MANNING

Developing iOS and Android Apps with JavaScript

React Native IN ACTION

MEAP

Nader Dabit

MEAP Edition
Manning Early Access Program
React Native in Action
Developing iOS and Android Apps with JavaScript
Version 1

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing *React Native in Action*! With the growing demand for app development and the increasing complexity that app development entails, React Native comes along at a perfect time, making it possible for developers to build performant cross platform native apps much easier than ever before, all with a single programming language: JavaScript. This book gives any iOS, Android, or web developer the knowledge and confidence to begin building high quality iOS and Android apps using the React Native Framework right away.

When React Native was released in February of 2015, it immediately caught my attention, as well as the attention of the lead engineers at my company. At the time, we were at the beginning stages of developing a hybrid app using Cordova. After looking at React Native, we made the switch and bet on React Native as our existing and future app development framework. We have been very pleased at the ease and quality of development that the framework has allowed the developers at our company, and I hope that once you are finished reading this book, you will also have a good understanding of the benefits that React Native has to offer.

Since I began working with React Native at its release, it's been a pleasure to work with it and to be involved with its community. I've spent much time researching, debugging, blogging, reading, building things with, and speaking about React Native. In my book, I boil down what I have learned into a concise explanation of what React Native is, how it works, why I think it's great, and the important concepts needed to build high quality mobile apps in React Native.

Any developer serious about app development or wanting to stay ahead of the curve concerning emerging and disruptive technologies should take a serious look at React Native, as it has the potential to be the holy grail of cross-platform app development that many developers and companies have been hoping for.

—Nader Dabit

brief contents

1

Getting started with React Native

This chapter covers

- Introducing React Native
- The strengths of React Native
- Creating components
- Creating a starter project

Native mobile application development can be complex. With the complicated environments, verbose frameworks, and long compilation times, developing a quality native mobile application is no easy task. It's no wonder that the market has seen its share of solutions come onto the scene that attempt to solve the problems that go along with native mobile application development, and try to somehow make it easier.

At the core of this complexity is the obstacle of cross platform development. The various platforms are fundamentally different and do not share much of the same development environments, APIs, or code. Because of this, you must have separate teams working on each platform, which is both expensive and inefficient.

This is where React Native stands out. React Native is an extraordinary technology. It will not only improve the way you work as a mobile developer, it will change the way you build and reason about mobile application development, and how you organize your engineering team.

This is a very exciting time in mobile application development. We are witnessing a new paradigm in the mobile development landscape, and React Native is on the forefront of this shift in how we build and engineer our mobile applications, as it is now possible to build native performing cross platform apps as well as web applications with a single language and team. With the rise of mobile devices and the subsequent increase in demand of talent driving

developer salaries higher and higher, React Native brings to the table a framework that offers the possibility of being able to deliver quality applications across all platforms at a fraction of the time and cost while still delivering a high quality user experience and a delightful developer experience.

1.1 Introducing React Native

React Native is a framework for build native mobile apps in JavaScript using the React JavaScript library. React Native has a lot going for it. Besides being backed and open sourced by Facebook, it also has a tremendous community of motivated people behind it. Facebook groups, with its hundreds of millions of users, is powered by React Native as well as Facebook Ads Manger. Discord and li.st are built with React Native and Discovery VR has harnessed the framework to build a beautiful and complex virtual reality video application. With React Native, developers can build native views and access native platform-specific components using JavaScript. This sets React Native apart from hybrid app frameworks, as hybrid apps package a web view into a native application and are built using HTML & CSS. React Native applications are built using JavaScript and JSX. JSX is something we will be discussing in depth in this book, but for now think of it as a JavaScript syntax extension that looks like html or xml.

To get started and begin understanding the flow of React Native, let's walk through a typical interaction and go over what happens in the React Native application flow:

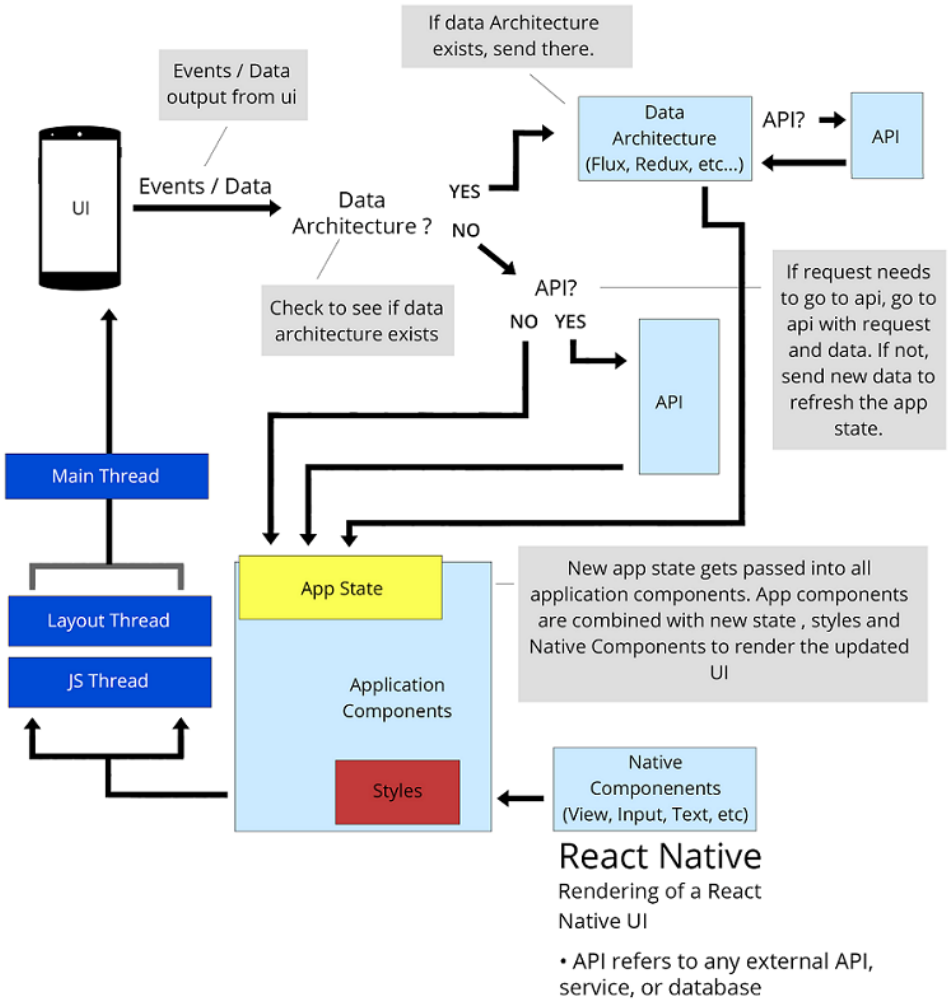

Figure 1.1 The program flow in a React Native program

We start with the app UI along with the current state of the app. Once a button is pressed, or an event is triggered, some data or state is passed into one of a few places: it either goes to an API, a data architecture layer (something like Redux, Flux, Mobx, or similar that holds the app state), or back to the app to update the state directly. The state is something we will be discussing in depth in this book, but it is basically just the current representation of your app (data, Boolean values, and so on being used in the app).

If there is a data architecture layer, something we will discuss in depth later but is essentially a library or pattern to manage complex state, it will make the necessary changes to

the data that is passed to it then communicate with the API if needed or send the data back to the update the main state of the app.

If there is no data architecture but there is an API that needs to be interacted with, a request is made, and when the response comes back it will be manipulated in the data architecture layer and then passed on to the main state of the app.

If there is no data architecture layer and no API, then the data will be either passed on to the main state of the app.

If there is no data architecture layer but there is an API, the request will go to the API. The response from the API will go directly back to the state of the app and supply the app with the new data that was returned from the API, updating the state of the app.

Before we go any further, let's set up a mock API that will return data and reset the state to see how all of this looks in React Native. We will not be getting into the data architecture piece just yet. Do not worry about understanding everything we are about to, as we will be discussing all of this in depth in the book.

Before we write any code, let's take a look at a diagram of the interaction we will be going over:

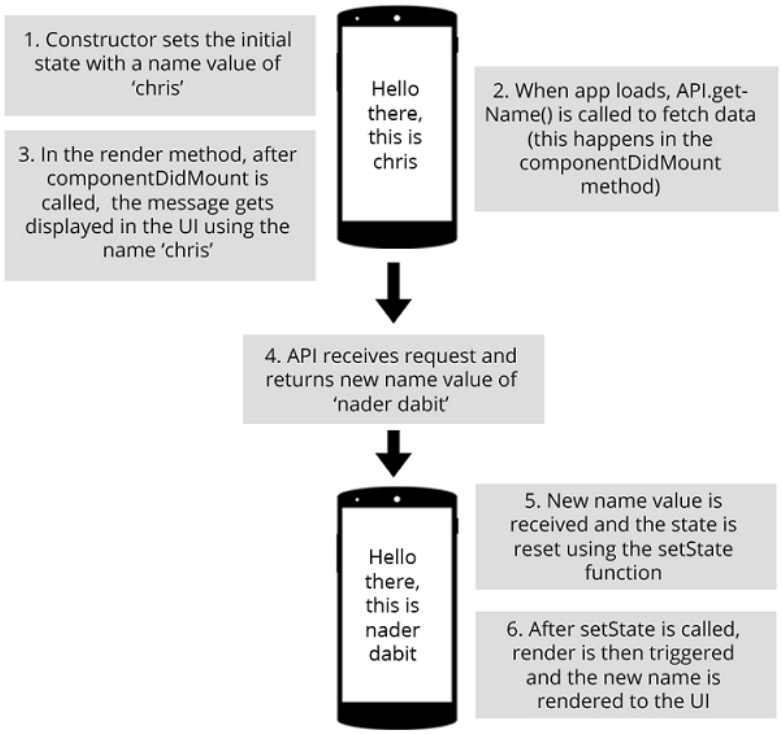

Figure 1.2 Mocking an API in React Native

To mock this functionality, we will have two files: a main entry file (index.ios.js or index.android.js) and a mock API (api.js).

