Part 1

Getting started with React Native

Chapter 1 will get you up and running by going over what React Native is, how it works, what its relationship with React is, and when you might want to use React Native (and when you might not). This chapter provides an overview of React Native's components, which are at the core of React Native. It concludes with creating a small React Native project.

Chapter 2 covers state and properties: what they are, how they work, and why they're important in React Native application development. It also covers the React Component specification and React lifecycle methods.

In chapter 3, you'll build your first React Native app—a Todo app—from the ground up. You'll also learn about using the developer menu in iOS and Android for, among other things, debugging apps.

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 developers face, developing a quality native mobile application is no easy task. It's no wonder 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 make it easier.

At the core of this complexity is the obstacle of cross-platform development. The various platforms are fundamentally different and don't share much of their development environments, APIs, or code. Because of this, we must have separate teams working on each platform, which is both expensive and inefficient.

But this is an exciting time in mobile application development. We're 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 mobile applications. It's now

possible to build native performing cross-platform apps as well as web applications with a single language and a single team. With the rise of mobile devices and the subsequent increase in demand for talent driving developer salaries higher and higher, React Native brings to the table the ability 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 and React Native

React Native is a framework for building native mobile apps in JavaScript using the React JavaScript library; React Native code compiles to real native components. If you're not sure what React is, it's a JavaScript library open sourced by and used within Facebook. It was originally used to build user interfaces for web applications. It has since evolved and can now also be used to build server-side and mobile applications (using React Native).

React Native has a lot going for it. In addition to being backed and open sourced by Facebook, it also has a tremendous community of motivated people behind it. Facebook groups, with their millions of users, are powered by React Native as well as Facebook Ads Manager. Airbnb, Bloomberg, Tesla, Instagram, Ticketmaster, SoundCloud, Uber, Walmart, Amazon, and Microsoft are some of the other companies either investing in or using React Native in production.

With React Native, developers can build native views and access native platform-specific components using JavaScript. This sets React Native apart from other hybrid app frameworks like Cordova and Ionic, which package web views built using HTML and CSS into a native application. Instead, React Native takes JavaScript and compiles it into a true native application that can use platform-specific APIs and components. Alternatives like Xamarin take the same approach, but Xamarin apps are built using C#, not JavaScript. Many web developers have JavaScript experience, which helps ease the transition from web to mobile app development.

There are many benefits to choosing React Native as a mobile application framework. Because the application renders native components and APIs directly, speed and performance are much better than with hybrid frameworks such as Cordova and Ionic. With React Native, we're writing entire applications using a single programming language: JavaScript. We can reuse a lot of code, thereby reducing the time it takes to ship a cross-platform application. And hiring and finding quality JavaScript developers is much easier and cheaper than hiring Java, Objective C, or Swift developers, leading to an overall less-expensive process.

> **NOTE** React Native applications are built using JavaScript and JSX. We'll discuss JSX in depth in this book, but for now think of it as a JavaScript syntax extension that looks like HTML or XML.

We'll dive much deeper into React in chapter 2. Until then, let's touch on a few core concepts as an introduction.

1.1.1 A basic React class

Components are the building blocks of a React or React Native application. The entry point of an application is a component that requires and is made of other components. These components may also require other components, and so on.

There are two main types of React Native components: *stateful* and *stateless*. Here's an example of a stateful component using an ES6 class:

```
class HelloWorld extends React.Component {
  constructor() {
    super()
    this.state = { name: 'Chris' }
  }

  render () {
    return (
      <SomeComponent />
    )
  }
}
```

And here's an example of a stateless component:

```
const HelloWorld = () => (
  <SomeComponent />
)
```

The main difference is that stateless components don't hook into any lifecycle methods and hold no state of their own, so any data to be rendered must be received as properties (props). We'll go through the lifecycle methods in depth in chapter 2, but for now let's take a first look at them and look at a class.

Listing 1.1 Creating a basic React Native class

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

class HelloWorld extends React.Component {
  constructor () {
    super()
    this.state = {
      name: 'React Native in Action'
    }
  }
  componentDidMount () {
    console.log('mounted..')
  }
  render () {
    return (
      <View style={styles.container}>
        <Text>{this.state.name}</Text>
      </View>
    )
  }
```

Constructor sets a state object with a name property

Final lifecycle method

Calls render()

```
}

const styles = StyleSheet.create({
  container: {
    marginTop: 100,
    flex: 1
  }
})
```

> **NOTE** Something to keep in mind when we discuss the following methods is the concept of *mounting*. When a component is created, the React component lifecycle is instantiated, triggering the methods used in listing 1.1.

At the top of the file, you require `React` from `'react'`, as well as `View`, `Text`, and `StyleSheet` from `'react-native'`. `View` is the most fundamental building block for creating React Native components and the UI in general and can be thought of like a `div` in HTML. `Text` allows you to create text elements and is comparable to a `span` tag in HTML. `StyleSheet` lets you create style objects to use in an application. These two packages (`react` and `react-native`) are available as npm modules.

When the component first loads, you set a `state` object with the property `name` in the constructor. For data in a React Native application to be dynamic, it needs to be either set in the state or passed down as props. Here, you set the state in the constructor and can therefore change it if desired by calling

```
this.setState({
  name: 'Some Other Name'
})
```

which rerenders the component. Setting the variable in state allows you to update the value elsewhere in the component.

`render` is then called: it examines the props and state and then must return a single React Native element, `null`, or `false`. If you have multiple child elements, they must be wrapped in a parent element. Here, the components, styles, and data are combined to create what will be rendered to the UI.

The final method in the lifecycle is `componentDidMount`. If you need to do any API calls or AJAX requests to reset the state, this is usually the best place to do so. Finally, the UI is rendered to the device, and you can see the result.

1.1.2 React lifecycle

When a React Native class is created, methods are instantiated that you can hook into. These methods are called *lifecycle methods*, and we'll cover them in depth in chapter 2. The methods in listing 1.1 are `constructor`, `componentDidMount`, and `render`, but there are a few more, and they all have their own use cases.

Lifecycle methods happen in sync and help manage the state of components as well as execute code at each step of the way, if you wish. The only required lifecycle method is `render`; all the others are optional. When working with React Native, you're fundamentally working with the same lifecycle methods and specifications you'd use with React.

1.2 What you'll learn

In this book, we'll cover everything you need to know to build robust mobile applications for iOS and Android using the React Native framework. Because React Native is built using the React library, we'll begin in chapter 2 by covering and thoroughly explaining how React works.

We'll 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'll dive deep into how flexbox works and discuss all the flexbox properties. If you've used flexbox in CSS for layout on the web, all of this will be familiar to you, but keep in mind that the flexbox implementation used by React Native isn't 100% the same.

We'll then go through many 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 are covered extensively throughout this book because they're the building blocks of a React Native application.

There are many ways to implement navigation, each with its own nuances, pros, and cons. We'll discuss navigation in depth and cover how to build robust navigation using the most important of the navigation APIs. We'll cover 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.

Next, we'll discuss in depth both cross-platform and platform-specific APIs available in React Native and how they work. It will then be time for you to start working with data using network requests, AsyncStorage (a form of local storage), Firebase, and WebSocket. Then we'll dive into the different data architectures and how each of them works to handle the state of the application. Finally, we'll look at testing and a few different ways to test in React Native.

1.3 What you should know

To get the most out of this book, you should have beginner to intermediate knowledge of JavaScript. Much of your 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'll be building in iOS, a basic understanding of Xcode is beneficial and will speed things along but isn't required. Similarly, if you're building for Android, a basic understanding of Android Studio will be beneficial but not required.

Fundamental knowledge of newer JavaScript features implemented in the ES2015 release of the JavaScript programming language is beneficial but not necessary. Some conceptual knowledge of MVC frameworks and single-page architecture is also good but not required.

1.4 Understanding how React Native works

Let's look at how React Native works by discussing JSX, the threading model, React, unidirectional data flow, and more.

1.4.1 JSX

React and React Native both encourage the use of JSX. JSX is basically a syntax extension to JavaScript that looks similar to XML. You can build React Native components without JSX, but JSX makes React and React Native a lot more readable and easier to maintain. JSX may seem strange at first, but it's extremely powerful, and most people grow to love it.

1.4.2 Threading

All JavaScript operations, when interacting with the native platform, are done in a separate thread, allowing the user interface as well as any animations to perform smoothly. This thread is where the React application lives, and where all API calls, touch events, and interactions are processed. When there's 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 it uses React. React is an open source JavaScript library that's also backed by Facebook. It was originally designed to build applications and solve problems on the web. This framework has become extremely popular since its release, with many established companies taking advantage of its quick rendering, maintainability, and declarative UI, among other things.

Traditional DOM manipulation is slow and expensive in terms of performance and should be minimized. React bypasses the traditional DOM with something called the *virtual DOM*: basically, a copy of the actual DOM in memory that only changes when comparing new versions of the virtual DOM to old versions of the virtual DOM. This minimizes the number of DOM operations required to achieve the new state.

1.4.4 Unidirectional data flow

React and React Native emphasize unidirectional, or one-way, data flow. Because of how React Native applications are built, this one-way data flow is easy to achieve.

1.4.5 Diffing

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

1.4.6 Thinking in components

When building a UI in React Native, it's useful to think of your application as being composed of a collection of components. Thinking about how a page is set up, you already do this conceptually, but using concepts, names, or class names like *header*, *footer*, *body*, *sidebar*, and so on. With React Native, you can give these components names that make sense to you and other developers who may be using your code, making it easy to bring new people into a project or hand a project off to someone else.

Suppose a designer has handed you the example mockup shown in figure 1.1. Let's think of how to conceptualize this into components.

The first thing to do is to mentally break the UI elements into what they represent. The example mockup has a header bar, and within the header bar are a title and a

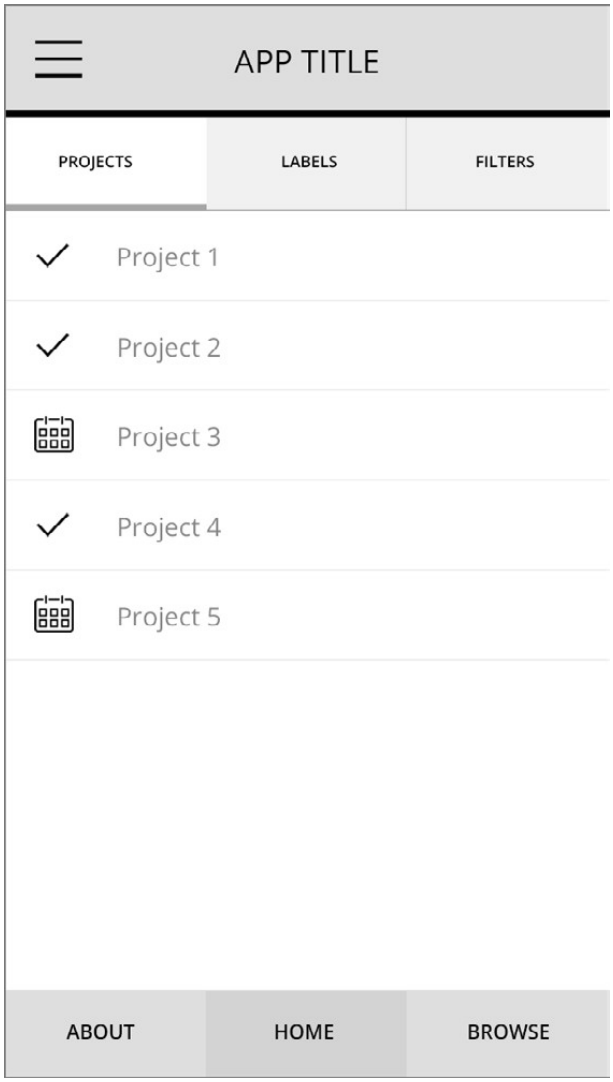


Figure 1.1 Example app design

menu button. Below the header is a tab bar, and within the tab bar are three individual tabs. Go through the rest of the mockup and think of what the other items are. These items you're identifying will be translated into components. This is the way you should think about composing a UI when working with React Native: break down common elements in the UI into reusable components, and define their interface accordingly. When you need an element in the future, it will be available for reuse.

Breaking UI elements into reusable components is good for code reuse and also makes your code declarative and understandable. For instance, instead of 12 lines of code implementing a footer, the element could be called `footer`. Looking at code built this way, it's much easier to reason about and know exactly what's going on.

Figure 1.2 shows how the design in figure 1.1 could be broken up as I just described. The names can be whatever makes sense to you. Some of the items are grouped together—I logically separated the items individually and grouped components conceptually.

Next, let's see how this would look using actual React Native code. First, let's look at how the main UI elements appear on the page:

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

Next, let's see how the child elements look:

```
TabBar:
      <TabBarItem  />
      <TabBarItem  />
      <TabBarItem />

ProjectList:
      // Add a Project component for each project in the list:
      <Project />
```

I've used the names declared in figure 1.2, but they could be whatever makes sense to you.

1.5 Acknowledging React Native's strengths

As discussed earlier, one of the main strengths React Native has going for it is that it uses React. React, like React Native, is an open source project backed by Facebook. As of the time of this writing, React has over 100,000 stars and more than 1,100 contributors on GitHub—that's 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, 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 probably won't be going away anytime soon.

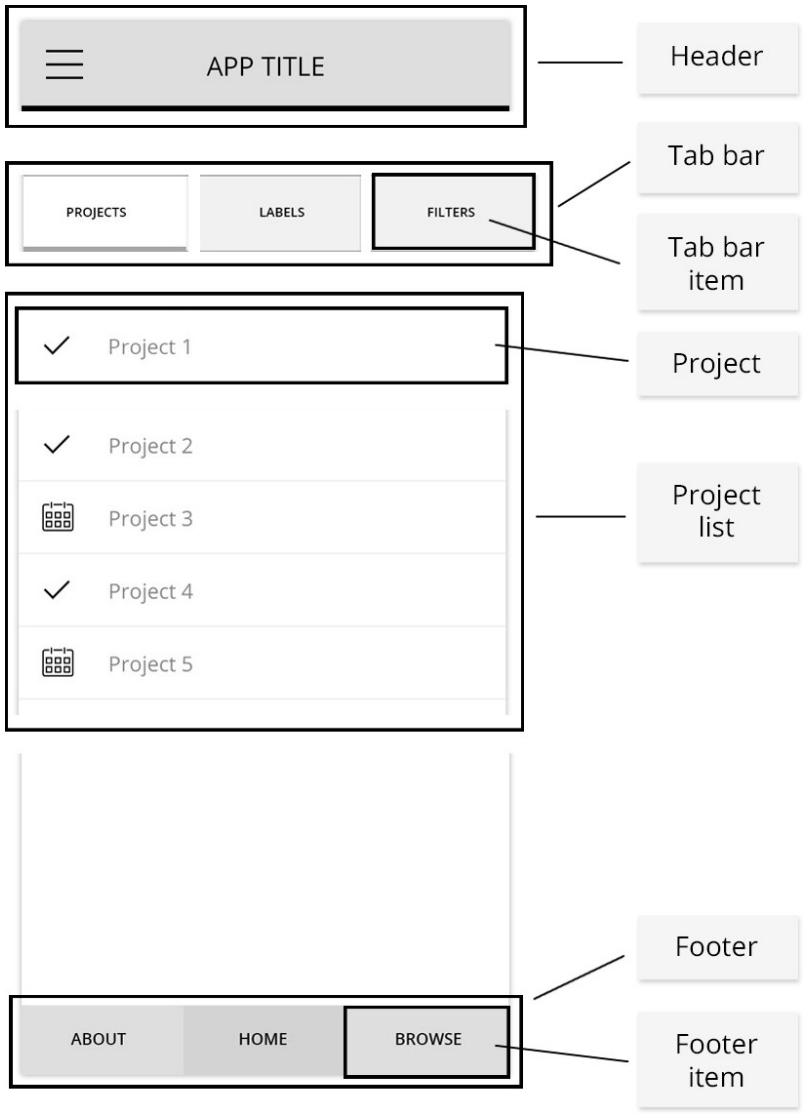


Figure 1.2 App structure broken down into separate components

1.5.1 Developer availability

With the rising cost and decreasing availability of native mobile developers, React Native enters the market with a key advantage over native development: it takes advantage of the wealth of existing talented web and JavaScript developers and gives them another platform on 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 and an iOS team. React Native allows you to build Android, iOS, and (soon) Windows applications using a single programming language, JavaScript, and possibly 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're 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 their time in one state of mind when switching between web and mobile projects. It's also a win for teams who were traditionally split between Android and iOS, because they can now work together on a single codebase. To underscore these points, you can share your data architecture not only cross platform, but also on the web, if you're using something like Redux (discussed in chapter 12).

1.5.3 Performance

If you follow other cross-platform solutions, you're probably aware of solutions such as PhoneGap, Cordova, and Ionic. Although these are also viable solutions, the consensus is that performance hasn't yet caught up to the experience a native app delivers. This is where React Native also shines, because the performance is usually not noticeably different from that of a native mobile app built using Objective-C/Swift or Java.

1.5.4 One-way data flow

One-way data flow separates React and React Native from most other JavaScript frameworks and also any MVC framework. React incorporates a one-way data flow from top-level components all the way down (see figure 1.3). This makes applications much easier to reason about, because there's one source of truth for the data layer as opposed to having it scattered about the application. We'll look at this in more detail later in the book.

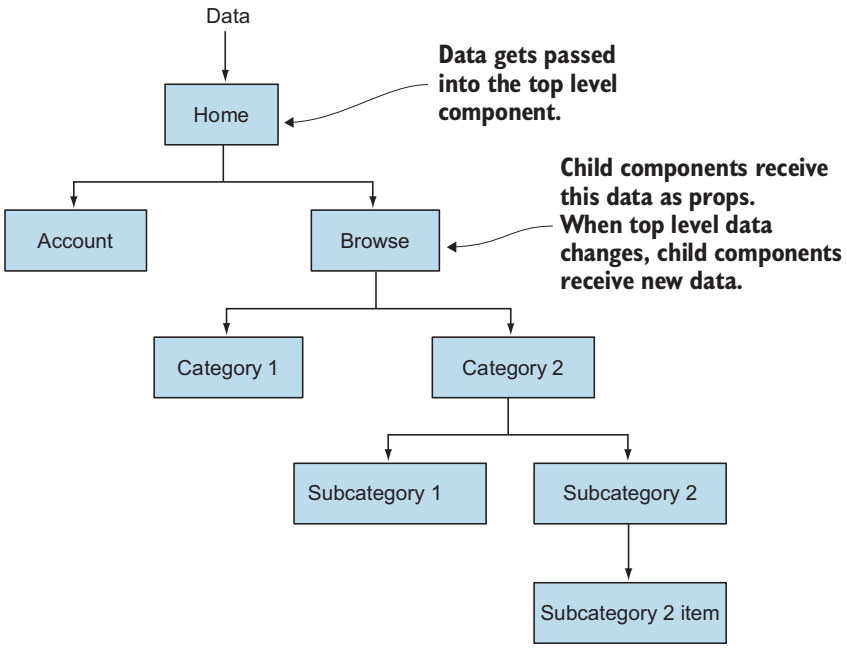


Figure 1.3 How one-way data flow works

1.5.5 Developer experience

The developer experience is a major win for React Native. If you've ever developed for the web, you're aware of the browser's snappy reload times. Web development has no compilation step: just refresh the screen, and your changes are there. This is a far cry from the long compile times of native development. One of the reasons Facebook decided to develop React Native was to overcome the lengthy compile times of the Facebook application when using native iOS and Android build tools. To make a small UI change or any other change, Facebook developers had to wait a long time while the program compiled to see the results. Long compilation times result 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.

React Native also has something called *hot reloading* built in. What does this mean? Well, while developing an application, imagine having to click a few times into your app to get to the place you're developing. While using hot reloading, when you make a code change, you don't have to reload and click back through the app to get to the current state. Using this feature, you save the file, and the application reloads only the component you've changed, instantly giving you feedback and updating the current state of the UI.

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 this case JavaScript, and producing transpiled 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's built in by default. Babel is an open source tool that transpiles the most bleeding-edge JavaScript language features into code that can be used today. You don't have to wait for the bureaucratic process of language features being proposed, approved, and then implemented before you can use them. You can start using a feature as soon as it makes it into Babel, which is usually very quickly. JavaScript classes, arrow functions, and object destructuring are all examples of powerful ES2015 features that haven't made it into all browsers and runtimes yet; but with Babel and React Native, you can use them today with no worries about whether they will work. If you like using the latest language features, you can use the same transpilation process to develop web applications.

1.5.7 Productivity and efficiency

Native mobile development is becoming more and more expensive, so engineers who can deliver applications across platforms and stacks will become increasingly valuable and in demand. Once React Native—or something similar, if it comes along—makes

developing desktop and web as well as mobile applications using a single framework mainstream, there will be a restructuring and rethinking of how engineering teams are organized. Instead of a developer being specialized in a certain platform, such as iOS or web, they'll oversee features across platforms. In this new era of cross-platform and cross-stack engineering teams, developers delivering native mobile, web, and desktop applications will be more productive and efficient and will therefore be able to demand a higher wage than a traditional web developer who can only deliver web applications.

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

1.5.8 Community

The React community, and by extension the React Native community, is one of the most open and helpful groups I've ever interacted with. When I've run into issues I couldn't resolve on my own by searching online or on Stack Overflow, I've 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, in addition to the Facebook team, hundreds of developers contribute to React Native. Bugs are pointed out much faster than in proprietary software, which has only the employees on a specific team working on bug fixes and improvements. Open source usually gets closer to what users want because the users can have a hand in making the software what they want it to be. Given 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're at the mercy of the app store approval process and schedule. This long, tedious process can take up to two weeks. Making a change, even if it's extremely small, is painful and requires releasing a new version of the 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 an app store approval process. If you're used to the web and the rapid release cycle it offers, you can now do the same thing with React Native and other hybrid application frameworks.

1.5.11 Other solutions for building cross-platform mobile applications

React Native isn't the only option for building a cross-platform mobile application. Multiple other options are available, with the main ones being Cordova, Xamarin, and Flutter:

- *Cordova* is basically a native shell around a web application that allows the developer to access native APIs within the application. Unlike traditional web

applications, Cordova apps can be deployed to the App Store and Google Play Store. The benefit of using something like Cordova is that there isn't much more to learn if you're already a web developer: you can use HTML, JavaScript, CSS, and your JavaScript framework of choice. The main drawback of Cordova is that you'll have a hard time matching the performance and smooth UI that React Native offers: you're relying on the DOM, because you're mainly working with web technologies.

- *Xamarin* is a framework that allows developers to build iOS, Android, Windows, and macOS applications using a single codebase written in C#. Xamarin compiles to a native app in different ways depending on the platform being targeted. Xamarin has a free tier that lets developers build and deploy mobile applications and a paid tier for larger or enterprise companies. Xamarin will probably appeal more to native developers because it doesn't have similarities to web technologies like React Native and Cordova.
- *Flutter* is a framework open sourced by Google that uses the Dart programming language to build applications that run on iOS and Android platforms.

1.6 React Native's drawbacks

Now that we've gone over the benefits of using React Native, let's look at a few reasons and circumstances where you may not want to choose the framework. First, React Native is still immature when compared to other platforms such as native iOS, Android, and Cordova. Feature parity isn't there yet with either native iOS or Cordova. Most functionality is now built in, but there may be times when you need functionality that isn't yet available, and this means you must dig into the native code to build it yourself, hire someone to do it, or not implement the feature.

Another thing to think about is the fact that you and/or your team must learn a completely new technology if you aren't familiar with React. Most people agree that React is easy to pick up; but if you're already proficient with Angular and Ionic, for example, and you have an application deadline coming up, it may be wise to go with what you already know instead of spending the time it takes to learn and train your team on a new tech. In addition to learning React and React Native, you must also become familiar with Xcode and the Android development environments, which can take some getting used to.

Finally, React Native is an abstraction built on top of existing platform APIs. When newer versions of iOS, Android, and other future platforms are released, there may be a time when React Native will be behind on new features, forcing you to either build custom implementations to interact with these new APIs or wait until React Native regains feature parity with the new release.

1.7 Creating and using basic components

Components are the fundamental building blocks of React Native, and they can vary in functionality and type. Examples of components in popular use cases include buttons,

headers, footers, and navigation components. They can vary in type from an entire view, complete with its own state and functionality, to a single stateless component that receives all its props from its parent.

1.7.1 An overview of components

As I've said, the core of React Native is the concept of components. Components are collections of data and UI elements that make up views and, ultimately, applications. React Native has built-in components that are described as *native components* in this book, but you can also build custom components using the framework. We'll go into depth on how to build, create, and use components.

As mentioned earlier, React Native components are built using JSX. Table 1.1 shows a few basic examples of what JSX in React Native looks like versus HTML. As you can see, JSX looks similar to HTML or XML.

Table 1.1 JSX components vs. HTML elements

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

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

Touchable highlight	`<button>`
	`  <span>Hello World 2</span>`
	`</button >`
		`<TouchableHighlight>`
		`  <Text>Hello World 2</Text>`
		`</TouchableHighlight>`

1.7.2 Native components

The framework offers native components out of the box, such as `View`, `Text`, and `Image`, among others. You can create components using these Native components as building blocks. For example, you can use the following markup to create a `Button` component using React Native `TouchableHighlight` and `Text` components.

Listing 1.4 Creating a `Button` component

```
import { Text, TouchableHighlight } from 'react-native'
const Button = () => (
  <TouchableHighlight>
    <Text>Hello World</Text>
  </TouchableHighlight>
)
export default Button
```

You can then import and use the new button.

Listing 1.5 Importing and using the `Button` component

```
import React from 'react'
import { Text, View } from 'react-native'
import Button from './components/Button'
const Home = () => (
  <View>
    <Text>Welcome to the Hello World Button!</Text>
    <Button />
  </View>
)
```

Next, we'll go through the fundamentals of what a component is, how components fit into the workflow, and common use cases and design patterns for building them.

1.7.3 Component composition

Components are usually composed using JSX, but they can also be composed using JavaScript. In this section, you'll create a component several different ways to see all the options. You'll be creating this component:

```
<MyComponent />
```

This component outputs "Hello World" to the screen. Now, let's see how to build this basic component. The only out-of-the-box components you'll use to build this custom component are the `View` and `Text` elements discussed earlier. Remember, a `View` component is similar to an HTML `<div>`, and a `Text` component is similar to an HTML `<span>`.

Let's look at a few ways to create a component. The entire application doesn't have to be consistent in its component definitions, but it's usually recommended that you stay consistent and follow the same pattern for defining classes throughout your application.

CREATECLASS SYNTAX (ES5, JSX)

This is the way to create a React Native component using ES5 syntax. You'll probably still see this syntax in use in some older documentation and examples, but it isn't being used in newer documentation and is now deprecated. We'll focus on the ES2015 class syntax for the rest of the book but will review the `createClass` syntax here in case you come across it in older code:

```
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>)
    }
})
```

CLASS SYNTAX (ES2015, JSX)

The main way to create stateful React Native components is using ES2015 classes. This is the way you'll create stateful components for the rest of the book and is now the approach recommended by the community and creators of React Native:

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

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

STATELESS (REUSABLE) COMPONENT (JSX)

Since the release of React 0.14, we've had the ability to create *stateless* components. We haven't yet dived into state, but just remember that stateless components are basically pure functions that can't mutate their own data and don't contain their own state. This syntax is much 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 <View><Text>HELLO FROM STATELESS</Text></View>
}
```

CREATEELEMENT (JAVASCRIPT)

`React.createElement` is rarely used, and you'll probably never need to create a React Native element using this syntax. But it may come in handy if you ever need more control over how you're creating a component, or if you're 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(type, props, children) {}
```

Let's walk through them:

- `type`—The element you want to render
- `props`—Any properties you want the component to have
- `children`—Child components or text

In the following example, you 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, you 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")
      )
    )
  }
}
```

This is the same as declaring the component as follows:

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

1.7.4 Exportable components

Next, let's look at another, more in-depth implementation of a React Native component. You'll create an entire component that you can export and use in another file:

```
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 the pieces that make up this component and discuss what's going on.

IMPORTING

The following code imports React Native variable declarations:

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

Here, you're importing React directly from the React library using a default import and importing `Component` from the React library using a named import. You're also using named imports to pull `Text` and `View` into your file.

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

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

This statement without using named imports 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 the component:

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

Here you're creating a new instance of a React Native `Component` class by extending it and naming it `Home`. Before, you declared `React.Component`; now you're just declaring `Component`, because you imported the `Component` element in the object destructuring statement, giving you access to `Component` as opposed to having to call `React.Component`.

THE RENDER METHOD

Next, look at the `render` method:

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

The code for the component is executed in the `render` method, and the content after the `return` statement returns what's rendered on the screen. 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 using state or props, or run any functions that don't

manipulate the state of the component, you can do so between the `render` method and the `return` statement.

EXPORTS

Now, you export the component to be used elsewhere in the application:

```
export default Home
```

If you want to use the component in the same file, you don't need to export it. After it's declared, you can use it in the file or export it to be used in another file. You may also use `module.exports = 'Home'`, which is ES5 syntax.

1.7.5 Combining components

Let's look at how to combine components. First, create `Home`, `Header`, and `Footer` components in a single file. Begin by creating the `Home` component:

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

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

      </View>)
    }
 }
```

In the same file, below the `Home` class declaration, build out a `Header` component:

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

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

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

Now, insert `Header` into the `Home` component:

```
  class Home extends Component {
    render() {
```

```
      return (
       <View>
         <Header />
       </View>
     )
   }
 }
```

Create a `Footer` and a `Main` view, as well:

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

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

Now, drop those into your application:

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

The code you 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 high-level overview of how you'll create components and views in React Native, but should give you a good idea of how the basics work.

1.8 Creating a starter project

Now that we've gone over a lot of details about React Native, let's dig into some more code. We'll focus on building apps using the React Native CLI, but you can also use the Create React Native App CLI to create a new project.

1.8.1 Create React Native App CLI

You can create React Native projects using the Create React Native App CLI, a project generator that's maintained in the React Community GitHub repository, mainly by the Expo team. Expo created the React Native App project as a way to allow developers to get up and running with React Native without having to worry about installing all the native SDKs involved with running a React Native project using the CLI.

To create a new project using Create React Native App, first install the CLI:

```
npm install -g create-react-native-app
```

Here's how to create a new project using `create-react-native-app` from the command line:

```
create-react-native-app myProject
```

1.8.2 React Native CLI

Before we go any further, check this book's appendix to verify that you have the necessary tools installed on your machine. If you don't have the required SDKs installed, you won't be able to continue building your first project using the React Native CLI.

To get started with the React Native starter project and the React Native CLI, open the command line and then create and navigate to an empty directory. Once you're 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 you choose. The CLI will then spin up a new project in whatever directory you're in. Open the project in a text editor.

First, let's look at the main files and folders this process has generated for you:

- *android*—This folder contains all the Android platform-specific code and dependencies. You won't need to go into this folder unless you're implementing a custom bridge into Android or you install a plugin that calls for some type of deep configuration.
- *ios*—This folder contains all the iOS platform-specific code and dependencies. You won't need to go into this folder unless you're implementing a custom bridge into iOS or you install a plugin that calls for some type of deep configuration.
- *node_modules*—React Native uses *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'll go here. These can be installed using either npm or yarn.
- *.flowconfig*—Flow (also open sourced by Facebook) offers type checking for JavaScript. Flow is like Typescript, if you're 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 you don't 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 in rare use cases.
- *index.js*—This is the entry point of the application. In this file, App.js is imported and `AppRegistry.registerComponent` is called, initializing the app.
- *App.js*—This is the default main import used in index.js containing the base project. You can change it by deleting this file and replacing the main import in index.js.
- *package.json*—This file holds your npm configuration. When you npm install files, you can save them here as dependencies. You can also set up scripts to run different tasks.

The following listing shows App.js.

Listing 1.6 App.js

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 * @flow
 */

import React, { Component } from 'react';
import {
  Platform,
  StyleSheet,
  Text,
  View
} from 'react-native';

const instructions = Platform.select({
  ios: 'Press Cmd+R to reload,\n' +
    'Cmd+D or shake for dev menu',
  android: 'Double tap R on your keyboard to reload,\n' +
    'Shake or press menu button for dev menu',
});

export default class App 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 App.js
        </Text>
        <Text style={styles.instructions}>
          {instructions}
        </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,
  },
});
```

This code looks much like what we went over in the last section. There are a couple of new items you haven't yet seen:

```
StyleSheet
Platform
```

`Platform` is an API that allows you to detect the current type of operating system you're running on: web, iOS, or Android.

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

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

This corresponds directly to

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

At the bottom of the index.js file, you see

```
AppRegistry.registerComponent('myProject', () => App);
```

This is the JavaScript entry point to running all React Native apps. In the index file is the only place you'll call 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 run the app when it's ready.

Now that we've gone over what's in the file, run the project in either your iOS simulator or your Android emulator (see figure 1.4). In the text element that contains "Welcome to React Native," enter "Welcome to Hello World!" or other text of your choice. Refresh the screen, and you should see your changes.

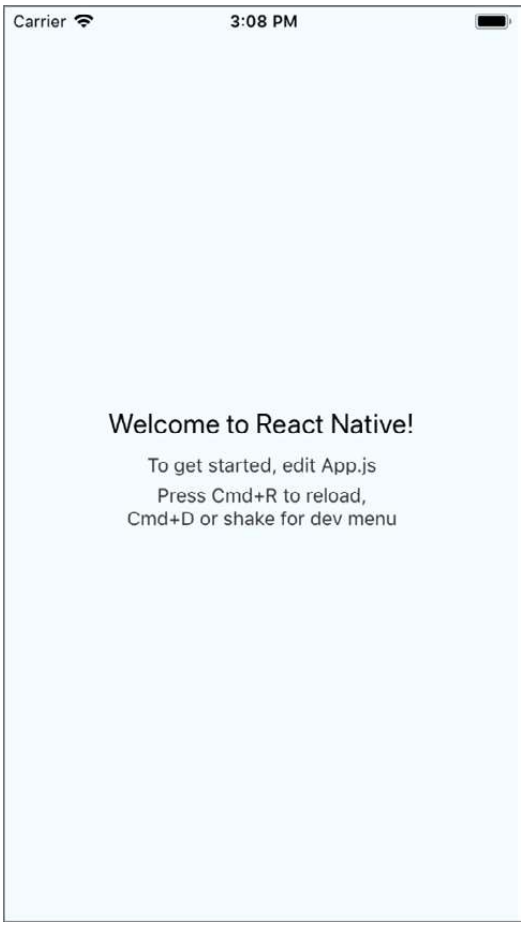


Figure 1.4 React Native starter project: what you should see after running the starter project on the emulator

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 a 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 that require state or lifecycle methods need to be created using a JavaScript class by extending the `React.Component` class.
- Stateless components can be created with less boilerplate for components that don't need to keep up with their own state.
- Larger components can be created by combining smaller subcomponents.

2 Understanding React

This chapter covers

- How state works and why it's important
- How properties work and why they're important
- Understanding the React component specification
- Implementing React lifecycle methods

Now that we've gone over the basics, it's time to dive into some other fundamental pieces that make up React and React Native. We'll discuss how to manage state and data, and how data is passed through an application. We'll also dive deeper by demonstrating how to pass properties (props) between components and how to manipulate these props from the top down.

After you're equipped with knowledge about state and props, we'll go deeper into how to use the built-in React lifecycle methods. These methods allow you to perform certain actions when a component is created or destroyed. Understanding them is key to understanding how React and React Native work and how to take full advantage of the framework. The lifecycle methods are also conceptually the biggest part of React and React Native.

NOTE You'll see both React and React Native referenced in this chapter. Keep in mind that when I mention React, I'm talking not about things that are specific to React Native, but concepts that are related to both React and React Native. For example, state and props work the same in both React and React Native, as do the React lifecycle and the React component specifications.

2.1 Managing component data using state

One of the ways data is created and managed in a React or React Native component is by using state. Component state is declared when the component is created, and its structure is a plain JavaScript object. State can be updated within the component using a function called `setState` that we'll look at in depth shortly.

The other way data can be handled is by using props. Props are passed down as parameters when the component is created; unlike state, they can't be updated within the component.

2.1.1 Correctly manipulating component state

State is a collection of values that a component manages. React thinks of UIs as simple state machines. When the state of a component changes using the `setState` function, React rerenders the component. If any child components are inheriting this state as props, then all of the child components are rerendered as well.

When building an application using React Native, understanding how state works is fundamental because state determines how stateful components render and behave. Component state is what allows you to create components that are dynamic and interactive. The main point to understand when differentiating between state and props is that state is mutable, whereas props are immutable.

SETTING *INITIAL STATE*

State is initialized when a component is created either in the constructor or with a property initializer. Once the state is initialized, it's available in the component as `this.state`. The following listing shows an example.

Listing 2.1 Setting state with a property initializer

```
import React from 'react'

class MyComponent extends React.Component {
  state = {
    year: 2016,
    name: 'Nader Dabit',
    colors: ['blue']
  }

  render() {
    return (
      <View>
        <Text>My name is: { this.state.name }</Text>
```

```
        <Text>The year is: { this.state.year }</Text>
        <Text>My colors are { this.state.colors[0] }</Text>
      </View>
    )
  }
}
```

The `constructor` function is called the moment a JavaScript class is instantiated, as shown in the next listing. This isn't a React lifecycle method, but a regular JavaScript class method.

Listing 2.2 Setting state with a constructor

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

class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      year: 2016,
      name: 'Nader Dabit',
      colors: ['blue']
    }
  }
  render() {
    return (
      <View>
        <Text>My name is: { this.state.name }</Text>
        <Text>The year is: { this.state.year }</Text>
        <Text>My colors are { this.state.colors[0] }</Text>
      </View>
    )
  }
}
```

The constructor and property initializer both work exactly the same, and which approach you use is based on preference.

UPDATING STATE

State can be updated by calling `this.setState(object)`, passing in an object with the new state you want to use. `setState` merges the previous state with the current state, so if you only pass in a single item (key-value pair), the rest of the state will remain the same, while the new item in the state will be overwritten.

Let's look at how to use `setState` (see listing 2.3). To do so, we'll introduce a new method, a touch handler called `onPress`. `onPress` can be called on a few types of "tappable" React Native components, but here you'll attach it to a `Text` component to get started with this basic example. You'll call a function called `updateYear` when the text is pressed, to update the state with `setState`. This function will be defined before the `render` function, because it's usually best practice to define any custom methods before the render method, but keep in mind that the order of the definition of the functions doesn't affect the actual functionality.

