GRADO DE INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

Desarrollo de una aplicación móvil usando las tecnologías React Native y Phonegap y diseño e implementación de un escenario de prueba con React Native

Ignacio Martín Velasco

2016

TRABAJO FIN DE GRADO

TÍTULO:	**Desarrollo de una aplicación móvil usando las tecnologías React Native y Phonegap y diseño e implementación de un escenario de prueba con React Native**
AUTOR:	**D. Ignacio Martín Velasco**
TUTOR:	**D. Joaquín Salvachúa**
DEPARTAMENTO:	**Ingeniaría de Sistemas Telemáticos**

TRIBUNAL:

Presidente:	**D. Juan Quemada Vives**
Vocal:	**D. Santiago Pavón Gómez**
Secretario:	**D. Gabriel Huecas Fernández-Toribio**
Suplente:	**David Fernández Cambronero**

FECHA DE LECTURA: _____

CALIFICACIÓN: _____

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGERNIEROS DE TELECOMUNICACIÓN

GRADO DE INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO DE FIN DE GRADO

DESARROLLO DE UNA APLICACIÓN MÓVIL USANDO LAS TECNOLOGÍAS REACT NATIVE Y PHONEGAP Y DISEÑO E IMPLEMENTACIÓN DE UN ESCENARIO DE PRUEBA CON REACT NATIVE

IGNACIO MARTÍN VELASCO

2016

Resumen

La aparición de los smartphones ha propiciado la proliferación de multitud de aplicaciones móviles para todas las plataformas existentes. En las distintas tiendas de aplicaciones podemos encontrar desde apps para acceder a las redes sociales o juegos, hasta algunas orientadas a profesionales de distintos sectores. Esto ha dado lugar a un creciente mercado dedicado al desarrollo de las mismas empleando lenguajes de programación como Java, Objective C o Swift.

Sin embargo, la existencia de plataformas muy distintas y propiedad de diferentes compañías ha provocado que una vez desarrollada una app para una de ellas, sea necesario rehacerla parcial o completamente si deseamos adaptarla a otro sistema.

Con el objetivo de facilitar la portabilidad entre plataformas surgió el concepto de las aplicaciones móviles híbridas. A través del uso de tecnologías web estándar como son HTML5, CSS o JavaScript, y mediante una API, se pueden desarrollar aplicaciones móviles multiplataforma de forma rápida y eficiente.

Actualmente existen dos tecnologías para el desarrollo de aplicaciones móviles híbridas: Phonegap, con una ampplia base de usuarios, y React Native, más reciente y en continua actualización. En este Trabajo de Fin de Grado se expondrán las principales diferencias entre ellas y se desarrollará una aplicación con ambas con objeto de poner de relieve dichas diferencias.

Asimismo se diseñará e implementará un escenario de prueba con React Native, que sirva de introducción a dicha tecnología. Se documentará todo el proceso, desde la instalación de las distintas herramientas necesarias hasta la ejecución de una aplicación en un terminal real.

Palabras clave: Aplicación Móvil, React, Phonegap, JavaScript.

Abstract

The recent emergence of smartphones has led to a XX in the amount of mobile applications available for all existing platforms. In the app markets a wide variety of applications can be found, ranging from games or social networks to some more career-oriented. The result is a rapidly growing market devoted to their development with programming languages such as Java, Swift or Objective C.

However, very different platforms exist, owned by different companies. This means that once an app has been developed for one of these platforms, in order to run it on another one, the application has to be completely doveloped almost from scratch again.

In order to ease the task of porting an app to a different computing environment emerged the concept of hybrid mobile applications. Using web technologies such as HTML5, CSS and JavaScript, and an API, cross-platform mobile applications can be developed in a a quick and efficient manner.

Currently, there are two technologies for the development of hybrid mobile applications: Phonegap, used by a broad amount of developers, and the more recent React Native, which is still under development. In this work, the biggest and most important differences between both technologies will be explained, and in order to hightlight them, the same application will be developed for them both.

A React Native test scenario will be designed and implemented as well, to act as an introduction to this technology. Every step of the process will be thoroughly documented, from installing the required software to running the app on a device.

Keywords: Aplicación Móvil, React, Phonegap, JavaScript.

Agradecimientos

Lo primero quiero agradecer a mis padres y a mi hermana todo el apoyo brindado a lo largo de estos años. Siempre me han apoyado, han confiado en mí y se han sacrificado para que hoy pueda estar aquí.

En segundo lugar, a mi novia, por todo el apoyo y la comprensión mostrada en esos largos días estudiando y trabajando. Me ayudaste a seguir cuando más desencantado estaba y has aguantado mi cabezonería. Espero poder devolvértelo siempre.

Gracias a mis abuelos por todo su cariño y sus consejos, siempre habeís estado ahí cuando os he necesitado.

A mis amigos, porque cuando he necesitado que me echaráis una mano nunca me habéis fallado. Especial mención a mi primo, por todo lo vivido juntos.

A mis compañeros y también amigos de esta Escuela, con los que ha sido un placer trabajar y

Y finalmente a Joaquín Salvachúa, mi tutor, que me ofreció la oportunidad de realizar este Trabajo, ha confiado en mí y jamás me ha puesto una pega.

Contents

List of Figures

Listings

CHAPTER 1

Introduction

1.1 Context

Not long ago, before developing a mobile application, or app, one of the most important decisions that had to be made was choosing the platforms that your app would run on. Because a different programming language is used for each platform, the same application would have to be developed twice, thrice or even more times. In fact, the amount of work involved could be such that would limit the efforts of the development team to a single platform.

Along with mobile applicationss, another way of adapting web services for smartphones emerged: web mobile applications. This type of applications use standard web technologies, generally HTML5, CSS and JavaScript, to adapt web pages and make them suitable for the smaller screens of smartphones and tablets. Though this concept saves time, some vital limitations remain, mostly access to native device functionalities such as camera or GPS.

Aiming to increase the productivity and efficiency of a development process, the concept of hybrid mobile applications was born. Through an Aplication Programming Interface, or API, it is possible to embed HTML5 apps inside a native container, so the same code can be ported across systems. Hybrid mobile applications combine the best elements of native and HTML5, and therefore represent a

new stage in the evolution of mobile applications development.

In this context, currently the most popular technology for this purpose is Adobe PhoneGap [1]. However, some of its own peculiarities make PhoneGap apps inefficient and slow, and somehow feel less polished than a native app. In an attempt to fix these issues and challenge PhoneGap's dominant position, React Native [33] came on the scene on April 2015. React Native's main feature is that it uses native components to achieve a completely native application user experience.

1.2 Project goals

This project aims to provide an explanation on two main topics: firstly, the differences betweeen PhoneGap and React Native, and secondly, a test scenario to help understand the most important features that revolve around React Native.

Among the main goals inside this project, we can find:

- Comparing current mobile application development options: their advantages and disadvantages will be exposed and detailed.
- Understanding what React Native: what it is and how it works, its architecture, runtime execution and lifecycle, among others.
- Design and build a simple test scenario to demonstrate React Native's capabilities.

1.3 Structure of this document

This project is divided in X chapters, arranged following a general-to-specific pattern. The structure is as follows:

Chapter 1: provides an introduction to the project, explains the main goals to be achieved and a strctural overview of this document.

Chapter 2: describes the main options currently available for developing a mobile application, and their advantages and disadvantages.

Chapter 3: brief introduction to PhoneGap in order to fully understand how it works and distinguish it from React Native.

Chapter 4: description of React [32], a JavaScript library for creating the user interface in a Model-View-Controller (MVC) architecture based web page.

Chapter 5: a detailed description of React Native and its motivation, architecture, runtime execution, and other relevant features.

Chapter 6: description and implementation of a test scenario with detailed explanations. This Chapter aims to serve as an introduction to the development of mobile applications using React Native and flatten the learning curve. It is not a showcase of all React Native features and capabilities, but will serve as a solid foundation for those with real interest in this technology.

Chapter 7: this paper ends with a reflection on React Native and the developer experience, summarizing the achieved goals and providing future lines of work.

CHAPTER 2

Understanding Mobile Application Development Options

2.1 Introduction

This chapter provides a detailed explanation on the distinct types of mobile applications that have been already introduced. As a result, the reader will be aware of the pros and cons that every scenario involves. Special attention will be payed to hybrid mobile applications, not only because the whole project is based on them, but also because they are the less well-known.

One section will be devoted for each and every scenario, discussing how are they developed and exposing their strengths and weaknesses. A brief summary of the technologies and tools needed for each will also be provided. The chapter will conclude with a table to sum up all the available scenarios.

2.2 Native Mobile Applications

A native mobile application is an application coded in a specific programming language. The most popular are Java for Android and Swift or Objective C for iOS operating systems. They provide the best usability, the best features, the best per-

formance, the best overall user experience, plus are highly reliable. Because of this, apps like video games tend to be developed this way, as they are resource intensive and a bad performance could hurt the user experience. However, this type of app is tied to one single operating system, forcing the development team to make duplicate versions that work on other platforms. Therefore this may not be the best option for apps intended to spread throughout the whole market.

Their key features are:

- **Multi touch handling:** native apps support all kinds of multi touch events, from double taps or pinches to pressure-sensitive taps.
- **Built-in components:** camera, GPS or encrypted storage are only a few of the features native to a device that can be smoothly integrated into an app. Built-in components are of such importance to some apps that may be a deciding factor on which mobile technology one should choose.
- **Fast graphics and fluid animations:** for those apps which are using a lot of data and require a fast refresh, are highly interactive or make use of intensely computational algorithms, native apps provide the highest efficiency and the best results.
- **High reliability:** the abstraction while coding is lower than on the other types of mobile applications. Therefore, as a developer, you got more control over what is going on inside your app, and nasty surprises are less likely to happen.
- **Improved user experience and familiarity:** people are accustomed to the native platform. Making use of all the native features will be easier for the average user and the app will be just plain easier to use.
- **Documentation:** apart from the official docuementation, there are thousands of books for Android and iOS development, and many more blogs, articles and websites with detailed information about every feature and little crevice.

After taking a look at the pros of developing a native app, it is clear that they are unbeatable on some aspects. However, they suffer from a number of issues:

- **System restricted:** the fact that native apps are developed using specific programming languages turns porting an app between platforms into a time-consuming task because most of the code will be simply useless.
- **Difficult to develop:** the programming languages used are not the easiest ones, so simply cutting and pasting Objective-C or Java will neither suffice nor work.