Listing 1.1. Mocking the functionality of an API

Index.ios.js

```
import React, {
  AppRegistry,
  Component,
  StyleSheet,
  View,
  Text
} from 'react-native'
import API from './api'  (1)

class blankRNApp extends Component {
  constructor () {
    super()
    this.state = { name: 'chris' }  (2)
  }
  componentDidMount () {
    API.getName().then((data) => {  (3)
      this.setState({ name: data }) (5)
    })
  }
  render () {
    return (
      <View style={styles.container}>
        <Text>Hello there, {this.state.name}</Text>  (6)
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF'
  }
})

AppRegistry.registerComponent('blankRNApp', () => blankRNApp)
```

api.js

```
export default {
   getName: function () {
     return new Promise((resolve, reject) => { (4)
       setTimeout(() => {
         resolve('nader dabit')
       }, 3000)
     })
   }
```

```
}
```

1. Import the API into the file
2. Set value of name as 'Chris' in the initial state
3. Once the component is mounted / loaded, we go to the API and retrieve a new name
4. API returns a promise, once resolved it will give us the name 'Nader Dabit'
5. We set the state of `name` with our returned value from the API
6. The view displays `this.state.name`, when the API comes back with the new value, you will see this change from 'chris' to 'nader dabit'

1.2 What You Will Learn

In this book, we will cover everything you will need to know to build robust Mobile Applications in iOS and Android using the React Native framework.

Because React Native is built using the React library, we will begin by covering and thoroughly explaining how React works.

We will then cover styling, touching on most of the styling properties available in the framework. Because React Native uses flexbox for laying out the UI, we will dive deep into how flexbox works and discuss all of the flexbox properties.

We will then go through all of the native components that come with the framework out of the box and walk through how each of them works. In React Native, a component is basically a chunk of code that provides a specific functionality or UI element and can easily be used in the application. Components will be covered extensively throughout this book as they are the building blocks of a React Native application.

There are many ways to implement navigation, each with their nuances, pros and cons. We will discuss navigation in depth and cover how to build navigation using the most important of the navigation APIs. We will be covering not only the native navigation APIs that come out of the box with React Native, but also a couple of community projects available through npm.

After learning navigation, we will then cover both cross platform and platform specific APIs available in React Native and discuss how they work in depth.

It will then be time for us to start working data using network requests, asyncstorage (a form of local storage), firebase, and websockets.

After that we will dive into the different data architectures and how each of them works to handle the state of our application

Finally, we will take a look at testing and a few different ways to do so in React Native.

1.3 What You Should Know

To get the most out of this book, you should have a beginner to intermediate knowledge of JavaScript. Much of our work will be done with the command line, so a basic understanding of how to use the command line is also needed. You should also understand what npm is and

how it works on at least a fundamental level. If you will be building in iOS, a basic understanding of Xcode is beneficial and will speed things along, but is not absolutely necessary. Fundamental knowledge of newer JavaScript features implemented in the es2015 release the JavaScript programming language is beneficial but not necessary. Some conceptual knowledge on MVC frameworks and Single Page Architecture is also good but not absolutely necessary.

1.4 Understanding how React Native works

1.4.1 JSX

React and React native both encourage the use of JSX. JSX is basically a preprocessor step that adds an XML like syntax to JavaScript. You can build React Native components without JSX, but JSX makes React and React Native a lot more readable, easier to read, and easier to maintain. JSX may seem strange at first, but it is extremely powerful and most people grow to love it.

1.4.2 Threading

All JavaScript operations, when interacting with the native platform, are done on separate a thread, allowing the user interface as well as any animations to perform smoothly. This thread is where the React application lives, and all API calls, touch events, and interactions are processed. When there is a change to a native-backed component, updates are batched and sent to the native side. This happens at the end of each iteration of the event-loop. For most React Native applications, the business logic runs on the JavaScript thread.

1.4.3 React

A great feature of React Native is that is uses React. React is an open-source JavaScript library that is also backed by Facebook. It was originally designed to build applications and solve problems on the web. This framework has become extremely popular and used since its release, with companies such as Airbnb, Box.com, CodeAcademy, and Dropbox taking advantage of its quick rendering, maintainability, and declarative UI among other things. Traditional DOM manipulation is very slow and expensive in terms of performance, and should be minimized. React bypasses the traditional DOM with something called the 'Virtual DOM'. The Virtual DOM is basically a copy of the actual DOM in memory, and only changes when comparing new versions of the Virtual Dom to old versions of the Virtual DOM. This allows the minimum number of DOM operations needed to achieve the new state.

