End-To-End Testing for React Native Applications

Research Report

Sebastian Wilczek

Fontys University of Applied Sciences

Venlo, January 8, 2019

Summary

In recent mobile application development, the *React Native* framework has gathered a large following because of it's cross-platform deployment approach. Developing and moving applications to production is made rather easy with the framework, both for the aforementioned cross-platform compatibility as well as the large availability of open source components that are free to use and offered on the internet.
However, a part often overlooked in *React Native* development is testing. Many developers use manual testing on their applications by deploying them to testing devices or emulators while simply executing the actions a user would. This makes the development inefficient and the product unreliable due to the possibility of non-tested errors.
This research aims to answer the question how applications developed in *React Native* can be tested properly, more specifically how end-to-end testing can be applied successfully in such a project. To do so, a collection of potential end-to-end testing frameworks is considered under defined circumstances, such as cross-platform compatibility, ease of use and necessary change to production code.
The testing framework are tested in the context of a project working on the application *Connected.Football*. Conclusions are drawn both as to whether the testing frameworks are applicable to the project in question as well as how they can be used in a general *React Native* project that is tested from the very beginning.

Contents

List of Figures

List of Listings

List of Abbreviations

ADB	Android Debug Bridge
AVD	Android Virtual Device
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
LLC	Limited Liability Company
OS	Operating System
UI	User Interface

1 Introduction

This document is a research report on the topic of end-to-end testing of *React Native* applications. *React Native* is a framework developed by Facebook that enables developers to write one singular base of *JavaScript* development code for a mobile based application and to then deploy them to various mobile platforms, predominately *Android* and *iOS*. The framework itself is based on *ReactJS* and is handled similarly (Facebook Inc. 2018b).
The research was conducted as part of a development project involving the mobile application *Connected.Football*, which is being developed using *React Native*. During development, the question came up as to how such an application can be properly tested, especially how it can be assured that a user of the application is able to use it as intended.
The following chapters will first of all detail how the research was planned to be conducted. To illustrate that, it will be defined why the research was done in the first place, what questions were asked and answered, and what methods were to be made use of during the research. What follows are the results of the research. The results contain mostly a collection of different testing frameworks and their advantages when compared to others. Each framework is tested regarding the requirements necessary, as they are defined by the project work and the environment of the *Connected.Football* application. The result chapter also contains sample code written for some of the researched framework. Furthermore, it is explained how the frameworks could be used in the application development in question as well as in general *React Native* development.
The report ends with a conclusion, summarising the results of the research and giving advice as to how the testing frameworks are to be used in the project. It also contains general advice as to how to approach *React Native* end-to-end testing.

2 Research Definition

This chapter contains details as to how the research was conducted. This includes an explanation as to why the research was necessary in the first place as well as a set of questions that the research aims to answer. Furthermore, it is detailed what kind of research methods were made use of during the research.

2.1 Research Motivation

The research is part of a project on the *Connected.Football* mobile application. The application itself is developed using *React Native*, a frontend framework developed by Facebook (Facebook Inc. 2018b).
During development, the development team realised that they do a lot of manual testing, as in continuously going through the same repetitive steps to test certain use cases, for example entering an email and a password every time they want to test something that requires a user to be logged in. Eventually, the question came up as to how this form of testing can be done more efficiently, since the developers lose a huge amount of time testing repetitive interactions. Furthermore, due to the repetitive nature of such tests, the developers may overlook some flaws in the developed product, which could also be avoided through proper tests.

2.2 Research Questions

As a guideline of what should be researched, the following research questions were defined:

1. How can React Native applications be end-to-end tested?
2. What framework is the best to use for the React Native application in question?
   (a) Does it fit the development environment?
   (b) Is it easy to use, in terms of setup, test development and execution?
   (c) Is it compatible with the already existing product?
3. Is end-to-end testing a feasible thing to do in the project in question?