- **High level of experience required:** because native apps are difficult to develop, the level of experience required is higher than other development scenarios. Indeed, the technological know-how of a developer is a very important consideration.
- **Require an integrated development environment:** an integrated development environment, or IDE, provides tools for building, debugging, managing, version control and some other critical tools that professional developers need. These tools are needed because of the difficulties involved in the development process. In fact, usually different IDEs are used for each programming language, and even though they tend to be very similar to one another, so they become another thing to add to the list of inconvenients.

After considering all the pros and cons concerning the development of a native mobile application, we can conlude that it is the best way to develop ambitious, compute-intensive, single platform mobile applications.

2.3 Web Mobile Applications

A web mobile application, also known as HTML5 mobile application, is basically a web page designed to work on the small screen of a smartphone. This makes them compatible with every modern mobile browser. Developing them is easier as well: the technological bar and the learning curve of website programming languages are singnificantly lower than those of Objective C, for example.

Web apps most remarkable aspects are the following:

- **Write once, run anywhere:** web applications take this concept to the extreme. As long as the device is running a modern browser, the user can use the app. This makes sure that it will reach the whole market.
- **Searchable:** because the content is on the web, it can be searched, which can be a huge boost to its popularity.
- **Easy to develop:** the complexity involved in developing a web page is usually diminished by the fact that is easier to understand and of a higher level than native apps programming languages.
- **Distribution and support:** native mobile applications are dependent upon marketplaces for their distribution and support. Nevertheless, web applications are hosted on a server kept under control by the development team, so fixing a bug or adding features becomes a much easier task.

However, these advantages come at a price:

- **Lack of access to native features:** web applications are not capable of accessing built-in components such as the camera or the GPS. This lack may be a determining factor when security is of the utmost importance, as offline storage and security are not available fopr use in a proper manner.
- **Multi-touch input events not supported:** today, no one has managed to capture multi-touch input events in this type of app. This is a clear limitation for any application that requires two or more touch events simultaneously, like games of other highly interactive apps.
- **Unfamiliar feeling:** users will not be as familiar with a web application as they would be with a native application, and therefore the user experience will resent.

As we just explained, web mobile applications excel at almost every aspect where native mobile applications suffer. In general, they are suited for simple tasks like adapting websites with a lot of text to the tiny screen of a mobile device, but tend to be bad a bad design choice for more specific tasks like highly interactive or tight security apps. Also, statistics show that 90% of the time on mobile devices is spent in apps, and not in the browser. Additionally, more and more companies are shifting to native apps because the overall superior user experience.

2.4 Hybrid Mobile Applications

Hybrid mobile applications combine the best of both the native and web application worlds. A hybrid app is built using HTML5, CSS and JavaScript, like a web application. Then, the app is wrapped inside a thin container that provides access to native features. This way the code will work on every platform, reducing the porting task to rewriting only a few, small parts of the app.

Before digging deeper into the definition of hybrid mobile apps, it is important to clarify that there are two subtypes, which will be classified in terms of this work as Native Hybrid and Web Hybrid.

Native Hybrid apps include one or more webviews, generally controlled by the native side, who is responsible of provinding the transitions and navigation between webviews.

On a Web Hybrid app, the whole content is wrapped in a webview. The thin native container acts as a bridge between the app and the device. The native side

does not provide any kind of user interface component. It is, therefore, a mere intermediary for native-to-webview communication.

However, as in most cases where things look perfect, the devil is in the details. This subject will be further discussed while describing the two technologies previously mentioned: PhoneGap and React Native.

2.5 Summary

Based on current technology, the scenarios examined previously are summarized in the following table:

	Native	**Web**	**Hybrid**
Programming Languages	Java, Swift, Objective C	HTML5, CSS, Javascript	HTML5, CSS, Javascript
Installed locally	Yes	No	Yes
Features			
Graphics	Native APIs	HTML, Canvas, WebGL, SVG	HTML, Canvas, SVG
Performance	Fast	Slow	Depends
Look and feel	Native	Emulated	Depends
Distribution	App Market	Web	App Market
Internet connection required	Depends	Yes	Depends
Built-in Components			
Camera	Yes	No	Yes
Geolocation	Yes	Yes	Yes
Storage	Secure offline file storage	Shared SQL	Both
Notifications	Yes	No	Yes
Contacts, calendar...	Yes	No	Yes
Gesture Recognition			
Tapping	Yes	Yes	Yes
Swipe	Yes	Yes	Yes
Pinch	Yes	No	Yes
Other multi-touch	Yes	No	Yes

Table 2.1: Features of the different scenarios available for mobile application development.

CHAPTER 3

PhoneGap

3.1 Introduction

Adobe PhoneGap is a web-based mobile development framework based on the open source Apache Cordova [9] project, which allows the creation of hybrid mobile applications using HTML5, CSS and JavaScript. It is currently under development by Adobe.

PhoneGap can build apps for Amazon Fire OS [4], Android [5], BlackBerry 10 [10], Firefox OS [14], iOS [23], Ubuntu [38], Windows [42], Windows Phone [43] and Tizen [37]. The user interface is built using the traditional web development skills already stated, which are executed within wrappers targeted to each platform, and rely on API bindings, to access each devices built-in components.

PhoneGap's container consists basically of a native WebView where the HTML5 code is embedded. Native resources such as the camera, accelerometer or geolocation are accessed through a foreign function interface. Its fucntionalities can be extended with native plugins that allow developers to add more capabilities that can be called from the JavaScript code.

PhoneGap's most relevant features will be reviewed next.

3.2 Apache Cordova

Apache Cordova is the framework on top of which PhoneGap is built. In short, PhoneGap is an extension of Cordova in terms of functionality and tools. A high-level view of a Cordova application architecture is shown in the following diagram.

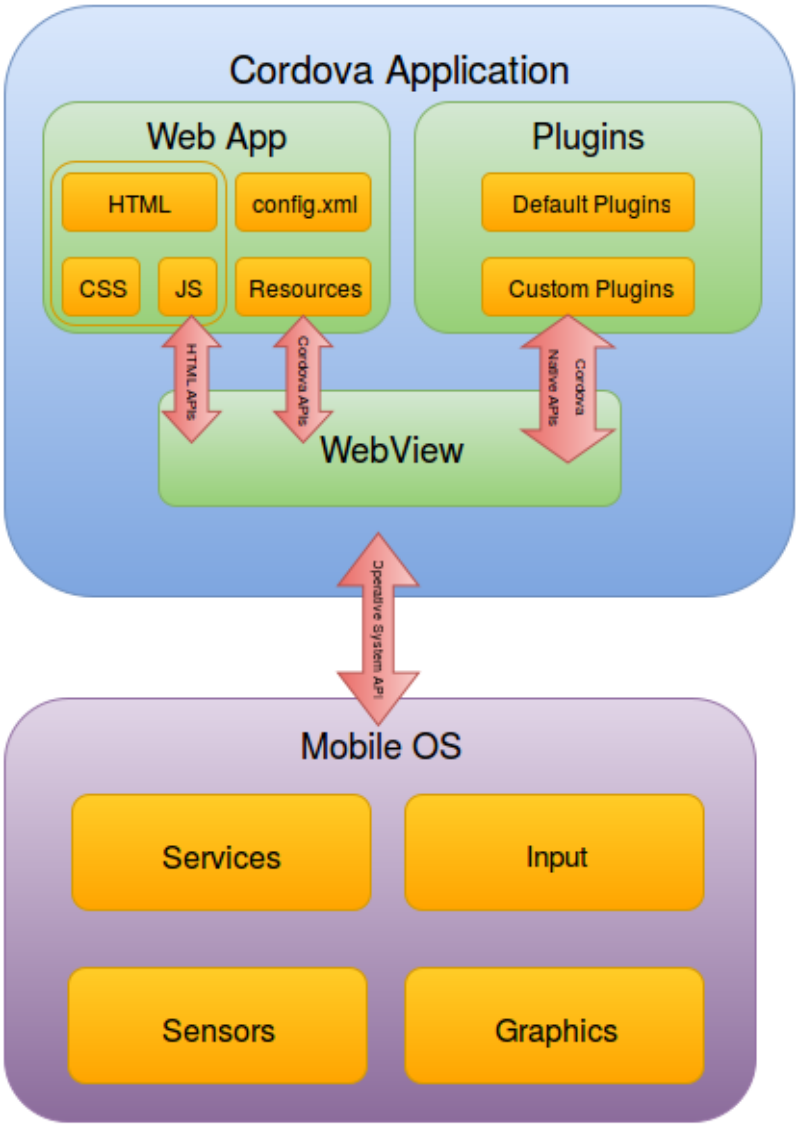

Figure 3.1: Cordova Architecture

As we can see in Figure 3.1, there are two clearly separated levels: the mobile operative system and the Cordova application.

The first one is the lowest as well, and is in charge of managin the device sensors, services, graphics and other input. To gain acces to all of these, the Cordova application makes use of a series of operative system APIs. This level has been kept intentionally simple because is out of the scope of this work.

The latter is described with a little more detail. There are three main elements:

- **Web App:** user interface (UI) and apps logic. The UI is developed as any web application frontend using HTML5 and CSS, while JavaScript takes care of the logic. "config.xml" [40] is a World Wide Web Consorctium (W3C) widget specification compliant file that allows the developer to specify metadata about the app. Images, audio, etc. would be part of the application resources.
- **HTML rendering engine:** also referred as the WebView. Cordova uses the native browser engine provided for each platform to render the application. For example, on iOS, this is the Objective-C UIWebView [24] class, while on Android this is the android.webkit.WebView [8]. This way of rendering views is much slower than the native rendering, usually hurting the apps performance and the user experience. Besides, there are differences in the native browser engine between operating systems which may result in errors, bugs and other inconveniences. Obviously, this is a major drawback of Apache Cordova and should be considered carefully.
- **Cordova plugins:** plugins allow invoking native code from JavaScript. A series of core plugins are provide access to the most basic device features such as the camera, contacts or battery. PhoneGap plugins will be further discussed in the following pages.

3.3 PhoneGap Main Features

As PhoneGap is built on top of Apache Cordova, it extends all of its features and provides some new ones as well.

3.3.1 Desktop app and command line interface

PhoneGap provides both a desktop app and a command line interface (CLI) as tools to ease the creation, compilation, test and deployment of an application.

The CLI is written on top of Apache Cordova and distributed on Node Package Manager (NPM) [30]. It is available on any platform and is the most powerful way to use PhoneGap.

However, a desktop app is provided as well for Mac and Windows. Way simpler than the CLI, allows to create and serve apps to a connected mobile device. The GUI makes it easier to use, but advanced functions are not implemented, rendering it considerably less powerful.

