communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder

Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Informatik 9
The present work was submitted to Learning Technologies Research Group

Evaluierung plattformübergreifender App-Entwicklung mittels React Native
Evaluation of Cross-Platform App Development using React Native

Bachelorarbeit
Bachelor-Thesis

von / presented by

Jaworski, Tobias
310414

Univ.-Prof. Dr.-Ing. Ulrik Schroeder

Univ.-Prof. Dr. rer. nat. Horst Lichter

Aachen, March 21, 2018

Contents

Abstract

The mobile web is continuously growing in popularity. Yet, mobile apps are preferred to mobile websites. Traditional app development for native platforms requires big resource investments. Thus, solutions that promise cross-platform development are of particular interest to many organizations. This thesis takes a closer look at one of these solutions, namely *React Native* by Facebook. To evaluate the performance, an example app is developed both for the Android and iOS platforms, using both native technology as well React Native. Using a set of benchmarked metrics as well as considering the developer experience, we'll conclude whether React Native can be a viable solution for cross-platform app development.

Part I

Introduction

Chapter 1 Motivation

As seen in figure 1.1, the web is increasingly used by mobile devices - nowadays, smart phones and similar devices are used more commonly than desktop computers.

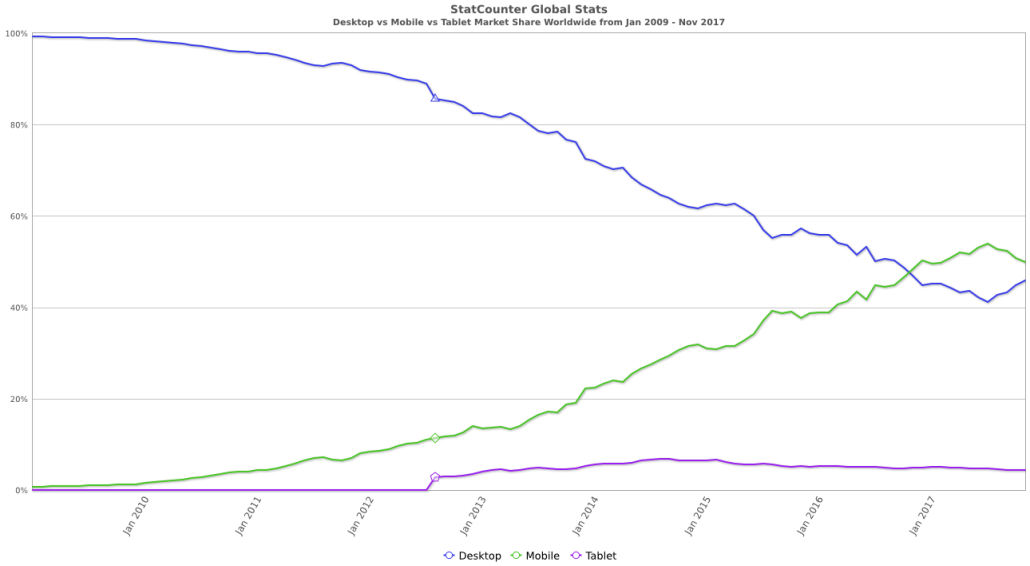

Figure 1.1: Web market share [Sta17b]

While many websites are designed in a *responsive* way - e.g., they can react to various screen sizes and display their content correctly - most users use dedicated mobile apps (see figure 1.2).

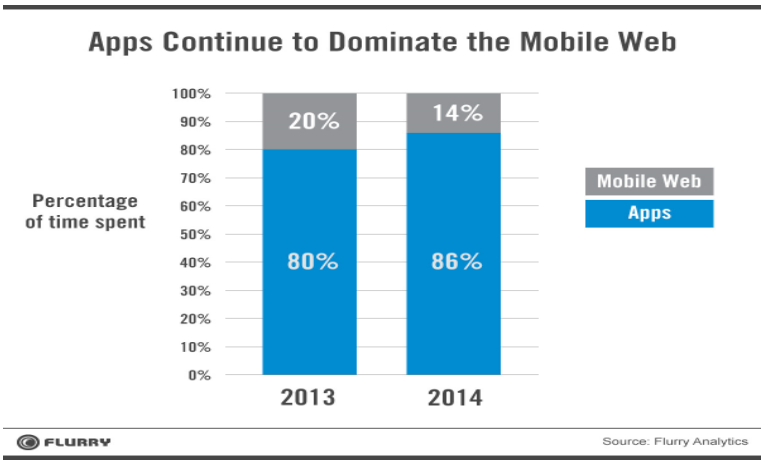

Figure 1.2: Mobile Web vs Apps [Sim14]

Despite all efforts to improve User Experience in mobile browsers, users seem to prefer native mobile apps - reasons for that could include a better perceived performance and a deeper integration into the mobile operating system. [CL11]
Development and support of an app for both the Android as well as the iOS platforms require a big investment of resources, which not every organization is ready to make. Not only does one have to consider the initial hiring of developers proficient in a platform and the initial development of the apps, but the cost of continued support with bug fixes and new features.
Thus for many organizations and developers, solutions that allow cross-platform development are interesting [Dal+13]: one code base, multiple platforms. Historically, there have been many attempts to deliver such a solution: From simple implementations like `WebView` which embeds browser-rendering into native Android apps to big enterprise-backed solutions like `Xamarin`, which enables writing C# for a multitude of platforms, there are many options to consider when developing a cross-platform application.

Chapter 2 Purpose

The aim of this thesis is to take a closer look at one potential solution to cross-platform mobile app development: *React Native*[1] by Facebook, which builds upon the popular React[2] JavaScript framework and promises development using platform independent JavaScript with near native performance [Facc].
We will investigate how viable React Native is as an approach to cross-platform app development, and whether it can be an alternative to developing native applications for both Android and iOS.

To do so, we will first take a look at the underlying technologies in part II and give a glimpse into app development on the React Native platform.
We will then establish the methodology to evaluating the viability of the React Native platform in part III. As a part of that, we will design an example application. This sample application will then be implemented in three different ways: For iOS using Swift[3], for Android using Kotlin[4] and finally for both platforms using React Native.
We will try to measure the apps' performance using various metrics, and also consider the user experience the author had during development.
Finally in part IV, we will conclude whether React Native can be a viable alternative to building separate native applications for Android and iOS and give a prospect to the future of React Native.

[1] `https://facebook.github.io/react-native`
[2] `https://reactjs.org`
[3] `https://developer.apple.com/swift`
[4] `https://kotlinlang.org`

Part II

BACKGROUND

Chapter 3 JavaScript

The sample application we are going to implement on the React Native platform will be exclusively written in JavaScript. In the following, we will give a glimpse into the language and how it has over time evolved from a simple scripting language to a powerful multi-paradigm programming language that is among the most used by developers worldwide [Sta17a].

3.1 Introduction

JavaScript as a language is "characterized as dynamic, weakly typed, prototype-based and multi-paradigm" [con18c]. Originally envisioned as a simple way to add dynamic elements to static HTML pages, Brendan Eich at Netscape completed the prototype for the language in just a week [Eic05]. Despite it's name might suggest, the language has little in common with Java, and the name JavaScript was merely chosen to benefit from the popularity of Java at that time [Eic05].
The dynamic and weakly typed nature of the language allows quick iterations and makes it rather easy to get started, but does not come without its drawbacks. Special care must be taken while writing code, as making wrong assumptions about the type of a value can have undesired consequences, as seen in code example 1.

```
var x = 1;
console.log(x + '1'); // result: 11
console.log(x - '1'); // result: 0
```

Code example 1: Multiple semantics for the '+' operator (string concatenation and integer addition) lead to unwanted behavior

As an effort to eliminate bugs caused by wrong typing, efforts exist to statically type check JavaScript. The most common solutions are TypeScript[1] by Microsoft, a superset of the language that compiles down to regular JavaScript; and Flow[2] by Facebook, an add-in to existing JavaScript code with the capability to dynamically infer types if not explicitly set.
Nowadays, the usage of JavaScript has grown far beyond adding dynamic elements to static HTML pages.
So called Single-Page-Applications (SPA) offer a desktop-like experience using modern web technology; entirely while running in the users browser. "We can think of an SPA as a fat client that is loaded from a web server" [MP13].

[1] https://www.typescriptlang.org
[2] https://flow.org

Furthermore, node.js[3] allows running JavaScript on the server. By some, it is considered "[the] perfect tool for developing fast, scalable network applications" [CKT15] and it is used by corporations like Netflix[4] and PayPal[5].
Recently, technologies like Electron[6] bundle a web browser and JavaScript to create desktop apps; and there are even attempts to write whole operating systems in JavaScript.[7]
Concluding the history of JavaScript, we can note that it is one of the most used languages by developers worldwide and continues to grow in popularity [Sta17a].
The focus of this thesis, namely React Native, is based on a concurrent JavaScript thread running in parallel to a platform specific native rendering thread. What follows is a brief history and the technical background of the technologies used.