2.3 Research Methods

To answer the aforementioned questions and to conduct the research, a set of methods is to be made use during the research. First of all, the research conduction will include mostly literature research. It will be researched what frameworks are available and what the advantages and disadvantages to using them are. Literature to be used comes from the official documentations of the various frameworks as well as by publications of developers making use of specific frameworks.
Furthermore, interviews with the *React Native* developers of the project in question are to be conducted. The interview topics will be how testing is currently done in development, how it could be made more efficient, what their development environment is like when it comes to for instance operating systems and whether or not existing code should be changed or more work put into future development for the sake of testing.

2.4 Interview Questions

As mentioned in *Chapter 2.3*, an interview with *React Native* developers is part of the research methods applied. The following questions are to be asked to the developers and answers to said questions will be referenced in the coming chapter.

1. How do you currently test artefacts developed in *React Native*?
2. Does the testing include any repetitive or unnecessary elements?
3. Could the testing process be made more efficient? If yes, how?
4. What is included in your development environment? (OS, Mobile OS, emulation, physical devices)
5. Is it worth it to change existing code or to put extra work into future work on source code, if tests can be made more accessible and easier to use this way?

A transcript of all interviews can be found in the Appendix (see *Appendices A, B, C*).

3 Research Results

The following chapter describes the results that the research brought forth. Structurally, the chapter is made up of sub-chapters detailing a list of various frameworks.
For each framework, it is described how the framework can be used to test a *React Native* application. Furthermore, depending on the advantages and disadvantages the frameworks brings, it is defined whether or not the framework in question is fit to be used during the *Connected.Football* project.
As far as the project itself is concerned, the developers work on a variety of operating systems, including *Windows 10*, *Gentoo Linux* and *macOS 12 Mojave*, according to the interviews conducted with the developers. All developers furthermore deploy and test the application using an *Android* based device or emulator. The developers would like the tests to make their work more efficient, by removing unnecessary repetition and navigation. Having additional unit tests to the end-to-end tests would also be an advantage (see *Appendices A, B, C*).

