the current values of variables, etc.

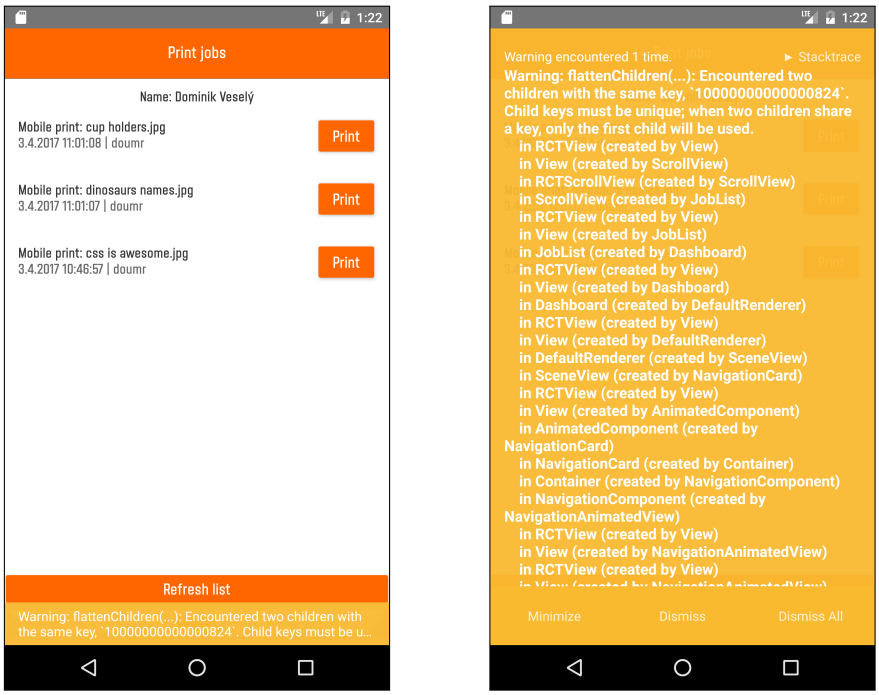

Figure 5.3: Warning message in React Native app. Collapsed warning message is on the left. Expanded warning message is on the right.

5.9 Build times

The measured times are shown in table 5.2. Again we can see that the standard deviation is very small so all build times are consistent. It is also important to note that in React Native there is actually almost no difference between livesync of the code and livesync of the view. As we can see the measured times are practically identical.

5.10 Community

Here we will mostly present the gathered data. Their meaning was described in the introduction in section 1.1.

GitHub statistics:

- Stars – 44198

Build type	$\overline{X}$	σ
Full compilation	01:19,77	00:01,07
Successive compilation	00:24,15	00:00,95
Livesync view	00:00,96	00:00,10
Livesync code	00:01,02	00:00,06

Table 5.2: The average time and standard deviation of compilation and synchronization speed in React Native.

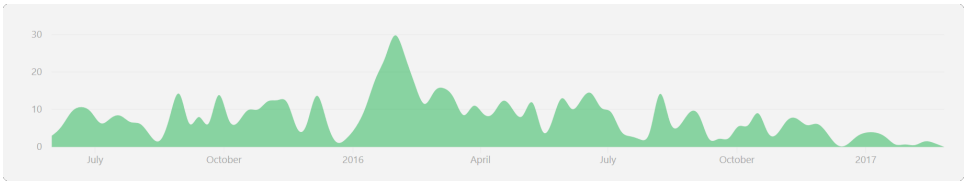

Figure 5.4: Graph showing contribution commits evolution in React Native GitHub repository [36].

- Contributors – 1211
- Issues – 998 opened (6672 closed, solved percentage 86.99%)

On Stack Overflow there are 11357 questions with the **react-native** tag.

In general React Native could be considered superior to any other mobile framework with these statistics. For example, we can compare it to the very popular WebView based framework Ionic which has around 28500 GitHub stars.

The evolution of contribution commits and commits in the last year are displayed in figures 5.5 and 5.4.

Figure 5.5: Graph showing commits evolution in the last year in React Native GitHub repository [37].

5.10.1 3rd party libraries

React Native has its own catalog of open source libraries [38]. It is hosted on the web `https://js.coach/`. There are currently 1902 plugins. Very important about this catalog is that there is a rating and a number of installations per month. With this informations we can easily find a promising library for us to use.

On the npm portal, there are over 4900 3rd party libraries created for React Native [39]. As with the NativeScript, some of those libraries might not even work or it might by targeting only one platform.

