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.

Contents

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 taken 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. However, some of its own peculiarities make PhoneGapapps 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 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 and expose their advantages and disadvantages.
- Understanding what React Native is and how it works.
- 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, a JavaScript library for creating the user interface in a Model-View-Controller (MVC) architecture based web page.

Chapter 5:

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 performance, 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 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 Android, BlackBerry 10, iOS, Windows, Windows Phone, Symbian, Tizen and webOS. 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.

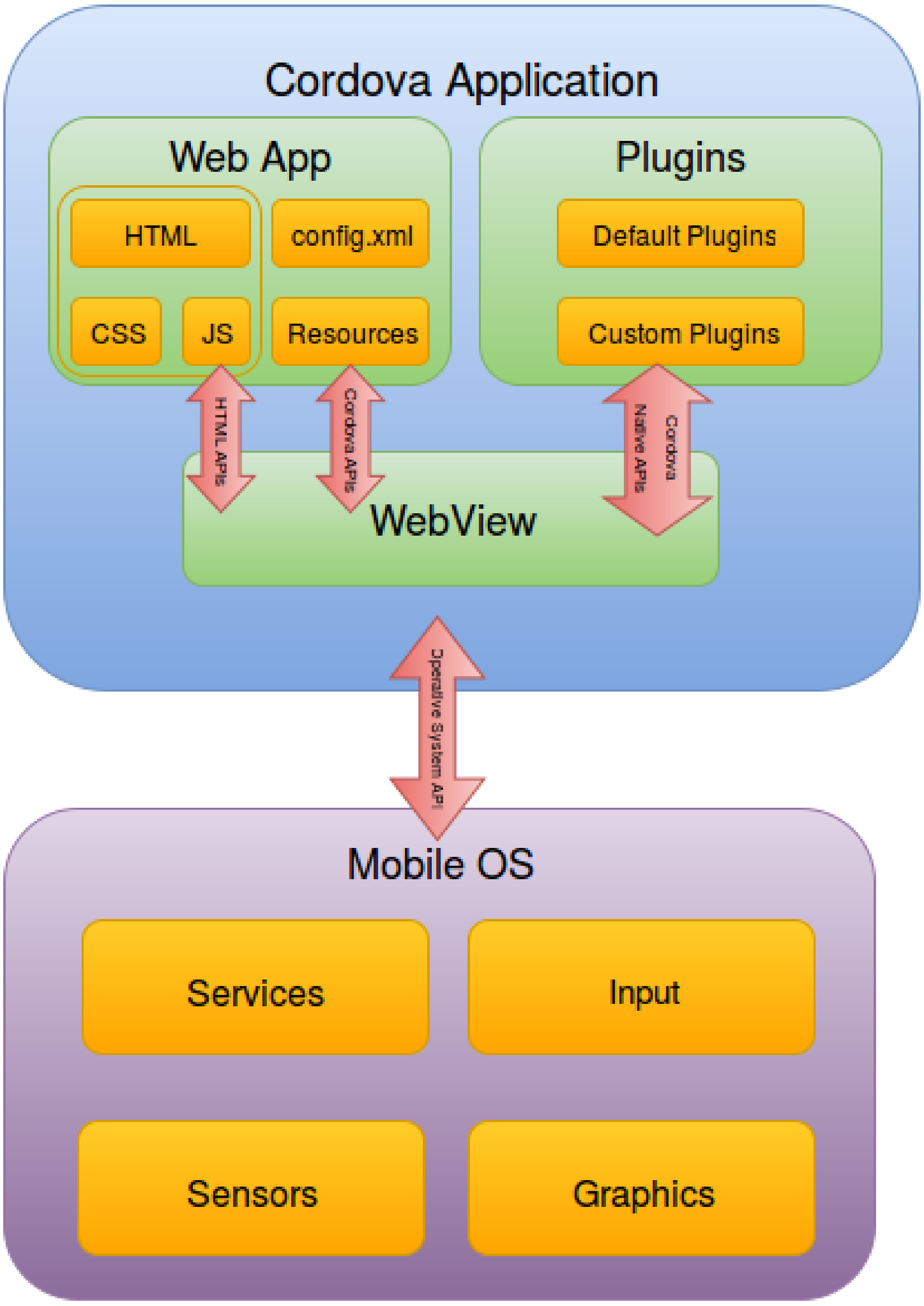

Figure 3.1: Cordova Architecture

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" is a 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 class, while on Android this is the android.webkit.WebView. 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). 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 99% of the market share: iOS, Android and Windows Phone.

3.3.3 Local and remote building

For most platforms, PhoneGap can compile a project locally, or can be sent to a PhoneGap Build server to be compiled remotely. Platform availability of each service is presented in the next table.

Platform	Android	Blackberry (6)	Blackberry 10	iOS	WP7	WP8	W8	Tizen	webOS	Symbian
Local Building	✓	✓	✓	✓	✓	✓	✓	✓		
Remote Building	✓	✓		✓	✓				✓	✓

Table 3.1: Platforms supported by PhoneGap's local and remote building

3.3.4 Plugins

A PhoneGap plugin 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.

Current native device features supported for each platform are shown in the next table:

Platform	Android	Blackberry (6)	Blackberry 10	iOS	WP7	WP8	W8	Tizen	webOS	Symbian
Accelerometer	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Camera	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Capture	✓	✓	✓	✓	✓	✓				
Compass	✓		✓	✓	✓	✓	✓	✓	✓	
Connection	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Contacts	✓	✓	✓	✓	✓	✓	✓			✓
Device	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Events	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
File	✓	✓	✓	✓	✓(No file transfer)	✓(No file transfer)	✓			
Geolocation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Globalization	✓	✓		✓		✓				
InAppBrowser	✓	✓	✓	✓	✓	✓				
Media	✓		✓	✓	✓	✓	✓	✓		
Notification	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spashscreen	✓		✓	✓	✓	✓	✓			
Storage	✓	✓	✓	✓	✓(local storage only)	✓(local storage only)	✓	✓	✓	

Table 3.2: Device native features currently supported for each platform.

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 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.

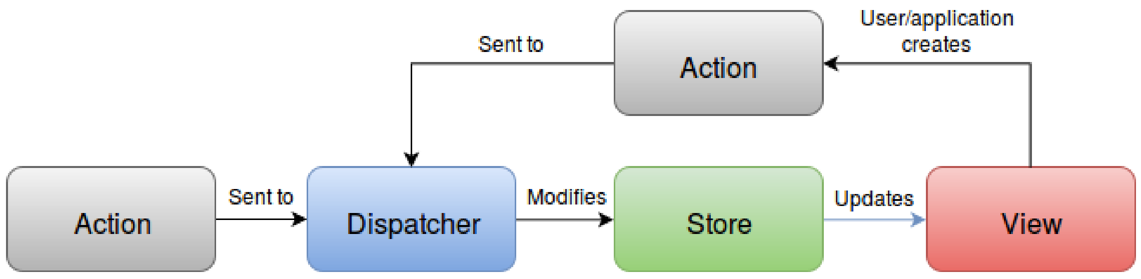

Figure 4.1: Flux unidirectional data flow

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 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. It has the following characteristics:

- **Faster code:** JSX code has to be compiled to JavaScript. However, the resulting code is faster than the equivalent code directly written in JavaScript, thanks to the optimizations performed by the compiler.
- **More strict:** JSX is mostly type-safe. This leads to many error being caught in the compiling process, and consequently not reaching the final product.
- **Easier:** JSX has a class system similar to Java so developers can avoid using the prototype-based inheritance system provided by JavaScript. However, expressions and statements remain almost the same, so it is easy for developers to switch to JSX.

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 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 World Wide Web Consorctium or 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.