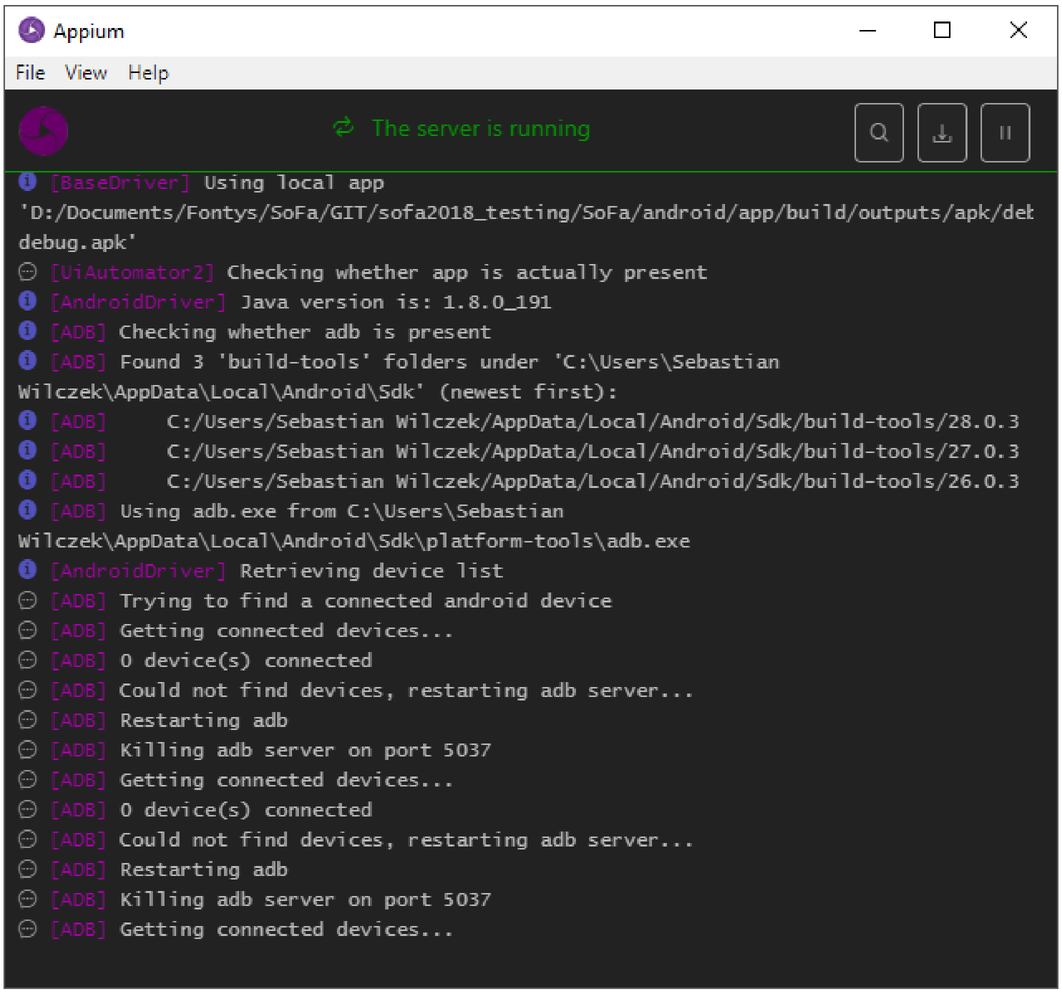
3.1 Jest

Jest is the testing framework that is automatically installed whenever a new *React Native* app is created. It is also developed by Facebook. Since version 14, *Jest* is able to create so-called snapshots. Snapshots are records of how a certain part of an application rendered the last time the tests ran successfully. This means that to test using *Jest*, it is defined what component is to be tested with certain properties. If this test has never run, the resulting component rendering tree is stored as a snapshot. If a snapshot already exists, the new rendering tree is compared against it, marking the differences to the snapshot. If the changes are intentional, the developer can store the result as a new snapshot. Ideally, these snapshots should always be committed and stored together with the sources. (Facebook Inc. 2018c)
The benefit to using *Jest* is that it is completely platform-independent. What is tested is the generated *React Native* rendering tree, rather that how the platform it is deployed on interprets it. Furthermore, the tests are not dependent on any emulation of an actual device: *"Because tests are run in a command line runner instead of a real browser or on a real phone, the test runner doesn't have to wait for builds, spawn browsers, load a page and drive the UI to get a component into the expected state which tends to be flaky and the test results become noisy."* (Negre 2016)
Furthermore, due to not being dependent on a platform, it is also rather simple to provide information that is not accessible through other means, for instance through mocking. State, properties and other values can be given to the tests instead of having to simulate them. This has the advantage than rather to having to replicating state logic in the tests, this logic can also be tested. (Gaare 2017)
However, despite the aforementioned advantages, *Jest* is most useful when testing single components or at most somewhat high-level components. This makes it most useful for unit and component testing. End-to-end testing is rather hard to do with a framework that can only compare old versions of components to new ones. Ideally, *Jest* would have to compare snapshots after executing a chain of commands that are processed by the application that is to be tested. This is also the reason why *Jest* is used in a similar way in some of the testing frameworks mentioned below.

3.2 Appium

Appium is an end-to-end testing framework which uses *Selenium* as a base. Since it was *".. released even before React.js, [making it] the number one in the industry"* (Bormanis 2018),