3.3.2 Mobile developer app

The developer mobile app enables the developer to quickly see how the mobile app works and feels on a real device. However, it currently supports only three platforms, which, on the other hand, comprise over 95% of the market share: iOS, Android and Windows Phone [28].

3.3.3 Local and remote building

For most platforms, PhoneGap can compile a project locally, or can be sent to a PhoneGap Build [2] server to be compiled remotely.

3.3.4 Plugins

A PhoneGap plugin [3] is a piece of code that provides a JavaScript interface to interact with native buil-in components. They allow an application to use all the native capabilities that are not available to web apps. These plugins give the developer the ability to further extend functionality and interact in more detail with device features and functions that are not already exposed thorugh the existing Apache Cordova plugin API.

CHAPTER 4

React

4.1 Introduction

React is a JavaScript library for creating composable user interfaces. It was built meant for big applications with tons of data that changes over time. React keeps track of the data and automatically updates the views when it changes. However, it only updates parts affected by those changes, improving performance.

Views are built using reusable components. These are encapsulated, making testing and isolating elements a very easy task.

4.2 Flux

Flux [17] is an architecture concept that implements a unidirectional data flow. This means that data only flows in one direction, progressing through stages until it reaches the end of the chain. In Flux, this is implemented as follows: and action propagates through a central dispatcher to the various stores that contain the application's logic and data, which then updates all the views that are affected. This structure if much more efficient than two-way data binding and makes a program much easier to reason about.

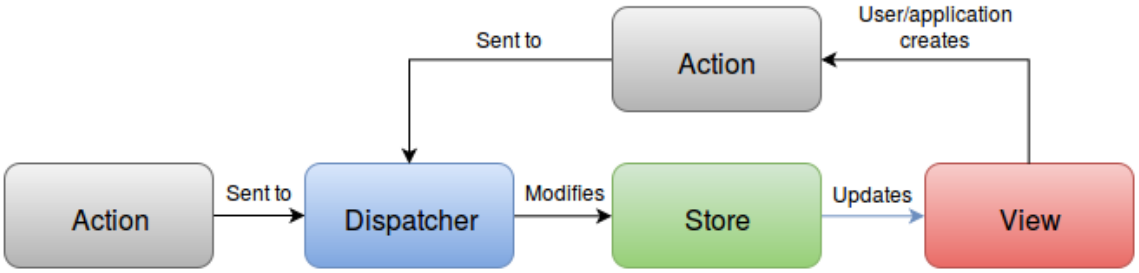

Figure 4.1: Flux unidirectional data flow

As shown in Figure 4.1, all data flows through the dispatcher. Actions usually originate from the user's interactions with the views. The dispatchers invokes callbacks according to the stores that have registered with it. Stores update the necessary data and proceed to alert the views, which in turn update their own data and perform a re-render of themselves and all their descendants in the component tree.

4.3 JSX

JSX [27] is a JavaScript extension to make it look similar to XML. Though using it is optional, its very concise and familiar syntax make it great for defining tree structures with attributes. Thanks to JSX, these tree structures are easier and faster to read.

Because all of these, applications developed using JSX instead of JavaScript will achieve an overall higher quality. A fragment of code is shown next, both in JSX and JavaScript, to show the differences. The translation was performed by Babel.js, a JavaScript library specially designed to translate JSX to JavaScript, among other features.

```
var HelloMessage = React.
    createClass({
  render: function() {
    return <div>Hello {this.props.
    name}</div>;
  }
});

ReactDOM.render(<HelloMessage name=
    "John" />, mountNode);
```

Listing 4.1: Sample JSX code

```
"use strict";

var HelloMessage = React.
    createClass({
  displayName: "HelloMessage",

  render: function render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
});

ReactDOM.render(React.createElement
    (HelloMessage, { name: "John" })
    , mountNode);
```

Listing 4.2: Sample JSX code translated to JavaScript using Babel

4.4 Virtual DOM

A Document Object Model [12] is an API for HTML and XML documents. It defines their logical structure and the way they are accessed and manipulated. The specifications have been defined by the W3C. DOM documents have a logical structure that resembles a tree, which is why it is usually referred as DOM tree. A root node serves as the root of the tree for the document.

To render a document such as an HTML page, browsers use an internal DOM. The whole HTML code is automatically parsed and a DOM of the page is generated. This enables JavaScript to create dynamic HTML, meaning it can add, modify or remove HTML elemts, change CSS styles, and create or react to events.

However, whenever a change has to be made, the whole DOM needs to be remade. This results in very low performances, specially on modern pages with thousands of nodes. In fact, accessing a node can take seconds sometimes.

To solve this issue, React makes use of an abstract, light-wight version of the DOM, known as Virtual DOM. Once changes have been saved in the Virtual DOM,

the browser compares it with the real DOM, finds the differences and only updates what should be changed. React is declarative, meaning that these updates happen whenever data changes. The whole process is much more efficient than directly modifying the DOM as it does not require all the heavyweight parts included in it.

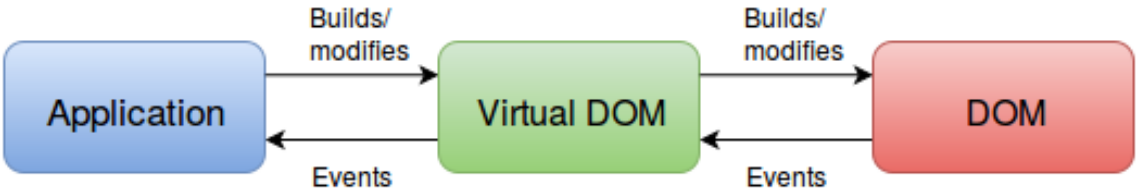

Figure 4.2: React interaction with Virtual and real DOM

4.5 Example Component

React components implement two very important features related to input data and internal state data: `props` and `state`. The first one provides a way of passing input data and can be accessed using a `render()` method which is available for every React component; while the latter is used to inform React of a data change. Every time the state of a component changes, a user interface update is triggered. In fact, React components can be seen as simple state machines.

An example component is shown next, where both `props` and `state` are used. This snippet simply salute a user called *Juan* and counts the elapsed seconds since the user loaded it. Trying it out is very simple, as many online live compilers exist.

```
var ExampleComponent = React.createClass({
  //Set initial state
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  //Updates elapsed time
  tick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed + 1});
  },
  //Once after intitial render; only client-side
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  //Before component is unmounted from the DOM
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {  //Renders the component
```

```
19      return (
20        <div>
21          //Renders the name stored in props
22          <div>Hola {this.props.name}</div>
23          //Renders elapsed time, stored in state
24          <div>Tiempo transcurrido: {this.state.secondsElapsed}</div>
25        </div>
26      );
27    }
28  });
29
30  ReactDOM.render(<ExampleComponent name="Juan"/>, mountNode);
```

Listing 4.3: Sample React component

As we can see, components are React classes. The first thing to do is setting the component's state, if any is needed at all. Keeping as many components as possible stateless is essential, as any state update triggers an update and affects performance. The next function is in charge of updating the state, while the next couple of functions are invoked at different stages of the component's lifecycle. Finally, the `render()` function specifies what to show. The last line of code renders the component/class into the DOM.

As we will see in the next chapter, React Native's component lifecycle and programming style is very much the same as React's.

CHAPTER 5

React Native

5.1 Introduction

React Native is a framework for building hybrid mobile applications based on React. It was announced on April 2015 by Facebook at the Facebook Developer Conference [13] and is currently under development, receiving updates every two weeks. React Native is constantly evolving, so new features not included in this work may become available as time passes.

React Native libraries provide the React architecture to iOS and Android applications. These two systems are the only ones available, but this should not be a worrying issue as they comprise around 90% of the market share [28].

Due to Apple policies, development of iOS mobile applications can only be done on a Mac machine. On the other hand, Android apps can be developed on Mac, Linux and Windows. However, React Native is still on a very early version, and development on a Windows environment can sometimes be unstable.

5.2 The Implications of Native

React Native stands out from other methods of cross-platform mobile application development due to the fact that it uses its host's standard rendering API's. Currently

existing frameworks that enable the development of mobile apps using HTML, CSS and JavaScript render their views using WebViews. As has been previously stated, this is a valid approach and can work, but implies a series of drawbacks, mainly bad performance. When these technologies try to mimic the native user interface elements, they feel a little off, and details like animations take enormous amount of work to implement and usually cause drops in the frame rate.

React Native's main goal is achieving native performance, feeling and capabilities, while keeping an easier and more friendly environment for development. React Native mobile applications are written using only JavaScript as opposed to PhoneGap, which uses HTML5, CSS and JavaScript. In practical terms, this means that a JavaScript layer must be added.

However, if this is not done properly, it could result in low frame rate, slow performance and bad user experience in general. To avoid all of these, React Native implements a few features which will described next.

5.2.1 Asynchronus and batched

The JavaScript layer is asynchronus and runs on a separate thread from the main one. This way, the rendering will not be affected by any operations that are performed in the JavaScript thread. Moreover, native instructions that have to be executed can be batched and run whenever the device can. Actions such as decoding an image or saving to disk could potentially block the user interface even for seconds, but this way, the app will remain fluid and responsive.

In addition, all the communication is fully serializable, which allows using Chrome Developer Tools [11] to debug the JavaScript code while running the complete app.

5.2.2 Declarative vs Imperative:

React is declarative, meaning that it focuses on what the application should do instead of how it must be done. However, native development uses an imperative paradigm, i.e. consists mostly of commands for the device to execute. So all changes that have to be made to the interface must be converted from declarative to imperative. This is achieved by using thin wrappers that run on background processes to take advantage of multicore architectures and avoid interfering with the main thread. For instance, lets take the following example based on Android:

 → <View>
 → <Text>
<img> → <Image>

Here, the well-known `div`, `span` and `img` HTML tags turn into their Android code equivalent. This way, React Native avoids using WebViews, so everything on screen is purely native. On PhoneGap the WebView caused most of the trouble such as frame rate drops or low performance, but React Native is not hampered by this.

5.2.3 Learn once, write everywhere

React Native does not aim to achieve the write one run everywhere idea where the same code powers every platform. Each system has a different feeling and behaviour, and therefore iOS and Android mobile applications feel and behave differently. Instead, the main goal is using the same set of principles for every system, so building the same app for iOS and Android requires no significant amount of extra work, but it feels like a native application on both platforms.

5.3 Architecture

As has been stated, React Native code is not exactly the same for an iOS app than for an Android app. The situation is very similar when we take a look at React Native's architecture, and must be explained separetely. However, they share a basic structure, which is shown in Figure 5.1.