Listing 2.3 Updating state

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

class MyComponent extends Component {
    constructor(){
      super()
      this.state = {
        year: 2016,
      }
    }
    updateYear() {
      this.setState({
        year: 2017
      })
    }
    render() {
      return (
        <View>
          <Text
            onPress={() => this.updateYear()}>
            The year is: { this.state.year }
          </Text>
        </View>
      )
    }
 }
```

Figure 2.1 shows how the state is updated each time the text element in listing 2.3 is pressed. Every time `setState` is called, React will rerender the component (calling the `render` method again) and any child components. Calling `this.setState` is the way to change a state variable and trigger the `render` method again, because changing the state variable directly won't trigger a rerender of the component and therefore no changes will be visible in the UI. A common mistake for beginners is updating the state variable directly. For example, something like the following doesn't work when trying

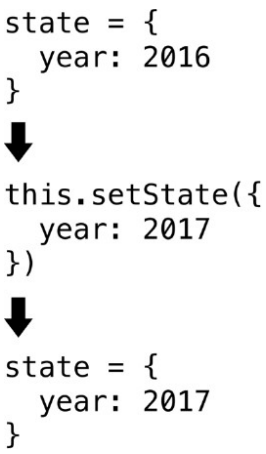


Figure 2.1 The flow of `setState`, with arrows indicating when the text element is pressed. The state `year` property is initialized to 2016 in the constructor. Each time the text is pressed, the state `year` property is set to 2017.

to update state—the `state` object is updated, but the UI isn't updated because `setState` isn't called and the component isn't rerendered:

```
class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      year: 2016,
    }
  }
  updateYear() {
    this.state.year = 2017
  }
  render() {
    return (
      <View>
        <Text
          onPress={() => this.updateYear()}>
          The year is: { this.state.year }
        </Text>
      </View>
    )
  }
}
```

But a method is available in React that can force an update once a state variable has been changed as in the previous snippet. This method is called `forceUpdate`; see listing 2.4. Calling `forceUpdate` causes `render` to be called on the component, triggering a rerendering of the UI. Using `forceUpdate` isn't usually necessary or recommended, but it's good to know about in case you run into it in examples or documentation. Most of the time, this rerendering can be handled using other methods such as calling `setState` or passing in new props.

Listing 2.4 Forcing rerender with `forceUpdate`

```
class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      year: 2016
    }
  }
  updateYear() {
    this.state.year = 2017
  }
  update() {
    this.forceUpdate()
  }
  render() {
    return (
      <View>
```

```
          <Text onPress={ () => this.updateYear() }>
              The year is: { this.state.year }
            </Text>
            <Text
             onPress={ () => this. update () }>Force Update
            </Text>
        </View>
      )
    }
}
```

Now that we've gone over how to work with state using a basic string, let's look at a few other data types. You'll attach a Boolean, an array, and an object to the state and use it in the component. You'll also conditionally show a component based on a Boolean in the state.

Listing 2.5 State with other data types

```
class MyComponent extends Component {
    constructor(){
      super()
      this.state = {
        year: 2016,
        leapYear: true,
        topics: ['React', 'React Native', 'JavaScript'],
        info: {
          paperback: true,
          length: '335 pages',
          type: 'programming'
        }
      }
    }
    render() {
      let leapyear = <Text>This is not a leapyear!</Text>
      if (this.state.leapYear) {
        leapyear = <Text>This is a leapyear!</Text>
      }
      return (
        <View>
          <Text>{ this.state.year }</Text>
          <Text>Length: { this.state.info.length }</Text>
          <Text>Type: { this.state.info.type }</Text>
          { leapyear }
        </View>
      )
    }
}
```

2.2 Managing component data using props

Props (short for *properties*) are a component's inherited values or properties that have been passed down from a parent component. Props can be either static or dynamic

values when they're declared, but when they're inherited they're immutable; they can only be altered by changing the initial values at the top level where they're declared and passed down. React's "Thinking in React" documentation says that props are best explained as "a way of passing data from parent to child." Table 2.1 highlights some of the differences and similarities between props and state.

Table 2.1 Props vs. state

Props	State
External data	Internal data
Immutable	Mutable
Inherited from a parent	Created in the component
Can be changed by a parent component	Can only be updated in the component
Can be passed down as props	Can be passed down as props
Can't change inside the component	Can change inside the component

A good way to explain how props work is to show an example. The following listing declares a book value and passes it down to a child component as a static prop.

Listing 2.6 Static props

```
class MyComponent extends Component {
  render() {
    return (
      <BookDisplay book="React Native in Action" />
    )
  }
}
class BookDisplay extends Component {
  render() {
    return (
      <View>
        <Text>{ this.props.book }</Text>
      </View>
    )
  }
}
```

This code creates two components: `<MyComponent />` and `<BookDisplay />`. When you create `<BookDisplay />`, you pass in a property called `book` and set it to the string "React Native in Action". Anything passed as a property in this way is available on the child component as `this.props`.

You can also pass down literals as you would variables, by using curly braces and a string value as shown next.

Listing 2.7 Displaying static props

```
class MyComponent extends Component {
  render() {
    return (
      <BookDisplay book={"React Native in Action"} />
    )
  }
}
class BookDisplay extends Component {
  render() {
    return (
      <View>
        <Text>{ this.props.book }</Text>
      </View>
    )
  }
}
```

DYNAMIC PROPS

Next, pass a dynamic property to the component. In the `render` method, before the `return` statement, declare a variable `book` and pass it in as a prop.

Listing 2.8 Dynamic props

```
class MyComponent extends Component {
    render() {
      let book = 'React Native in Action'
      return (
        <BookDisplay book={ book } />
      )
    }
  }

class BookDisplay extends Component {
  render() {
    return (
      <View>
        <Text>{ this.props.book }</Text>
      </View>
    )
  }
}
```

Now, pass a dynamic property to the component using state.

Listing 2.9 Dynamic props using state

```
class MyComponent extends Component {
  constructor() {
    super()
    this.state = {
      book: 'React Native in Action'
```

```
    }
  }
  render() {
    return (
      <BookDisplay book={this.state.book} />
    )
  }
}
class BookDisplay extends Component {
  render() {
    return (
      <View>
        <Text>{ this.props.book }</Text>
      </View>
    )
  }
}
```

Next, let's look at how to update the state and, consequently, the value passed down as the prop to `BookDisplay`. Remember, props are immutable, so you'll change the state of the parent component (`MyComponent`), which will supply a new value to the `BookDisplay` `book` prop and trigger a rerender of both the component and the child component. Breaking this idea into individual parts, here's what needs to be done:

1. Declare the state variable:

```
this.state = {
  book: 'React Native in Action'
}
```

2. Write a function that will update the state variable:

```
updateBook() {
  this.setState({
    book: 'Express in Action'
  })
}
```

3. Pass the function and the state down to the child component as props:

```
<BookDisplay
  updateBook={ () => this.updateBook() }
  book={ this.state.book } />
```

4. Attach the function to the touch handler in the child component:

```
<Text onPress={ this.props.updateBook }>
```

Now that you know the pieces you need, you can write the code to put this into action. You'll use the components from the previous examples and add the new functionality.

Listing 2.10 Updating dynamic props

```
class MyComponent extends Component {
  constructor(){
    super()
```

```
    this.state = {
      book: 'React Native in Action'
    }
  }
  updateBook() {
    this.setState({
    book: 'Express in Action'
    })
  }
  render() {
    return (
      <BookDisplay
       updateBook={ () => this.updateBook() }
       book={ this.state.book } />
    )
  }
}
class BookDisplay extends Component {
  render() {
    return (
      <View>
        <Text
         onPress={ this.props.updateBook }>
          { this.props.book }
        </Text>
      </View>
    )
  }
}
```

DESTRUCTURING PROPS AND STATE

Constantly referring to state and props as `this.state` and `this.props` can get repetitive, violating the DRY (don't repeat yourself) principle that many of us try to follow. To fix this, you can try using destructuring. *Destructuring* is a new JavaScript feature that was added as part of the ES2015 spec and is available in React Native applications. The basic idea is that you can take properties from an object and use them as variables in an app:

```
const person = { name: 'Jeff', age: 22 }

const { age } = person

console.log(age)   #22
```

Write a component using destructuring, as shown next.

Listing 2.11 Destructuring state and props

```
class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      book: 'React Native in Action'
    }
```

```
  }
  updateBook() {
    this.setState({ book: 'Express in Action' })
  }
  render() {
    const { book } = this.state
    return (
      <BookDisplay
       updateBook={ () => this.updateBook() }
       book={ book } />
    )
  }
}
class BookDisplay extends Component {
  render() {
    const { book, updateBook } = this.props
    return (
      <View>
        <Text
         onPress={ updateBook }>
          { book }
        </Text>
      </View>
    )
  }
}
```

You no longer have to refer to `this.state` or `this.props` in the component when referencing the book; instead, you've taken the `book` variable out of the state and the props and can reference the variable itself. This starts to make more sense and will keep your code cleaner as your state and props become larger and more complex.

PROPS WITH STATELESS COMPONENTS

Because stateless components only have to worry about props and don't have their own state, they can be extremely useful when creating reusable components. Let's see how props are used in a stateless component.

To access props using a stateless component, pass in `props` as the first argument to the function.

Listing 2.12 Props with stateless components

```
const BookDisplay = (props) => {
  const { book, updateBook } = props
  return (
    <View>
      <Text
        onPress={ updateBook }>
        { book }
      </Text>
    </View>
  )
}
```

You can also destructure props in the function argument.

Listing 2.13 Destructuring props in a stateless component

```
const BookDisplay = ({ updateBook, book }) => {
  return (
    <View>
      <Text
        onPress={ updateBook }>
        { book }
      </Text>
    </View>
  )
 }
```

That looks much nicer and cleans up a lot of unnecessary code! You should use stateless components wherever you can, simplifying your codebase and logic.

> **NOTE** Stateless components are often referred to as *functional* components, because they can be written as functions in JavaScript.

PASSING ARRAYS AND OBJECTS AS PROPS

Other data types work exactly as you might expect. For example, to pass an array, you pass in the array as a prop. To pass an object, you pass in the object as a prop. Let's look at a basic example.

Listing 2.14 Passing other data types as props

```
class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      leapYear: true,
      info: {
        type: 'programming'
      }
    }
  }
  render() {
    return (
      <BookDisplay
       leapYear={ this.state.leapYear }
       info={ this.state.info }
       topics={['React', 'React Native', 'JavaScript']} />
    )
  }
}
const BookDisplay = (props) => {
  let leapyear
  let { topics } = props
  const { info } = props
  topics = topics.map((topic, i) => {
    return <Text>{ topic }</Text>
```

```
  })
  if (props.leapYear) {
    leapyear = <Text>This is a leapyear!</Text>
  }
  return (
    <View>
         { leapyear }
         <Text>Book type: { info.type }</Text>
         { topics }
    </View>
  )
}
```

2.3 React component specifications

When creating React and React Native components, you can hook into several specifications and lifecycle methods to control what's going on in your component. In this section, we'll discuss them and give you a good understanding of what each one does and when you should use them.

First we'll go over the basics of the component specifications. A component *specification* basically lays out how a component should react to different things happening in the lifecycle of the component. The specifications are as follows:

- `render` method
- `constructor` method
- `statics` object, used to define static methods available to a class

2.3.1 Using the render method to create a UI

The `render` method is the only method in the component specification that's required when creating a component. It must return either a single child element, `null`, or `false`. This child element can be a component you declared (such as a `View` or `Text` component), or another component you defined (maybe a `Button` component you created and imported into the file):

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

You can use the `render` method with or without parentheses. If you don't use parentheses, then the returned element must of course be on the same line as the `return` statement:

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

The `render` method can also return another component that was defined elsewhere:

```
render() {
   return <SomeComponent />
}
#or
render() {
   return (
     <SomeComponent />
   )
}
```

You can also check for conditionals in the `render` method, perform logic, and return components based on their value:

```
render() {
  if(something === true) {
    return <SomeComponent />
  } else return <SomeOtherComponent />
}
```

2.3.2 Using property initializers and constructors

State can be created in a constructor or using a *property initializer*. Property initializers are an ES7 specification to the JavaScript language, but they work out of the box with React Native. They provide a concise way to declare state in a React class:

```
class MyComponent extends React.Component {
  state = {
    someNumber: 1,
    someBoolean: false
  }
```

You can also use a `constructor` method to set the initial state when using classes. The concept of classes, as well as the `constructor` function, isn't specific to React or React Native; it's an ES2015 specification and is just syntactic sugar on top of JavaScript's existing prototype-based inheritance for creating and initializing an object created with a class. Other properties can also be set for a component class in the constructor by declaring them with the syntax `this.property` (`property` being the name of the property). The keyword `this` refers to the current class instance you're in:

```
constructor(){
  super()
  this.state = {
    someOtherNumber: 19,
    someOtherBoolean: true
  }
  this.name = 'Hello World'
  this.type = 'class'
  this.loaded = false
}
```

When using a constructor to create a React class, you must use the `super` keyword before you can use the `this` keyword, because you're extending another class. Also, if you need access to any props in the constructor, they must be passed as an argument to the constructor and the `super` call.

Setting the state based on props usually isn't good practice unless you're intentionally setting some type of seed data for the component's internal functionality, because the data will no longer be consistent across components if it's changed. State is only created when the component is first mounted or created. If you rerender the same component using different prop values, then any instances of that component that have already been mounted won't use the new prop values to update state.

The following example shows props being used to set state values within the constructor. Let's say you pass in "Nader Dabit" as the props to the component initially: the `fullName` property in the state will be "Nader Dabit". If the component is then rerendered with "Another Name", the constructor won't be called a second time, so the state value for `fullName` will remain "Nader Dabit":

```
constructor(props){
   super(props)
   this.state = {
     fullName: props.first + ' ' + props.last,
   }
 }
```

2.4 React lifecycle methods

Various methods are executed at specific points in a component's lifecycle: these are called the *lifecycle methods*. Understanding how they work is important because they allow you to perform specific actions at different points in the creation and destruction of a component. For example, suppose you wanted to make an API call that returned some data. You'd probably want to make sure the component was ready to render this data, so you'd make the API call once the component was mounted in a method called `componentDidMount`. In this section, we'll go over the lifecycle methods and explain how they work.

The life of a React component has three stages: creation (mounting), updating, and deletion (unmounting). During these three stages, you can hook into three sets of lifecycle methods:

- *Mounting (creation)*—When a component is created, a series of lifecycle methods are triggered and you have the option to hook into any or all of them: `constructor`, `getDerivedStateFromProps`, `render`, and `componentDidMount`. The one such method you've used so far is `render`, which renders and returns a UI.
- *Updating*—When a component updates, the update lifecycle methods are triggered: `getDerivedStateFromProps` (when props change), `shouldComponentUpdate`, `render`, `getSnapshotBeforeUpdate`, and `componentDidUpdate`. An update can happen in one of two ways:
  - When `setState` or `forceUpdate` is called within a component
  - When new props are passed down into the component
- *Unmounting*—When the component is unmounted (destroyed), a final lifecycle method is triggered: `componentWillUnmount`.

2.4.1 The static getDerivedStateFromProps method

`getDerivedStateFromProps` is a static class method that is called both when the component is created and when it receives new props. This method receives the new props and most up-to-date state as arguments and returns an object. The data in the object is updated to the state. The following listing shows an example.

Listing 2.15 `static getDerivedStateFromProps`