it gathered a large following both of *React Native* developers as well as developers using it in different environments. Similar to *Detox*, it makes use of a selection API which sends inputs to specified parts of the application, which is running in an emulator. *Appium* is also able to run tests on actual devices. If an *Android* device is connected to the development machine, *"providing there is only a single device connected, the test will pick up that device for execution."* (Verma 2017, p. 169)
To run *Appium*, a service is started which connects to the emulator. This service is responsible for propagating the inputs to the actual app. The service itself is actually an HTTP server, as seen in *Figure 1*, developed in *Node.js* which creates *WebDriver* sessions (Helppi n.d.). These sessions can be used by client applications which use certain drivers, depending on the programming language the tests are written in. To test the project in question, the tests are written in *JavaScript*, just like the actual application is. The driver that was made use of is *Webdriver.IO* which comes with its own selection API (JS.Foundation 2019).



```
[BaseDriver] Using local app
'D:/Documents/Fontys/SoFa/GIT/sofa2018_testing/SoFa/android/app/build/outputs/apk/deb
debug.apk'
[UiAutomator2] Checking whether app is actually present
[AndroidDriver] Java version is: 1.8.0_191
[ADB] Checking whether adb is present
[ADB] Found 3 'build-tools' folders under 'C:\Users\Sebastian
Wilczek\AppData\Local\Android\Sdk' (newest first):
[ADB]     C:/Users/Sebastian Wilczek/AppData/Local/Android/Sdk/build-tools/28.0.3
[ADB]     C:/Users/Sebastian Wilczek/AppData/Local/Android/Sdk/build-tools/27.0.3
[ADB]     C:/Users/Sebastian Wilczek/AppData/Local/Android/Sdk/build-tools/26.0.3
[ADB] Using adb.exe from C:\Users\Sebastian
Wilczek\AppData\Local\Android\Sdk\platform-tools\adb.exe
[AndroidDriver] Retrieving device list
[ADB] Trying to find a connected android device
[ADB] Getting connected devices...
[ADB] 0 device(s) connected
[ADB] Could not find devices, restarting adb server...
[ADB] Restarting adb
[ADB] Killing adb server on port 5037
[ADB] Getting connected devices...
[ADB] 0 device(s) connected
[ADB] Could not find devices, restarting adb server...
[ADB] Restarting adb
[ADB] Killing adb server on port 5037
[ADB] Getting connected devices...
```

Figure 1: *Appium* Server

Defining tests in *Appium* is simple. As seen in *Listing 1*, test cases can be described which in turn contain instructions on how to interact with the application, or rather how the *Appium* service should interact, including setup and tear down functions to start and end sessions with the service. The actual definition of interaction can make use of the driver's selection API. In the case of testing with *Appium*, this selection is limited to so-called *Accessibility Labels*. These labels are defined in the *React Native* source code and can be read in an *Appium* test using the ' ˜' character

```
describe("Basic Android interactions", function() {
  let client;

  beforeEach(function() {
    client = webdriverio.remote(opts);
    return client.init().pause(15000);
  });

  afterEach(function() {
    return client.end();
  });

  it("should click a tab that opens more functionality", async function() {
    return client
      .waitForExist('~More', 5000)
      .element('~More')
      .click()
      .waitForExist("~Privacy", 5000)
      .getText("~Privacy", function(result) {
        assert.equal(result.value, "Privacy");
      });
  });
});
```

Listing 1: *Appium WebdriverIO* Test Example

Using the labels has two major downsides. First of all, the development team would be forced to create such an identification for each element that would be involved in the testing process. As stated in the interviews, it would be preferable to test using as little change to actual developed work as necessary.
Furthermore, the *Accessibility Label* is also exposed to the user. In the accessibility options of *Android*, it is possible to enable a setting that enables screen reading applications to read out the functions of an application to vision-impaired users, as described by the aforementioned labels. (Google LLC 2018)
Unfortunately, the labels are also the reason why the testing framework can not be applied to the project in question. While it was possible to define and execute tests using *Appium*, including interacting with the application, the application also makes use of various open source *React Native* components which do not support the usage of *Accessibility Labels*. In particular, the module *react-native-navigation* caused problems this way, making it impossible for tests to ever leave the starting view of the application.

3.3 Espresso

Espresso is a library developed and maintained by Google designed for *Android* test automation (Lipps 2018). Since it is developed only for *Android*, it can not be used to test applications running on an *iOS* device or emulator.

The testing framework was developed having *Android* native applications in mind, with the possibility of testing *React Native* applications added later on. Given this change, *Espresso* was also made available to be used as a driver for *Appium*, similar to *Webdriver.IO* and *UIAutomator2*, which is also developed by Google.
An advantage of using *Espresso* is the way it manages timing while interacting with the application. *"Espresso requires no waiting method calls to ensure synchronization of the UI, [making it] clearly [one of] the fastest of the tools"* (Lämsä 2017).
However, in the same way as *Appium*, the framework is only able to access components through given identification, which is again defined through labels. This again makes it impossible to use *Espresso* in the given project. For detailed information about the problem posed by the *Accessibility Labels*, see *Chapter 3.2 Appium*.