3.2 ECMAScript and beyond

Since its first appearance in 1995, the requirements for JavaScript have shifted from simple scripts to full-fledged applications. As outlined in the previous section, the language initially not designed to accommodate needs beyond simple scripts, so feature additions seemed inevitable.
A challenge in adding new features to JavaScript (as run in the browser) arises from the fact that it relies on the client to support the language. While nowadays most people use recent versions of fully-featured browsers like Chrome or Firefox [Sta17b], there is a considerable amount of users still using legacy browser versions like Internet Explorer 11 or Safari 8. While newer browsers might get support for new language features, older browsers are for the most part stuck with the "basic" JavaScript version, now commonly known as ES5.
ES5 is an acronym for "ECMAScript 5", which is a standardization of scripting languages set forth by Ecma International, a "standards organization for information and communication systems" [con18a]. Ecma International are responsible for various standards, such as ECMA-404 (JSON), ISO 9660 (File System for optical disks) and ECMA-262 (ECMAScript, based on JavaScript).
ECMAScript is a living standard versioned by numbers. Almost all browsers in active use support ES5, and many support features from ES6 (also called ES2015). New versions of ECMAScript bring useful features such as support for lambda functions, promises, classes, iterators, generators, data structures such as `Map` and `Set`, meta-programming capabilities and much more [Sta17].
While browser vendors usually add support for these features after a while, there is no guarantee that every user of the site would have a recent enough version to use all ES6+ features. As a consequence, most JavaScript code in the web is served as the compatible ES5 version.
The feature additions to the language make it more attractive to developers, so there are efforts to bring ES6+ features to all browsers. The most prominent project is Babel[8], a JavaScript compiler that can translate the latest and experimental ES6+ features to commonly understood ES5. Babel currently is in its sixth major version and a very robust tool that seems to unify the best of two worlds: being compatible with nearly all browsers

[3] https://nodejs.org
[4] https://medium.com/the-node-js-collection/netflixandchill-how-netflix-scales-with-node-js-and-containers-cf63c0b92e57
[5] https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal
[6] https://electronjs.org
[7] https://node-os.com
[8] https://babeljs.io

while still using the latest features - so it is no surprise that more developers now use ES6 that ES5 [RSM].[9]

[9] For this reason, we are using ES6 syntax for all JavaScript code examples in this thesis as well as the actual React Native sample application.

Chapter 4 React

As React Native is at its core an extension of the JavaScript framework React [Ale15], it is essential to understand the technology behind it.
The following will give a brief history of its conception and an introduction to the technology behind the framework.

4.1 History

The first public information about React dates back to a talk of Facebook Software Engineers Jordan Walke and Tom Occhino in 2013 [JT13]. In it, they talk about the technical UI challenges Facebook faced internally, in particular to display the same piece of state in multiple places and keeping those views in sync.
This of course was not a new problem, and various JavaScript libraries such as AngularJS[1], Ember[2], Backbone.js[3] and many other solutions existed to help developing Single-Page-Applications in JavaScript. These popular libraries admittedly work well enough to power complex and commercially successful web apps such as WolframAlpha[4] (AngularJS), LinkedIn[5] (Ember) and AirBnb[6] (Backbone), so the need for yet another solution was not apparent to everyone. (Ben Alman, creator of the popular GruntJS build tool, put it this way: "Facebook: Rethink established best practices" [Alm13].)
What all these popular JavaScript frameworks have in common is that they follow the Model View Controller[7] (MVC) pattern in some way or the other. They prominently feature so called *Two-Way Data Bindings*: When properties in the model get updated, the view changes accordingly, and if the view changes due to user interaction (e.g. filling in a form), the underlying model changes as well. This pattern allows for quick iteration on developing user interfaces and remains popular to this day.
React shares a central concept with these frameworks: it abstracts away the need to update the DOM manually.

> "The Document Object Model (DOM) is an application programming interface for valid HTML and well-formed XML documents. [...] With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions." [Woo+04]

[1] `https://angular.io`
[2] `https://www.emberjs.com`
[3] `http://backbonejs.org`
[4] `https://www.wolframalpha.com`
[5] `https://www.linkedin.com`
[6] `https://www.airbnb.com`
[7] "The well-known Model/View/Controller (or MVC) design pattern is a useful way to architect interactive software systems. [...] The key idea is to separate user interfaces from the underlying data represented by the user interface." [LR01]

Manually updating the DOM with JavaScript can be error-prone and is very verbose (due to the need to traverse the whole DOM), so by sheer complexity of the DOM SPAs almost universally rely on a framework to do their updates. React however takes this concept a step further, and abstracts away the DOM in a so called *Virtual DOM*.
Instead of relying on the developer to insert appropriate bindings into the DOM, React stores a representation of the DOM in memory, and decides if and what changes to propagate to the actual DOM. All that is required by the developer is supplying React with a new state to display, and React will compute the actual changes from the old state. This step is called *reconciliation* and will be further explained in subsection 4.2.6.
In any case, this declarative[8] approach to developing JavaScript applications instead of the usual imperative quickly caught traction within the JavaScript community. It manifested itself not only as the most popular JavaScript framework (see figure 4.1), but also as the most liked framework among developers worldwide. (Figure 4.2)

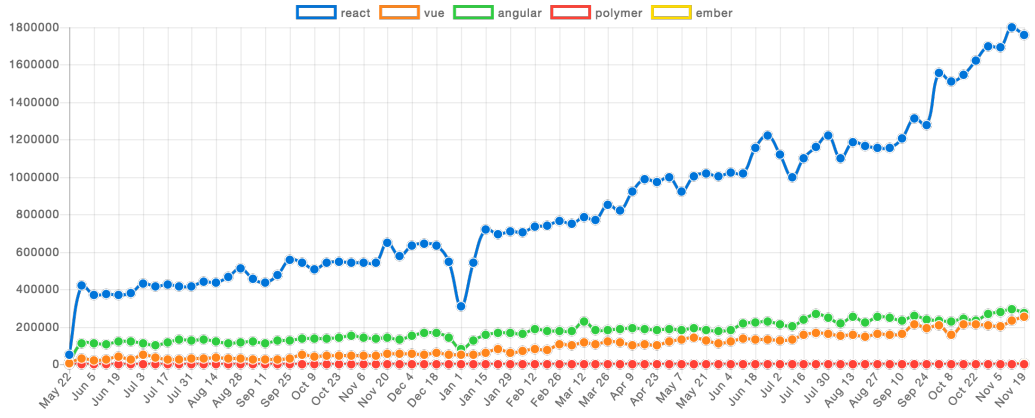

Figure 4.1: Amount of downloads of selected JavaScript frameworks on the de-facto standard JavaScript package repository npm [Joh17]

[8] "Declarative programming involves stating what is to be computed but not necessarily how it is to be computed." [Llo94]

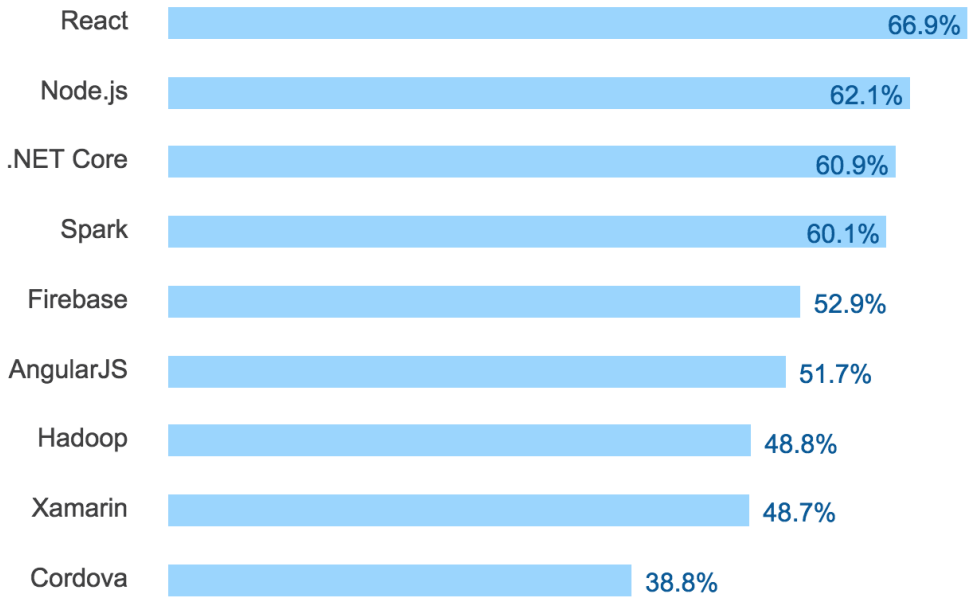

Figure 4.2: Percentage of developers using a framework that would like to continue working with the framework [Sta17a]

4.2 Technology

Let us now take a closer look at the technology behind React.

4.2.1 Single DOM Entry point

Code sample 2 serves as a minimal working entry point for a React application.
In the first two lines, we import the necessary vendor libraries `React` and `ReactDOM`[2]. `ReactDOM` is one possible render target for the `React` library and allows rendering into a HTML DOM, making it the most useful for web applications. There are other possible targets for a React app, for example for rendering into a text terminal[9], for VR[10] or for native mobile apps using React Native, which will be of special interest in this thesis. In any case, the `React` library contains the actual rendering, Virtual DOM, the reconciliation algorithm and support for JSX, an optional syntax extension that allows embedding of HTML into JavaScript functions.
Anyone familiar with JavaScript will notice the DOM selector function `document.getElementById` in the last line - this is a standard way to get access to the HTML DOM within JavaScript file, e.g. to dynamically alter the contents of a HTML element. However, this is for the most part the **only** DOM binding required. Every other component is supposed to be contained within the root Component, called `App` in this case.[11]
In this trivial example, the root component `App` simply contains of a `div` HTML element with text contents `Hello, World!`.