For instance, consider the following HTML piece of code, and a graphical representation of the DOM that would be generated.

```
<table>
  <tr>
    <td>Text1</td>
    <td>Text2</td>
  </tr>
  <tr>
    <td>Text3</td>
    <td>Text4</td>
  </tr>
```

```
</table>
```

Listing 4.3: Sample HTML code

```
HTML
    HEAD
    BODY
        TABLE
            #text:
            TBODY
                TR
                    #text:
                    TD
                        #text: Text1
                    #text:
                    TD
                        #text: Text2
                    #text:
                #text:
                TR
                    #text:
                    TD
                        #text: Text3
                    #text:
                    TD
                        #text: Text4
                    #text:
                #text:
        #text:
```

Listing 4.4: DOM code generated

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.

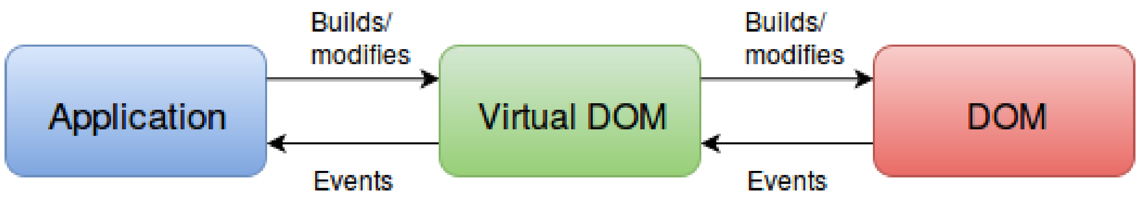

Figure 4.2: React interaction with Virtual and real DOM

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
    return (
      <div>
        //Renders the name stored in props
        <div>Hola {this.props.name}</div>
        //Renders elapsed time, stored in state
        <div>Tiempo transcurrido: {this.state.secondsElapsed}</div>
      </div>
    );
  }
});