3.4 Detox

Detox is another testing framework that simulates user interaction. Originally it was developed to run only on *macOS* and *iOS* platforms. Recently, the developers decided to make the framework available to *Android* developers as well, releasing in-development artefacts as they are continuing to make it fully functional. (Wix Engineering 2018)
Similar to *Espresso*, *Detox* tries to improve test stability using a gray box approach, by only running interactions once other interactions have been processed and the application is running in an idle state. *Figure 2* shows the check performed by *Detox* every few milliseconds. The framework will wait until it is asserted that the application's resources are running in an idle state or until a callback makes *Detox* run regardless.

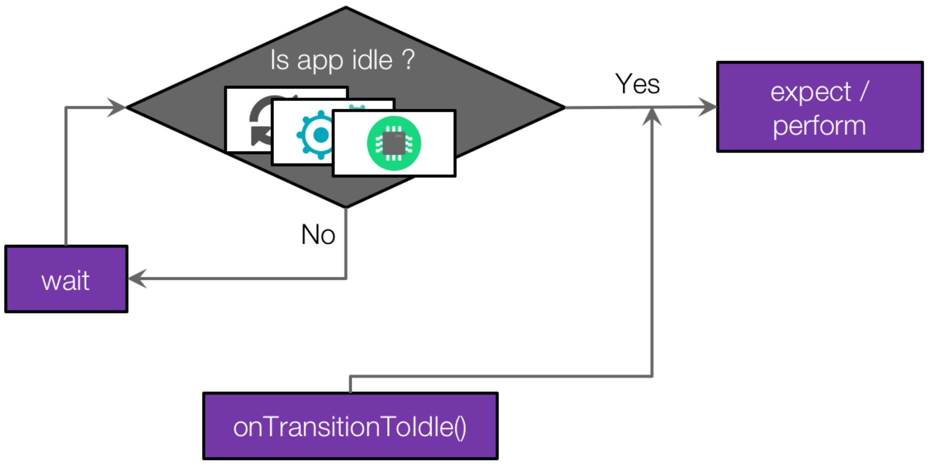


Figure 2: *Detox* Idle State Check (Mizrachi-Meidan 2017)

Detox is very simple to setup and use in a project. Once it has been installed as part of the modules of a *React Native* project, multiple configurations can be created to test in different environments, such as different devices running *Android* and *iOS*. *Listing 2* shows two such configurations.

```
"detox": {
  "configurations": {
    "ios.sim.debug": {
      "binaryPath":
          "ios/build/Build/Products/Debug-iphonesimulator/example.app",
      "build": "xcodebuild -project ios/example.xcodeproj -scheme
          example -configuration Debug -sdk iphonesimulator
          -derivedDataPath ios/build",
      "type": "ios.simulator",
      "name": "iPhone 7"
    },
    "android.emu.debug": {
      "binaryPath": "android/app/build/outputs/apk/debug/app-debug.apk",
      "build": "cd android && gradlew assembleDebug assembleAndroidTest
          -DtestBuildType=debug && cd ..",
      "type": "android.attached",
      "name": "emulator_5554"
    }
  },
  "test-runner": "jest"
}
```

Listing 2: *Detox* Configuration Example

These configurations include definitions on how to run the test builds as well as on what platform to run them. For instance, the second configuration run the test using ADB on an *Android* device with the name *emulator_5554*, which in this case is the running AVD emulator.
Just like most other frameworks, *Detox* has an API to select components. However, there is at the time of writing no documentation that details how this API can be used with a *React Native* application that is built for and deployed to an *Android* device. It could however be assumed that it might have to rely on *Accessibility labels*, just like *Appium*.
Furthermore, when trying to run an example test, just to see if it fails, *Detox* failed to run with multiple different errors depending on the chosen configuration. No documentation or reproduction of these errors could be found either. It is not known if the low support and early development stage of the *Android* functionality or the compatibility to the given project is to blame, however, the high amount of errors with no apparent solution makes *Detox* unsuitable for the project, despite being a promising product.