[9] `https://github.com/Yomguithereal/react-blessed`
[10] `https://facebook.github.io/react-vr`
[11] There sometimes is a need to display some element outside the rendering tree of the main application, e.g. for a modal dialog or similar. React allows this by rendering into so called "Portals", which are secondary DOM bindings outside the application.

React will render this root component (and all siblings) into the DOM automatically as required.

```
import React from "react" // rendering and JSX support
import ReactDOM from "react-dom" // render target

// root Component
const App = () => (
  <div>
    <span>Hello, World!</span>
  </div>
)

// assume a div with id "app" exists in the HTML embedding this script
ReactDOM.render(<App />, document.getElementById("app"))
```

Code example 2: Minimal working example of a React application

4.2.2 Component Composition

As our application grows, it is useful to modularize our code, and React helps us by letting us define custom components. For example, our application could feature a commonly used layout by additionally having a header and a navigation:

```
// ...
const Header = () => <h1>Some Title</h1>
const Navigation = () => <div>...</div>
const Content = () => <div><span>Hello, World!</span></div>

const App = () => (
  <div>
    <Header />
    <Navigation />
    <Content />
  </div>
)
// ...
```

Code example 3: Composition of multiple components

In code example 3, we defined additional components that we compose together to create the final application. These components could of course be defined in another file and consist of multiple components each, and so on. All of these components can be considered pure functions: they take a set of inputs and deterministically produce an output depending only on the inputs.

```
// ...
const Content = ({ greeting }) => (
  <div>
    <span>{ greeting }, World!</span>
  </div>
)

const App = () => (
  <div>
    <Header />
    <Navigation />
    <Content greeting="Olà" />
  </div>
)
// Olà, World!
// ...
```

Code example 4: Introduction of function argument ("prop")

4.2.3 Pure Functions as Stateless Components

In the previous example, all the components take no inputs, but we can easily modify a component to react to changes in input:
In code example 4, our `<Content />` component takes an argument (also called *prop* in React lingo) identified by `greeting` that allows us to set the type of greeting to be displayed. Defining custom components is a core concept of React and a very powerful tool to building applications. A common application of this pattern is defining a set of custom components that we can then use throughout the app, e.g. we'd create a set of custom components that are then used in multiple places, adapting to our needs by passing in different props.

4.2.4 Class-based Components with State

The components we have used so far were *stateless* components: They have no inner state and their output solely depends on the input passed to them. This functional approach allows easy reasoning about the app and is heavily optimized by React (because if the input does not change, we won't have to re-render the component). However, some sort of state is inevitable for most applications, so if we could only use stateless pure functions in our React app, the framework would not be that useful in real-life scenarios.
For this reason, React offers a second, class-based approach to components.
The slightly more complex example 5 shows a text message displaying how often the button next to it has been clicked. As we can see, the `<App />` component no longer is a pure function, but rather a class extending the `Component` class of the `React` library. This means we additionally get access to both an internal `state` of the component, that we can use to store intrinsic data, as well as special lifecycle hooks.
In the constructor of the class we have to call the constructor of the `Component` parent class to initialize the component and then set the initial state.

```
// ...
class App extends React.Component {

  constructor() {
    // setup lifecycle
    super()
    // initial state
    this.state = { clicked: 0 }
  }

  increaseCounter = () => {
    // calculate the next state by looking at the previous state
    this.setState(prevState => ({ clicked: prevState.clicked + 1 }))
  }

  render() {
    return (
      <div>
        The button has been clicked { this.state.clicked } times.
        <button type="button" onClick={this.increaseCounter}>
          +1
        </button>
      </div>
    )
  }

}
// ...
```

Code example 5: Class-Based approach to React components featuring internal state

The custom class method `increaseCounter` allows us to increment the internal counter. It is important to note that we must not mutate the state directly once the component is constructed (e.g. by setting `this.state = { clicked: x }` for some `x`), as React relies on calculating the difference between old and new state to decide whether a component needs to re-render or not. Thus `increaseCounter` uses a `Component` method `setState`, which takes a function as argument that receives the old state and returns the new.
The `render` then gets called by React as needed (in this case: when the state updates due to the user clicking on the button) and displays the current `clicked` counter by inspecting the internal state of the `<App />` component. Note how we bind our previously defined `increaseCounter` method as an event handler for the button `onClick` event.

4.2.5 One-Way Data Flow

As outlined in the previous section, JavaScript frameworks commonly use a two-way data binding approach. The previous example hinted at how React handles state differently: State flows from top down in one direction only and if we need to change the state, we must do it at the highest level. The changed state then propagates down until all children are notified of it. This may seem overly complicated and verbose in comparison

to the commonly used two-way data bindings, but allows us to easily reason about state changes and encapsulate the responsibilities of each component.
For example, we could have a top level `<Auth />` component that contains all state regarding authentication, methods to check credentials against an API, and so forth. This `<Auth />` component would then pass its state down to sibling containers via props: If we had the `<Greeting />` component from previous examples, it would only care about the username of the currently logged in user and would not need access to methods. Another `<Settings />` component could allow the user to change his password, so we would pass it methods to do that via props.
This way, state resides in one component only (the `<Auth />` component) and sibling component get access to this state as needed. This concept, also called *Lifting State Up*, is a core design pattern of React.[12]
Of course, this requires passing down the state from the top component through all siblings. As the app grows in size, other ways of managing state may be useful. A popular alternative to managing state in React is Redux[13] by Dan Abramov, which by some is considered the de-facto state solution for React [VS16]. Redux keeps all state inside an immutable, serializable object. Components can hook into this state from any place in the component hierarchy and modify it only by dispatching so called *actions*. Actions can be of any structure but commonly are JavaScript objects of the from `{ type: 'SOME_ACTION', payload: {} }`. So called *reducers* listen for these actions and determine the change in state - e.g. in the previous counter example, we could dispatch an action `{ type: 'INC_COUNTER', payload: 1 }`, and the corresponding reducer picks up the action and returns a **new** state by copying the previous and increasing the counter by `1`.
While introducing another layer of abstraction, this enables many useful features "for free". Because we never mutate the state but instead return copies of the old one, we have access to a serializable state timeline. This not only gives us undo/redo functionality but is extremely useful to debug an application - even remotely, we can catch any error the user encounters and upload a history of all actions that triggered the error.

4.2.6 Virtual DOM and Reconcilation

> "The virtual DOM [...] is a programming concept where [...] [a] 'virtual' representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation." [Faca]

As outlined in subsection 4.2.1, React massively simplifies the process of updating the DOM for the developer: Instead of requiring manual bindings of JavaScript variables into the HTML DOM, we bind a single root element into the DOM and let React figure out how to update it according to our app.
The Virtual DOM is represented as a tree data structure in memory, and any time the applications render output changes, React has to compute the differences between the currently displayed (real) DOM tree and the new (virtual) tree.
Traditional tree diffing algorithms have a runtime complexity in the order of $\mathcal{O}(n^3)$ [Bil05], which make them less useful for large HTML documents. The React reconciler uses a heuristic approach with runtime complexity in the order of $\mathcal{O}(n)$ in most cases [Facb].

[12] `https://reactjs.org/docs/lifting-state-up.html`
[13] `https://redux.js.org`

What is interesting to the Virtual DOM approach used by React is that the actual rendering is asynchronous and does not make any guarantees with regards to the temporal sequence of rendering - e.g. a components render output may change but not get updated for a few rendering cycles, if the reconciler decides this would improve the performance of the application. This allows automatic optimization of rendering performance - for example, the interface the user currently is interacting with may be given higher priority than some background animation. However, for this exact reason one has to take special care to not trigger any side effects in the `render` function, as there are no guarantees about the state other components may be in.

Chapter 5 React Native

Now that we have a solid understanding of the fundamentals of React, we will see how React Native utilizes the framework to deliver platform-native applications to mobile devices.

5.1 Technology

React Native can be understood as an extension to React, so all technologies explained in the previous section apply to React Native as well.
In fact, a React Native app runs both a JavaScript thread running React as well as a Native thread in parallel. Communication is managed by a bridge, over which a serializable payload is relayed (see illustration 5.1).
In practice, events in the native thread, such as touch inputs by the user or network responses, get collected and batched to reduce the throughput in the bridge middleware. Every batch of native data is then serialized in a protocol similar to JSON and relayed to the JavaScript thread, where it is deserialized and processed in a JavaScript and React environment, using the techniques outlined in chapter 4.2. However, the components act as interfaces to internal native methods - any calls to native methods are again batched, serialized and sent through the bridge to the native thread, where they get deserialized and actually call the methods as needed.
This of course has a performance overhead, and special care has to be taken by the developer to ensure each iteration of this cycle takes less than `16.67ms` to achieve a performance target of `60fps`, which is perceived as smooth motion and a responsive app by the end user (we will elaborate on that in section 6.2).