```
export default class App extends Component {
  state = {
    userLoggedIn: false
  }
  static getDerivedStateFromProps(nextProps, nextState) {
    if (nextProps.user.authenticated) {
      return {
        userLoggedIn: true
      }
    }
    return null
  }
  render() {
    return (
      <View style={styles.container}>
        {
          this.state.userLoggedIn && (
            <AuthenticatedComponent />>
          )
        }
      </View>
    );
  }
}
```

2.4.2 The componentDidMount lifecycle method

`componentDidMount` is called exactly once, just after the component has been loaded. This method is a good place to fetch data with AJAX calls, perform `setTimeout` functions, and integrate with other JavaScript frameworks.

Listing 2.16 `componentDidMount`

```
class MainComponent extends Component {
  constructor() {
    super()
    this.state = { loading: true, data: {} }
  }
  componentDidMount() {
      #simulate ajax call
    setTimeout(() => {
      this.setState({
        loading: false,
        data: {name: 'Nader Dabit', age: 35}
```

```
      })
    }, 2000)
  }
  render() {
    if(this.state.loading) {
      return <Text>Loading</Text>
    }
    const { name, age } = this.state.data
    return (
      <View>
        <Text>Name: {name}</Text>
        <Text>Age: {age}</Text>
      </View>
    )
  }
}
```

2.4.3 The shouldComponentUpdate lifecycle method

`shouldComponentUpdate` returns a Boolean and lets you decide when a component renders. If you know the new state or props won't require the component or any of its children to render, you can return `false`. If you want the component to rerender, return `true`.

Listing 2.17 `shouldComponentUpdate`

```
class MainComponent extends Component {
   shouldComponentUpdate(nextProps, nextState) {
     if(nextProps.name !== this.props.name) {
       return true
     }
     return false
   }
   render() {
     return <SomeComponent />
   }
 }
```

2.4.4 The componentDidUpdate lifecycle method

`componentDidUpdate` is invoked immediately after the component has been updated and rerendered. You get the previous state and previous props as arguments.

Listing 2.18 `componentDidUpdate`

```
class MainComponent extends Component {
   componentDidUpdate(prevProps, prevState) {
     if(prevState.showToggled === this.state.showToggled) {
          this.setState({
         showToggled: !showToggled
       })
     }
   }
```

```
  render() {
    return <SomeComponent />
  }
}
```

2.4.5 The componentWillUnmount lifecycle method

`componentWillUnmount` is called before the component is removed from the application. Here, you can perform any necessary cleanup, remove listeners, and remove timers that were set up in `componentDidMount`.

Listing 2.19 `componentWillUnmount`

```
class MainComponent extends Component {

  handleClick() {
    this._timeout = setTimeout(() => {
      this.openWidget();
    }, 2000);
  }
  componentWillUnmount() {
    clearTimeout(this._timeout);
  }
  render() {
    return <SomeComponent
             handleClick={() => this.handleClick()} />
  }
}
```

Summary

- State is a way to handle data in React components. Updating state rerenders the UI of the component and any child component relying on this data as props.
- Properties (props) are how data is passed down through a React Native application to child components. Updating props automatically updates any components receiving the same props.
- A React component specification is a group of methods and properties in a React component that specifies the declaration of the component. `render` is the only required method when creating a React component; all other methods and properties are optional.
- There are three main stages in a React component's lifecycle: creation (mounting), updating, and deletion (unmounting). Each has its own set of lifecycle methods.
- React lifecycle methods are available in a React component and are executed at specific points in the component's lifecycle. They control how the component functions and updates.

3 Building your first React Native app

This chapter covers

- Building a todo app from the ground up
- Light debugging

When learning a new framework, technology, language, or concept, diving directly into the process by building a real app is a great way to jump-start the learning process. Now that you understand the basics of how React and React Native work, let's put these pieces together to make your first app: a todo app. Going through the process of building a small app and using the information we've gone over so far will be a good way to reinforce your understanding of how to use React Native.

You'll use some functionality in the app that we haven't yet covered in depth, and some styling nuances we've yet to discuss, but don't worry. Instead of going over these new ideas one by one now, you'll build the basic app and then learn about these concepts in detail in later chapters. Take this opportunity to play around with the app as you build it to learn as much as possible in the process: feel free to break and fix styles and components to see what happens.

3.1 Laying out the todo app

Let's get started building the todo app. It will be similar in style and functionality to the apps on the TodoMVC site (http://todomvc.com). Figure 3.1 shows how the app will look when you're finished, so you can conceptualize what components you need and how to structure them. As in chapter 1, figure 3.2 breaks the app into components and container components. Let's see how this will look in the app using a basic implementation of React Native components.

Listing 3.1 Basic todo app implementation

```
<View>
  <Heading />
  <Input />
  <TodoList />
  <Button />
  <TabBar />
</View>
```

The app will display a heading, a text input, a button, and a tab bar. When you add a todo, the app will add it to the array of todos and display the new todo beneath the input. Each todo will have two buttons: Done and Delete. The Done button will mark it as complete, and the Delete button will remove it from the array of todos. At the bottom of the screen, the tab bar will filter the todos based on whether they're complete or still active.

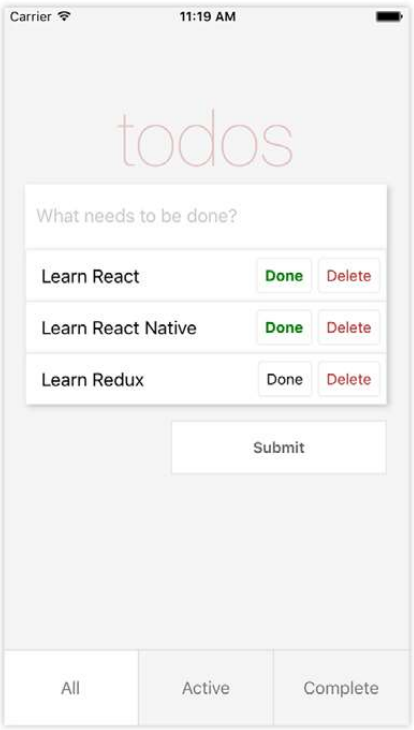


Figure 3.1 Todo app design

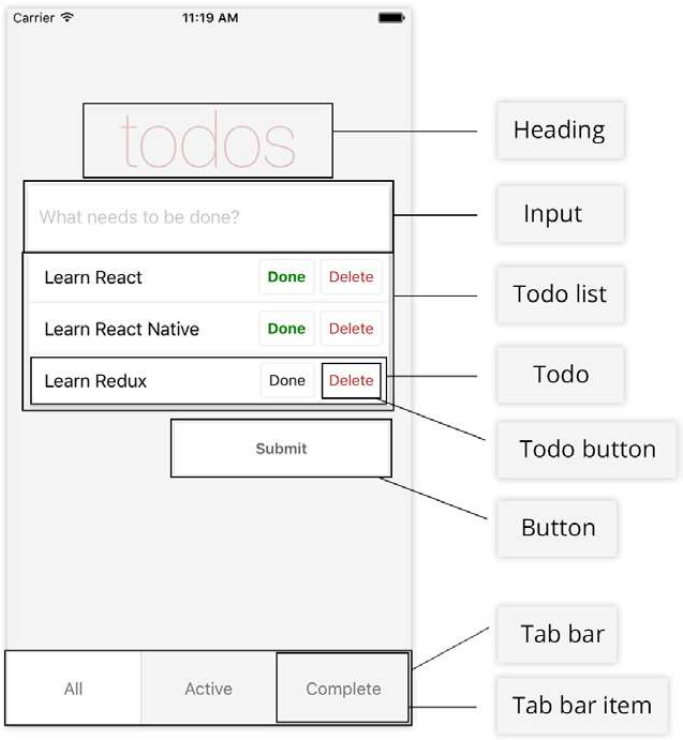


Figure 3.2 Todo app with descriptions

3.2 Coding the todo app

Let's get started coding the app. Create a new React Native project by typing `react-native init TodoApp` in your terminal (see figure 3.3). Now, go into your index file: if you're developing for iOS, open index.iOS.js; and if you're developing for Android, open index.Android.js. The code for both platforms will be the same.

> **NOTE** I'm using React Native version 0.51.0 for this example. Newer versions may have API changes, but nothing should be broken for building the todo app. You're welcome to use the most recent version of React Native, but if you run into issues, use the exact version I'm using here.

In the index file, import an `App` component (which you'll create soon), and delete the styling along with any extra components you're no longer using.

Listing 3.2 index.js

```
import React from 'react'
import { AppRegistry } from 'react-native'
import App from './app/App'

   const TodoApp = () => <App />

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

Here, you bring in `AppRegistry` from `react-native`. You also bring in the main `App` component, which you'll create next.

In the `AppRegistry` method, you initiate the application. `AppRegistry` is the JS entry point to running all React Native apps. It takes two arguments: the `appKey`, or the name of the application you defined when you initialized the app; and a function that returns the React Native component you want to use as the entry point of the app. In this case, you're returning the `TodoApp` component declared in listing 3.2.

Now, create a folder called app in the root of the application. In the app folder, create a file called App.js and add the basic code shown in the next listing.

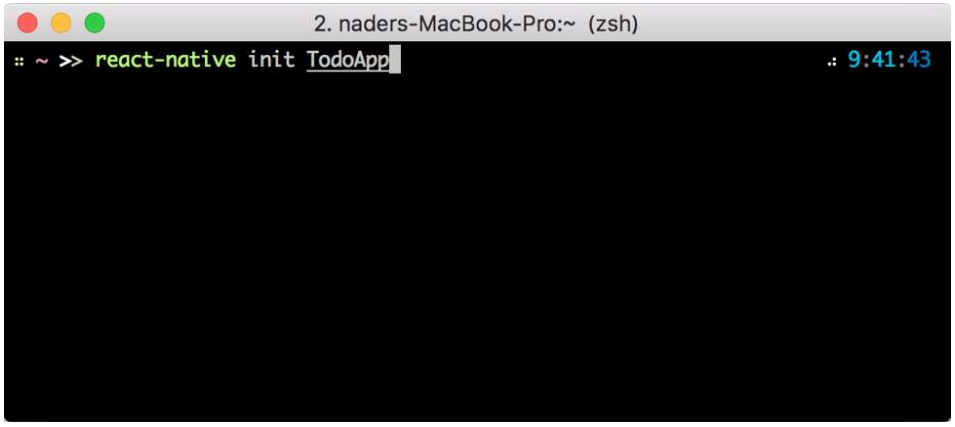


Figure 3.3 Initializing a new React Native app

Listing 3.3 Creating the `App` component

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

class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <ScrollView keyboardShouldPersistTaps='always'
                    style={styles.content}>
          <View/>
        </ScrollView>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f5f5f5'
  },
  content: {
    flex: 1,
    paddingTop: 60
  }
})

export default App
```

You import a new component called `ScrollView`, which wraps the platform `ScrollView` and is basically a scrollable `View` component. A `keyboardShouldPersistTaps` prop of `always` is added: this prop will dismiss the keyboard if it's open and allow the UI to process any `onPress` events. You make sure both the `ScrollView` and the parent `View` of the `ScrollView` have a `flex:1` value. `flex:1` is a style value that makes the component fill the entire space of its parent container.

Now, set up an initial state for some of the values you'll need later. You need an array to keep your todos, which you'll name `todos`; a value to hold the current state of the `TextInput` that will add the todos, named `inputValue`; and a value to store the type of todo that you're currently viewing (All, Current, or Active), named `type`.

In App.js, before the `render` function, add a constructor and an initial state to the class, and initialize these values in the state.

Listing 3.4 Setting the initial state

```
...

class App extends Component {
  constructor() {
    super()
```

```
    this.state = {
      inputValue: '',
      todos: [],
      type: 'All'
    }
  }
  render() {
    ...
  }
}

...
```

Next, create the `Heading` component and give it some styling. In the app folder, create a file called Heading.js. This will be a stateless component.

Listing 3.5 Creating the `Heading` component

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

const Heading = () => (
  <View style={styles.header}>
    <Text style={styles.headerText}>
      todos
    </Text>
  </View>
)

const styles = StyleSheet.create({
  header: {
    marginTop: 80
  },
  headerText: {
    textAlign: 'center',
    fontSize: 72,
    color: 'rgba(175, 47, 47, 0.25)',
    fontWeight: '100'
  }
})

export default Heading
```

Note that in the styling of `headerText`, you pass an `rgba` value to `color`. If you aren't familiar with RGBA, the first three values make up the RGB color values, and the last value represents the alpha or opacity (red, blue, green, alpha). You pass in an alpha value of 0.25, or 25%. You also set the font weight to `100`, which will give the text a thinner weight and look.