3.5 Cavy

Cavy is the newest available framework to test *React Native* applications at the time of writing. It tests applications by running an application in a simulated environment, in the same way *Appium* and *Detox* does, finding components using a selection API and interacting with them in a specified way.
The difference between *Cavy* and other similar tools is the way it finds and accesses compo-

nents. The most notable *".. difference is that Appium uses native hooks to access components (accessibility IDs), wheras [sic] Cavy uses React Native refs. This means that Cavy sits directly within your React Native environment [...] without much overhead."* (Pixie Labs 2018)
While it is a good idea to use only functionality that is already available in the *React Native* framework, this also means that every component that has to be tested has to be marked as such. To do so, *"Cavy (ab)uses React ref generating functions to give you the ability to refer to, and simulate actions upon, deeply nested components within your application."* (Pixie Labs 2018)
In other terms, that means that every component written in *React Native* needs to include a specific syntax when used. This syntax appears to be rather complicated, as shown in *Listing 3*, and can become hard to read when used in many components.

```
<TextInput
  ref={this.props.generateTestHook('Scene.TextInput')}
  onChangeText={...}
/>
```

Listing 3: *Cavy* Hook Example

There is a small workaround available to make the source code more readable. This workaround makes use of *Higher-Order Components*, which *"is an advanced technique in React for reusing component logic"* (Facebook Inc. 2018a). Making use of this technique, the rather complex looking syntax can be replaced by the word *testable* and the name of the components to be referenced with. Essentially this is the same as the previous syntax, just far more readable. (Parreira 2018)
In any case, the entire app must also be wrapped in a test related component, which receives the testing hooks. This another change to production code that is undesirable.
While the creation of test hooks looks rather complicated, the definition of tests is rather simple. The selection API provides, even if it is a bit simplistic, enough ways to select and interact with *React Native* components. An example test script can be found in *Listing 4*.

```
export default function(spec) {
  spec.describe('My feature', function() {
    spec.it('works', async function() {
      await spec.fillIn('Scene.TextInput', 'some string')
      await spec.press('Scene.button');
      await spec.exists('NextScene');
    });
  });
}
```

Listing 4: *Cavy* Test Example

While *Cavy* looks great in theory, aside from the massive change to the existing source code, it is not possible to make use of it in the project in question. The project makes heavy use of functional components, and since *"functional components cannot be assigned a ref since they don't have instances"* (Pixie Labs 2018), *Cavy* will not be able to access these components. Using *Recompose*, it might be possible to convert functional components to *React Native* classes, however that is more change to existing code and goes against the principle and goal of functional components.

4 Conclusion

As part of this research, many testing frameworks were considered whether they are easy and efficient to test with and if their usage is applicable in the project in question. According to the interviews conducted with *React Native* developers working on the project, this means that the framework should ideally be cross-platform compatible both for the mobile OS, as in *Android* and *iOS*, although *Android* is the preferred platform, as well as the development platform, meaning *Windows*, *Linux* and *mac OS*. Furthermore, the framework should require as little change to the source code as possible, both to existing code as well as how code should be written in the future of the project.
All examined platforms have their advantages and disadvantages. However, the research results in the conclusion that, at least for the project in question, none of the testing frameworks can be applied and used in an efficient way.
Jest offers great functionality when it comes to testing singular components. The snapshot feature makes it test at a very high speed without the need for any emulation, and it's platform independence makes it usable almost everywhere in the context of *React Native*. However, the framework was designed with a more component-based testing approach in mind, and it is best used that way. The simulation of user input and information processing can not be created in a test environment using only *Jest*.
Appium has the advantage of having a great interaction API and tests that are easily readable. It is also able to run cross-platform. However, the dependency on *Accessibility Labels* makes it incompatible with some *React Native* components, as it is the case with the already developed product. Furthermore, using the labels is more change to the source code and also indirectly perceivable by the user of the application.
Espresso does have a good approach with it's synchronisation. However, this feature is also available in other frameworks, for instance *Detox*, and it might not be the most up-to-date framework available. It's limitation to the Android platform also makes it less than ideal.
Detox combines the easy usability of *Appium* and the great synchronisation features of *Espresso*. It is also simple to set up, offering the possibility of defining many different testing configurations. However, the API is more limited than the one of *Appium* and, same as *Appium*, components are not recognised during the testing procedure. For now, the functionality is also limited when it comes to the *Android* platform.
Lastly, *Cavy* has a great approach when referencing components when one thinks about performance, since the references used by *Cavy* are only making use of the *React Native* framework, instead of native platform identification. However, just like *Appium* and *Detox*, this requires every component to be marked as a testable component during development, making the selection API very limited, with the added disadvantage of needing to wrap the entire application in a test hooking function. Furthermore, *Cavy* does not support functional components, another paradigm that is made heavily use of in the project in question.
For the reason reiterated above, no testing framework is completely applicable to the project. However, if one were to create a new *React Native* application, the following suggestions could be made:
Every major component should be unit or component tested making use of *Jest*'s snapshot feature. For end-to-end testing, developers should decide between *Appium* and *Detox*, depending on the needed complexity of the written tests, with *Appium* for more complex tests. If the required complexity level of tests is not as high, *Detox* can be used for the sake of an easy setup and configuration. In any case, it should be paid attention to the fact that the components that are made use of need to be able to be referenced by the chosen test framework, for example through *Accessibility Labels*. If it can be made sure that no functional components will ever be used as part of the application, *Cavy* may also be used as an alternative.