5.11 3D support

Currently, there is no support for displaying 3D objects. But the situation at React Native is not so hopeless because there is a library called gl-react-native supporting OpenGL in React Native [40]. It supports a lot of complex effects over images and components but it does not support anything with 3D as they state in their root library gl-react: „gl-react primary goal is not to do 3D. The library currently focuses on stacking fragment shaders (that runs with a static vertex) and exposes these features in a simple API applying React paradigm.“ [41]. There was an opened issue for support for 3D but it was dropped as the whole library is currently being rewritten from scratch [42].

5.12 Created application

Here we show two example pages of the created application in React Native. There are always two versions of the image – one for Android and one for iOS. There is a login screen displayed in figure 5.6. Also we have a job list screen displayed in figure 5.7.

During the development of the final app, we encountered few issues. Mainly it was connected with the fact that the code of the React Native app was written in pure JavaScript where it's harder to keep track of all the invisible properties of objects as opposed to more strictly typed TypeScript with better code hinting. Also when building the app it sometimes randomly crashes during the full compilation or even during the livesync. This is mainly caused by the fact that the

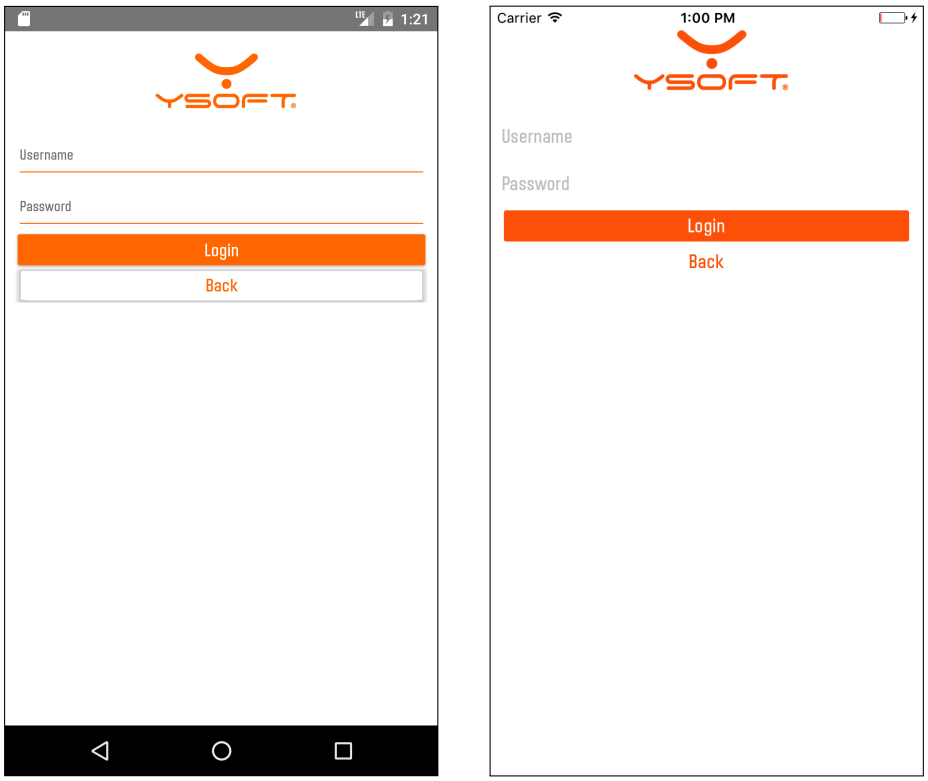

Figure 5.6: The login screen of the application created in React Native on Android (left) and iOS (right).

React Native is more optimized for the development on the Mac as it was originally available only on Mac.

5.12.1 Memory consumption

Measured memory consumption of the running app is displayed in table 5.3. The standard deviation is very small so the results can be considered consistent.