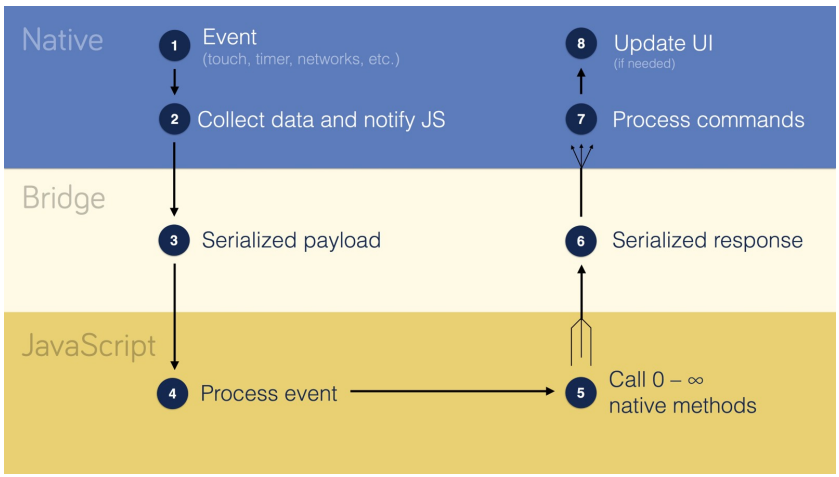

Figure 5.1: Illustration of React Native internals [Ale15]

```
1  import React from 'react'
2  import { Text, View } from 'react-native'
3
4  const HelloWorld = () => (
5    <View>
6      <Text>
7        Hello, World!
8      </Text>
9    </View>
10 )
11
12 export default HelloWorld
```

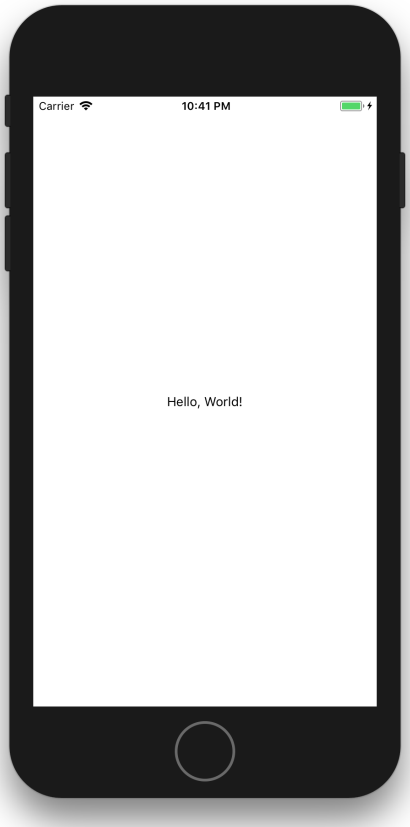

Figure 5.2: Minimal working example of a React Native app. Shown is the code (left) and the visual representation on an iOS device (right).

In order to adapt our "Hello World" example from a standard React app to a React Native version, a few changes are needed.

1. We do not use the `ReactDOM` library anymore.
2. Instead of using HTML elements like `<div />` or `<span />`, we use their React Native alternatives `<View />` and `<Text />`.
3. Any styling (if it exists) is translated from CSS to a React Native internal styling language.

The necessity of these changes arise from not working within a HTML DOM anymore. React Native components like `<View />` are interfaces to platform-native modules. Most of these components are available for both the iOS and Android platforms[1], but there a few platform specific components that are not available for the respective other platform. For example, for Android targets one has access to a `<ToolbarAndroid />` component that controls the commonly used toolbar with "Hamburger" style menu on the Android platform, while for iOS, one can use a `<TabBarIOS />` component that displays a commonly used tabbed bar on top.

[1] In fact, the sample React Native application developed during this thesis shares 100% of the code between the two platforms.

Thus if one wants to implement a true cross-platform component using React Native, the `render` method must output exactly the right representation for each platform. A common pattern for designing such a component is to implement all shared functionality within a common file `index.js` and specialize the `render` function in platform-specific files, e.g. `index.ios.js` for iOS, `index.android.js` for Android and potentially `index.web.js` for the HTML DOM.
Due to the need for platform specific code, this approach may seem to under-deliver on React Native's promise to deliver cross-platform app development. However, when considering the inherit differences between the two platforms Android and iOS, this seems like a sensible approach. The platforms differ not only massively in hard- and software, but in user expectations in terms of UI and UX, too. Thus instead for aiming for maintaining a single look and feel throughout all end-devices, one should ideally consider the differences between the platforms and deliver a platform-idiomatic application to the end user.
This approach is paraphrased by the tagline "learn once, write anywhere" as opposed to "write once, run anywhere" as one of the goals of the React Native framework [Tom15].

Part III

EVALUATION

Chapter 6 Methodology

In order to evaluate whether React Native is a viable alternative to native mobile app development, we will have to look at both raw performance as well as the development platform itself.
To do so, we'll implement the same application on the different platforms and then evaluate their performance based on selected metrics. We'll also consider the development platforms itself.

6.1 Test Application Design

When designing an application to be used in benchmarking, we are proposing the following constraints:
Firstly, the application must lend itself to being benchmarked. A simple "Hello World" application would typically render instantly on any platform, and we would not get any meaningful benchmark results.
Secondly, it should aim to closely resemble real-life usage scenarios. A benchmark of features not commonly found in smartphone usage patterns will not be relevant to most users.
Finally, it should be managable in terms of scope. While it is certainly possible to design a highly complex app that tests performance in a plethora of different ways, the time it takes to implement is critical for this thesis, as it will have to be implemented a total of three times.
For the reasons outlined in the following, we are proposing that the sample application designed for this thesis fulfills the constraints outlined above and is a good test case for our evaluation:

The application we are going to use as a test case is a simple client for the popular tech-focused social news aggregator Hacker News[1]. Hacker News like Reddit[2] and similar social news sites allow users to submit text posts and links to web pages, which other users then can vote on to influence how prominently the respective link or text is displayed to other users [Sto15].
While prior art evaluates the performance based on a simple implementation of Bubble Sort in the different platforms [FW16], we argue this is not a sufficient measurement of the experience the user would face in a typical mobile application, as few users would use a mobile app simply to sort integers.
Thus, we will evaluate a more sophisticated application that more closely resembles what an actual user would typically do on their mobile device.
The application will fetch the current 100 most popular links in a scrollable list, and allow users to click on a link to see top level comments made by other users.

[1] `https://news.ycombinator.com`
[2] `https://www.reddit.com`

This sufficiently complex application incorporates many techniques typically used in mobile applications, like fetching data from a remote server, displaying it in an interactive way to the user, and so on. It also is managable in terms of scope, as development of an application for three different platforms is a considerable time investment.

6.2 Performance Goals

> "Without a doubt, performance is a cornerstone of a great user experience." [CL11]

How do we accurately judge performance? While it is possible to quantify some metrics like latency and execution time [CL11], the actual user experience is highly dependent on the end user. What may seem like acceptable performance to one user, may be jarring to the other.
Google proposes a model to measure performance of web applications called RAIL, short for **R**esponse **A**nimation **I**dle **L**oad. "RAIL is a user-centric performance model that breaks down the user's experience into key actions" [MAK]. While RAIL is targeting web applications, we will apply the performance goals set by it to our React Native mobile application as well. Specifically, we are going to focus on the following performance goals:

- Respond: respond to user input within 100ms
- Animation: produce every frame within 10ms to consistently hit 60fps
- Idle: maximize idle time
- Load: Become interactive in 5s

If the sample React Native application is able to hit these performance targets, we propose its performance is viable.

6.3 Benchmarking

The main metric for benchmarking the app will be the time it takes the app to successfully fetch the 100 most popular links and display them interactively in a scrollable list. We will also test the response time from switching from the list of all links to the detail view of a specific link. Finally, we will check each implementation against a set of performance goals and also track hardware resource allocation.
In order to take fluctuations due to the network out of the equation, we build a simple mock API server using node.js.
The mock API server fetches the current 100 most popular links (including top-level comments) on Hacker News and save them to disk. Upon startup, the mock API server will load those items into memory cache for later consumption, mirroring the real API endpoints. The sample application will then use this local mock server instead of the live API to fetch all network requests. The mock API server also features helpful endpoints `/benchmark/set` and `/benchmark/reset`, which the applications will additionally call before and after every test run - these endpoints time the duration the app needed to fulfill the rendering and export the data in an easily machine parsable format.
Even with a local server there will be some fluctuations, so we will run each test a sufficient number of times and use the median test result as a final data point to compare to other platforms to.

```
03/10/18 23:29:02 1520720942186 [mockapi]   GET   '/topstories.json'
03/10/18 23:29:02 1520720942254 [benchmark] SET   'rn:activities:PostsTop:refreshPosts'
03/10/18 23:29:02 1520720942258 [mockapi]   GET   '/item/16473729.json'
03/10/18 23:29:02 1520720942261 [mockapi]   GET   '/item/16472774.json'
[...]
03/10/18 23:29:02 1520720942806 [mockapi]   GET   '/item/16465090.json'
03/10/18 23:29:02 1520720942810 [benchmark] RESET 'rn:activities:PostsTop:refreshPosts'. d=556
```

Code example 6: Sample output of mock API / benchmark server, indicating the client has finished fetching the top posts within 556ms

Chapter 7 Developer Experience

While a survey on developer experience with regards to the different platforms would be beyond the scope of this thesis, one cannot ignore the platforms themselves when determining whether React Native can be a viable alternative.
Thus in the following, the author will compare his personal user experience (UX) in using the recommended platform tools to develop the sample application outlined in the previous section.
It is noteworthy that the author was exposed to both Android and iOS platform development for the first time during this thesis, and has previous experience in both JavaScript and React (but strictly not React Native). Due to React Native fundamentally being based on React, the author most certainly had an easier time getting started on the React Native platform compared to the other two.