Overall, each testing framework does what it is intended to do, which is testing a *React Native* application. This research has returns some of the caveats and disadvantages of the tested frameworks. However, it also shows some of the positive aspects of each as well as potential reasons why and environments to use them in. One thing that can be taken away from this research is that testing is a part of software development that should be considered from the very beginning of a project, even when working with a very frontend focused framework such as *React Native*. Introducing tests late in a project results mostly in errors and compatibility issues, like it did in this project. If tests are written and executed properly from the start, both the product and the tests written for it can develop into software that works as intended.

References

Bormanis, J. Y. (2018). *Detox vs. Appium: automated UI tests in React Native.* Available at: `https://medium.com/@borman/detox-vs-appium-ui-tests-in-react-native-2d07bf1e244f/` [Accessed 05 Jan. 2019].

Facebook Inc. (2018a). *Higher-Order Components.* Available at: `https://reactjs.org/docs/higher-order-components.html` [Accessed 06 Jan. 2019].

– (2018b). *React Native.* Available at: `https://facebook.github.io/react-native/` [Accessed 26 Dec. 2018].

– (2018c). *Testing React Native Apps.* Available at: `https://jestjs.io/docs/en/tutorial-react-native` [Accessed 06 Jan. 2019].

Gaare, J. (2017). *Learning to test React Native with Jest — part 1.* Available at: `https://medium.com/react-native-training/learning-to-test-react-native-with-jest-part-1-f782c4e30101` [Accessed 06 Jan. 2019].

Google LLC (2018). *Make apps more accessible — Android Developers.* Available at: `https://developer.android.com/guide/topics/ui/accessibility/apps` [Accessed 05 Jan. 2019].

Helppi, V.-V. (n.d.). *The Basics of Mobile Web Testing on Real Devices Using Selenium.* Available at: `https://bitbar.com/the-basics-of-mobile-web-testing-using-appium-selenium/` [Accessed 05 Jan. 2019].

JS.Foundation (2019). *WebdriverIO API Docs.* Available at: `https://webdriver.io/docs/api.html` [Accessed 05 Jan. 2019].

Lämsä, T. (2017). "Comparison of GUI testing tools for Android applications". MA thesis. University of Oulu.

Lipps, J. (2018). *Using Espresso With Appium.* Available at: `https://appiumpro.com/editions/18` [Accessed 06 Jan. 2019].

Mizrachi-Meidan, R. (2017). *Detox: Gray Box End to End Testing Framework for Mobile Apps.* Available at: `https://hackernoon.com/detox-gray-box-end-to-end-testing-framework-for-mobile-apps-196ccd9564ce` [Accessed 06 Jan. 2019].

Negre, F. (2016). *Testing React Native with the *new* Jest — Part I.* Available at: `https://blog.callstack.io/unit-testing-react-native-with-the-new-jest-i-snapshots-come-into-play-68ba19b1b9fe` [Accessed 06 Jan. 2019].