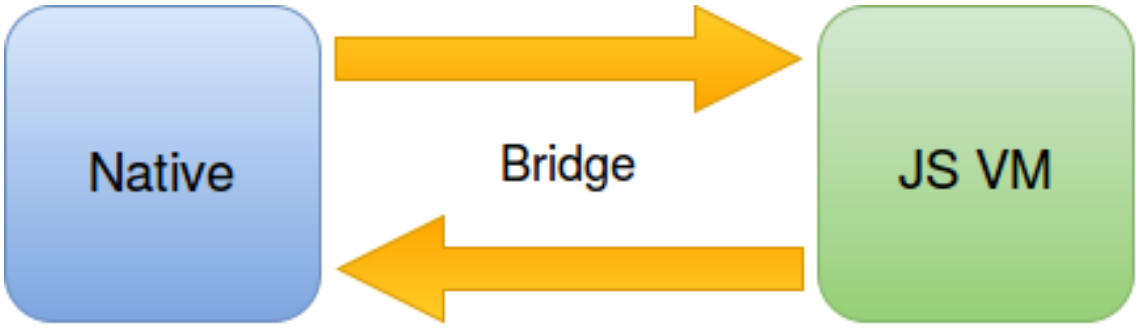

Figure 5.1: React Native basic architecture

As we can see, there are 3 main layers:

- **Native:** renders the user interface and manages native events.
- **Bridge:** the Bridge connects the Native layer with the JS VM.

- **JS VM:** the JavaScript Virtual Machine runs the code. In this case, the virtual machine used is JavaScriptCore [26]. It needs to be shipped with every app, which adds around 3.5MB to its size on Android, while on iOS JavaScriptCore is part of the system.

5.3.1 Runtime

The problem of concurrence has been traditionally tackled by using threads. A process can have many threads, each managed independently by a scheduler. As it is a very well-known concept, there is no need for going into details. However, React Native uses queues to solve the issue. Queues are First-In-First-Out data structure that allows the insertion and deletion of elements, also known as enqueueing and dequeueing. Queues hold the data until the receiver can retrieve it.

There are 3 main queues in React Native:

- **Shadow queue:** creates the layout.
- **Main thread:** thread run by default in a native app.
- **JavaScript thread:** runs the JavaScript code.

In addition, every native module, such as networking, images, etc. has its own queue unless it is specified otherwise. Figure 5.2 shows a graph of what happens when a React Native application is run. The action that appear in the graph are described next. [47]

- **Run a React Native app:** when the user runs the application, three actions are triggered: loading the JavaScript bundle, loading the Native Modules and starting the JavaScript Virtual Machine.
- **Load the JavaScript bundle:** grabs all the dependencies and puts them into one single script, so React will only deal with this file. To avoid hanging up, a separate thread takes care of the process.
- **Load the Native Modules:** loads all the necessary native modules such as networking, image, local storage, etc. This process is split between the bridge and the module itself: the native module is loaded into the memory and then it calls the bridge to register this module. This way, the bridge knows from the very beggining all the modules that exist, and can create any required instances.
- **Start the JavaScript VM:** a new instance of the JavaScript VM is created in a new thread, and all the native hooks that React Native has are provided.

These include high precision timers, logging, shotcuts for optimization and all the synchronous methods that React Native exports, among others.

- **Load the JSON configuration:** once all the native modules are loaded and the JavaScript VM has started, all the modules and the mothods they have are put into a JSON object. JavaScript will use this information to create objects at runtime.
- **Execute the application's JavaScript bundle:** once the JSON configuration is finished, the JS VM starts executing the code.
- **Abstract JavaScript Execution:** JavaScript has four possible entry points: you can call a callback, call a method, execute an application script or load configuration. To avoid having a graph with four descendants, all of them have been placed together in this node.
- **Create Shadow Views:** once the JavaScript code has been executed and created the component objects, each one will have a shadow view which contains the layout information.
- **Layout:** the layout is computed in a background thread. An absolute position and size is generated for each view.
- **Create the Native Views:** in parallel with the creation of the shadow views and the layout computing, the native views that will be used to render the app screen are create.
- **Render to screen:** once both the layout and the native views have been created, they are combined into the final layout of the application.

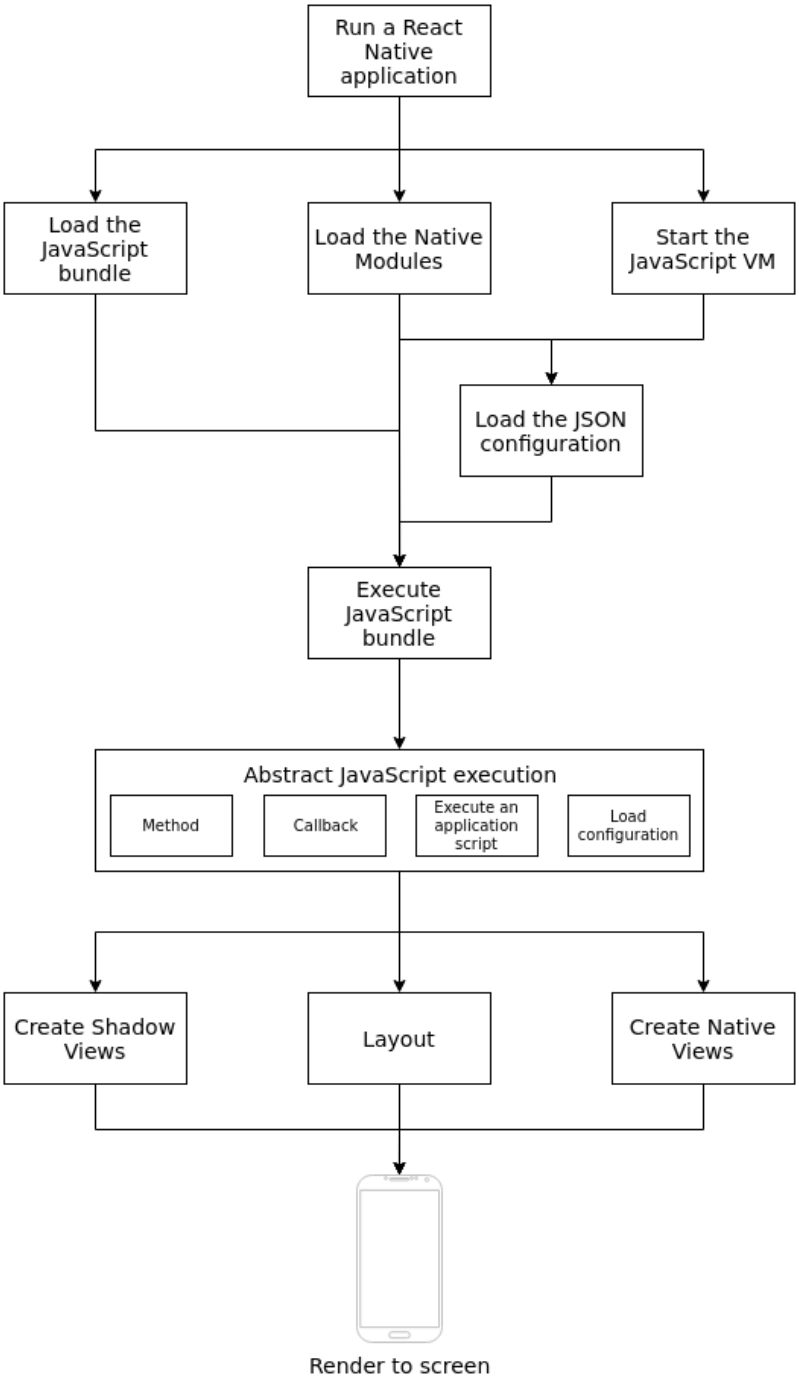

Figure 5.2: React Native application runtime graph

5.3.2 Call cycle

The typical call cycle is shown in Figure 5.3. JavaScript is event-driven, so once the device detects that event (network, touch, etc.), it collects all the necessary data and sends it to the JavaScript Virtual Machine, encoded as JSON. The Bridge serializes the data and pushes it. The JS VM handles the event and decides what native methods should be called. These are bundled and sent to the Native layer

through the Bridge, which serializes the response. In this case, the JavaScript output is enconded as JSON, and is then converted into the right type depending on the system. In turn, the native device processes the commands and updates the user interface. Step by step, the actions taken are the following:

- **User Interaction and Device:** one of the two two entry points in the call cycle. The user interacts with the device and generates an event such as a touch, so device notifies the native system.
- **Native:** the native layer processes the information generated by the device and decides which native module should call. For example, React Native has its own touch handler, that would send out to the appropriate native module.
- **Native Event:** the second entry point to the call cycle. A native event refers to any action
- **Native Module:** this module dispatches the event to React. The call can directly come from another native module such as an alert, or from a user interaction with the application.
- **Abstract JavaScript Execution:** the JS VM executes the necessary code to respond to the event, which may require a call to another native module. In this case, JavaScript executes a call to native; but if no more native calls are required, it proceeds to re-render the screen in the same manner as explained in Paragraph 5.3.1.
- **JavaScript call to native:** JavaScript calls back to native to require a native module.
- **Bridge:** the Bridge takes the array of calls, matches them to the right modules and runs the necessary threads. After that, it gets back to the Native Module, which in turn dispatches the action to React.