7.1 Android

Since it's first public release in 2014, Google has been offering Android Studio as a feature-rich IDE (integrated development environment) for development of Android applications. It is based on the IntelliJ Idea IDE and open source [Goo]. With official support for the previous vendor solution Eclipse ending in 2015 [Jam15], it has since become the de facto standard development environment for Android.
Downloading and installing the software required no effort on the authors part - everything from the actual IDE to required drivers and emulation software worked from the beginning on the development machine, even though it is running macOS. The developers seem to have put a great emphasis on multi-platform compatibility, as Android Studio will run on all major operating systems. This is in stark contrast to the iOS development setup, which exclusively runs on macOS.
Nowadays, Android developers can not only develop in Java, but choose from a plethora of options, including but not limited to C, C++ and Kotlin.
Like Java, Kotlin is a statically typed language that can compile to byte code for the JVM (Java Virtual Machine). With Kotlin gaining first-class support for Android in 2017 [Max17], the author was set on using Kotlin for the Android implementation of the sample app.
Due to the limited amount of resources on programming for Android using Kotlin in the web and the authors prior experience in the platform, the author relied on the book "Kotlin for Android Developers" by Antonio Leiva [Ant16] to get started with the app.
Without it, the author highly doubts whether he would've figured out the rather complex project and build settings. Even with the book, he had to research many platform-specific details on his own, e.g. the *Intent* and *Action* system to be able to display a different content to the user on clicking.

A hard requirement for the app was also the capability to run it with a different build configuration ("benchmark") to fetch network requests from the mocked API and trigger benchmark runs. This proved to be exceptionally difficult with little non-outdated resources on the web and the needed settings buried deep in one of Android Studio's numerous sub menus.

On the positive site, the author found designing the layout of the app using the integrated layout manager to see the results without compilation to be very helpful. To his knowledge, there is no such thing for React Native, which requires editing the source code and then recompiling the app to see any changes. Granted, due to incremental builds, this takes as little as a few seconds on React Native, but still requires a context switch.

Overall, the authors personal experience with developing the sample app for the Android platform using Kotlin has been very good. Android Studio has been very helpful with getting started and build times were acceptable (~1 minute for the finished app), while still being the slowest out of any platform tested.

The final implementation consists of **333** lines of Kotlin code and **559** lines of XML, so **892** lines of code in total.

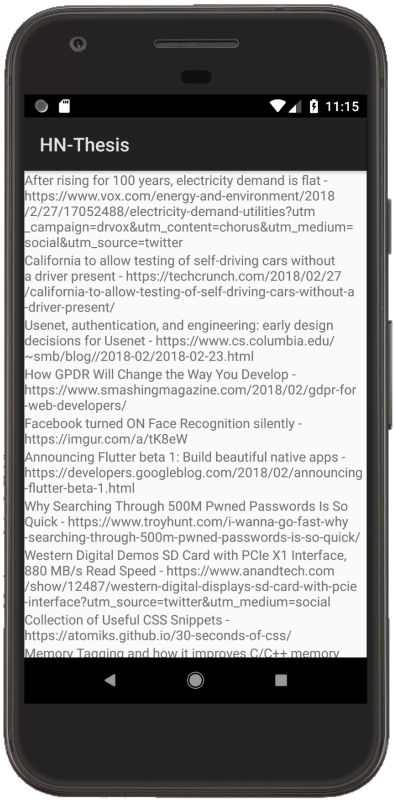

Figure 7.1: Native Android variant of sample app

7.2 iOS

While the first version of the mobile operating system that is now known as iOS was released in June 2007 [con18b], the core of the platform tools date back much longer: They are based on modern macOS (which used to be known as Mac OS X), which in turn is based on NeXTSTEP, which dates as far back as 1989 [GR11].

Traditionally, software for the platform was developed using Objective-C[1]. Objective-C is a super-set of C, with feature additions from the object-oriented programming language Smalltalk [Koc11].

In 2014, Apple has introduced the Swift programming language. It features a more concise and easily readable syntax, advanced memory management, support for dynamic libraries and a better performance, which makes it the recommended language to use going forward for the iOS and related platforms [Sol15]. Thus, the native iOS implementation of our sample app is developed using the Swift language.

For developing for the platform, Apple exclusively is offering XCode[2], a fully-featured IDE which supports not only Swift and Objective-C but a variety of platform-unspecific programming languages as well. Unfortunately, XCode exclusively runs on macOS, and to the authors knowledge, there is no way to develop native iOS applications on either Windows or Linux, as the SDK only targets macOS. Additionally, releasing an application

[1] `https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`
[2] `https://developer.apple.com/xcode`

for the public via the App Store requires not only a paid account at the so called Apple Developer Program[3] but also explicit approval by Apple. This "walled garden"[4] approach has been the subject of criticism over the years due to restricting the freedom of the end user [Jos07], but it is possible that the tight grip of Apple on the platform helps ensuring a high quality standard of applications.
In any case, developing locally without using the App Store does not have any specific requirements apart from having access to a macOS compatible machine.
Implementing the actual app presented no major roadblocks to the author. We use the common MVC pattern to structure the app into separate **M**odels, **V**iews and **C**ontrollers. As an example: the `Post` model (code 7) describes the structure of a Post, while the `PostsTopController` (see code sample 8) calls the API, maps the returned values to posts according to the model, and displays in a `UITableView`.

```
struct Post: Codable {
  let title: String // every post has a title
  let url: String?  // URL may be optional for self-posts
}
```

Code example 7: A simple `Post` model encodes the neccessary information for displaying a Post in a scrollable list

Using the built in `UITableView` class, it is trivial to display a dynamic list of cells, in our case containing the title and link of a story. All that is left to get the result as seen in figure 7.2 is to define a layout for each `ViewCell`, either using the graphical editor built into XCode or manually writing a so called `.xib` XML resource file.
Because of quick incremental builds and solid tooling, implementing the iOS variant of the sample application was overall a pleasant experience.
The final implementation consists of **244** lines of Swift code and **168** lines of resource files, which makes for **412** lines of code in total.

[3] `https://developer.apple.com/programs`
[4] "A walled garden is a system where an entity controls as many aspects of a product as possible and where features are only available if approved by a central authority." [Wol09]

```
class PostsTopController: UITableViewController {
  // store array of Posts
  var postsTop = [Post]()

  override func viewDidLoad() {
    super.viewDidLoad()
    // manually trigger refresh of data on first load
    refresh()
  }

  // refresh everything
  func refresh() {
    DispatchQueue.main.async {
      self.postsTop.removeAll()
      self.tableView.reloadData()
    }
    self.fetchData()
  }

  // fetch data from API and populate Posts array
  func fetchData() {
    HNAPI.getPostsTop { (postsTop) in
      for postId in postsTop {
        HNAPI.getPost(itemID: postId) { (post) in
          self.postsTop.append(post)
          DispatchQueue.main.async {
            self.tableView.reloadData()
          }
        }
      }
    }
  }

  /**
   * Control PostViewCell
   */
  // display all posts we have fetched
  override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
    -> Int
  {
    return postsTop.count
  }

  // map Post Array to PostViewCell
  override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    -> UITableViewCell
  {
    let cell = Bundle.main.loadNibNamed("PostViewCell", owner: self, options: nil)?
                .first as! PostViewCell
    cell.titleView.text = postsTop[indexPath.row].title
    cell.URLView.text = postsTop[indexPath.row].url

    return cell
  }

}
```

Code example 8: Simplified example of the controller that maps the `Post` model to the correct view