Parreira, L. (2018). *React Native e2e testing with Cavy.* Available at: `https://medium.com/magnetis-backstage/react-native-e2e-testing-with-cavy-1f1d5be1d3be` [Accessed 06 Jan. 2019].

Pixie Labs (2018). *Cavy: An integration test framework for React Native.*
Available at: `https://github.com/pixielabs/cavy/` [Accessed 06 Jan. 2019].

Verma, N. (2017). *Mobile Test Automation with Appium.* Birmingham: Packt Publishing Ltd.

Wix Engineering (2018). *Android Support Status.*
Available at: `https://github.com/wix/Detox/blob/master/docs/More.AndroidSupportStatus.md` [Accessed 05 Jan. 2019].

Appendices

Appendix A: Interview Lucas Gehlen

The following is a transcript of the interview conducted with the *React Native* developer Lucas Gehlen. Formalities at the beginning and the end of the interview that are not related to any relevant information have been omitted from the transcript.

Interviewer: How do you currently test artefacts developed in *React Native*?

L. Gehlen: I don't, really. All I do is manually enter actions and values into the application to test certain use cases.

Interviewer: Does the testing include any repetitive or unnecessary elements?

L. Gehlen: Yes. I often run through the same actions to test the same cases over and over again, wasting time.

Interviewer: Could the testing process be made more efficient? If yes, how?

L. Gehlen: Probably. I don't really know how though.

Interviewer: What is included in your development environment? (OS, Mobile OS, emulation, physical devices)

L. Gehlen: I use *macOS 10 Sierra* on my development machine, as well as *Android 8 on an Android Emulator*. I do not use any physical devices.

Interviewer: Is it worth it to change existing code or to put extra work into future work on source code, if tests can be made more accessible and easier to use this way?

L. Gehlen: No, it is not worth it and way too much effort.

Appendix B: Interview Marco Kull

The following is a transcript of the interview conducted with the *React Native* developer Marco Kull. Formalities at the beginning and the end of the interview that are not related to any relevant information have been omitted from the transcript.

Interviewer: How do you currently test artefacts developed in *React Native*?

M. Kull: I only test by hand, using the application like a normal user would, to see if it works as intended. Very tedious.

Interviewer: Does the testing include any repetitive or unnecessary elements?

M. Kull: Yes, I have to wait very long times for the application to be deployed to a device before it is ready. It takes forever to navigate to the things I want to test.

Interviewer: Could the testing process be made more efficient? If yes, how?

M. Kull: Yes. A more clear separation of functionality and user interface is needed, this way it would also be possible to do unit testing, which would be nice. It would be great if the navigation could be quicker by using tests.

Interviewer: What is included in your development environment? (OS, Mobile OS, emulation, physical devices)

M. Kull: I run *Gentoo Linux* and I have a physical *Nexus 6 P* device running *Android 8*.

Interviewer: Is it worth it to change existing code or to put extra work into future work on source code, if tests can be made more accessible and easier to use this way?

M. Kull: Yes. In a professional setting this should be done in a certain extent, as long as the process of doing so does not put unnecessary strain on the development team.

Appendix C: Interview Patrick Richter

The following is a transcript of the interview conducted with the *React Native* developer Patrick Richter. Formalities at the beginning and the end of the interview that are not related to any relevant information have been omitted from the transcript.

Interviewer: How do you currently test artefacts developed in *React Native*?

P. Richter: I don't test at all.

Interviewer: Does the testing include any repetitive or unnecessary elements?

P. Richter: Nope, since I do not test the application.

Interviewer: Could the testing process be made more efficient? If yes, how?

P. Richter: Yes, of course, there needs to be a lot more testing overall.

Interviewer: What is included in your development environment? (OS, Mobile OS, emulation, physical devices)

P. Richter: I have *macOS Mojave* and an *Android 8* device, which is a *Nexus 6 P*.

Interviewer: Is it worth it to change existing code or to put extra work into future work on source code, if tests can be made more accessible and easier to use this way?

P. Richter: Of course, that would be reasonable. Seems alright to me.