Go back into App.js, bring in the `Heading` component, and place it in the `ScrollView`, replacing the empty `View` you originally placed there.

Run the app to see the new heading and app layout: see figure 3.4. To run the app in iOS, use `react-native run-ios`. To run in Android, use `react-native run-android` in your terminal from the root of your React Native application.

Listing 3.6 Importing and using the `Heading` component

```
import React, { Component } from 'react'
import {View, ScrollView, StyleSheet} from 'react-native'
import Heading from './Heading'

class App extends Component {
  ...
  render() {
    return (
      <View style={styles.container}>
        <ScrollView
         keyboardShouldPersistTaps='always'
         style={styles.content}>
          <Heading />
        </ScrollView>
      </View>
    )
  }
}
...
```

Figure 3.4 Running the app

Next, create the `TextInput` component and give it some styling. In the app folder, create a file called Input.js.

Listing 3.7 Creating the `TextInput` component

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

const Input = () => (
  <View style={styles.inputContainer}>
    <TextInput
      style={styles.input}
      placeholder='What needs to be done?'
      placeholderTextColor='#CACACA'
      selectionColor='#666666' />
  </View>
)

const styles = StyleSheet.create({
  inputContainer: {
    marginLeft: 20,
    marginRight: 20,
    shadowOpacity: 0.2,
    shadowRadius: 3,
    shadowColor: '#000000',
    shadowOffset: { width: 2, height: 2 }
  },
  input: {
    height: 60,
    backgroundColor: '#ffffff',
    paddingLeft: 10,
    paddingRight: 10
  }
})

export default Input
```

You're using a new React Native component called `TextInput` here. If you're familiar with web development, this is similar to an HTML `input`. You also give both the `TextInput` and the outer `View` their own styling.

`TextInput` takes a few other props. Here, you specify a `placeholder` to show text before the user starts to type, a `placeholderTextColor` that styles the placeholder text, and a `selectionColor` that styles the cursor for the `TextInput`.

The next step, in section 3.4, will be to wire up a function to get the value of the `TextInput` and save it to the state of the `App` component. You'll also go into App.js and add a new function called `inputChange` below the `constructor` and above the `render` function. This function will update the state value of `inputValue` with the value passed in, and for now will also log out the value of `inputValue` for you to make sure the function is working by using `console.log()`. But to view `console.log()` statements in React Native, you first need to open the developer menu. Let's see how it works.

3.3 Opening the developer menu

The developer menu is a built-in menu available as a part of React Native; it gives you access to the main debugging tools you'll use. You can open it in the iOS simulator or in the Android emulator. In this section, I'll show you how to open and use the developer menu on both platforms.

> **NOTE** If you aren't interested in the developer menu or want to skip this section for now, go to section 3.4 to continue building the todo app.

3.3.1 Opening the developer menu in the iOS simulator

While the project is running in the iOS simulator, you can open the developer menu in one of three ways:

- Press Cmd-D on the keyboard.
- Press Cmd-Ctrl-Z on the keyboard.
- Open the Hardware > Shake Gesture menu in the simulator options (see figure 3.5).

When you do, you should see the developer menu, shown in figure 3.6.

> **NOTE** If Cmd-D or Cmd-Ctrl-Z doesn't open the menu, you may need to connect your hardware to the keyboard. To do this, go to Hardware > Keyboard > Connect Hardware Keyboard in your simulator menu.

Figure 3.5 Manually opening the developer menu (iOS simulator)

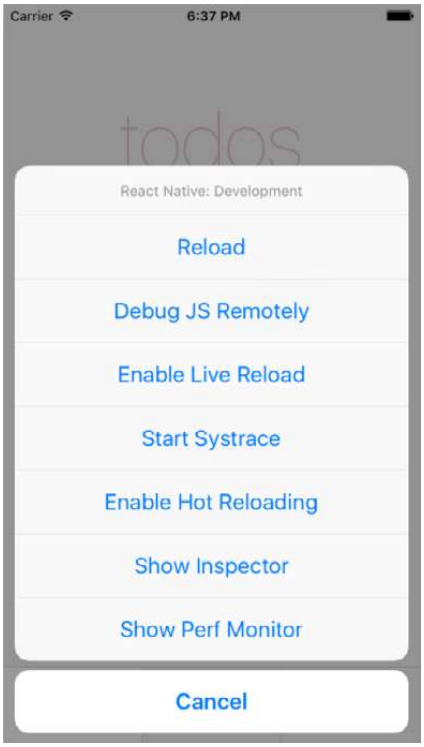


Figure 3.6 React Native developer menu (iOS simulator)

3.3.2 Opening the developer menu in the Android emulator

With the project open and running in the Android emulator, the developer menu can be opened in one of three ways:

- Press F2 on the keyboard.
- Press Cmd-M on the keyboard.
- Press the Hardware button (see figure 3.7).

When you do, you should see the developer menu shown in figure 3.8.

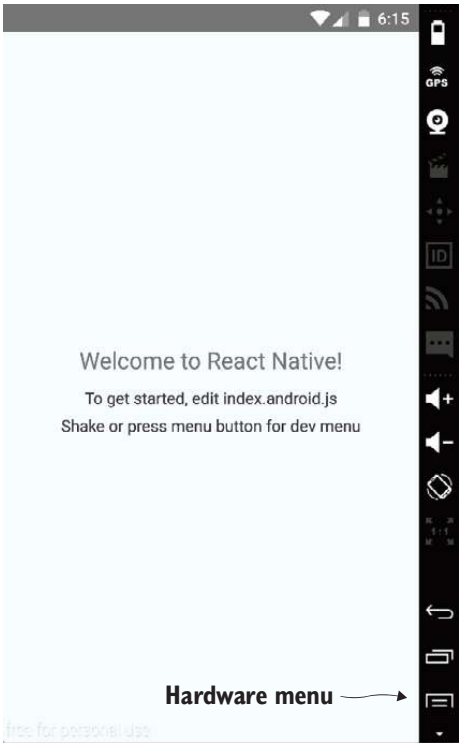


Figure 3.7 Manually opening the hardware menu (Android emulator)

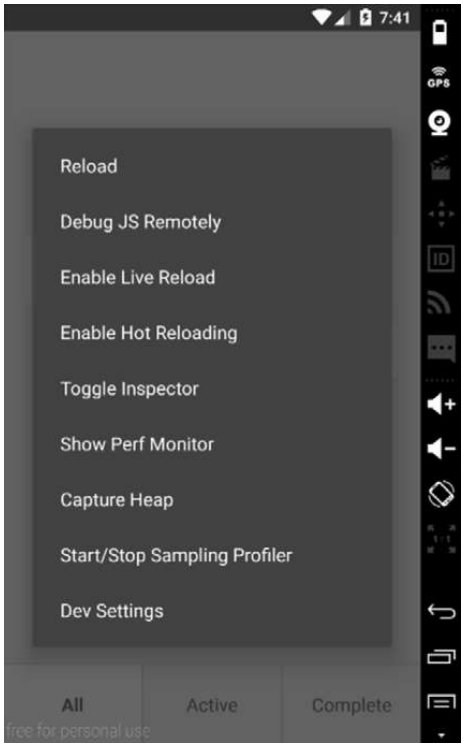


Figure 3.8 React Native developer menu (Android emulator)

3.3.3 Using the developer menu

When the developer menu opens, you should see the following options:

- *Reload (iOS and Android)*—Reloads the app. This can also be done by pressing Cmd-R on the keyboard (iOS) or pressing R twice (Android).
- *Debug JS Remotely (iOS and Android)*—Opens the Chrome dev tools and gives you full debugging support through the browser (figure 3.9). Here, you have access not only to logging statements in your code, but also to breakpoints and whatever you're used to while debugging web apps (with the exception of the DOM). If you need to log any information or data in your app, this is usually the place to do so.

Figure 3.9 Debugging in Chrome

- *Enable Live Reload (iOS and Android)*—Enables live reload. When you make changes in your code, the entire app will reload and refresh in the simulator.
- *Start Systrace (iOS only)*—Systrace is a profiling tool. This will give you a good idea of where your time is being spent during each 16 ms frame while your app is running. Profiled code blocks are surrounded by start/end markers that are then visualized in a colorful chart format. Systrace can also be enabled manually from the command line in Android. If you want to learn more, check out the docs for a very comprehensive overview.
- *Enable Hot Reloading (iOS and Android)*—A great feature added in version .22 of React Native. It offers an amazing developer experience, giving you the ability to see your changes immediately as files are changed without losing the current state of the app. This is especially useful for making UI changes deep in your app without losing state. It's different than live reloading because it retains the current state of your app, only updating the components and state that have been changed (live reloading reloads the entire app, therefore losing the current state).
- *Toggle Inspector (iOS and Android)*—Brings up a property inspector similar to what you see in the Chrome dev tools. You can click an element and see where it is in the hierarchy of components, as well as any styling applied to the element (figure 3.10).

Figure 3.10 Using the inspector (left: iOS, right: Android)

- *Show Perf Monitor (iOS and Android)*—Brings up a small box in the upper-left corner of the app, giving some information about the app's performance. Here you'll see the amount of RAM being used and the number of frames per second at which the app is currently running. If you click the box, it will expand to show even more information (figure 3.11).
- *Dev Settings (Android emulator only)*—Brings up additional debugging options, including an easy way to toggle between the `__DEV__` environment variable being `true` or `false` (figure 3.12).

Figure 3.11 Perf Monitor (left: iOS, right: Android)

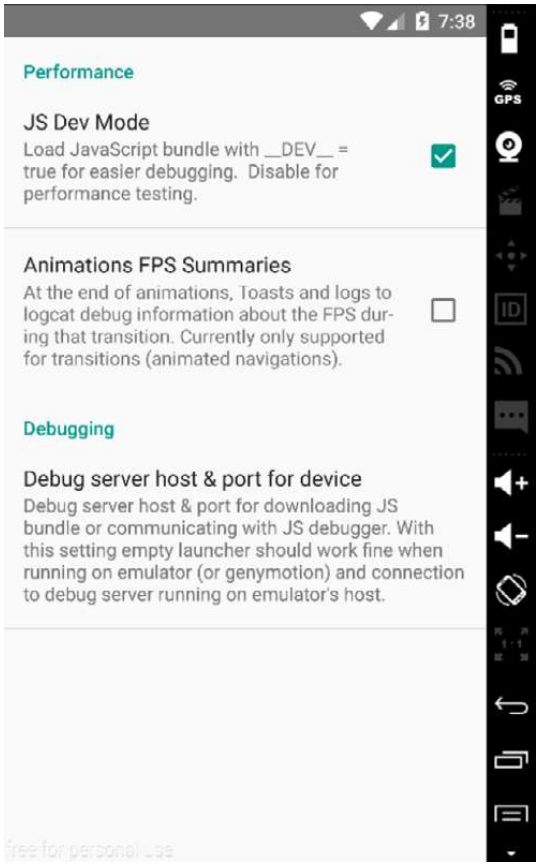


Figure 3.12 Dev Settings (Android emulator)

3.4 Continuing building the todo app

Now that you know how the developer menu works, open it and press Debug JS Remotely to open the Chrome dev tools. You're ready to start logging information to the JavaScript console.

You'll import the `Input` component into app/App.js and attach a method to `TextInput`, which you'll give as a prop to the `Input`. You'll also pass the `inputValue` stored on the state to `Input` as a prop.

Listing 3.8 Creating the `inputChange` function

```
...
import Heading from './Heading'
```

```
import Input from './Input'
class App extends Component {
  constructor() {
    …
  }
  inputChange(inputValue) {
    console.log(' Input Value: ' , inputValue)
    this.setState({ inputValue })
  }
  render() {
    const { inputValue } = this.state
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
          <Input
            inputValue={inputValue}
            inputChange={(text) => this.inputChange(text)} />
        </ScrollView>
      </View>
    )
  }}
```

Creates the inputChange method, which takes inputValue as an argument

Logs out the inputValue value to make sure the method is working

Sets the state with the new value—same as this.setState({inputValue: inputValue})

Passes inputValue as a property to the Input component

Passes inputChange as a property to the Input component

`inputChange` takes one argument, the value of the `TextInput`, and updates the `inputValue` in the state with the returned value from the `TextInput`.

Now, you need to wire up the function with the `TextInput` in the `Input` component. Open app/Input.js, and update the `TextInput` component with the new `inputChange` function and the `inputValue` property.

Listing 3.9 Adding `inputChange` and `inputValue` to the `TextInput`