5.13 Summary

The React Native framework seems to be a solid tool for developing a cross-platform mobile application. At its core, it is a very lightweight

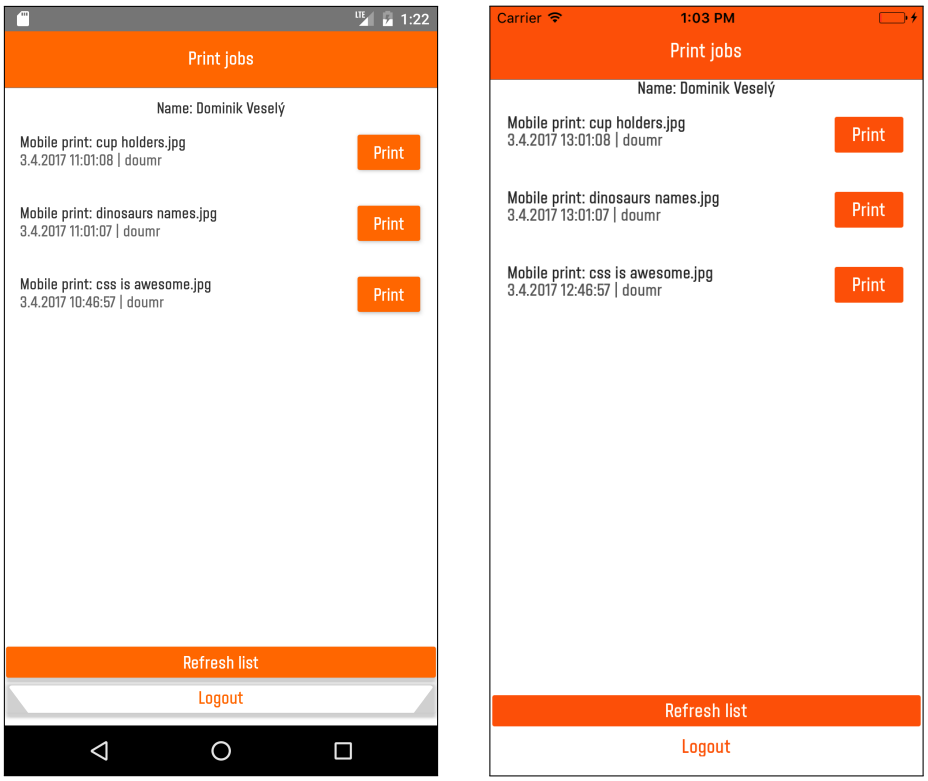

Figure 5.7: The job list screen of the application created in React Native on Android (left) and iOS (right).

framework that solves the basic stuff of creating the core native app, connecting the JavaScript with the native API, providing the error handling and providing the basic components for the development of a mobile app. For any JavaScript-related stuff, it leaves everything to the second level framework React. So it does not provide a huge number of features and additional solutions but the core stuff it does very well.

Since React Native was originally targeting only the iOS platform it was also designed to only work on Mac. This involves some problems with the compilation of the app on a non-Mac device as was described in 5.12. Another thing is that it still provides several components that are iOS only as there is no alternative solution with equivalent functionality provided for the Android platform. Even in the docu-

Consumed memory type	$\overline{X}$	σ
Total	19,962	0,25
Allocated	18,57	0,25
Free	1,392	0,35

Table 5.3: The recorded memory consumption of our React Native app.

mentation they often provide some easier solution when working on iOS only project which is not useful when creating a cross-platform app.

Another problem noticed was that until version 0.38 there was a problem with starting the packager[3] and they were encouraging developers to change some values in the file under *node_modules* folder [43] which is folder containing all project dependencies (eg. React, React Native) and there should not be done any changes under this folder. On the other hand in the newer versions of the framework, there is no longer such problem. This shows that the framework is doing a major progress and they are gradually solving all the issues.

Pros

- General advantages of the React framework.
- Huge community surrounding this framework.
- Developer menu inside the app for advanced setup during development process
- Very fast synchronization of the changed code with the app during development.
- Microsoft is working on the Windows support for React Native on its own.

3. Packager is a part of the React Native that handles the preprocessing and joining of the JavaScript code which is necessary before the code is transferred to a mobile device.

- The 3rd party libraries can be very nicely integrated into our React Native projects.
- Good error reporting to the developer.

Cons

- The Getting Started guide contains only descriptions of the separate techniques and components but does not put them into the context of creating some app.
- A limited amount of the core components provided by the React Native[4].
- It is harder to prevent errors in pure JavaScript code and the Flow language does not have a good support in some multi-platform IDE.
- There are relics from the time when React Native was iOS only framework[5].

4. For example, there is no support for the tab view component which is very useful in mobile apps.
5. There are still places in the documentation where they offer a lot of functionality that is available only on iOS.

6 Comparison

6.1 Common features

Because these two frameworks share the same main idea – that is to have one code base and support multiple target platforms – they also share a lot of common features. These common features were discussed in section 3. Even though they both support for example livesync of the code, the actual time it takes to sync the code differs and it will be covered in section 6.3.