As we can see, the JavaScript Execution - JavaScript call to native - Bridge - Native Module loop can be repeated as many times as necessary before re-rendering the view. [47]

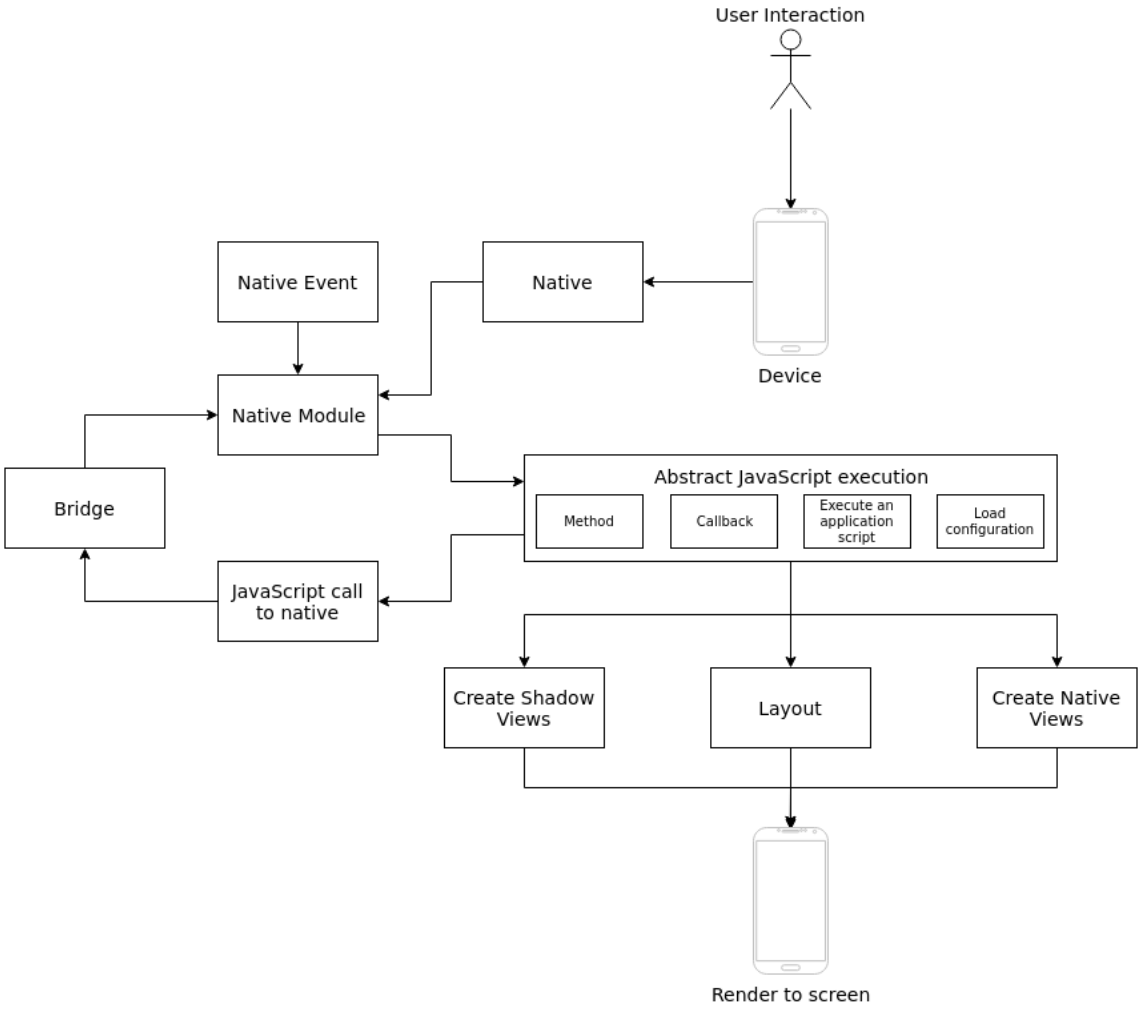

Figure 5.3: React Native app call cycle

5.4 Components

React Native components are very simmilar to React's. However, there are some differences that must be explained.

5.4.1 Native Components

As has already been stated, React Native is declarative, and a simple example on React web components and their React Native equivalents was shown. But React Native uses native components, which are invoked from the JavaScript code. Listing 5.1 provides an example of a very simple React Native application.

```
1 import React, { AppRegistry, Component, StyleSheet, Text, View } from 'react
    -native';
2
3 class RNExample extends Component {
```

```
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to React Native!</Text>
      </View>
);}}

const styles = StyleSheet.create({
  container: { ... },
  welcome: { ... },
});

AppRegistry.registerComponent('RNExample', () => RNExample);
```

Listing 5.1: Sample React Native app

Parts of the code have been left out to highlight the structure. Every React Native app must include the following four parts:

- **Imports:** modules and components that will be used in the application must be imported before creating the React Native class. In this case, React is imported along with the necessary components from the React Native library.
- **Class:** here is where both the logic and view of the component are implemented. The `render()` method is mandatory, and should be always implemented. In this case, there are only two components: a parent `View` which contains a `Text` welcoming us to React Native.
- **Styles:** style and graphic design of every component are placed in a stylesheet to separate the logic from the views. In fact, the styles could be placed in a separate file, and even one developer can work on the logic while a graphic designer takes care of the user interface. This is one of the features that makes React Native easy to work with in large developer teams, keeping high cohesion and low coupling and producing high-quality projects.
- **Register the component:** every React Native component must be registered so it can be loaded into the JavaScript bundle. This is generally done by using `module.exports`, but in this case, as our component is the root component of the application and will server as its entry point, we must register it using `AppRegistry.registerComponent`.

5.4.2 Component lifecycle

React Native components have props and state, which must be set when the component is rendered for the first time, but can also be updated. To get a clear idea

on how the lifecycle works, we need to differentiate between the initial creation or mounting phase of a component, props and state triggered updates and the unmounting phase. In each scenario, a series of methods are called, which allow to configure, update or perform any necessary cleanup. These scenarios are briefly described next. To better separate what can and cannot be done in each scenario, a description of the methods that are executed is provided, along with a graph to clarify the order of execution.

- **Creating or Mounting:** methods called when mounting the component.
  - `getInitialState`: invoked once before the component is mounted to set the initial value of `this.state`.
  - `getDefaultProps`: invoked once and cached when the class is created, before any instances are created. Values not specified in `this.props` when creating an instance of the class will default to the ones inserted here.
  - `componentWillMount`: invoked once before the component is mounted. Modifying `this.state` in this method will not trigger another execution of the `render()` method, but instead, the component will be rendered with the updated state values.
  - `componentDidMount`: invoked once right after the component is mounted. The `componentDidMount()` of child components is invoked before that of parent components.
- **Updating:** methods called when updating the component.
  - `componentWillReceiveProps`: invoked when the component is receiving new props. This will trigger a render, but updating `this.state` will not queue an additional render.
  - `shouldComponentUpdate`: invoked before rendering when new props or state are received. Should be used to avoid rendering the component again when the developer is certaing that the update will not require another component render, therefore improving performance.
  - `componentWillUpdate`: invoked right before rendering the component when new props or state are received. Should be used to perform any required preparation before the update occurs.
  - `componentDidUpdate`: called inmediately after the component is updated.

- **Unmounting:** methods called when unmounting the component.
  - `componentWillUnmount`: called inmediately before the component is unmounted. Should be used to perform any necessary cleanup.

The updating scenario has been split in two different possibilities, totaling four scenarios where these methods can be called: Initial Render (Creation or Mounting), Props Change (Updating), State Change (Updating) and Component Unmount (Unmounting). A summary comparing them is shown in Figure 5.4.

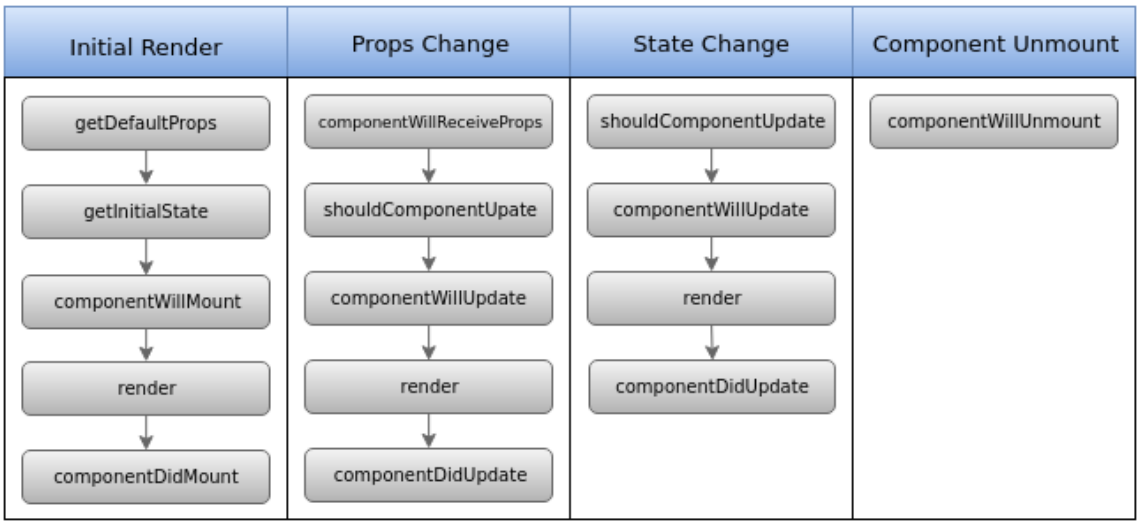

Figure 5.4: Component lifecycle scenarios summary

5.5 Risks and Drawbacks

The largest risk that comes with using React Native is its low maturity level. As a young project, React Native still contains some bugs and unoptimizations. However, this risk is somewhat offset by the fact that Facebook is using React Native in production for their own apps. The community is also constantly contributing to the project, and updates are released every two weeks. Nevertheless, working with bleeding edge techolonogy does mean that you can experience some trouble that will get fixed over time.

Another major drawback is what happens when you encounter a limitation in React Native's native supported API's. If you are only comfortable in JavaScript, the learning curve of programming languages such as Java or Objective-C is very steep, and could potentially delay a project.

CHAPTER 6

Test Scenario

6.1 Introduction

This test scenario was designed for newcomers who want to know how to start developing a React Native app. It is based on the Cronometer example developed in PhoneGap from the subject Computación en Red. The app will be developed for Android , and will fearute a very simple cronometer: the user will be able to increase and decrease the amount of time, and start and stop it. The PhoneGap example displays a single view, but this is quite unrealistic, as most current apps have more than one view. Therefore, navigation between views will be added to the example.

The project will be devidided in three stages:

1. **Introduction:** how to create a new project, file system and other important concepts.
2. **Creating the first view:** a simple cronometer view will be implemented in this stage step by step.
3. **Results:** show the results obtained in the test scenario.

React Native is still a young project in constant development. Currently, installing and getting it to work can be a little tricky, but a good installation guide

is available here [36]. A very detailed guide has not been included in this work as teaching how to get React Native to work is not its purpose, but instead learning how to develop a hybrid mobile application.