ReactDOM.render(<ExampleComponent name="Juan"/>, mountNode);
```

Listing 4.5: 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 2015 by Facebook 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.

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 Main Goal

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. 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.

5.3 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 to debug the JavaScript code while running the complete app.

5.4 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:

<div>	→	<View>
<span>	→	<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.

5.4.1 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.

5.5 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.

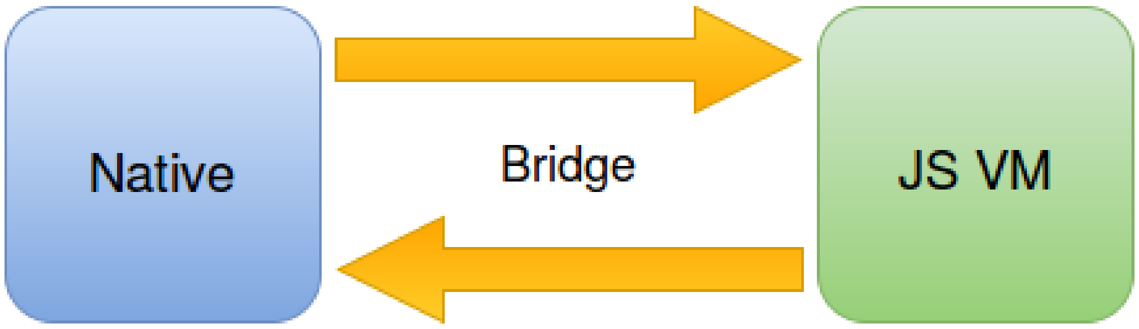

Figure 5.1: React Native basic architecture

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. 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.

5.5.1 Runtime

The problem of concurrence has been traditionally tacked 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.

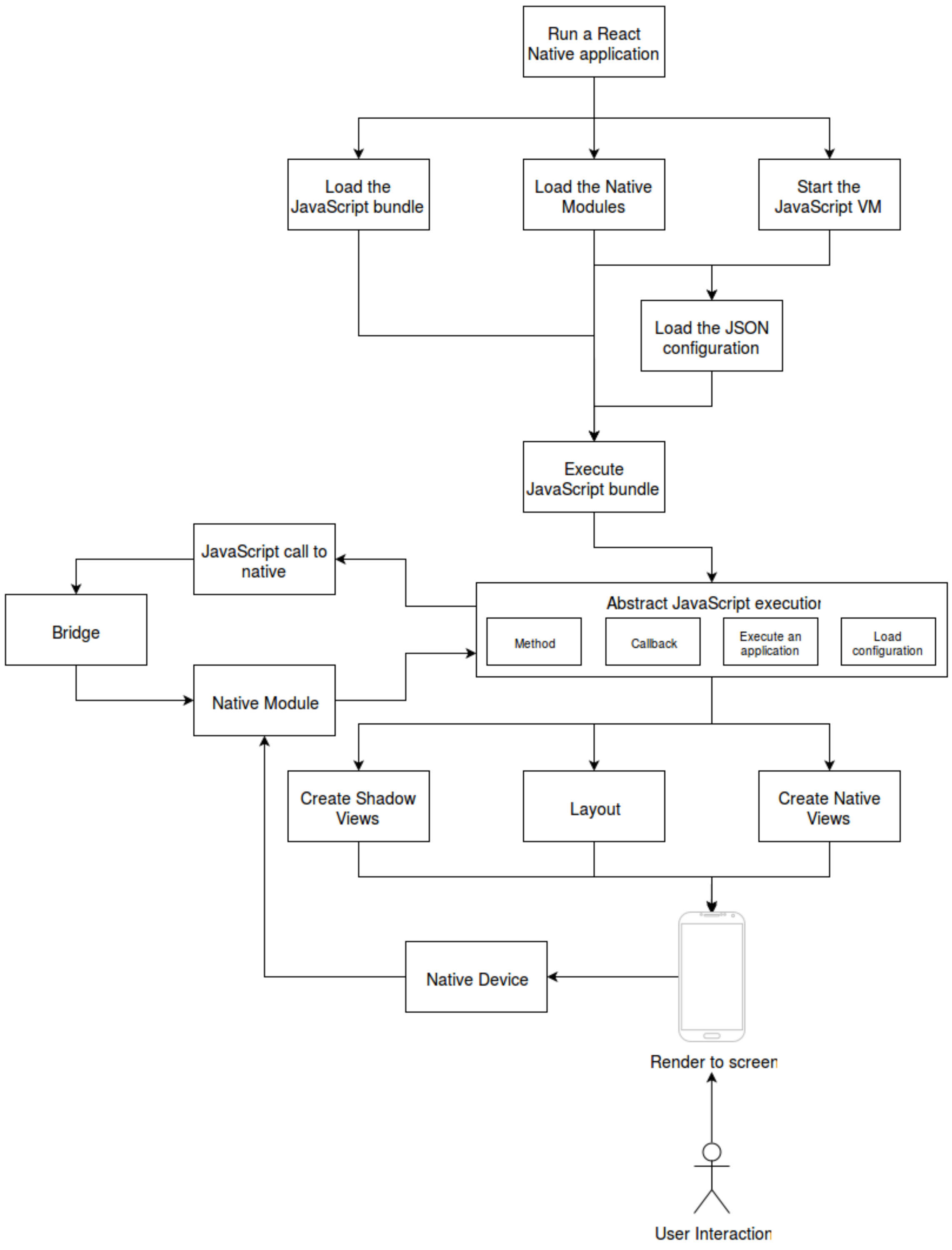

Figure 5.2: React Native application runtime graph

- **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 og the application.
-
-
-

-
-
-
-

5.5.2 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.

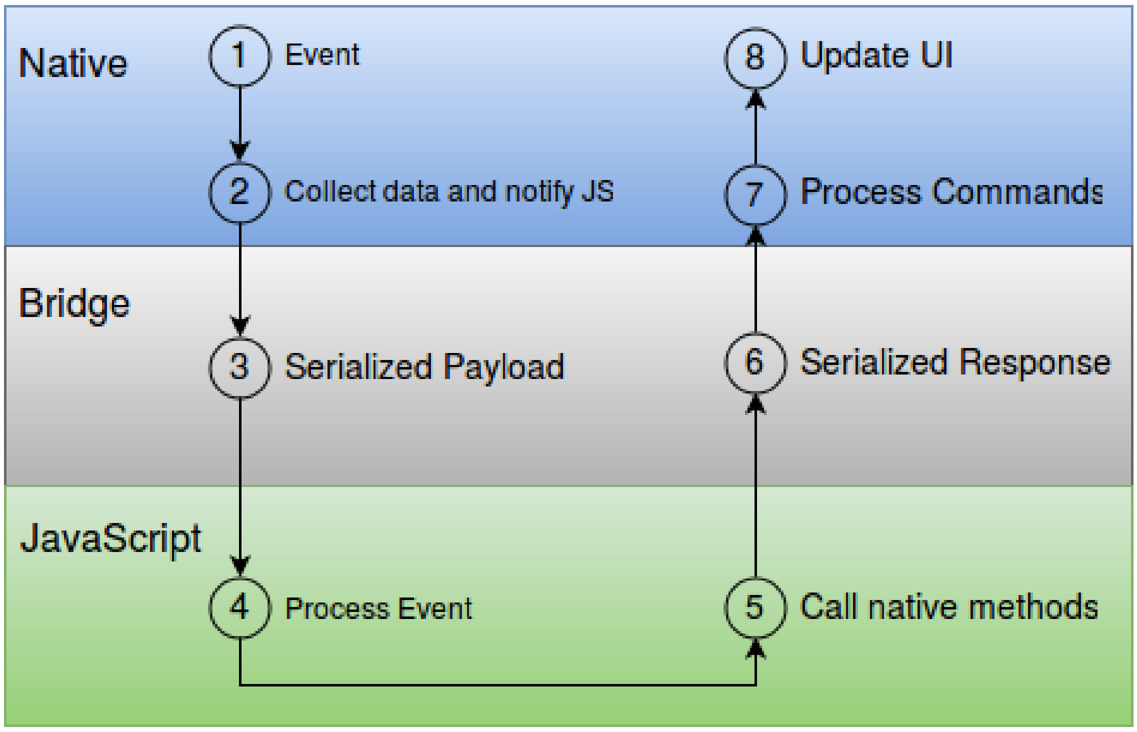

Figure 5.3: React Native app call cycle

CHAPTER 6

Test Scenario

6.1 Introduction

This test scenario was designed for newcomers who want to know how to develop a React Native app.

6.2 Dependencies

React Native makes use of a series of separate technologies that enable features like real-time update of the debugging mobile application, so the developer can get a quick preview of her work. In this work only development for Android in a Linux operative system will be considered.

6.2.1 React Native Command Line Tools

The React Native Command Line Tools (CLI) allow to create, initialize, update, debug and deploy and application.

6.2.2 Node Package Manager

As its name says, Node Package Manager or NPM is a package manager for JavaScript. It eases the task of installing, sharing and distributing code, manages dependencies in a project and helps giving and receiving feedback with others.

6.2.3 Gradle

Gradle is an open source general-purpose build automation system for the building, testing, publishing and deployment of software packages, websites, applications and others. It uses a domain-specific language based on Groovy. It takes the best features of other builing systems, such as Maven or Ant, and combines them into one.

6.2.4 Watchman

Watchman is a system designed to watch files and record when they change. It can also trigger actions when a change is made.

6.2.5 Flow

6.2.6 Android SDK

6.2.7 Nuclide

6.2.8 Genymotion

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

7.2 Achieved Goals

7.3 Future Work