6.2 Supported target platforms

Supported target platforms were discussed in section 4.2 for NativeScript and in section 5.2 for React Native.

In conclusion, both frameworks currently support Android and iOS platform. This can be considered sufficient as it covers about 99.3% of the mobile phone market share which was discussed in section 1. Both frameworks also have a plan of future Windows Universal platform support. NativeScript is stuck in this progress and postponed the expected Windows support to summer 2017 while in React Native the Windows platform support is actively worked on so the official release can be expected earlier than with NativeScript.

Even though both frameworks currently support the same set of the target platforms, the minimal required version of operating systems on these platforms differs for each framework. The minimal version of each framework and each platform is displayed in table 6.1. From the table, we can see that the minimal version for iOS is the same for both frameworks where for Android the React Native supports a wider range of Android versions. React Native is therefore able to run on Android 4.1 while NativeScript requires at least Android 4.2.

6.3 Build times

The build times in each framework were covered in 4.12 for NativeScript and in section 5.9 for React Native. The overview summary of

OS version	NativeScript	React Native
Android API version	17	8
iOS version	16	8

Table 6.1: Minimal versions of the target platform OS required by the apps created in NativeScript and React Native.

Build type	NativeScript	React Native	Difference
Full compilation	02:17,03	01:19,77	71.77%
Successive compilation	00:21,27	00:24,15	−11.93%
Livesync view	00:04,90	00:00,96	410.21%
Livesync code	00:11,58	00:01,02	1031.05%

Table 6.2: The comparison of build times in NativeScript and React Native.

build times is displayed in table 6.2 where we can see the average times for NativeScript and React Native and the percentage difference between NativeScript time and React Native time for each build type. From the table, we can see that React Native is much faster in most cases except for the successive compilation where it lacks behind NativeScript by about 12%. But the most important part is the livesync where the React Native is faster by 410% in the sync of the view and by 1000% in the sync of the code. Since the development of the front-end is an iterative process where the developer makes a minor change, waits for the change to be propagated to the app and then repeats this process, it will be much faster in React Native as the developer has to wait shorter time before the change gets applied.

6.4 Styling

Styling in each framework was covered in section 4.6 for NativeScript and in section 5.4 for React Native.

The styling in both frameworks is essentially similar in a sense that the rules for the styling are about the same as in pure CSS.

NativeScript offers true CSS styling as we actually style the app in CSS which brings all the useful features of this approach covering style inheritance, usage of CSS preprocessors like SASS, etc. We can also define animations right in the CSS.

React Native, on the other hand, has more of a JavaScript based approach to the styling which is more strict. One of the good approaches React Native does is to report a misuse of styling to the developer when trying to apply an unknown rule or invalid rule value while NativeScript quietly ignores such badly composed styles.

6.5 Animations

Animations in each framework were covered in section 4.7 for NativeScript and in section 5.5 for React Native.

In NativeScript we have two ways of defining animation (declarative and imperative) whereas in React Native we have only the imperative one and it is still less convenient than the NativeScript's imperative approach. Also, there is no support for animating a fluid color change in React Native. On the other hand, in React Native we animate the „value of a variable" which we then apply to some component's property (eg. opacity). This means we can use this animated variable for more use-cases than just animating some visible component's property. Also in React Native, we have a lot more options to do, like compose two animated values and we can subscribe to the update of the animated value.

Overall the definition of animations is much easier in NativeScript and it provides additional animation of background color. React Native offers few extra options to work with as we can create a wider variety of animations, but these are useful only in some cases.

The performance of animations in both frameworks works seems to be smooth. Except for few exceptions that will be discussed in section 6.13.

6.6 Navigation

Navigation in each framework was covered in section 4.8 for NativeScript and in section 5.6 for React Native.

We will skip the built-in navigator in React Native, and we will compare the Angular 2 router in NativeScript with the third-party router for React Native.

Generally considering usage and capabilities it is fairly similar in both frameworks. But with NativeScript this router is built into the Angular which creates more opportunities for optimizations. On the other hand, we came across few difficulties when working with the router in NativeScript and there were no problems with using the router in React Native.

6.7 Custom fonts

How to work with custom fonts it was described in section 4.9 for NativeScript and in section 5.7 for React Native.

It is possible to have custom fonts in both frameworks. They both support the standard versions of font files. The only difference is that in NativeScript it is more simple to include custom fonts to the project as in React Native we have to manually perform several steps in order to add the font to the iOS app.