6.2 Introduction

The next tools are necessary to develop an Android app in a Linux environment. Note that how to install them is explained on the tutorial previously mentioned. However, there are a few things missing that on some machines might be required. Because of that, I recommend running the following commands prior to installing anything:

```
sudo apt-get install -y python-dev
sudo apt-get install -y automake
```

Listing 6.1: Missing Guide Dependencies

- **Required:**
  - Git: free and open source distributed version control system. Needed to clone Git projects and to install some of the other tools. [20]
  - Node and Node Package Manager (NPM): NPM is a package manager for JavaScript that allows developers to find and share and packages of code. [29] [30]
  - React Native Command Line Tools (CLI): the React Native CLI allow to create, initialize, update, debug and deploy and application. [34]
  - Java Development Kit 1.8: React Native requires JDK 1.8 or higher. I recommend the default version from OpenJDK, which can be installed running `sudo apt-get install -y default-jdk`. [25]
  - Android Studio: provides both the Android SDK Tools and an Android emulator (see below). The first ones are needed to compile and generate the android app, while the latter provide a way to quickly see how the mobile app works and feels on a real device. [6]
- **Recommended:**
  - Watchman: system designed to watch files and record when they change. It can also trigger actions when a change is made. [41]
  - Flow: static type checker, designed to quickly find errors in JavaScript applications. Current version is 0.27, but React Native uses 0.25. This

version mismatch may cause an error, which can be fixed by running `npm install -g flow-bin@ReactNativeVersion`. As the current React Native version is 0.25, we would substitute *ReactNativeVersion* with 0.25, but this version is subject to changes as new versions are released. [16]

  - Nuclide: IDE that provides a first-class development environment for writing, running and debugging React Native applications. Currently available for Max and Linux. [31]

- **Emulator:**
  - Android Virtual Device (AVD): stock Google Android device emulator that comes with Android Studio. Usually performs well, but can inexplicably fail to run on some systems and sometimes performs badly. [7]
  - Genymotion: altough Android Studio provides a way to emulate a real device, I recommend using Genymotion as it tends to be more stable, runs smoother and is easier to install as well. There are many tutorials for installing it, but I personaly recommend this one [19]. Genymotion requires VirtualBox to be installed on your system, so I recommend running this command before: `sudo apt-get install -y virtualbox`. [18]

6.2.1 Creating the project

The first thing to do is to create a new project. We can do this by opening a terminal, moving to the desired folder and running the following commands:

```
1 react-native init Cronometer
2 cd Cronometer
```

Listing 6.2: React Native project folder

This will create a sample project with all the required files and configuration in a folder named *Cronometer*, and move us to the project folder. This folder will contain a series of folders and files very similar to the ones shown in Figure 6.1. The whole process might take a few minutes to complete.

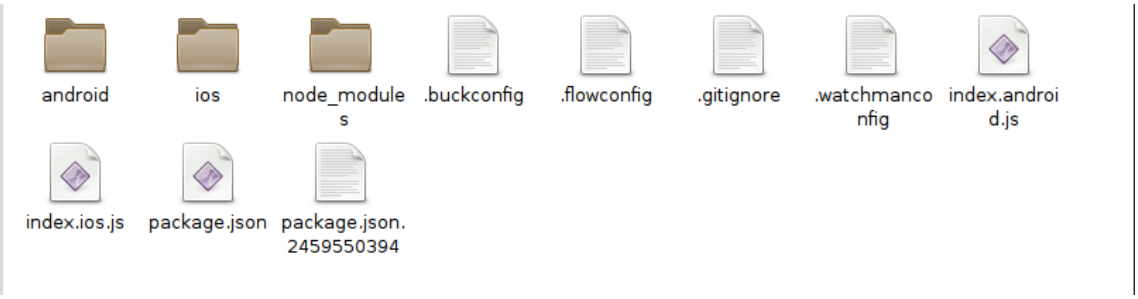

Figure 6.1: Project folder

- **Folders**
  - android: folder containing the generated code for the Android app. Inside we can find Gradle [22] configuration, Java file sources and the apk generated.
  - ios: folder containing the generated code for the iOS app. Inside we can find all the necessary information to open the project in XCode [44] and to generate the app.
  - node_modules: contains the required node modules, including React Native ones among others.
- **Files**
  - index.android.js: main file for the Android app. Here we will start writing our code before spreading it over multiple files as our project grows.
  - index.ios.js: main file for the iOS app. In this project this file will be left unmodified because it is not required for the Android app.
  - package.json: file containing vital information about the project like the own project name, dependencies, versions, and a long etc. Current dependencies are `react` and `react-native`, but if we want to add new modules and automatically include them in this file, we should add the `--save` option when running a `npm install` command. For this project, hoewever, the two dependencies mentioned will suffice as they provide all the core components and functionalities for a mobile application.

6.2.2 Running the app for the first time on the emulator

To check that the project we just created runs fine and mistakes were made, we are going to try it out on the Android device emulator. Run your device, either from the Android Virtual Device window or from Genymotion. Once the emulator has

finished loading, it is time to try our app. Open two more terminals and navigate to the project folder in both of them. Type the following:

First Terminal: `react-native start`.

Second Terminal: `react-native run-android`

On the first terminal we start the packager, which bundles the whole application as a single JavaScript file and delivers it when receiving a GET request on port 8081. We can take a look at the bundle by typing `http://localhost:8081/index.android.bundle` on the browser. The second one tells React Native to run the Android version of our project. It automatically installs the application on the virtual device, which requests the bundle. If everything is ok, one should see something similar to what is shown on Figure 6.2.

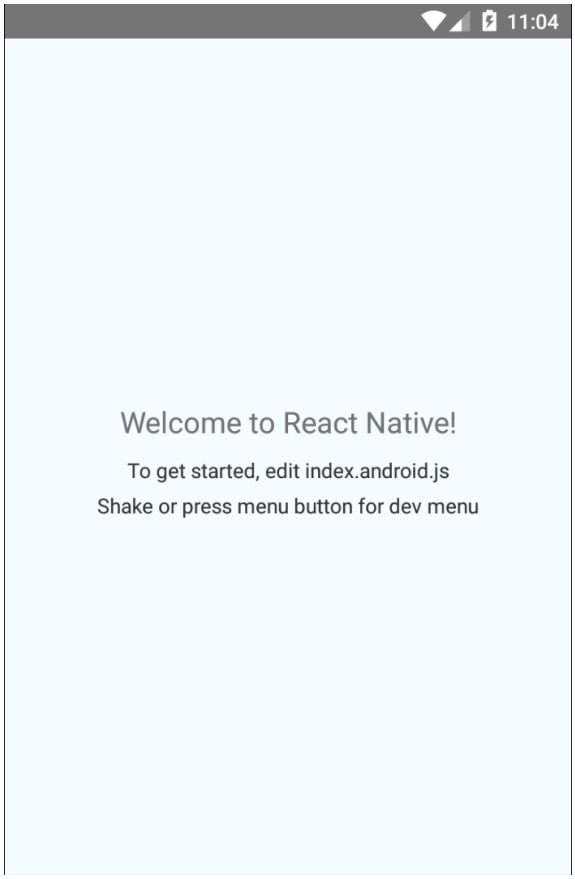

Figure 6.2: First view of the app

To open de developer tools in the emulator, simply press Control + M on Linux or Control + ⌘+ Z on Mac. However, the key-bindings may vary between systems, so for more information click here [35].

6.3 Creating the first view

In this work, Nuclide was used as the IDE, but any text editor will do the trick. However, using Nuclide is highly recommended since it was designed for developing React Native apps and enables many features like Flow that otehrwise would be useless. Flow provides type checking, which is helps avoiding errors and other nuisances. It also allows tabbing differente files, global searches and working on a remote server, among other features.

In case we are using Nuclide, open it and click on *Add Project Folder*, add our project folder and double click on the file *index.android.js*. The code we can see is the default application that React Native creates when Initializing a project. As we can see, the structure is the same as the described on Paragraph 5.4.1. There are a few imports, a single class which comprises the whole view shown in Figure 6.2 and some styling.

Before running around like a headless chicken, it is highly recommended to spend a few minutes thinking about what are we going to do and how. For this purpose, we can use a very simple class diagram, based on the Unified Modeling Language (UML) [39] [46].

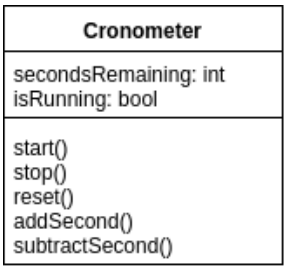

Figure 6.3: First class diagram

As we can see in Figure 6.3, we will only need one class, named *Cronometer*. This class will have two attributes and five methods, which names are self-explanatory. However, when working with React Native and mobile applications, we must also think about the user interface and how the user will interact with out application. A quick example: if we want to start and stop the cronometer, we could place two buttons, one to start it and one to stop it; but we could also use the same button for both actions, depending on the current state of the cronometer. This way, when the cronometer is not running, the button could read *Play*, and once the user pressed it and the cronometer would start running, the text could switch to *Stop*, and vice versa.

To implement a cronometer, we will need some kind of timer to measure seconds. React Native does include some very basic and low-level timing functions, but are hard to use and tend to cause trouble. For this reason, we are going to make use of a module available through NPM. To install it, open a terminal, navigate to the project folder and type `npm install --save react-native-timer`. This will install the package and directly save it into the *package.json* as a dependency.

Now, lets get out hands dirty and start modifying our app. Except otherwise stated, we will only modify the *index.android.js*. Firstly, we will modify the imports to include a few new components and the timer module we downloaded. Logically, using a module which has not been imported will crash the app. The resulting code is shown on Figure 6.4.

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

const timer = require('react-native-timer');
```

Listing 6.3: Imports

Now, lets move to the core class of our app. Our *Cronometer* class extends the *Component* class from React, so we can use all the methods inherited such as the lifecycle methods explained on Chapter 5 and the `render()` method, and add our own to the app's logic.