```
...
const Input = ({ inputValue, inputChange }) => (
  <View style={styles.inputContainer}>
    <TextInput
      value={inputValue}
      style={styles.input}
      placeholder='What needs to be done?'
      placeholderTextColor='#CACACA'
      selectionColor='#666666'
      onChangeText={inputChange} />
  </View>
)
...
```

Destructures the inputValue and inputChange props

Sets the onChangeText method to inputChange

You destructure the props `inputValue` and `inputChange` in the creation of the stateless component. When the value of the `TextInput` changes, the `inputChange` function is called, and the value is passed to the parent component to set the state of `inputValue`. You also set the value of the `TextInput` to be `inputValue`, so you can later control and reset the `TextInput`. `onChangeText` is a method that will be called every time the value of the `TextInput` component is changed and will be passed the value of the `TextInput`.

Run the project again and see how it looks (figure 3.13). You're logging the value of the input, so as you type you should see the value being logged out to the console (figure 3.14).

Figure 3.13 Updated view after adding the `TextInput`

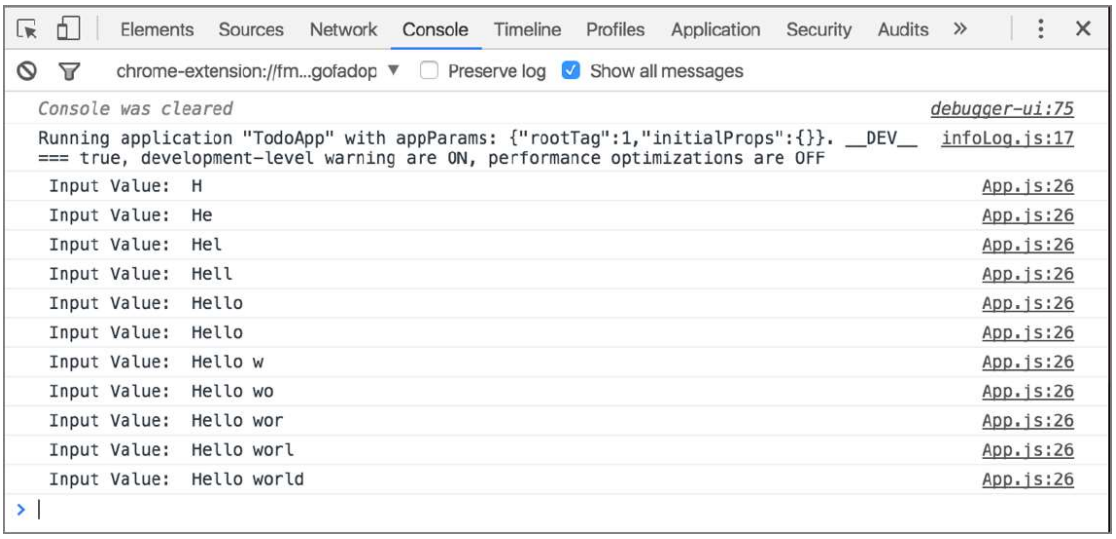
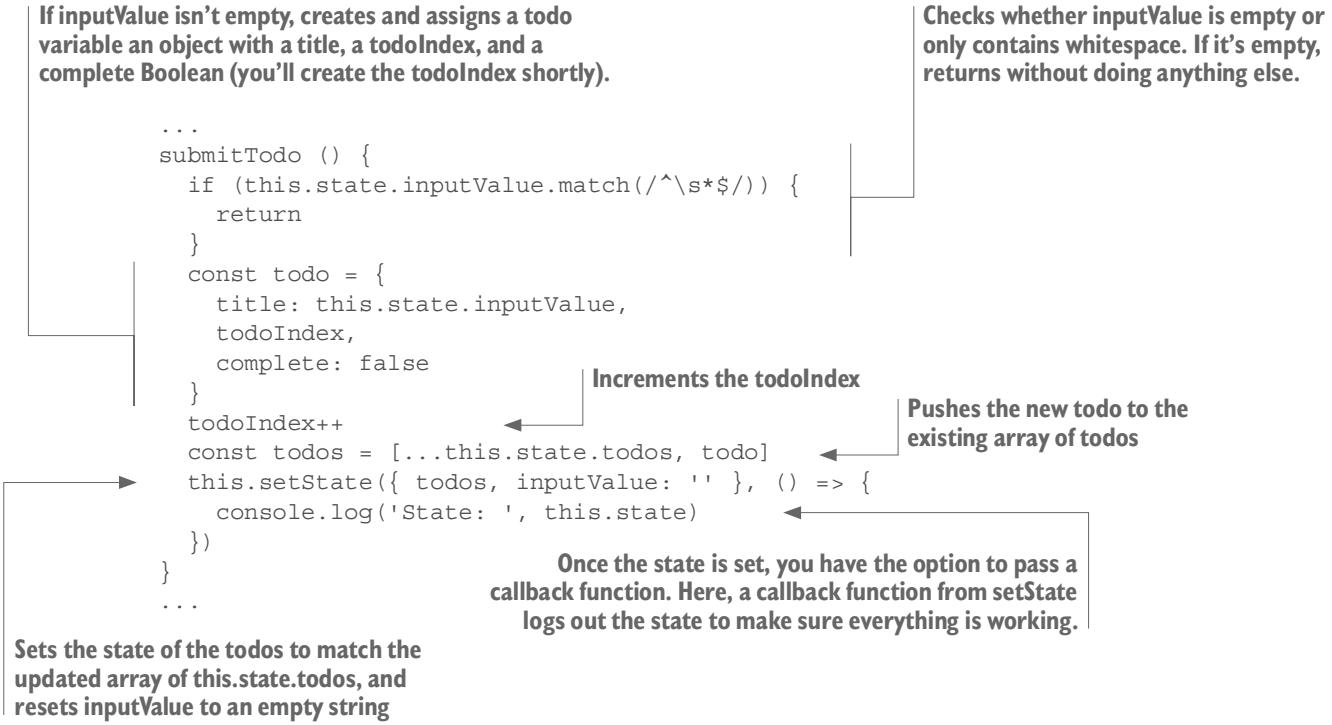


Figure 3.14 Logging out the `TextInput` value with the `inputChange` method

Now that the value of the `inputValue` is being stored in the state, you need to create a button to add the items to a list of todos. Before you do, create a function that you'll bind to the button to add the new todo to the array of todos defined in the constructor. Call this function `submitTodo`, and place it after the `inputChange` function and before the `render` function.

Listing 3.10 Adding the `submitTodo` function




```
...
submitTodo () {
  if (this.state.inputValue.match(/^\s*$/)) {
    return
  }
  const todo = {
    title: this.state.inputValue,
    todoIndex,
    complete: false
  }
  todoIndex++
  const todos = [...this.state.todos, todo]
  this.setState({ todos, inputValue: '' }, () => {
    console.log('State: ', this.state)
  })
}
...
```

Next, create the `todoIndex` at the top of the App.js file, below the last `import` statement.

Listing 3.11 Creating the `todoIndex` variable

```
...
import Input from './Input'

let todoIndex = 0

class App extends Component {
...
```

Now that the `submitTodo` function has been created, create a file called Button.js and wire up the function to work with the button.

Listing 3.12 Creating the `Button` component

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

```
const Button = ({ submitTodo }) => (
  <View style={styles.buttonContainer}>
    <TouchableHighlight
      underlayColor='#efefef'
      style={styles.button}
      onPress={submitTodo}>
      <Text style={styles.submit}>
        Submit
      </Text>
    </TouchableHighlight>
  </View>
)

const styles = StyleSheet.create({
  buttonContainer: {
    alignItems: 'flex-end'
  },
  button: {
    height: 50,
    paddingLeft: 20,
    paddingRight: 20,
    backgroundColor: '#ffffff',
    width: 200,
    marginRight: 20,
    marginTop: 15,
    borderWidth: 1,
    borderColor: 'rgba(0,0,0,.1)',
    justifyContent: 'center',
    alignItems: 'center'
  },
  submit: {
    color: '#666666',
    fontWeight: '600'
  }
})

export default Button
```

Destructures the submitTodo function, which was passed as a prop to the component

Attaches submitTodo to the onPress function available to the TouchableHighlight component. This function will be called when the TouchableHighlight is touched or pressed.

In this component, you use `TouchableHighlight` for the first time. `TouchableHighlight` is one of the ways you can create buttons in React Native and is fundamentally comparable to the HTML `button` element.

With `TouchableHighlight`, you can wrap views and make them respond properly to touch events. On press down, the default `backgroundColor` is replaced with a specified `underlayColor` property that you'll provide as a prop. Here you specify an `underlayColor` of `'#efefef'`, which is a light gray; the background color is white. This will give the user a good sense of whether the touch event has registered. If no `underlayColor` is defined, it defaults to black.

`TouchableHighlight` supports only one main child component. Here, you pass in a `Text` component. If you want multiple components in a `TouchableHighlight`, wrap them in a single `View`, and pass this `View` as the child of the `TouchableHighlight`.

NOTE There's also quite a bit of styling going on in listing 3.12. Don't worry about styling specifics in this chapter: we cover them in depth in chapters 4 and 5. But do look at them, to get an idea how styling works in each component. This will help a lot in the in-depth later chapters, because you'll already have been exposed to some styling properties and how they work.

You've created the `Button` component and wired it up with the function defined in App.js. Now bring this component into the app (app/App.js) and see if it works!

Listing 3.13 Importing the `Button` component

```
...
import Button from './Button'

let todoIndex = 0

...
constructor() {
    super()
    this.state = {
      inputValue: '',
      todos: [],
      type: 'All'
    }
    this.submitTodo = this.submitTodo.bind(this)
  }
...
render () {
    let { inputValue } = this.state
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
          <Input
            inputValue={inputValue}
            inputChange={(text) => this.inputChange(text)} />
          <Button submitTodo={this.submitTodo} />
        </ScrollView>
      </View>
    )
  }
```

Imports the new Button component

Binds the method to the class in the constructor. Because you're using classes, functions won't be auto-bound to the class.

Place the Button below the Input component, and pass in submitTodo as a prop.

You import the `Button` component and place it under the `Input` component in the `render` function. `submitTodo` is passed in to the `Button` as a property called `this.submitTodo`.

Now, refresh the app. It should look like figure 3.15. When you add a todo, the `TextInput` should clear, and the app state should log to the console, showing an array of todos with the new todo in the array (figure 3.16).

Now that you're adding todos to the array of todos, you need to render them to the screen. To get started with this, you need to create two new components: `TodoList` and `Todo`. `TodoList` will render the list of `Todos` and will use the `Todo` component for each individual todo. Begin by creating a file named Todo.js in the app folder.

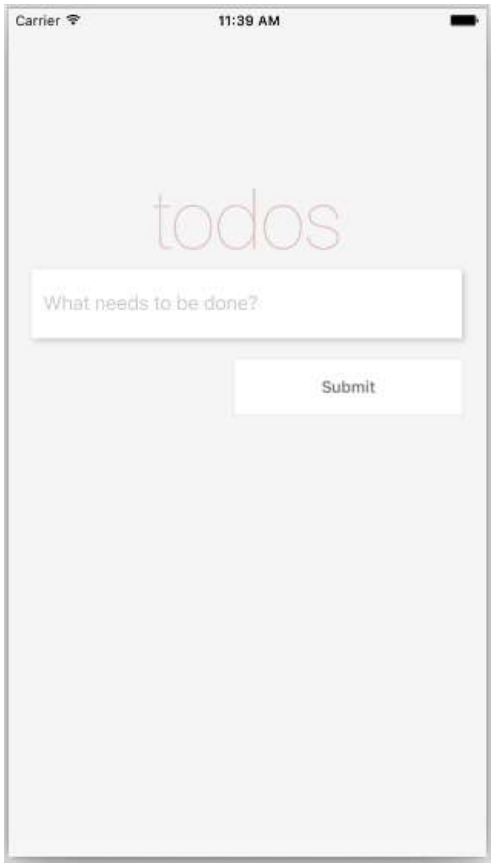


Figure 3.15 Updated app with the `Button` component

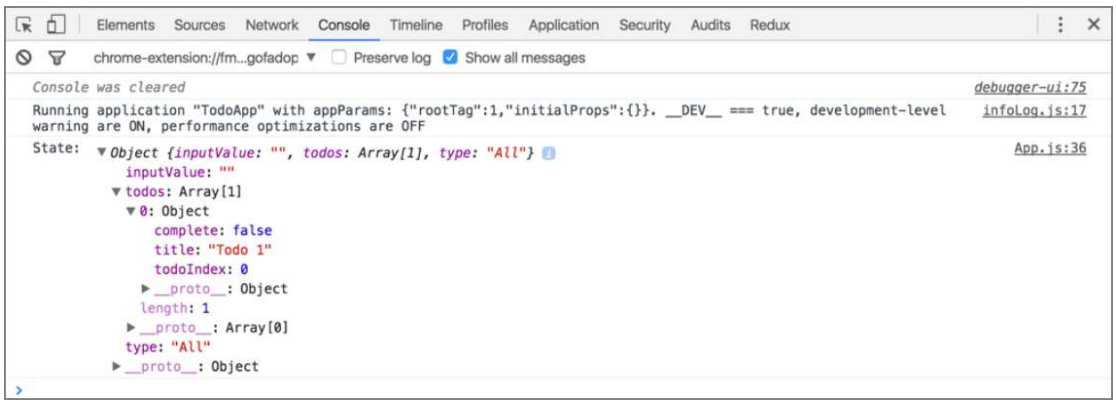


Figure 3.16 Logging the state

Listing 3.14 Creating the `Todo` component

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

const Todo = ({ todo }) => (
  <View style={styles.todoContainer}>
    <Text style={styles.todoText}>
      {todo.title}
    </Text>
  </View>
)

const styles = StyleSheet.create({
  todoContainer: {
    marginLeft: 20,
    marginRight: 20,
    backgroundColor: '#ffffff',
    borderTopWidth: 1,
    borderRightWidth: 1,
    borderLeftWidth: 1,
    borderColor: '#ededed',
    paddingLeft: 14,
    paddingTop: 7,
    paddingBottom: 7,
    shadowOpacity: 0.2,
    shadowRadius: 3,
    shadowColor: '#000000',
    shadowOffset: { width: 2, height: 2 },
    flexDirection: 'row',
    alignItems: 'center'
  },
  todoText: {
    fontSize: 17
  }
})

export default Todo
```

The `Todo` component takes one property for now—a todo—and renders the title in a `Text` component. You also add styling to the `View` and `Text` components.