6.8 3D support

The ability to display 3D objects was discussed in section 4.14 for NativeScript and in section 5.11 for React Native.

As we found out it is not possible to display any 3D object in NativeScript nor React Native. It would require the support of advanced OpenGL functionality provided as a cross-platform library. Currently, there is no such library for either framework. The only chance would be to create a custom component with native implementations in both Android and iOS. React Native is anyways closer to be able to display 3D objects as there is already that library gl-react-native that currently handles 2D support very nicely but does not yet support 3D (as described in section 5.11).

6.9 Barcode scanning

In our result apps, we needed the barcode scanning to be able to scan QR codes of the endpoints we want to connect to. We found cross-platform implementations of barcode scanners for both frameworks. Ease of implementation was also very reasonable in both frameworks.

The only difference was the way the barcode scanner is included into the framework. In Native Script the scanning screen pops up as a full screen when requested and the control is given back to our app when the scanner finishes scanning a code or when it's closed. In React Native we can put the scanning screen wherever we need. We can, for example, place it as only a part of the screen. In the end, we display the scanning screen as a full screen in both apps but the possibility to change this is only in React Native.

6.10 Community

The community of each framework has been discussed in section 4.13 for NativeScript and in section 5.10 for React Native.

Comparing the numbers of both frameworks the React Native seems to be four times more famous then NativeScript. Also, React Native has 17 times more contributors and 7 times more questions on Stack Overflow. These statistics speak for itself. The only positive outcome for NativeScript is the higher percentage of solved issues (90.77% over 86.99% for React Native).

If we compare the graphs of commits both are pretty stable with few fluctuations again with React Native going in higher numbers. Here the higher numbers in React Native does not necessarily mean better outcome as NativeScript is bit older framework meaning it is supposed to be a bit more mature and therefore not so huge development leads to less breaking changes.

As React Native has this huge community of developers around it, this brings new features and ideas of it. One nice example of this is the Redux library [44]. It is a library for managing the state of the app (mostly in React based projects) and it became widely popular. Anyway, it is important to note that the community form Angular quickly reacted to this popular library by creating a sort of similar

Consumption type	NativeScript	React Native	Difference
Total	39.96	19.96	95.93%
Allocated	24.10	18.57	29.84%

Table 6.3: The measured memory consumption of NativeScript app and React Native app and their difference.

library called ngrx/store [45] for managing the state of the apps in Angular based projects.

6.11 Memory consumption

In this section, we compare the consumption of the memory of the apps when they run. Also, we compare the sizes of the resulting apps.

The memory consumption of the running apps was shown in table 4.3 for Native Script and in table 5.3 for React Native. We will summarize and compare the most important values in table 6.3. From the table, we can see that the allocated memory of NativeScript app is by 30% higher than React Native app, but the total memory consumed is almost twice as big in NativeScript app compared to React Native app.

We compare the app sizes of the apk files for Android. As for iOS, it would be pretty difficult to obtain the installation files since it would require us to be registered as official iOS developers in some company. The size of the resulting app is 16.4MB for Native Script and 7.87MB for React Native. We can see that the React Native app is much smaller while having the same functionality. This means that the React Native framework is considerably more lightweight.

6.12 Security when working with certificates

Since security, in general, is a very large topic and it covers a lot of areas we focused on the secure communication. The most important aspect of this security for the Y Soft company was to elaborate the ways

of handling digital certificates. We encounter certificates in our apps only when working with HTTPS[1] secured network communication.

HTTPS provides secure communication over the Internet between client and server. This protects the data that we transmit from being compromised or changed and also it protects us from various attacks like man-in-the-middle. At the beginning of the communication, to ensure the client that the server we are connecting to is actually the server we want to communicate with, the server provides its certificate for its identification[2]. For the certificate to be valid it needs to be signed by some authority we trust, it must not be expired and it must not be revoked. This way we don't need to trust every single server, we only trust several certification authorities that issue certificates for those servers. This way those servers inherit our trust from the trust in those certification authorities.

The main goal was to elaborate how these frameworks handle communication through HTTPS when there is some problem with the certificate. The results were the same for both of the frameworks as it is not possible to communicate through HTTPS protocol when the target server's certificate is not trusted by the client. It is simply forbidden to communicate with any server that has the certificate signed by a certification authority we don't trust or if the certificate is invalid (eg. expired). This involves the inability to communicate with servers with self-signed certificates etc.