```
class Cronometer extends Component {
  constructor(props) { ... }
  componentDidMount() { ... }
  measureCronoImage() { ... }
  onSubtractPressed() { ... }
  onAddPressed() { ... }
  onNextOpButtonPressed() { ... }
  startCrono() { ... }
  stopCrono() { ... }
  render() { ... }
}
```

Listing 6.4: Application's logic summary

Our app will have nine methods which are described next.

6.3.1 constructor(props) and componentDidMount()

```
constructor(props) {
  super(props);
  this.state = {
    play: false,
    time: 0,
    x: 0,
    y: 0,
    w: 0,
    h: 0,
  }
}

componentDidMount() {
  setTimeout(this.measureCronoImage.bind(this));
}
```

Listing 6.5: constructor(props) and componentDidMount()

React Native uses the ES6 class syntax [45] along with an API very simmilar to `React.createClass`. However, nstead of providing a separate `getInitialState` method, you set up your own state property in the constructor. `propsTypes` and `defaultProps` are defined as properties on the constructor as well.

Invoking `super(props)` is mandatory in the constructor so the props are properly set. After that, we set the state with 6 variables:

- play: boolean, records if the cronometer's state: `false` means it is stopped, while `true` means it is running.
- time: records the amount of seconds left.
- x, y, w, h: x coordinate, y coordinate, width and height respectively. React Native uses flexbox, among other polyfills, to create adaptable user interfaces that keep their proportions independently from the device screen size (see Paragraph **??**). This variables will be used to store the clock sphere image position and size, se we are able to place the clockhand right in the middle.

The `componentDidMount()` method will be executed once, right after the component is mount. Its only purpose is calling the method `measureCronoImage()`, which will be explained next. However, this method is called from a `setTimer()`. At first glance, this may seem weird, but lets recall the component initialization in React (see Figure 5.4). We cannot measure a component that has not been already rendered, and this is done almost at the end, right before the `componentDidMount()` method. Ok, so we measure it after the user interface has been rendered. How-

ever, this leaves us with no clockhand at all until teh state changes, because all the state variables were assigned a value of 0, and no additional render is triggered in `componentDidMount()`. Therefore, we need to set a timer that calls the `measureCronoImage()` method right after the component initialization has finished, so it triggers another render and we can see the clockhand from the beggining.

6.3.2 measureCronoImage()

```
measureCronoImage() {
  this.refs.cronoImage.measureInWindow((a, b, width, height) => {
    this.setState({
      x: a,
      y: b,
      w: width,
      h: height
  })});
}
```

Listing 6.6: measureCronoImage()

This method measures the view contaning the clock sphere and updates the state variables, therefore triggering an update. The view is accessed through a reference set in the `render()` method (see Paragraph 6.3.6), and the `measureInWindow()` method is inherited from the Component class. It is important to note that axis coordinates in React Native work like is shown in Figure 6.4. It is also important to remmeber that, when giving a view absolute coordinates, these are relative to the parent view, and not to the root layout.

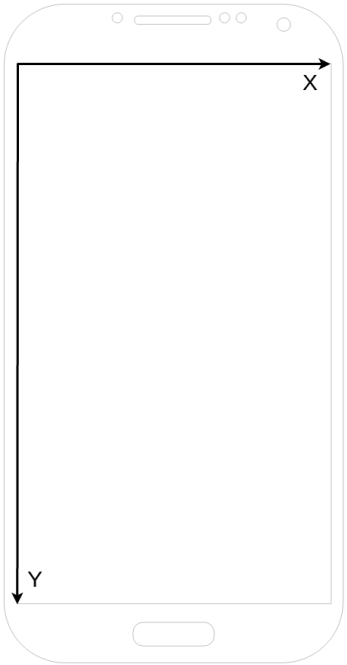

Figure 6.4: Coordinate axis in React Native

6.3.3 onSubtractPressed() and onAddPressed()

```
onSubtractPressed() {
  var time = this.state.time;
  (time > 0) ? this.setState({time: time-1}) : null;
}

onAddPressed() {
  this.setState({time: this.state.time+1});
}
```

Listing 6.7: onSubtractPressed() and onAddPressed()

`onSubtractPressed()` will be called when the subtract button is pressed, while `onAddPressed()` does the opposite. Each time one of them is called, they modify `this.state.time` and an update is triggered, rendering the view again and displaying the changes.

6.3.4 onNextOpButtonPressed()

```
onNextOpButtonPressed() {
  if(this.state.time == 0) {
    return;
  }
  var play = !this.state.play;
  this.setState({play: play});
  (play == true) ? this.startCrono() : this.stopCrono();
```

```
}
```

Listing 6.8: onNextOpButtonPressed()

This method starts or stops the cronometer depending on the current state. The methods in charge of doing it are described in the next paragraph. In case that time left has already reached 0, it does nothing; otherwise, it simply negates the current value of `this.state.play` and calls the appropirate method.

6.3.5 startCrono() and stopCrono()

```
startCrono() {
  timer.setInterval('crono', () => {
    this.setState({time: this.state.time-1});
    if(this.state.time == 0) {
      this.setState({play: false});
      Alert.alert("DING DING DING");
      setTimeout(this.stopCrono.bind(this));
    }
  }, 1000);
}

stopCrono() {
  timer.clearInterval('crono');
}
```

Listing 6.9: startCrono() and stopCrono()

`startCrono()` sets and interval which decreases the amount of time left by 1 second and checks if the amount if time left is 0. In that case, it shows an alert and stops the cronometer.

`stopCrono()` is a very simple method as it only clears the timer, therefore stopping the cronometer.

6.3.6 render()

```
render() {
  //Preparation and container
  var time = (this.state.time < 10) ? "0"+this.state.time+"s" : this.state.
    time+"s";
  return (

    <View style={styles.container}>

{/* Title */}
      <View style={styles.titleContainer}>
        <Text style={styles.title}>Temporizador</Text>
```

```
      </View>

{/* Add and subtract buttons and seconds remaining */}
      <View style={styles.bar}>
        <TouchableOpacity style={styles.barButton}
          onPress={this.onSubtractPressed.bind(this)}>
          <Text style={styles.buttonText}>-</Text>
        </TouchableOpacity>
        <Text style={styles.time}> {time} </Text>
        <TouchableOpacity style={styles.barButton}
          onPress={this.onAddPressed.bind(this)}>
          <Text style={styles.buttonText}>+</Text>
        </TouchableOpacity>
      </View>

{/* Clock sphere */}
      <View style={styles.cronoContainer}>
        <Image
          ref="cronoImage"
          style={styles.cronoImage}
          source={require('./img/crono.png')}/>
      </View>

{/* Clochand container */}
      <View style={{
        top: this.state.y+this.state.h/5,
        left: this.state.x+this.state.w/2-10,
        position: 'absolute',
        transform: [
           {rotate: (this.state.time*6)+'deg'}
        ]
      }}>

{/* Clockhand */}
        <View style={{
          width: 20,
          height: this.state.h*3/5,
          borderWidth: 1,
          borderTopLeftRadius: 5,
          borderTopRightRadius: 5,
          borderColor: '#111111',
       }}
       />
       <View style={{
          width: 6,
          height: 20,
          top: 0,
          left: 7,
          backgroundColor: 'red',
```

```
60            position: 'absolute'
61          }}
62          />
63        </View>
64
65 {/* Start/stop button */}
66        <View style={styles.bottomBar}>
67          <TouchableOpacity
68            style={styles.nextOpButton}
69            onPress={this.onNextOpButtonPressed.bind(this)}>
70            <Image source={(this.state.play) ? require('./img/pause.png') :
    require('./img/play.png')}
71              style={styles.nextOpButtonImage}
72            />
73          </TouchableOpacity>
74        </View>
75
76 {/* End container */}
77      </View>
78    );
79 }
```

Listing 6.10: render()

In the render method is where the user interface is implemented. It is by far the largest method, and so it was split in sections using comments. React Native uses nested components and JSX syntax, so you just need to put `{ }` around a comment when you are within the children section of a tag.

Preparation and container: `this.state.time` is read. The only view in this fragment is the root parent View, which fills the screen and contains all of the other elements of the user interface. Following views will be placed by default one below another, filling the space specified in the StyleSheet descibed in Paragraph 6.3.7.

Title: a single View containing a Text.

Add and subtract buttons: another View contains three elements: two TouchableOpacity elements and one Text. The simplest one is, of course, the Text, which displays the value of the variable created at the beggining of the method and contains the time remaining. In this case, the parent `style` has been defined to place its elements in a row, instead of a column. There are four kinds of elements for making views respond properly to touches, but it is highly recommended to stick with two first ones.

- TouchableOpacity: on press down, the opacity of the wrapped view is decreased, dimming it. This is done without actually changing the view

hierarchy, and in general is easy to add to an app without weird side-effects. Should be the first choice when implementing a touch-sensitive view.

- TouchableHighlight: on press down, the opacity of the wrapped view is decreased, which allows the underlay color to show through. This is done by adding a view to the view hierarchy, which can sometimes cause unwanted results if not used correctly.
- TouchableNativeFeedback: similar to the previous ones, but uses native state drawable to display touch feedback. Should only be used when there are no alternatives.
- TouchableWithoutFeedback: provides touch responsiveness without visual feedback, which is why using it is strongy discouraged.

In this case, the first TouchableOpacity element is the subtract button, and calls the `onSubtractPressed()` method when pressed; while the second TouchableOpacity element is the add button, which calls the `onAddPressed()` method. Inside each of them there is a Text element with a plus or minus symbol.

As a side note, simply pointing out that these four elements onyl support a single child view.

Clock sphere: this fragments contains a single View and an Image element which fills the whole view. A reference has been created to make the it visible to the rest of the class and be able to measure it in the `measureCronoImage` method. The Image source can come from the hybrid app's resources or from the network. In this case we are working with an image from the resources, as we can see in the line `source=require('./img/crono.png')`. To load a network image, is as simple as using a URI, e.g `source={{uri:'https://goleta.etsit.upm.es/actas/images/escudos/logoescuela.jpg'}}`

Clockhand container: parent view for the clockhand. In this case, the positiong is absolute to the parent view, which in this case is the root layout view as well. Its style has been directly defined because it depends on the current state. This view contains two more, the clockhand and the red tip, so when we rotate the parent view within the `transform` section, the whole view will rotate as if it was a single one. This saves us from the task of calculating the right position for both of the children views and rotating them independently.

Clockhand: made of two different views: the clockhand itself is the first one, while the latter will render the red tip. Again, like their parent view, their

positioning is absolute, but in this case it is absolute to their parent view, instead of the root layout view.

Start/stop button: finally, another view contains the TouchableOpacity element that acts as the Start/Stop button. When pressed, it will trigger the `onNextOpButtonPressed()` method that was detailed on Paragraph 6.3.4. The source image depends on the current state of the cronomenter.

6.3.7 Styles

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  title: {
    fontSize: 25,
    textAlign: 'center'
  },
  bar: {
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  barButton: {
    borderColor: 'gray',
    borderWidth: 1,
    height: 20,
    width: 20,
  },
  buttonText: {
    textAlign: 'center',
  },
  time: {
    fontSize: 20,
  },
  cronoContainer: {
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  cronoImage: {
    height: 300,
    width: 300,
    justifyContent: 'center',
    alignItems: 'center',
  },
  bottomBar: {
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  nextOpButton: {
    margin: 5,
  },
  nextOpButtonText: {
    fontSize: 20,
  },
  nextOpButtonImage: {
    width: 75,
    height: 75,
  }
});
```

Listing 6.11: Application's StyleSheet.

On Listing 6.3.7 the styles that each user interface view will adopt are specified. Some of those tags are common with HTML, but some others like `flex` or `flexDirection` will not. Therefore I will not go into detail with most of the attributes, but instead will describe some of the polyfills that React Native uses.