1.4.4 Diffing

React takes this idea of diffing and applies it to native components. It takes your UI and sends the smallest amount of data to the main thread to render it with native components. Your UI

is declaratively rendered based on the state, and React uses diffing to send the necessary changes over the bridge.

1.4.5 Thinking in components

When building your UI in React Native, it is useful to think of your application being as composed of a collection of components, and then build your UI with this in mind. If you think about how a page is set up, we already do this conceptually, but instead of component names, we normally use concepts, names or class names like header, footer, body, sidebar, and so on. With React Native, we can give these components actual names that make sense to us and other developers who may be using our code, making it easy to bring new people into a project, or hand a project off to someone else. Let's take a look at an example mockup that our designer has handed us. We will then think of how we can conceptualize this into components.

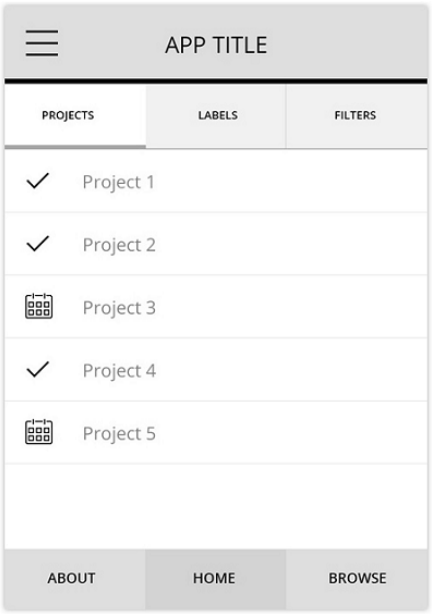

Figure 1.3 Final example app design / structure

The first thing to do is to mentally break the UI elements up into what they actually represent. So, in the above mockup, we have a header bar, and within the header bar we have a title and a menu button. Below the header we have a tab bar, and within the tab bar we have three individual tabs. Go through the rest of the mockup and think of what the rest of the items might be as well. These items that we are identifying will be translated into components. This

is the way you should think about composing your UI. When working with React Native, you should break down common elements in your UI into reusable components, and define their interface accordingly. When you need this element any time in the future, it will be available for you to reuse.

Breaking up your UI elements in to reusable components is not only good for code reuse, but will also make your code very declarative and understandable. For instance, instead of twelve lines of code implementing a footer, the element could simply be called footer. When looking at code that is built in this way, it is much easier to reason about and know exactly what is going on.

Let's take a look at how the design in Figure 1.2 could be broken up in the way we just described:

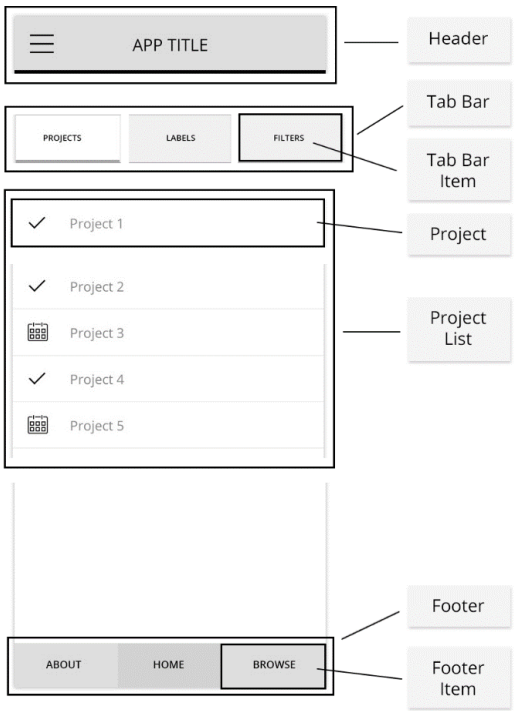

Figure 1.3 App structure broken down into separate components

The names I have used here could be whatever makes sense to you. Look at how these items are broken up and some of them are grouped together. We have logically separated these items into individual and grouped conceptual components. Next, let's see how this would look using actual React Native code.

First, let's look at how the main UI elements would on our page:

```
<Header />
<TabBar />
<ProjectList />
<Footer />
```

Next, let's see how our child elements would look:

```
TabBar:

      <TabBarItem  />
      <TabBarItem  />
      <TabBarItem />

ProjectList:
      // Loop through projects array, for each project return:
      <Project />
```

As you can see, we have used the same names that we declared in figure 1.3, though they could be whatever makes sense to you.

1.5 Acknowledging the strengths

As discussed earlier, one of the main strengths React Native has going for it is that is that it uses React. React, like React Native, is an open source project that is backed by Facebook. As of the time of this writing, React has over 45,000 stars and 700 contributors on Github, which shows that there is a lot of interest and community involvement in the project, making it easier to bet on as a developer or as a project manager. Because React is developed and maintained and used by Facebook, it has some of the most talented engineers in the world overseeing it, pushing it forward and adding new features, and it will probably not be going anywhere anytime soon.