This is very well supported by the underlying platforms of Android and iOS which forbid such insecure communications as well. The main reason is that our frameworks use the underlying native implementations for the HTTPS communication. Based on this, our frameworks does not need to put any further constraints to the HTTPS communication as all of the security aspects are handled by the underlying native platforms.

1. HTTPS is the secured version of the HTTP for transferring of hypertext.
2. The server also has to prove that it owns the given certificate which prevents it from pretending to be someone else by providing the wrong certificate.

6.13 Gathered feedback from UX expert

We presented our apps to a UX expert at Y Soft company. Overall the feed was positive for both frameworks and there were no serious drawbacks detected.

The main information was that the NativeScript app looks more native on Android platform. This information is important mainly when we do not need to highly customize the components and when we want to keep them simple and as native as possible. Also, the barcode scanner in the NativeScript app was rated very positively as it has a good feedback for the user when the code is scanned. Performance wise both apps are very smooth only the React Native app has negligible lags in rotation animation. In NativeScript app the job list item rows (shown in figure 4.6) has a feedback when clicked on them even though there is no action on them. Also, there is a glitch in the NativeScript when using the slide transition during the navigation as the content briefly flashes during the slide animation. And also in NativeScript, the animations on text inputs are run a second time when the user starts typing.

6.14 Summary

We found out that we can build decent cross-platform mobile apps in both of our frameworks. The fact is we encountered a bit more problems while developing the app in NativeScript but the difference was not that severe.

Both of these frameworks have a lot in common as well as they are different in many ways. It is not easy to objectively say which one is better since some things are harder to achieve in NativeScript others are harder to do in React Native. It seems to be rather a trade-off between specific features each framework offers.

Also, since each of the frameworks uses a different second level framework, the choice could be a very well between the Angular 2 and React which play a big part in our frameworks. There are also a lot of differences in approaches these two frameworks take as the Angular 2 is fully equipped framework providing all sorts of functionality where

the React is rather more lightweight providing solutions for the core concepts.

In general, both of our apps had rather positive feedback about their usability and performance. There were few complaints about some minor things but nothing was problematic in a way that it would eliminate either of our frameworks as unusable.

Anyways, based on the success and the size of the community around each framework we can state that the React Native is far more popular. This makes React Native a safe choice as such big community will not disappear from day to day. Nevertheless, the size of the community around NativeScript is still pretty big compared to other mobile frameworks so even this choice would not go to waste.

7 Conclusion

We explored the two frameworks for the cross-platform mobile development that are NativeScript and React Native.

At the beginning of this thesis, we started with the definition of objectives to be measured and compared. We also created a specification for the final applications that were created in each of the frameworks.

Then we summarized the approaches of creating mobile applications while we focused mostly on the cross-platform development. We placed our frameworks into this perspective and introduce the general shortcomings they are trying to solve.

In the next chapter, we introduced all the features these frameworks have in common. We mostly focused on the architecture and the code compilation which are the two areas these frameworks are the most different from the majority of cross-platform mobile frameworks.

In the following two chapters we presented each of our frameworks with a detailed description of the objectives we were focusing on. We concluded several statements during these chapters but no comparison has been done here. We also briefly presented the applications that were created in each of the frameworks and we listed the problems we encountered during the development of the application in given framework.

The final chapter was all about comparing the two frameworks. We did the comparison based on the previously defined objectives. We tried to objectively compare the approaches these frameworks took in solving given objective. Mainly, we highlighted the main differences and we tried to state which framework is better at solving given objective. At the end of this chapter, we summarized what we found about these frameworks and we claimed that neither of the frameworks is generally superior one to another.