A polyfill is a piece of code that implements a feature that is not supported. React Native uses them to implement features that are available for web development. These libraries can be installed using `npm install`. Current imlemented polyfills are the following:

- **Flexbox:** by far the most useful and used, 100% apps written in React Native use it. Flexboxmodule (currently a W3C Last Call Working Draft) aims at providing a more efficient way to lay out, align and distribute space among items in a container, even when their size is unknown and/or dynamic. [15]

  The main idea behind the flex layout is to give the container the ability to alter its items' width/height (and order) to best fill the available space (mostly to accommodate to all kind of display devices and screen sizes). A flex container expands items to fill available free space, or shrinks them to prevent overflow.

  Flexbox properties are separated in two categories, depending on the type of element: parent properties and children properties. Obviously, the parent or flex container contains children or flex items. Some of the most importante properties are:

  - flex-direction (parent): establishes the main axis, defining the direction that flex items will be placed in the flex container.
  - justify-content (parent): defines the alignment along the main axis, be it in the left, center, right, equally spaced...
  - align-items (parent): defines the default behaviour for how flex items are laid out along the cross axis on the current line.
  - align-content (parent): aligns a flex container's lines within when there is extra space in the cross-axis.
  - align-self (child): this allows the specified alignment to be overridden for individual flex items.

- **ShadowPropTypesIOS:** enables the UIView shadow properties.
- **Geolocation:** available for Android and iOS. Enables the use of native geolocation functions.
- **Network:** one of the most important APIs. It supports REST requests, full-duplex communication over TCP and an XML request method.
- **Timers:** this API has already been used in this project, so there is not much more to say. Aside from the timers, it also implements an `InteractionManager`

which can be used to schedule expensive operations and avoid delaying active animations.

- **Colors:** quite self-explanatory, it enables a dozen commands for color declaration and over a hundred pre-made colors.[?]

6.4 Results

A simple mobile application has been developed using React Native. All the code has been uploaded to a GitHub repository and is available for download here [21].The source images can be found in the folder *img* inside the project. The results should be the following:

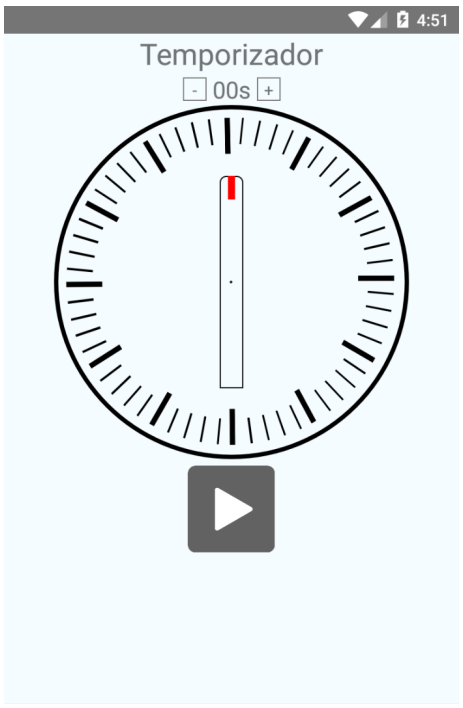

Figure 6.5: Cronometer stopped

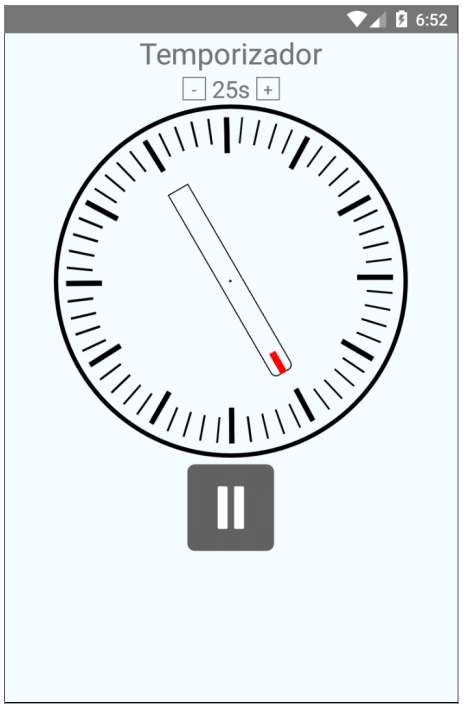

Figure 6.6: Cronometer running

When the user presses de + button, the clockhand will rotate clockwise, and vice versa when the user presses the - button. Clicking on the play button will start the crono, and clicking again will stop it.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

This project has allowed us to take a deep insight into React Native. We have learnt the differences between the two main current technologies for developing hybrid mobile applications and their inner workings.

React Native's architecture has been explained in detail to ease the understanding of the technology as a whole. Runtime execution and lifecycles were detailed so a new developer can immediately start writing code and testing her app without unpleasant and unexplained bugs related to the execution order of the code.

We have also developed a functional React Native app from scratch and tested it on an Android device. Other key aspects such as React Native APIs or syntax have been addressed as well.

From my own experience as a mobile applications developer, the results attained with React Native are quite remarkable. Although the learning curve is steeper than PhoneGap's, the fact that everything the user sees and experiences is native makes React Native a decisive advantage.

I have also had my own share of native developing for Android, and is obviously more powerful than React Native. But you can also spend countless hours trying to implement a view that only takes half an hour on React Native. The native

environment can sometimes be a little hostile, but developing in React Native has turned out as a satisfying and even fun activity. In addition, React Native's reusable components increase productivity, reward contributing to repositories and sharing your own work with the community.

React Native is still a young project in constant development, so anyone interested in developing an application should keep up to date. Myself, when I started this work, React Native's version was 0.18. Guides, resources and explanations were scarce and hard to find, but the situation is improving. Today, on June 21st, it is 0.27, and another release is scheduled for this week. Detailed guides have been added, example applications and more and more components and APIs are included with every update.

We can say without a doubt that React Native is a major improvement in the field of hybrid mobile applications. It is a techonology meant for most apps, which adds compatibility between the two biggest systems in the market without almost sacrificing performance. We will be seeing more and more apps developed using React Native in the near future.

7.2 Achieved Goals

In Chapter 1 a list of goals was mentioned. The achieved goals can be summarized as follows:

Comparing current mobile application development options: an introduction to this subject was made in Chapter 2 and expanded on Chapter 3.

Understanding what React Native: React Native is based on React, which was introduced on Chapter 4. After that, we moved to React Native's key features, architecture, processes, runtime, call cycle, etc. This goal was successfully achieved.

Design and build a simple test scenario to demonstrate React Native's capabilities: this goal has been achieved in Chapter 6, where a simple app was developed in React Native, detailing the whole process.

7.3 Future Work

This project can serve as an introduction to React Native and as a solid foundation for future mobile applications. These are some of the areas where it be useful:

- Analyze the impact of React Native on other technologies.
- Analyze the amount of hybrid mobile applications developed with React Native.
- Expand the test scenario with navigation between views and integration with native modules.
- Develpment of a wide range of mobile applications related to other projects, in fields such as home automation, business, social networking, smart-cities, etc.
- Integration with different subjects like Computación en Red or Ingeniería Web.

  React Native's maturity level ys still low, and as it developes, new functionalities and tools will be added. All of these will result in new ideas for future lines of work.

Bibliography

[1] Adobe PhoneGap. www.phonegap.com.

[2] Adobe PhoneGap Build. https://build.phonegap.com/.

[3] Adobe PhoneGap Plugins. https://build.phonegap.com/plugins.

[4] Amazon Fire OS. https://developer.amazon.com/public/solutions/platforms/android-fireos.

[5] Android. https://www.android.com.

[6] Android Studio. https://developer.android.com/studio/index.html.

[7] Android Virtual Device. https://developer.android.com/studio/run/managing-avds.html.

[8] Android WebView Reference. https://developer.android.com/reference/android/webkit/WebView.html.

[9] Apache Cordova. https://cordova.apache.org/.

[10] Blackberry 10. http://global.blackberry.com/es/software.html.

[11] Chrome Developer Tools. https://developer.chrome.com/devtools.

[12] Document Object Model Specification. https://www.w3.org/DOM/.

[13] Facebook Developer Conference. https://www.fbf8.com/.

[14] Firefox OS. https://www.mozilla.org/en-US/firefox/os/.

[15] Flexbox Reference. http://www.w3schools.com/css/css3_flexbox.asp.

[16] Flow. https://flowtype.org/.

[17] Flux. https://facebook.github.io/flux/.

[18] Genymotion. https://www.genymotion.com/.

[19] Genymotion Installation Guide. http://techapple.net/2014/07/tutorial-installsetup-genymotion-android-emulator-linux-ubuntulinuxmintfedoraarchlinux/#.

[20] Git. https://git-scm.com/.

[21] GitHub repository of the project. https://github.com/IgnacioMV/Cronometer.

[22] Gradle. https://docs.gradle.org.

[23] iOS. http://www.apple.com/ios/.

[24] iOS UIWebView Reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/.

[25] Java Development Kit 1.8. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

[26] JavaScriptCore Reference. https://developer.apple.com/library/tvos/documentation/Carbon/Reference/WebKit_JavaScriptCore_Ref/index.html.

[27] JSX. https://facebook.github.io/jsx/.

[28] Mobile/Tablet Operating System Market Share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1&qpsp=2016&qpnp=1&qptimeframe=Y.

[29] Node. https://nodejs.org/en/.

[30] Node Package Manager. https://www.npmjs.com/.

[31] Nuclide. https://nuclide.io/.

[32] React. https://facebook.github.io/react/.

[33] React Native. https://facebook.github.io/react-native/.

[34] React Native Command Line Tools. https://www.npmjs.com/package/react-native-cli.

[35] React Native debugging. https://facebook.github.io/react-native/docs/debugging.html.

[36] React Native Installation Guide. https://facebook.github.io/react-native/docs/getting-started.html.

[37] Tizen. https://www.tizen.org/.

[38] Ubuntu. http://www.ubuntu.com/.

[39] Unified Modeling Language. http://www.uml.org/.

[40] W3C Configuration Document Specification. https://www.w3.org/TR/widgets/#configuration-document-0.

[41] Watchman. https://facebook.github.io/watchman/.

[42] windows. https://www.microsoft.com/en-us/windows.

[43] Windows Phone. https://www.microsoft.com/en-us/windows/phones.

[44] XCode 8. https://developer.apple.com/xcode/.

[45] ECMA International. ECMAScript 2015 Language Specification, 2015. En Línea. Disponible en: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

[46] C. Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 2004.

[47] Tadeu Zagallo. React Native Architecture Overview. https://www.youtube.com/watch?v=Ah2qNbI40vE.