1.5.1 Developer availability

With the rising cost and falling availability of native mobile developers, React Native enters the market with a key advantage over native development: it leverages the wealth of existing talented web and JavaScript developers and gives them another platform in which to build without having to learn a new language.

1.5.2 Developer productivity

Traditionally, to build a cross platform mobile application, you needed both an Android team as well as an iOS team. React Native allows you to build your both Android, iOS, and soon Windows applications using only a single programming language, JavaScript, and possible even a single team, dramatically decreasing development time and development cost, while increasing productivity. As a native developer, the great thing about coming to a platform like this is the fact that you are no longer tied down to being only an Android or iOS developer,

opening the door for a lot of opportunity. This is great news for JavaScript developers as well, allowing them to spend all of their time in one state of mind when switching between web and mobile projects. It is also a win for teams who were traditionally split between Android and iOS, as they can now work together on a single codebase.

To underscore these points, you can also share your data architecture not only cross platform, but also on the web, if you are using something like Redux, which we will look at in a later chapter.

1.5.3 Performance

If you follow other cross platform solutions, you are probably aware of solutions such as PhoneGap, Cordova and Ionic. While these are also viable solutions, the overall consensus is that the performance has not yet caught up to the experience a native app delivers. This is where React Native also really shines, as the performance is usually unnoticeable from that of a native mobile app built using Objective-C or Java.

1.5.4 One-way data flow

One-way data flow separates React and React Native from not only most other JavaScript frameworks, but also any MVC framework. React differs in that it incorporates a one-way data flow, from top-level components all the way down. This makes applications much easier to reason about, as there is one source of truth for your data layer as opposed to having it scattered about your application. We will look at this in more detail later in the book.

1.5.5 Developer experience

The developer experience is a major win for React Native. If you've ever developed for the

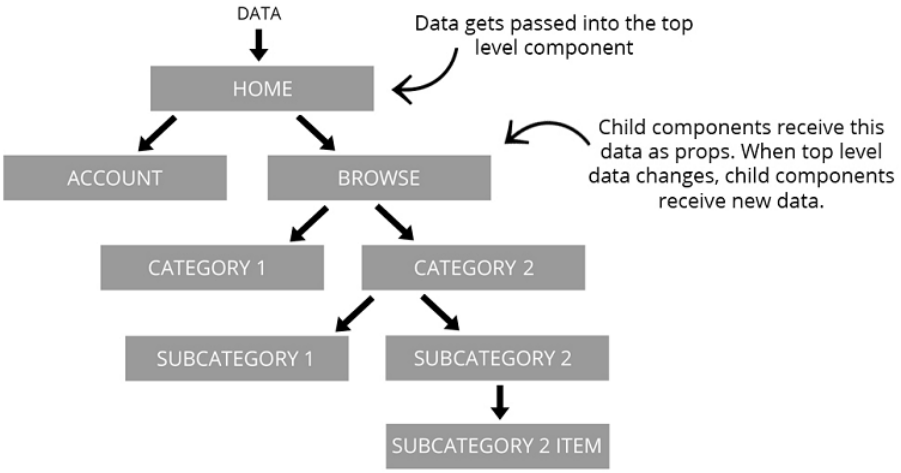

Figure 1.4 Explanation of how one-way data flow works

web, you're aware of the snappy reload times of the browser. Web development has no compilation step, just refresh the screen and your changes are there. This is a far cry from long the compile times of native development. One of the reasons Facebook decided to develop React Native was the lengthy compile times the Facebook application was giving them. Even if they needed to make a small UI change, or any change, they would have to wait a long time while the program compiled to see the results. This long compilation time results in decreased productivity and increased developer cost. React Native solves this issue by giving you the quick reload times of the web, as well as Chrome and Safari debugging tools, making the debugging experience feel a lot like the web.

1.5.6 Transpilation

Transpilation is typically when something known as a transpiler takes source code written in one programming language and produces the equivalent code in another language. With the rise of new EcmaScript features and standards, transpilation has spilled over to also include taking newer versions and yet to be implemented features of certain languages, in our case JavaScript, and producing compiled standard JavaScript, making the code usable by platforms that can only process older versions of the language.

React Native uses Babel to do this transpilation step, and it is built in by default. Babel is an open source tool that transpiles the most bleeding edge JavaScript language features in to code that can be used today. This means that we do not have to wait for the bureaucratic process of language features being proposed, approved, and then implemented before we can use them. We can start using it as soon as the feature makes it into Babel, which is usually very quickly. JavaScript classes, arrow functions and object destructuring are all examples of powerful ES2015 features that have not made it into all browsers yet, but with Babel and React Native, you can use them all today with no worries about whether or not they will work. If you like this, it is also available on the web.