Bibliography

1. *IDC: Smartphone OS Market Share 2016, 2015* [online]. IDC, 2016 [visited on 2017-05-07]. Available from: http://www.idc.com/promo/smartphone-market-share/os.
2. *Integrated development environment* [online]. Wikipedia, 2017 [visited on 2017-05-07]. Available from: https://en.wikipedia.org/wiki/Integrated_development_environment.
3. *Visual Studio Code – Code Editing. Redefined* [online]. Microsoft, 2017 [visited on 2017-05-07]. Available from: https://code.visualstudio.com/.
4. *app Meaning in the Cambridge English Dictionary* [online]. Cambridge Dictionary, 2017 [visited on 2017-05-12]. Available from: http://dictionary.cambridge.org/dictionary/english/app.
5. *Representational state transfer* [online]. Wikipedia, 2017 [visited on 2017-05-12]. Available from: https://en.wikipedia.org/wiki/Representational_state_transfer.
6. *Application programming interface* [online]. Wikipedia, 2017 [visited on 2017-05-12]. Available from: https://en.wikipedia.org/wiki/Application_programming_interface.
7. *Graphical user interface* [online]. Wikipedia, 2017 [visited on 2017-05-12]. Available from: https://en.wikipedia.org/wiki/Graphical_user_interface.
8. *React Native | A framework for building native apps using React* [online]. Facebook, 2017 [visited on 2017-01-18]. Available from: https://facebook.github.io/react-native/.
9. *NativeScript* [online]. Telerik, 2017 [visited on 2017-01-17]. Available from: http://www.telerik.com/nativescript.
10. *Installation of CLI* [online]. NativeScript, 2017 [visited on 2017-01-18]. Available from: http://docs.nativescript.org/start/quick-setup#step-2-install-the-nativescript-cli.
11. *Install Android Studio | Android Studio* [online]. Android, 2017 [visited on 2017-01-18]. Available from: https://developer.android.com/studio/install.html.

12. *Requirements for Android* [online]. NativeScript, 2017 [visited on 2017-01-17]. Available from: https://docs.nativescript.org/runtimes/android/requirements.
13. *NativeScript Advanced Setup for Windows* [online]. NativeScript, 2017 [visited on 2017-01-17]. Available from: http://docs.nativescript.org/start/ns-setup-win.
14. *Requirements for iOS* [online]. NativeScript, 2017 [visited on 2017-01-17]. Available from: https://docs.nativescript.org/runtimes/ios/requirements.
15. *Windows support · Issue #254 · NativeScript/NativeScript* [online]. GitHub, 2017 [visited on 2017-03-07]. Available from: https://github.com/NativeScript/NativeScript/issues/254#issuecomment-272184207.
16. *nativescript-sass package* [online]. npm, 2017 [visited on 2017-03-07]. Available from: https://www.npmjs.com/package/nativescript-sass.
17. *Chapter 6—Accessing Native APIs* [online]. NativeScript, 2017 [visited on 2017-05-14]. Available from: https://docs.nativescript.org/angular/tutorial/ng-chapter-6.html.
18. *Contributors to NativeScript/NativeScript* [online]. GitHub, 2017 [visited on 2017-05-07]. Available from: https://github.com/NativeScript/NativeScript/graphs/contributors.
19. *Commit Activity · NativeScript/NativeScript* [online]. GitHub, 2017 [visited on 2017-05-07]. Available from: https://github.com/NativeScript/NativeScript/graphs/commit-activity.
20. *Official source for NativeScript plugins* [online]. Telerik, 2017 [visited on 2017-01-18]. Available from: http://plugins.telerik.com/nativescript.
21. *NativeScript UI Controls* [online]. Telerik, 2017 [visited on 2017-01-18]. Available from: http://www.telerik.com/nativescript-ui.
22. *nativescript packages* [online]. npm, 2017 [visited on 2017-01-17]. Available from: https://www.npmjs.com/search?q=nativescript.

23. *OpenGL support · Issue #289 · NativeScript/NativeScript* [online]. GitHub, 2017 [visited on 2017-01-17]. Available from: https://github.com/NativeScript/NativeScript/issues/289.

24. *NativeScript/nativescript-canvas: HTML5-like 2D and WebGL canvas implementation for NativeScript* [online]. GitHub, 2017 [visited on 2017-01-17]. Available from: https://github.com/NativeScript/nativescript-canvas.

25. *facebook/react-native: A framework for building native apps with React.* [online]. GitHub, 2017 [visited on 2017-01-18]. Available from: https://github.com/facebook/react-native#react-native---.

26. *Getting Started – React Native* [online]. Facebook, 2017 [visited on 2017-01-18]. Available from: https://facebook.github.io/react-native/docs/getting-started.html.

27. *Microsoft/react-native-windows: A framework for building native UWP and WPF apps with React.* [online]. GitHub, 2017 [visited on 2017-01-18]. Available from: https://github.com/ReactWindows/react-native-windows.

28. *Thinking in React – React* [online]. Facebook, 2017 [visited on 2017-01-18]. Available from: https://facebook.github.io/react/docs/thinking-in-react.html.

29. *Style – React Native* [online]. Facebook, 2017 [visited on 2017-01-22]. Available from: https://facebook.github.io/react-native/docs/style.html.

30. *Limited Style Inheritance – Text – React Native* [online]. Facebook, 2017 [visited on 2017-01-22]. Available from: https://facebook.github.io/react-native/docs/text.html#limited-style-inheritance.