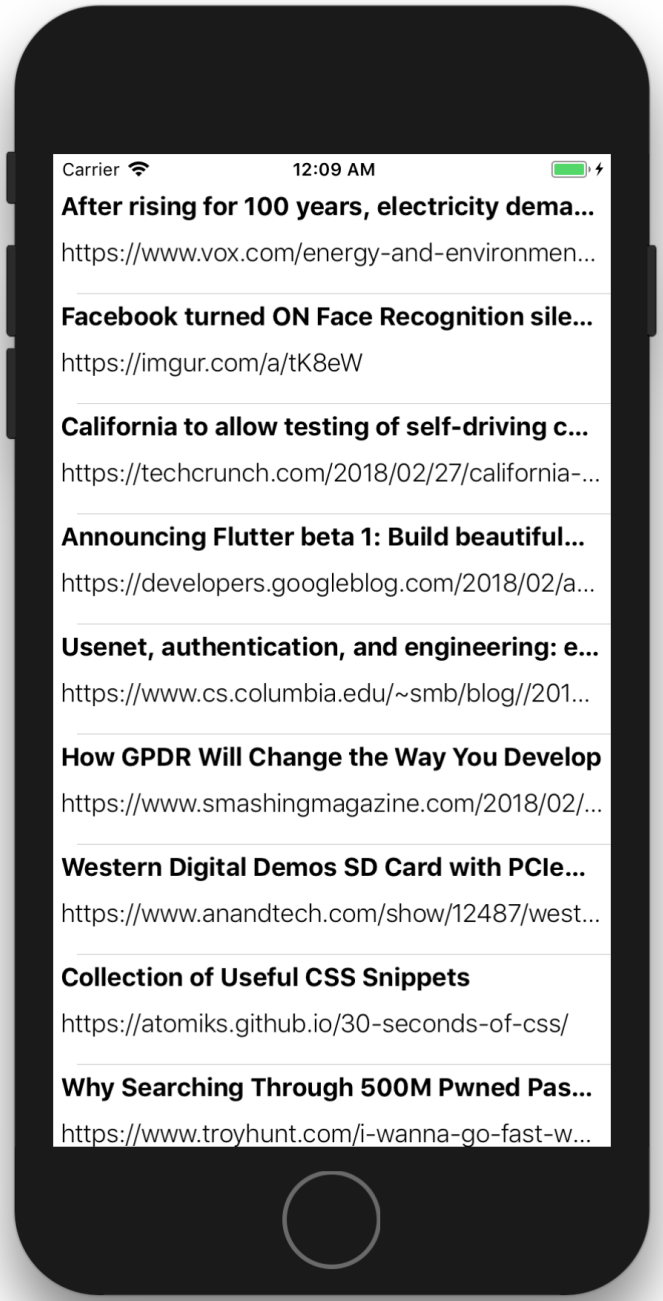

Figure 7.2: Native iOS variant of sample app

7.3 React Native

The initial setup with React Native is very quick and easy. All that is needed is to install the JavaScript CLI package `create-react-native-app`, and use that to initialize a new project. Because this setup has no further dependencies on any platform specific tools, any node.js compatible environment is usable for developing a React Native application. This is a clear advantage over the approach Android and even more so iOS take, as they require installation of special platform tools that in the latter case are exclusively available on computers running macOS.

The initial app build took about 1 minute, and subsequent incremental builds were the fastest among all platforms, with them taking only a couple of seconds. Due to a special technology called Hot Module Reloading (HMR), the application would even keep its state after incremental changes. This makes very quick iterations on the application possible and for an enjoyable developer experience.

In total, the final source code consists of **985** lines of JavaScript code.

However, it is to note that the first party tooling is not as robust as with the other platforms. There is no bundled or even recommended full-featured IDE shipped with React Native, and it is up to the user to setup linting, command completion and so on in their text editor of choice. While the author is proficient in JavaScript and the React platform, he was very thankful to have Android Studio/X-Code to help him out as a total beginner in these platforms. It is possible someone starting out with React Native and JavaScript in general would miss an IDE badly in the React Native platform.

So the tooling in React Native is arguably lacking in comparison, and during development of the React Native variant of the sample application the author would encounter bugs that broke the build process entirely without any connected action and required a re-initialization to resolve.

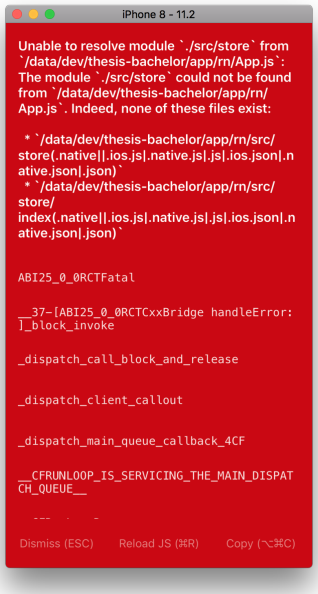

Figure 7.3: An error that the author encountered multiple times and which required re-initialization of the React Native project

While the author can't exclude the possibility of error on his part, it is arguable that the React Native platform as a whole is not as mature as the offerings of Google and Apple. One has to keep in mind React Native still is Alpha software and has not yet matured for over a decade like the native mobile platforms.

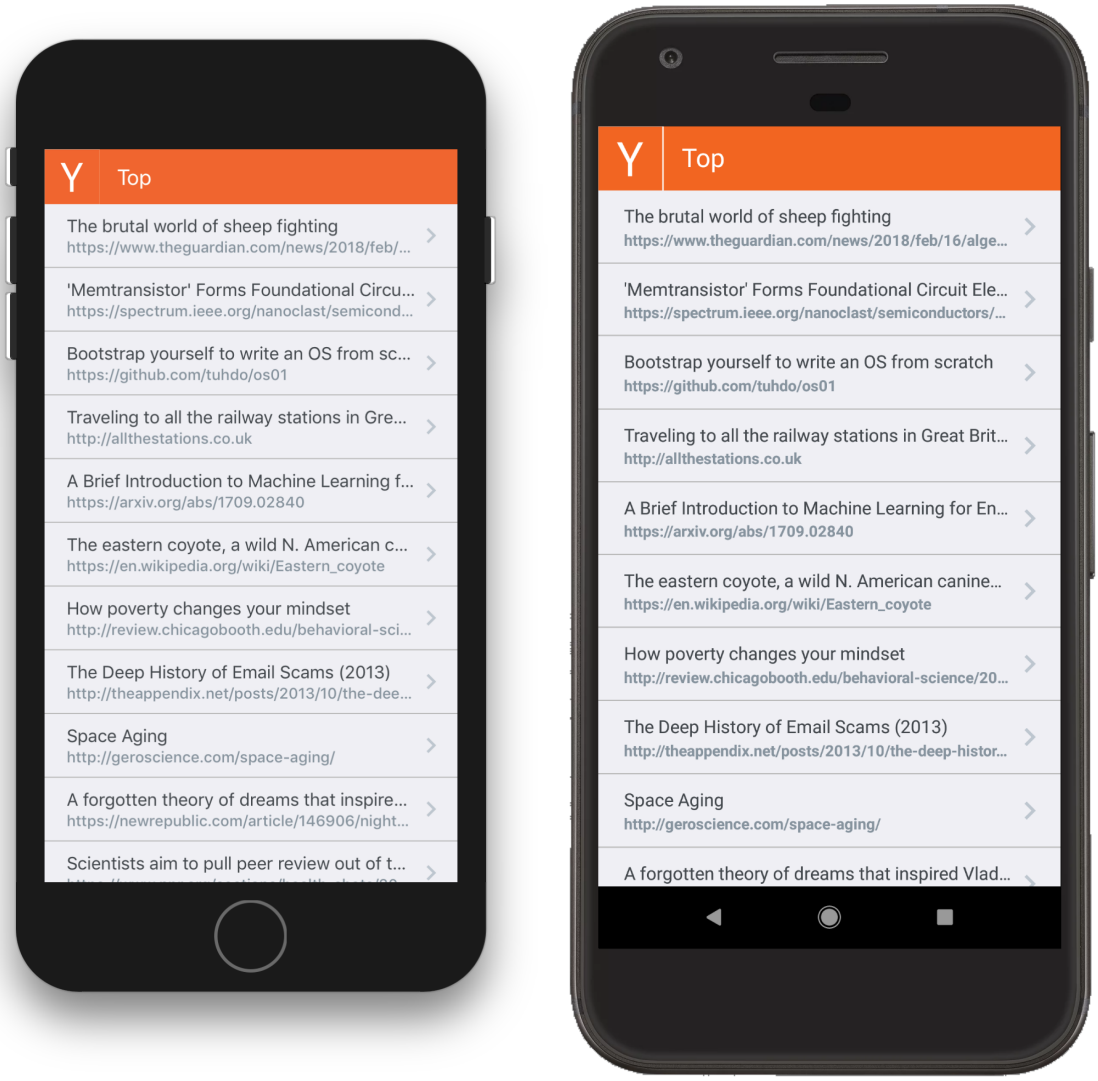

Figure 7.4: React Native implementations on iOS (left) and Android (right). Note the consistent look across platforms due to not using any platform-specific components.

Chapter 8 Benchmarking

In the following, we will test the performance of our React Native sample application not only against a set of performance goals but also directly compare the React Native implementations versus their native counterpart on the Android and iOS platform.

8.1 Setup

Due to not having physical access to recent iOS and Android devices, the author has opted to use emulation tools to run the benchmarks in. Both iOS with XCode as well as Android with Android Studio ship with bundled emulators that allow choosing the target device. We will opt for high-end devices (at the time of writing) as an emulation target, namely an iPhone 8 for iOS and a Pixel 2 for Android, as they seem to be comparable in specs and performance.
However it is to note that we cannot accurately compare between the two platforms[1] - e.g. if an implementation runs slower on one OS than the other, that does not have any significance with regards to the performance level of the two platforms. The only relevant result we will get is from comparing a different implementation on the same platform - e.g. we will compare how well the React Native app runs on Android compared how well the native version runs on Android, and vice versa for iOS.

8.2 Results

In the following, we will see how well our React Native implementation has performed.

8.2.1 Performance Goals

We are going to evaluate each performance goal set by RAIL (see 6.2) in the React Native implementation.

1. Respond

   Because the UI thread in React Native runs natively, user touch inputs experience no additional delay. To test the response time of application, we measure the delta in ms between the user clicking on a top link and the application starting the transition to a detailed view of that link.

Table 8.1: Delta in ms between user clicking on link and application responding by opening detail view

Platform	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
iOS, RN	23	66	80	70	63	39	38	43	30	39
Android, RN	45	104	103	91	39	73	72	140	50	79

[1] Performance of the Android emulator was well below the performance of the iOS emulator. The author suspects that is due to him running the emulator on macOS, which shares the kernel with iOS and thus could be better optimized to emulate the mobile OS.

Platform	Mean
iOS, RN	49.1
Android, RN	79.6

As chart 8.1 shows, the application on average successfully responded to all inputs well within 100 ms, thus meeting the respond performance goal of RAIL.