Next, create the `TodoList` component (app/TodoList.js).

Listing 3.15 Creating the `TodoList` component

```
import React from 'react'
import { View } from 'react-native'
import Todo from './Todo'

const TodoList = ({ todos }) => {
  todos = todos.map((todo, i) => {
    return (
      <Todo
        key={todo.todoIndex}
        todo={todo} />
```

```
    )
  })
  return (
    <View>
      {todos}
    </View>
  )
}

export default TodoList
```

The `TodoList` component takes one property for now: an array of todos. You then map over these todos and create a new `Todo` component (imported at the top of the file) for each todo, passing in the todo as a property to the `Todo` component. You also specify a key and pass in the index of the todo item as a key to each component. The `key` property helps React identify the items that have changed when the diff with the virtual DOM is computed. React will give you a warning if you leave this out.

The last thing you need to do is import the `TodoList` component into the App.js file and pass in the todos as a property.

Listing 3.16 Importing the `TodoList` component

```
...
import TodoList from './TodoList'
...
render () {
    const { inputValue, todos } = this.state
    return (
      <View style={styles.container}>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.content}>
          <Heading />
          <Input inputValue={inputValue} inputChange={(text) => this.
     inputChange(text)} />
          <TodoList todos={todos} />
          <Button submitTodo={this.submitTodo} />
        </ScrollView>
      </View>
    )
  }
...
```

Run the app. When you add a todo, you should see it pop up in the list of todos (figure 3.17).

The next steps are to mark a todo as complete, and to delete a todo. Open App.js, and create `toggleComplete` and `deleteTodo` functions below the `submitTodo` function. `toggleComplete` will toggle whether the todo is complete, and `deleteTodo` will delete the todo.

Figure 3.17 Updated app with the `TodoList` component

Listing 3.17 Adding `toggleComplete` and `deleteTodo` functions

Binds the toggleComplete method to the class in the constructor

Binds the deleteTodo method to the class in the constructor

deleteTodo takes the todoIndex as an argument, filters the todos to return all but the todo with the index that was passed in, and then resets the state to the remaining todos.

```
constructor () {
  ...
  this.toggleComplete = this.toggleComplete.bind(this)
  this.deleteTodo = this.deleteTodo.bind(this)
}
...
deleteTodo (todoIndex) {
  let { todos } = this.state
  todos = todos.filter((todo) => todo.todoIndex !== todoIndex)
  this.setState({ todos })
}
```

```
toggleComplete (todoIndex) {
  let todos = this.state.todos
  todos.forEach((todo) => {
    if (todo.todoIndex === todoIndex) {
      todo.complete = !todo.complete
    }
  })
  this.setState({ todos })
}
...
```

toggleComplete also takes the todoIndex as an argument, and loops through the todos until it finds the todo with the given index. It changes the complete Boolean to the opposite of complete's current setting, and then resets the state of the todos.

To hook in these functions, you need to create a button component to pass in to the todo. In the app folder, create a new file called TodoButton.js.

Listing 3.18 Creating TodoButton.js

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

const TodoButton = ({ onPress, complete, name }) => (
  <TouchableHighlight
    onPress={onPress}
    underlayColor='#efefef'
    style={styles.button}>
    <Text style={[
      styles.text,
      complete ? styles.complete : null,
      name === 'Delete' ? styles.deleteButton : null ]}
     >
      {name}
    </Text>
  </TouchableHighlight>
)

const styles = StyleSheet.create({
  button: {
    alignSelf: 'flex-end',
    padding: 7,
    borderColor: '#ededed',
    borderWidth: 1,
    borderRadius: 4,
    marginRight: 5
  },
  text: {
    color: '#666666'
  },
  complete: {
    color: 'green',
    fontWeight: 'bold'
  },
  deleteButton: {
    color: 'rgba(175, 47, 47, 1)'
  }
})
export default TodoButtton
```

Takes onPress, complete, and name as props

Checks whether complete is true, and applies a style

Checks whether the name property equals "Delete" and, if so, applies a style

Now, pass the new functions as props to the `TodoList` component.

Listing 3.19 Passing `toggleComplete` and `deleteTodo` as props to `TodoList`

```
render () {
  ...
        <TodoList
          toggleComplete={this.toggleComplete}
          deleteTodo={this.deleteTodo}
          todos={todos} />
        <Button submitTodo={this.submitTodo} />
  ...
}
```

Next, pass `toggleComplete` and `deleteTodo` as props to the `Todo` component.

Listing 3.20 Passing `toggleComplete` and `deleteTodo` as props to `ToDo`

```
...
const TodoList = ({ todos, deleteTodo, toggleComplete }) => {
  todos = todos.map((todo, i) => {
    return (
      <Todo
        deleteTodo={deleteTodo}
        toggleComplete={toggleComplete}
        key={i}
        todo={todo} />
    )
  })
...
```

Finally, open Todo.js and update the `Todo` component to bring in the new `TodoButton` component and some styling for the button container.

Listing 3.21 Updating Todo.js to bring in `TodoButton` and functionality

```
import TodoButton from './TodoButton'
...
const Todo = ({ todo, toggleComplete, deleteTodo }) => (
  <View style={styles.todoContainer}>
    <Text style={styles.todoText}>
      {todo.title}
    </Text>
    <View style={styles.buttons}>
      <TodoButton
        name='Done'
        complete={todo.complete}
        onPress={() => toggleComplete(todo.todoIndex)} />
      <TodoButton
        name='Delete'
        onPress={() => deleteTodo(todo.todoIndex)} />
    </View>
  </View>
)
```

```
const styles = StyleSheet.create({
...
  buttons: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'flex-end',
    alignItems: 'center'
  },
...
)}
```

You add two `TodoButtons`: one named Done, and one named Delete. You also pass `toggleComplete` and `deleteTodo` as functions to be called as the `onPress` you defined in TodoButton.js. If you refresh the app and add a todo, you should now see the new buttons (figure 3.18).

If you click Done, the button text should be bold and green. If you click Delete, the todo should disappear from the list of todos.

You're now almost done with the app. The final step is to build a tab bar filter that will show either all the todos, only the complete todos, or only the incomplete todos. To get this started, you'll create a new function that will set the type of todos to show.

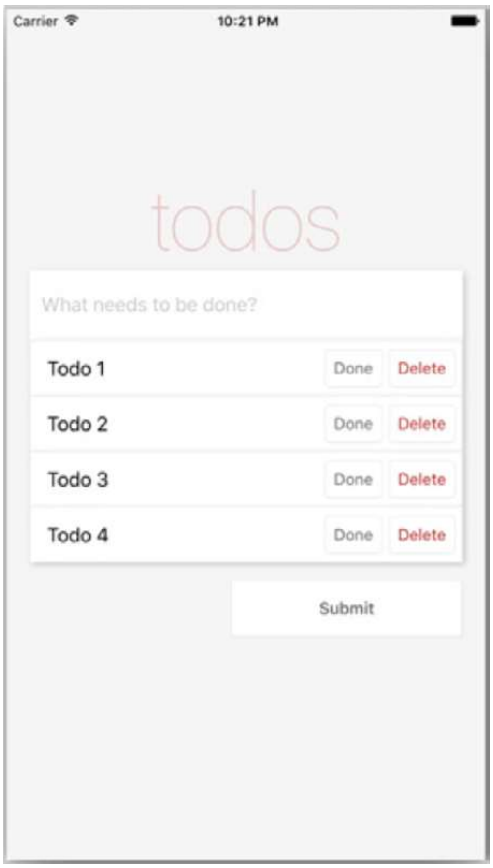


Figure 3.18 App with `TodoButtons` displayed

In the constructor, you set a state `type` variable to `'All'` when you first created the app. You'll now create a function named `setType` that will take a type as an argument and update the type in the state. Place this function below the `toggleComplete` function in App.js.

Listing 3.22 Adding the `setType` function

```
constructor () {
  ...
  this.setType = this.setType.bind(this)
}
...
setType (type) {
  this.setState({ type })
}
...
```

Next, you need to create the `TabBar` and `TabBarItem` components. First, create the `TabBar` component: add a file in the app folder named TabBar.js.

Listing 3.23 Creating the `TabBar` component

```
import React from 'react'
import { View, StyleSheet } from 'react-native'
import TabBarItem from './TabBarItem'

const TabBar = ({ setType, type }) => (
  <View style={styles.container}>
    <TabBarItem  type={type} title='All'
      setType={() => setType('All')} />
    <TabBarItem type={type} border title='Active'
      setType={() => setType('Active')} />
    <TabBarItem type={type} border title='Complete'
      setType={() => setType('Complete')} />
  </View>
)

const styles = StyleSheet.create({
  container: {
    height: 70,
    flexDirection: 'row',
    borderTopWidth: 1,
    borderTopColor: '#dddddd'
  }
})

export default TabBar
```

This component takes two props: `setType` and `type`. Both are passed down from the main `App` component.

You're importing the yet-to-be-defined `TabBarItem` component. Each `TabBarItem` component takes three props: `title`, `type`, and `setType`. Two of the components also take a `border` prop (Boolean), which if set will add a left border style.

Next, create a file in the app folder named TabBarItem.js.

Listing 3.24 Creating the `TabBarItem` component

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

const TabBarItem = ({ border, title, selected, setType, type }) => (
  <TouchableHighlight
    underlayColor='#efefef'
    onPress={setType}
    style={[
      styles.item, selected ? styles.selected : null,
      border ? styles.border : null,
      type === title ? styles.selected : null ]}>
    <Text style={[ styles.itemText, type === title ? styles.bold : null ]}>
      {title}
    </Text>
  </TouchableHighlight>
)

const styles = StyleSheet.create({
  item: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  border: {
    borderLeftWidth: 1,
    borderLeftColor: '#dddddd'
  },
  itemText: {
    color: '#777777',
    fontSize: 16
  },
  selected: {
    backgroundColor: '#ffffff'
  },
  bold: {
    fontWeight: 'bold'
  }
})

export default TabBarItem
```

In the `TouchableHighlight` component, you check a few props and set styles based on the prop. If `selected` is `true`, you give it the style `styles.selected`. If `border` is `true`, you give it the style `styles.border`. If `type` is equal to the `title`, you give it `styles.selected`.

In the `Text` component, you also check to see whether `type` is equal to `title`. If so, add a bold style to it.

To implement the `TabBar`, open app/App.js, bring in the `TabBar` component, and set it up. You'll also bring in `type` to the `render` function as part of destructuring `this.state`.

Listing 3.25 Implementing the `TabBar` component

```
...
import TabBar from './TabBar'
class App extends Component {
...
render () {
  const { todos, inputValue, type } = this.state
  return (
    <View style={styles.container}>
      <ScrollView
        keyboardShouldPersistTaps='always'
        style={styles.content}>
        <Heading />
        <Input inputValue={inputValue}
               inputChange={(text) => this.inputChange(text)} />
        <TodoList
          type={type}
          toggleComplete={this.toggleComplete}
          deleteTodo={this.deleteTodo}
          todos={todos} />
        <Button submitTodo={this.submitTodo} />
      </ScrollView>
      <TabBar type={type} setType={this.setType} />
    </View>
  )
}
...
```

Here, you bring in the `TabBar` component. You then destructure `type` from the state and *pass it not only to the new* `TabBar` *component, but also to the* `TodoList` *component*; you'll use this `type` variable in just a second when filtering the todos based on this type. You also pass the `setType` function as a prop to the `TabBar` component.

The last thing you need to do is open the `TodoList` component and add a filter to return only the todos of the type you currently want back, based on the tab that's selected. Open TodoList.js, destructure the type out of the props, and add the following `getVisibleTodos` function before the `return` statement.

Listing 3.26 Updating the `TodoList` component

```
...
const TodoList = ({ todos, deleteTodo, toggleComplete, type }) => {
  const getVisibleTodos = (todos, type) => {
    switch (type) {
      case 'All':
        return todos
      case 'Complete':
        return todos.filter((t) => t.complete)
      case 'Active':
```

```
        return todos.filter((t) => !t.complete)
      }
    }

    todos = getVisibleTodos(todos, type)
    todos = todos.map((todo, i) => {
...
```

You use a `switch` statement to check which type is currently set. If `'All'` is set, you return the entire list of todos. If `'Complete'` is set, you filter the todos and only return the complete todos. If `'Active'` is set, you filter the todos and only return the incomplete todos.

You then set the `todos` variable as the returned value of `getVisibleTodos`. Now you should be able to run the app and see the new `TabBar` (figure 3.19). The `TabBar` will filter based on which type is selected.

Figure 3.19 Final todo app

Summary

- `AppRegistry` is the JavaScript entry point to running all React Native apps.
- The React Native component `TextInput` is similar to an HTML `input`. You can specify several props, including a `placeholder` to show text before the user starts to type, a `placeholderTextColor` that styles the placeholder text, and a `selectionColor` that styles the cursor for the `TextInput`.
- `TouchableHighlight` is one way to create buttons in React Native; it's comparable to the HTML `button` element. You can use `TouchableHighlight` to wrap views and make them respond properly to touch events.
- You learned how to enable the developer tools in both iOS and Android emulators.
- Using the JavaScript console (available from the developer menu) is a good way to debug your app and log useful information.