31. *Using the native driver – Animations – React Native* [online]. Facebook, 2017 [visited on 2017-05-15]. Available from: https://facebook.github.io/react-native/docs/animations.html#using-the-native-driver.

32. *Navigation – React Native* [online]. Facebook, 2017 [visited on 2017-04-13]. Available from: http://facebook.github.io/react-native/releases/0.43/docs/navigation.html#navigation.

33. *NavigationExperimental – Navigation – React Native* [online]. Facebook, 2017 [visited on 2017-04-13]. Available from: http://facebook.github.io/react-native/releases/0.43/docs/navigation.html#navigationexperimental.

34. *aksonov/react-native-router-flux: React Native Router based on new React Native Navigation API* [online]. GitHub, 2017 [visited on 2017-04-13]. Available from: https://github.com/aksonov/react-native-router-flux.

35. *Flow: A Static Type Checker for JavaScript* [online]. Facebook, 2017 [visited on 2017-05-07]. Available from: https://flow.org/.

36. *Contributors to facebook/react-native* [online]. GitHub, 2017 [visited on 2017-05-07]. Available from: https://github.com/facebook/react-native/graphs/contributors.

37. *Commit Activity · facebook/react-native* [online]. GitHub, 2017 [visited on 2017-05-07]. Available from: https://github.com/facebook/react-native/graphs/commit-activity.

38. *React Native / JS.coach* [online]. js.coach, 2017 [visited on 2017-05-06]. Available from: https://js.coach/react-native.

39. *react native packages* [online]. npm, 2017 [visited on 2017-05-06]. Available from: https://www.npmjs.com/search?q=react+native.

40. *ProjectSeptemberInc/gl-react-native: OpenGL bindings for React Native to implement complex effects over images and components, in the descriptive VDOM paradigm* [online]. GitHub, 2017 [visited on 2017-01-18]. Available from: https://github.com/ProjectSeptemberInc/gl-react-native.

41. *Focus – ProjectSeptemberInc/gl-react: OpenGL / WebGL bindings for React to implement complex effects over images and content, in the descriptive VDOM paradigm* [online]. GitHub, 2017 [visited on 2017-01-18]. Available from: https://github.com/ProjectSeptemberInc/gl-react#focus.

42. *gre/gl-react: [gl-react v3 alpha] — a React library to write and compose WebGL shaders* [online]. GitHub, 2017 [visited on 2017-01-18]. Available from: https://github.com/gre/gl-react.

43. *Getting Started (for version 0.38) – React Native* [online]. Facebook, 2017 [visited on 2017-01-18]. Available from: `http://facebook.github.io/react-native/releases/0.38/docs/getting-started.html#testing-your-react-native-installation`.

44. *reactjs/redux: Predictable state container for JavaScript apps* [online]. GitHub, 2017 [visited on 2017-04-26]. Available from: `https://github.com/reactjs/redux/`.

45. *ngrx/store: RxJS powered state management for Angular applications, inspired by Redux* [online]. GitHub, 2017 [visited on 2017-04-26]. Available from: `https://github.com/ngrx/store`.

A An appendix

A.1 Additional code samples

```
export default class AnimationTest extends Component {
    constructor(props) {
        super(props);
        this.state = { animationValue: new Animated.Value(0) };
    }

    render() {
        return (
            <View>
                <YButton
                    onPress={() => {
                        this.state.animationValue.setValue(0);
                        Animated.timing(
                            this.state.animationValue,
                            {toValue: 1, duration: 1000}
                        ).start();
                    }}>
                    Start
                </YButton>
                <Animated.View style={{opacity: this.state.animationValue}}>
                    <YText>This will fade in</YText>
                </Animated.View>
            </View>
        );
    }
}
```

Listing A.1: An example of defining the animations in React Native.

```
export default class NavigationExample extends Component {
    render() {
        return (
            <Navigator
                initialRoute={{ path: 'home', index: 0 }}
                renderScene={(route, navigator) => {
                    switch(route.path) {
                        case 'home':
                            return <HomeScreen />;
                        case 'login':
                            return <LoginScreen />;
                        default:
                            return <InvalidScreen />;
                    }
                } }
                />
        );
    }
}
```

Listing A.2: An example of using the built in navigator in React Native.

A.2 Attachment

There is the attachment to this thesis in form of source codes of the two apps we created in NativeScript and React Native. We also included some additional files mainly containing the measured values we obtained while comparing those two frameworks.