2. Animation

   Unfortunately, the tooling of React Native is lacking the capability to accurately assess each frame render time. The only indication of the animation frame rate is a built-in performance monitor displaying basic stats such as the current frame rate for both UI and JS thread.

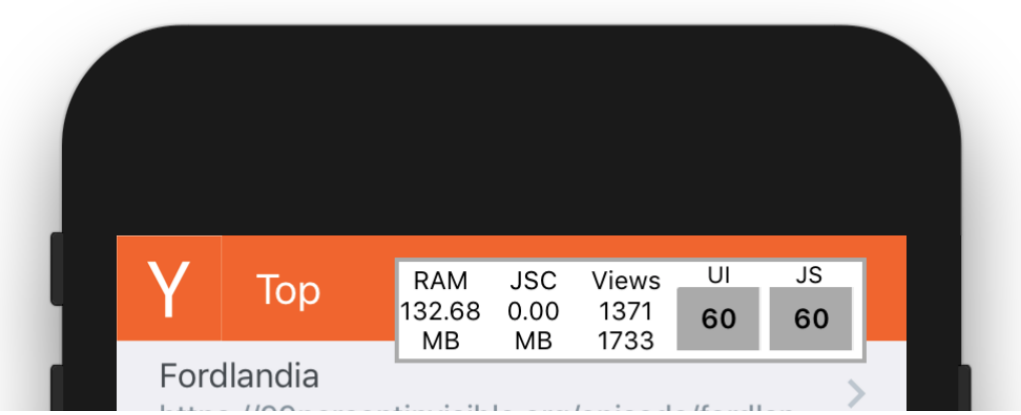

Figure 8.1: A glimpse into React Native rendering performance

The author could visually verify that the frame rates would stay at a constant 60 with momentary dips to 59 every now and then, even when scrolling the list and navigating to another link. However, due to lack of details regarding frame render time (which must stay below 10ms), we cannot verify for sure that the React Native app hits the Animation goal of RAIL.

3. Idle

   By attaching a debugger to the running iOS React Native instance and vice versa for Android,we were able to verify that the app returns to idle after every user action.

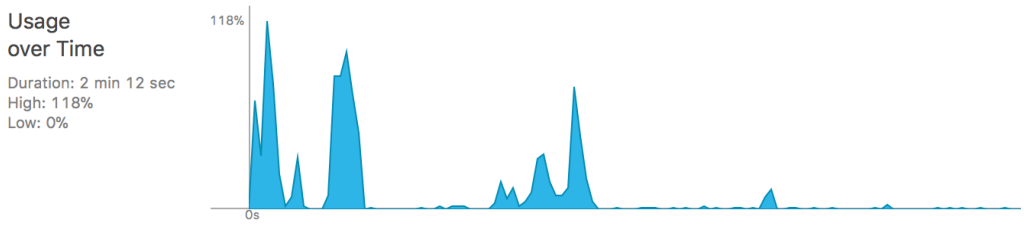

Figure 8.2: React Native iOS implementation CPU usage over time. First peak is the initial opening of the app, followed by the fetch and rendering of 100 top links. The peak in the middle indicates a switch from the main view to a detail view. Note how the usage returns to near zero after every action.

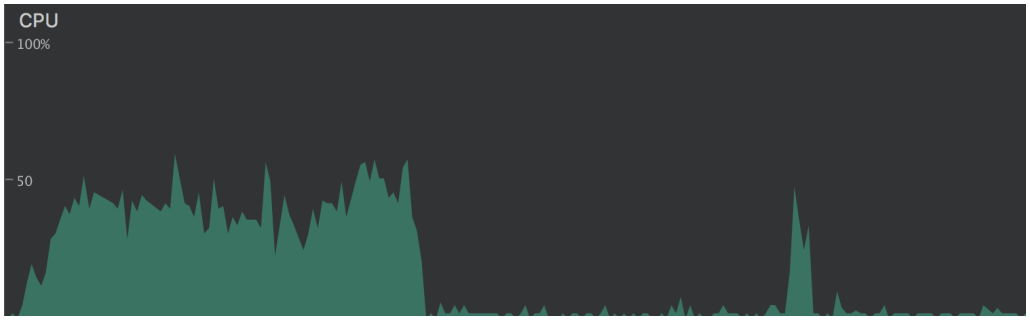

Figure 8.3: The android version of the React Native application displays similar idle behavior - the baseline load returns to under 5 percent after all user actions.

4. Load

   Results on Android were taken using `adb logcat` and listening for `ActivityManager` timings.[2]

Table 8.2: Time taken to launch application in ms. Note how the first data point is a *cold launch* after device reboot and subsequent launches benefit from the app assets being cached.

Platform	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Android, RN	2807	436	673	395	232	552	290	342	183	800

Platform	Mean
Android, RN	671

As chart 8.2 shows, the time to become interactive in Android after user launched the app is well within the goal set by RAIL.

Unfortunately, there was no reliable way to measure iOS startup time. Due to the iOS implementation outperforming the Android implementation in all measured benchmarks and the time to launch in Android being well under the target goal, it is safe to assume that the iOS platform performs within the goal, too.

8.2.2 Fetching and Rendering 100 Top Links

Initial test results of the performance during fetching and rendering 100 top links were very disappointing in the React Native implementation.

Table 8.3: Initial test results of fetching and rendering 100 top links. Shown are the two platforms using both native and React Native platform and the duration of each test run in ms

Platform	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
iOS	1176	450	599	412	653	789	663	675	779	1194
iOS, RN	4294	4755	4389	5875	4214	5077	4820	4738	4556	4410
Android	3278	3627	2187	2478	2997	1998	2664	2008	1443	1359
Android, RN	9701	9238	9653	10373	10495	10541	8916	8807	8505	8733

As we can see in charts 8.3 and 8.4, the React Native implementations ran about a factor of 6 (iOS) and 4 (Android) **slower** than the native implementations.

[2] `https://developer.android.com/topic/performance/launch-time.html#profiling`

Table 8.4: Initial test results of fetching and rendering 100 top links. Shown are the two platforms using both native and React Native platform and the mean duration of each test run in ms

Platform	Mean
iOS	739
iOS, RN	4712.8
Android	2403.9
Android, RN	9496.2

Of course, some performance penalty is expected due to the overhead of the parallel JavaScript thread and the message bridge, but the author had not expected such a significant slowdown. This huge disparity in performance led him to believe there was an issue with either the test setup or the React Native implementation, so investigation of the culprit began.
We can quickly rule out the test setup as the cause of this disparity. While the mock API approach still has some variance even when run locally, the apps would handle many more network requests to fetch 100 links without any problem - the two additional network requests used to set and reset the benchmark clock would be negligible.
As the author began looking into the React Native implementation, he was happy to discover debug support. It is possible to attach the rather powerful Chrome debugger to the currently running React Native application, which has helped to isolate the problem. For some reason, the reconciliation (subsection 4.2.6) of the list of top links takes about 200ms - which would not be a big problem if we didn't call it every time we fetched a new link - e.g. 100 times.
The fix as seen in code example 9 was rather simple: Instead of triggering the reconciliation on each fetch, we trigger it only at the end, when we have finished fetching all links.

```
// app/rn/src/containers/PostsTop/index.js
render() {
  const posts =
    this.props.postsHaveFetched &&
+   !this.state.refreshing &&
    Object.entries(this.props.posts)
      .map(([id, post]) => ({ id, ...post }))

  return (
    <PostList
      posts={posts}
      navigation={this.props.navigation}
      refreshing={this.state.refreshing}
      onRefresh={this.onRefresh}
      onPostClick={this.props.onPostClick}
    />
  )
}
```

Code example 9: simple performance optimization of the render path

This sped up the performance significantly, as seen in tables 8.5 and 8.6:

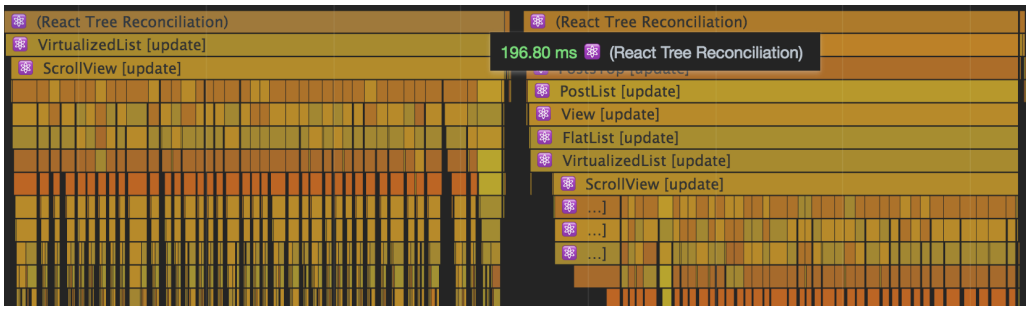

Figure 8.4: Debugging the performance problems of the React Native implementation using Chrome developer tools

Table 8.5: Second test results of fetching and rendering 100 top links. Shown are the two platforms using both native and React Native platform and the duration of each test run in ms. The React Native implementation has been optimized to improve performance.

Platform	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
iOS	1176	450	599	412	653	789	663	675	779	1194
iOS, RN	471	854	553	912	753	1130	656	817	649	648
Android	3278	3627	2187	2478	2997	1998	2664	2008	1443	1359
Android, RN	3462	2610	2264	2414	3306	1654	1869	2969	2806	1824

Table 8.6: Second test results of fetching and rendering 100 top links. Shown are the two platforms using both native and React Native platform and the mean duration of the test runs in ms. The React Native implementation has been optimized to improve performance.