1.5.7 Productivity and efficiency

Companies that are hiring developers for mobile development stand to benefit the most out of using React Native. Having everything written in once language makes hiring a lot easier and less expensive. Productivity also soars when your team is all on the same page, working within a single technology, which makes collaboration and knowledge sharing easier.

1.5.8 Community

The React community, and by extension the React Native community, is one of the most open and helpful groups I have ever interacted with. When running into issues that I have not been able to resolve on my own, by searching online or Stack Overflow, I have reached out directly to either a team member or someone in the community and have had nothing but positive feedback and help .

1.5.9 Open source

React Native is open source. This offers a wealth of benefits. First of all, in addition to the Facebook team there are hundreds of developers that contribute to React Native. If there are bugs, they are pointed out much faster than proprietary software, which will only have the employees on that specific team working on bug fixes and improvements. Open source usually gets closer to what users really want because the users themselves can have a hand in actually making the software what they want it to be. Between the cost of purchasing proprietary software, licensing fees, and support costs, open source also wins when measuring price.

1.5.10 Immediate updates

Traditionally when publishing new versions of an app, you are at the mercy of the app store approval process and schedule. This is a long a tedious process, and can take up to two weeks. If and when you have a change, even if it is something extremely small, it is a painful process to release a new version of your application. React Native, as well as hybrid application frameworks, allow you to deploy mobile app updates directly to the user's device, without going through and app store approval process. If you are used to the web, and the rapid release cycle it has to offer, you can now also do this with React Native and other hybrid application frameworks.

1.5.11 Conclusion

React Native has come a long way at a fairly fast pace. Considering all of the knowledgeable people both in the community and at Facebook working together to improve React Native, I do not see any serious roadblocks stopping the team and the community from handling most issues that have not yet been addressed, or that may have yet come up. Many companies are betting big on this framework, yet many have chosen to stay native or hybrid at the moment. It would be a good idea to look at your individual situation and see what type of mobile implementation works best for you, your company, or your team.

1.6 Creating and using basic components

Components are the fundamental building blocks in React Native, and they can vary in functionality and type. Examples of components in popular use cases could be a header, footer, or navigation component. They can vary in type from an entire View, complete with its own state and functionality, all the way to a single stateless component that receives all of its props (properties) from its parent.

1.6.1 Components

At the core of React Native is the concept of components. React Native has built in component that you will see me describe as Native components, and you will also build components using

the framework. Components are a collection of data and UI elements that make up your views and ultimately your application. We will go in depth on how to build, create and use components in this book.

Let's run through a couple of basic examples of what JSX in React Native looks like vs HTML:

Table 1.1 JSX components vs HTML elements

1. Text Component

HTML	React Native JSX
`<span>Hello World</span>`	`<Text>Hello World</Text>`

2. View Component

HTML	React Native JSX

`<div>`
`  <span>Hello World 2</span>`
`</div>`
	`<View>`
	`  <Text>Hello World 2</Text>`
	`</View>`

1.6.2 Native components

The framework offers native components out of the box, such as View, Text, and Image, among others. In this chapter, we go through the fundamentals of what a component is, how they fit into the workflow, common use cases, and design patterns for building them.

1.6.3 Component composition

Components are usually composed using JSX, but can also be composed using JavaScript. They can be stateful, or since the release of React 0.14, stateless.

Below, we will be creating a component in a number of different ways so we can go over all of the options when creating components.

We'll be creating this component:

```
<MyComponent />
```

This component simply outputs 'Hello World' to the screen. Now, let's look at how we can build this basic component. The only out of the box components we will be using to build this custom component are the View and Text elements we discussed earlier. Remember, a `<View>` component is similar to an html `<div>`, and a `<Text>` component is similar to an html `<span>`.

Let's take a look at a few ways that you can create a component.

1. CREATECLASS SYNTAX (ES5, JSX)

This is the way to create a React Native component using es5 syntax. While you will probably still see this syntax in use a lot on the web and in some older documentation, this syntax is not being used as much in newer documentation and most of the community seems to be heading to using the es2015 class syntax. Because of this, we will be focusing on the ES2015 class syntax for most of the rest of the book.

The entire application does not have to be consistent in its component definitions, but it is usually recommended that you do try to stay mostly consistent with either one or the other.

```
const React = require('react')
const ReactNative = require('react-native')
const { View, Text } = ReactNative

const MyComponent = React.createClass({
  render() {
    return (
      <View>
         <Text>Hello World</Text>
        </View>)
    }
})
```

2. CLASS SYNTAX (ES2015, JSX)

Another way to create React Native components is using ES2015 classes.

```
import  React from 'react'
import { View, Text } from 'react-native'

class  MyComponent extends React.Component {
  render() {
    return (
      <View>
        <Text>Hello World</Text>
      </View>)
  }
}
```

3. STATELESS (REUSABLE) COMPONENT (JSX)

Since the release of React 0.14, we have had the ability to create what are called stateless or reusable components. We have yet dived into state, but just remember that these components are basically pure functions of their props, and do not contain their own stat, so their state cannot be mutated. This syntax looks a lot cleaner than the class or `createClass` syntax.

```
import React from 'react'
import { View, Text } from 'react-native'
```

```
const MyComponent = () => (
  <View>
    <Text>Hello World</Text>
  </View>
)

or

import React from 'react'
import { View, Text } from 'react-native'

function MyComponent () {
  return <Text>HELLO FROM STATELESS</Text>
}
```

4. CREATEELEMENT (JAVASCRIPT)

`React.createElement` is rarely used, and you will probably never need to create a React Native element using this syntax, but may come in handy if you ever need more control over how you are creating your component or you are reading someone else's code. It will also give you a look at how JavaScript compiles JSX.