Platform	Mean
iOS	739
iOS, RN	744.3
Android	2403.9
Android, RN	2517.8

After the optimization, the React Native app was within 2% (iOS) and 5% (Android) of the performance of the native implementations. We have to keep in mind that a modification of the specification of the app was necessary - while the native versions of the app have no performance penalty displaying the fetched links as they arrive, the React Native version cannot deliver the same performance unless batching the reconciliation at the end. This would probably make not a notable difference for the end user, but gives us an indication that one has to take special care when developing for the React Native platform.

Part IV

Discussion

Chapter 9 Conclusion

We have shown that within specific constraints, React Native can be a viable alternative to developing separate mobile applications.
The sample application used to test the claims set forth by React Native is arguably a good baseline of features commonly expected in a smartphone app. But we have to note that we did not use any lower level hardware API like the camera, GPS and so on. While bindings exist for some of these features in the React Native API, we did not assess their performance. It is also reasonable to assume that applications that rely on the highest real-time performance like resource-intensive 3D games are the wrong fit for the React Native platform, due to the inherit overhead of the message bridge.
Within the basic feature set, the sample application was not only able to hit the performance goals set by the RAIL model (as verifiable), but in actual benchmarking it was within 5% of the performance of the native implementations. In order to achieve quasi-native level of performance, optimizations of the rendering path were necessary that were of no concern in the native implementations. These optimizations lead to a slight regression in features (due to the top links not rendering asynchronously as they loaded), but did not diminish the overall user experience of the application in a meaningful way.
In terms of resource investment, React Native is the clear winner to develop an application for two platforms at once. The implementation of the sample app amounted to about a thousand lines of code, which is on par with the Android implementation and slightly more than the iOS implementation, while running on two platforms instead of one. It is also noteworthy that we were able to share 100% of the code between the two platforms, no platform-specific code was necessary in our case.
While getting started on the React Native platform was the easiest and quickest of any platforms, this benefit may be negated due to the arguably worst tooling of the tested platforms.
Quick iteration cycles due to the fastest incremental builds and HMR make developing on React Native an enjoyable experience for the developer, but one may encounter bugs in the build process. By nature of JavaScript being an dynamically typed language, one also has to be aware of run-time bugs that would be caught during compilation on the Android and iOS platforms.
Overall, React Native can be an attractive platform for cross-platform app development. It proposes a compromise with slightly diminished performance and a less mature environment, but a quicker start and smaller resource investment. The trade off must be evaluated per use case, but React Native is an option to consider.

Chapter 10 Future Work

The features tested in the sample application only used a small set of the offerings of the React Native platform. There is future work to be done to assess the performance of hardware-near features like the camera and GPS, and to see how viable a highly complex cross-platform application in React Native could be.
While we compared the viability of React Native for cross-platform mobile development, we did not assess the viability of targeting the web as well. As React Native fundamentally is based on React with DOM support, it would be worthwhile to investigate multi-platform viability, including both mobile and the web.
As React Native matures as a platform, it will be interesting to see how the platform tooling will improve and whether it will match the offerings of Apple and Google. To address the shortcomings of the platform, a comprehensive survey among developers for mobile applications would be helpful.
The future of the platform is unclear. While React Native is backed by a huge company, there is no guarantee of continued support. Developing for a specific platform can be a large investment for any organization, and the success of React Native depends on regular updates to ensure compatibility with future versions of iOS and Android. However, React Native is licensed under the open source MIT license, so continued support by third parties is a possibility in case Facebook decides to stop active support.

Part V

BIBLIOGRAPHY

[Llo94] John W Lloyd. “Practical Advtanages of Declarative Programming.” In: *GULP-PRODE (1)*. 1994, pp. 18–30.

[LR01] Avraham Leff and James T Rayfield. “Web-application development using the model/view/controller design pattern”. In: *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*. IEEE, 2001, pp. 118–127.

[Woo+04] Lauren Wood et al. *Document Object Model (DOM) level 3 core specification*. W3C Recommendation, 2004.

[Bil05] Philip Bille. “A survey on tree edit distance and related problems”. In: *Theoretical computer science* 337.1-3 (2005), pp. 217–239.

[Eic05] Brendan Eich. “JavaScript at ten years”. In: *ACM SIGPLAN Notices*. Vol. 40. ACM, 2005, pp. 129–129.

[Jos07] Joshua Gay. *iPhone restricts users, GPLv3 frees them*. June 2007. URL: https://www.fsf.org/news/iphone-gplv3 (visited on 03/19/2018).

[Wol09] Michael H Wolk. “The iPhone Jailbreaking Exemption and the Issue of Openness”. In: *Cornell JL & Pub. Pol'y* 19 (2009), p. 795.

[CL11] Andre Charland and Brian Leroux. “Mobile application development: web vs. native”. In: *Communications of the ACM* 54.5 (2011), pp. 49–53.

[GR11] Mark H Goadrich and Michael P Rogers. “Smart smartphone development: iOS versus Android”. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 607–612.

[Koc11] Stephen G Kochan. *Programming in objective-C*. Addison-Wesley Professional, 2011.

[Alm13] Ben Alman. *Facebook: Rethink established best practices™*. May 2013. URL: https://twitter.com/cowboy/status/339858717451362304.

[Dal+13] Isabelle Dalmasso et al. “Survey, comparison and evaluation of cross platform mobile application development tools”. In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. IEEE, 2013, pp. 323–328.

[JT13] Jordan Walke and Tom Occhino. *JS Apps at Facebook*. May 2013. URL: https://www.youtube.com/watch?v=GW0rj4sNH2w.

[MP13] Michael S Mikowski and Josh C Powell. “Single page web applications”. In: *B and W* (2013).

[Sim14] Simon Khalaf. *Apps Solidify Leadership Six Years into the Mobile Revolution*. Jan. 2014. URL: http://flurrymobile.tumblr.com/post/115191864580/apps-solidify-leadership-six-years-into-the-mobile.

[Ale15] Alexander Kotliarskyi. *React Native: Under the Hood*. Sept. 2015. URL: https://speakerdeck.com/frantic/react-native-under-the-hood.

[CKT15] Ioannis K Chaniotis, Kyriakos-Ioannis D Kyriakou, and Nikolaos D Tselikas. “Is Node. js a viable option for building modern web applications? A performance evaluation study”. In: *Computing* 97.10 (2015), pp. 1023–1044.

[Jam15] Jamal Eason. *An update on Eclipse Android Developer Tools*. June 2015. URL: https://android-developers.googleblog.com/2015/06/an-update-on-eclipse-android-developer.html.

[Sol15] Paul Solt. “Swift vs. Objective-C: 10 reasons the future favors Swift”. In: *InfoWorld. May* 11 (2015).

[Sto15] Greg Stoddard. “Popularity and quality in social news aggregators: A study of reddit and hacker news”. In: *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015, pp. 815–818.

[Tom15] Tom Occhino. *React Native: Bringing modern web techniques to mobile*. Mar. 2015. URL: https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/.

[Ant16] Antonio Leiva. *Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App*. Mar. 2016.

[FW16] Martin Furuskog and Stuart Wemyss. *Cross-platform development of smartphone applications : An evaluation of React Native*. UPTEC STS 16029. Uppsala University, Division of Computer Systems, 2016.

[VS16] AM Vipul and Prathamesh Sonpatki. *ReactJS by Example-Building Modern Web Applications with React*. Packt Publishing Ltd, 2016.

[Joh17] John Potter. *react vs vue vs angular vs polymer vs ember | npm trends*. Nov. 2017. URL: http://www.npmtrends.com/react-vs-vue-vs-angular-vs-polymer-vs-ember.

[Max17] Maxim Shafirov. *Kotlin on Android. Now official*. May 2017. URL: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/.

[Sta17a] Stack Exchange Inc. *Stack Overflow Developer Survey 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted.

[Sta17] ECMA Standard. “262: ECMAScript Language Specification 8th Edition, June 2017”. In: *Web link: http://www.ecma-international.org/publications/standards/Ecma-262.htm* (2017).

[Sta17b] StatCounter. *Desktop vs Mobile vs Tablet Market Share Worldwide*. Nov. 2017. URL: http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide.

[con18a] Wikipedia contributors. *Ecma International — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=Ecma_International&oldid=828986257.

[con18b] Wikipedia contributors. *IOS version history — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=IOS_version_history&oldid=830748862.

[con18c] Wikipedia contributors. *JavaScript — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=828219989.

[Faca] Facebook Inc. *React Docs - Internals*. URL: https://reactjs.org/docs/faq-internals.html.

[Facb] Facebook Inc. *React Docs - Reconciliation*. URL: https://reactjs.org/docs/reconciliation.html.

[Facc] Facebook Inc. *React Native*. URL: https://facebook.github.io/react-native/.

[Goo] Google Inc. *Building Android Studio*. URL: http://tools.android.com/build/studio.

[MAK] Meggin Kearney, Addy Osmani, and Kayce Basques. *Measure Performance with the RAIL Model*. URL: https://developers.google.com/web/fundamentals/performance/rail.

[RSM] Raphaël Benitte, Sacha Greif, and Michael Rambeau. *The State of JavaScript 2017*. Tech. rep. URL: https://stateofjs.com/2017/flavors/results.