`React.createElement` takes a few arguments:

```
React.createElement(class type, props, children) {}
```

Let's walk through these arguments:

- **class type**: The element you want to render
- **props**: any properties you want the component to have
- **children**: child components or text

As you can see below, we pass in a View as the first argument to the first instance of `React.createElement`, an empty object as the second argument, and another element as the last argument.

In the second instance, we pass in Text as the first argument, an empty object as the second argument, and "Hello" as the final argument.

```
  class MyComponent extends React.Component {
  render() {
       return (
          React.createElement(View, { },
        React.createElement(Text, {}, "Hello")
      )
    )
    }
  }
```

1.6.4 Exportable components

Next, let's look at another more in depth implementation of a React Native component. Let's create an entire component that we can export and use in another file. We will walk through each piece of this code:

```
import React, { Component } from 'react'
import {
  Text,
  View
} from 'react-native'

class Home extends Component {
  render() {
    return (
      <View>
        <Text>Hello from Home</Text>
      </View>)
  }
}

  export default Home
```

Let's go over all of the different pieces that make up the above component, and discuss what's going on.

IMPORTING

The following code imports and React Native variable declarations:

```
import React, { Component } from 'react'
import {
    Text,
    View
} from 'react-native'
```

Here, we are importing React from the React library. We are also using ES6 object destructuring and an import statement to pull Component, Text, and View into our file, importing these components from React Native.

The `import` statement using ES5 would look like this:

```
var React = require('react')
```

The above statement without object destructuring would look like this:

```
import React = from 'react'
const Component = React.Component
import ReactNative from 'react-native'
const Text = ReactNative.Text
const View = ReactNative.View
```

The import statement is used to import functions, objects, or variables that have been exported from another module, file, or script.

COMPONENT DECLARATION

The following code declares a component:

```
class Home extends Component { }
```

Here we are creating a new instance of a React Native Component class by extending it, and naming it Home. As you can see, before we declaring React.Component, we are now just declaring Component. This is because we have imported the Component element in the object destructuring statement, giving us access to Component as opposed to having to call React.Component.

THE RENDER METHOD

Next, take a look at the Render method:

```
  render() {
    return (
      <View>
        <Text>Hello from Home</Text>
      </View>)
  }
```

The code for the component gets executed in the render method, and the content after the return statement makes up the final component. When the render method is called, it should return a single child element. Any variables or functions declared outside of the render function can be executed here. If you need to do any calculations, declare any variables, or run any functions that do not manipulate the state of the component, you can do so here in between the render() method and the return statement.

EXPORTS

Here, we export the component to be used elsewhere in our application. If you want to use the component in the same file, you do not need to export it. After it is declared, you can use it in your file, or export it to be used in another file. You may also use `module.exports = 'Home'` which would be es5 syntax.

```
export default Home
```

1.6.5 Combining components

Let's look at how we might combine components. First, let's create a Home, Header and Footer component. In a single file, let's start by creating the Home Component:

```
import React, { Component } from 'react'
import {
```

```
    Text,
    View
} from 'react-native'

class Home extends Component {
  render() {
    return (
      <View>

      </View>)
    }
 }
```

Below the Home class declaration, let's start by building out a Header component:

```
class Header extends Component {
    render() {
      return <View>
               <Text>HEADER</Text>
             </View>
    }
  }
```

This looks nice, but let's rewrite the Header into a stateless component. We will discuss when and why it is good to use a stateless component versus a regular React Native class in depth later in the book. As you will begin to see, the syntax and code is much cleaner when we use stateless components:

```
const Header = () => (
  <View>
    <Text>HEADER</Text>
  </View>
)
```

Now, let's insert our Header into our Home component:

```
  class Home extends Component {
    render() {
      return (
       <View>
         <Header />
       </View>
     )
   }
 }
```

We'll then create a Footer and a Main view as well:

```
const Footer = () => (
  <View>
    <Text>Footer</Text>
  </View>
)

const Main = () => (
  <View>
```

```
    <Text> Main </Text>
  </View>
)
```

Now, we'll just drop those into our application:

```
class Home extends Component {
  render() {
    return (
      <View>
        <Header />
        <Main />
        <Footer />
      </View>
   }
}
```

As you can see, the code we just wrote is extremely declarative, meaning it's written in such a way that it describes what you want to do, and is easy to understand in isolation. This is a very high level overview of how we will be creating our components and views in React Native, but should give you a very good idea of how the basics work.

1.7 Creating a starter project

Now that we have gone over a lot of basic details about React Native, let's start digging into some more code. The best way to get started is by looking at the starter project that the React Native CLI gives you and going through it piece by piece, and explaining what everything does and means.

Before we go any further, make sure you see the appendix to make sure you have the necessary tools installed on your machine.

To get started with the React Native starter project and the React Native CLI, open your command line and then create and navigate to an empty directory. Once you are there, install the react-native cli globally by typing the following:

```
npm install -g react-native-cli
```

After React Native is installed on your machine, you can initialize a new project by typing react-native init followed by the project name:

```
react-native init myProject
```

`myProject` can be any name that you choose.

The CLI will then spin up a new project in whatever directory you are in. Once this has finished, open up the project in a text editor.

First, let's take a look at the files and folders this has generated for us:

- `android` – This folder contains all of the android platform specific code and dependencies. You will not need to go into this folder unless you are either implementing a custom bridge into android, or if you install a plugin that calls for some

type of deep configuration

- `ios` - This folder contains all of the ios platform specific code and dependencies. You will not need to go into this folder unless you are either implementing a custom bridge into android, or if you install a plugin that calls for some type of deep configuration
- `node_modules` – React Native uses something called npm (node package manager) to manage dependencies. These dependencies are identified and versioned in the .package.json file, and stored in the node_modules folder. When you install any new packages from the npm / node ecosystem, they will go in here
- `.flowconfig` – Flow (also open sourced by Facebook) offers type checking for JavaScript. Flow is similar to Typescript, if you are familiar with that. This file is the configuration for flow, if you choose to use it.
- `.gitignore` – This is the place to store any file paths that you do not want in version control
- `.watchmanconfig` – Watchman is a file watcher that React Native uses to watch files and record when they change. This is the configuration for Watchman. No changes to this will be needed except for rare use cases.
- `.index.android.js`- this is the entry point for the Android build of the application. When you run your Android application, this is the first JavaScript file to get executed.
- `.index.ios.js`- This is the entry point for the iOS build of the application. When you run your iOS application, this is the first JavaScript file to get executed.
- `.package.json`- This file holds all of our npm configuration. When we npm install files, we can save them here as dependencies. We can also set up scripts to run different tasks.

Now, let's take a look at one of the index files. Open either index.ios.js or index.android.js:

Listing 1.2 Index.ios.js / index.android.js

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 */
'use strict';
import React, {
  AppRegistry,
  Component,
  StyleSheet,
  Text,
  View
} from 'react-native';
class book extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
        <Text style={styles.instructions}>
```

```
          To get started, edit index.ios.js
        </Text>
        <Text style={styles.instructions}>
          Press Cmd+R to reload,{'\n'}
          Cmd+D or shake for dev menu
        </Text>
      </View>
    );
  }
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});

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

As you can see, the above project looks very similar to the one we went over in the last section. There are a couple of new items we have not yet seen:

- StyleSheet
- AppRegistry

`StyleSheet` is an abstraction similar to CSS stylesheets. In React Native you can declare styles either inline or using Stylesheets. As you can see in the first view, there is a container style declared:

```
<View style={styles.container}>
```

This corresponds directly to:

```
container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
}
```

At the bottom of the file, you see

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

This is the JavaScript entry point to running all React Native apps. In the index file is the only place you will be calling this function. The root component of the app should register itself with `AppRegistry.registerComponent()`. The native system can then load the bundle for the app and then actually run the app when it is ready.

Now that we have gone over what is in the file, let's run the project in either our iOS simulator or our Android Emulator.

Figure 1.5 React Native starter project – what you should see after running the starter project on the emulator.

In the Text element that contains 'Welcome to React Native', replace that with 'Welcome to Hello World!' or some text of your choice. Refresh the screen. You should see your changes.

1.8 Summary

- React Native is a framework for building native mobile apps in JavaScript using the React JavaScript library.
- Some of React Native's strengths are its performance, developer experience, ability to build cross platform with a single language, one-way data flow, and community. You may consider React Native over hybrid mainly because of its performance, and over Native mainly because of the developer experience and cross platform ability with a single language.
- JSX is a preprocessor step that adds an XML like syntax to JavaScript. You can use JSX

to create a UI in React Native.

- Components are the fundamental building blocks in React Native. They can vary in functionality and type. You can create custom components to implement common design elements.
- Components can be created using either the ES2015 (class definition) syntax or the ES5 (createClass) syntax, with the es2015 (class definition) syntax being widely recommended and used by the community and in newer documentation.
- Stateless components can be created with less boilerplate for components that do not need to keep up with their own state.
- Larger components can be created by combining smaller subcomponents together.