MANNING

Developing iOS and Android Apps with JavaScript

React Native IN ACTION

MEAP

Nader Dabit

MEAP Edition
Manning Early Access Program
React Native in Action
Developing iOS and Android Apps with JavaScript
Version 9

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

welcome

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

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

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

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

—Nader Dabit

brief contents

1

Getting started with React Native

This chapter covers

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

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

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

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

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

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

1.1 Introducing React and React Native

React Native is a framework for building native mobile apps in JavaScript using the React JavaScript library and compiles to real native components. If you're not sure what React is, it is a JavaScript library open sourced by and used within Facebook, and 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. Besides being backed and open sourced by Facebook, it also has a tremendous community of motivated people behind it. Facebook groups, with its millions of users, is powered by React Native as well as Facebook Ads Manager. Discord and li.st are also built with React Native and Discovery VR has harnessed the framework to build a beautiful and complex virtual reality video application.

With React Native, developers can build native views and access native platform-specific components using JavaScript. This sets React Native apart from hybrid app frameworks, as hybrid apps package a web view into a native application and are built using HTML & CSS.

There are many benefits to choosing React Native as a mobile application framework. Because the application renders native components and APIs directly, the speed and performance are much better than hybrid frameworks such as Cordova or Ionic. With React Native we are writing our entire application using a single programming language, Javascript, so a lot of code can be reused, therefore reducing the time it takes to get a cross platform application shipped. Hiring and finding quality JavaScript developers is much easier and cheaper than hiring Java or Objective C / Swift developers, leading to an overall less expensive process.

React Native applications are built using JavaScript and JSX. JSX is something we will be discussing in depth in this book, but for now think of it as a JavaScript syntax extension that looks like html or xml.

1.1.1 A Basic React Native Class

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

There are two main types of React Native components, stateless and stateful.

Listing 1.1 Stateful component using React.createClass

```
var HelloWorld = React.createClass({
  render () {
    return (
```

```
<SomeComponent />
    )
  }
})
```

Listing 1.2 Stateful component using ES6 class

```
class HelloWorld extends React.Component ({
  render () {
    return (
<SomeComponent />
    )
  }
})
```

Listing 1.3 Stateless component

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

The main difference is that the stateless components do not hook into any lifecycle methods and therefore hold no state of their own, so any data to be rendered has to be passed down as props. We will discuss all of this in depth later in this chapter, but for now, we will talk about creating React Native components using ES6 classes.

To get started and begin understanding the flow of React Native, let's walk through and go over what happens when a basic React Native Component class is created and rendered (figure 1.1, listing 1.4).

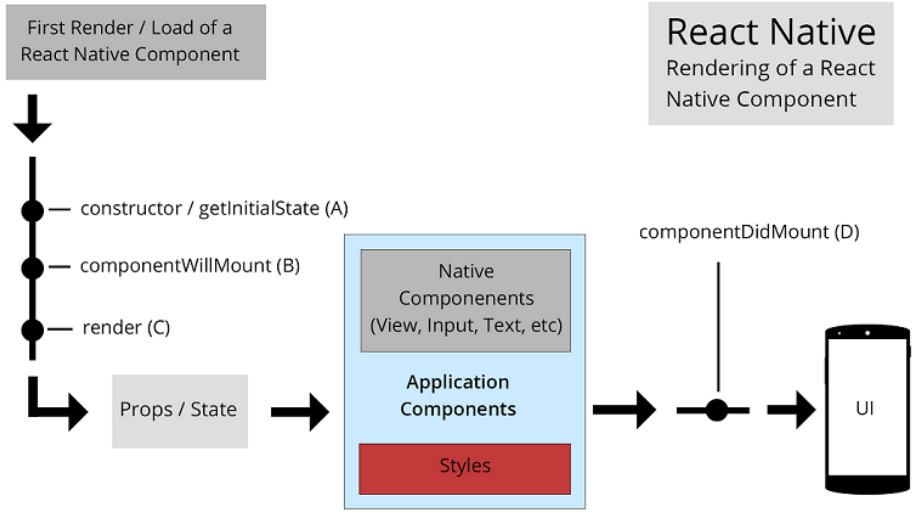


Figure 1.1 Rendering a React Native class

Listing 1.4 Creating a basic Reacct Native class component

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

class HelloWorld extends React.Component {
  constructor () {  ❶
    super()
    this.state = {
      name: 'React Native in Action'
    }
  }
  componentWillMount () { ❷
    console.log('about to mount..')
  }
  componentDidMount () { ❹
    console.log('mounted..')
  }
  render () { ❸
    return (
<View style={styles.container}>
<Text>{this.state.name}</Text>
</View>
    )
  }
}

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

> **Something to keep in mind when we discuss the following methods it the concept of mounting. When a component is mounted, or created, the react component lifecycle is instantiated, triggering the methods we used above and will discuss below.**

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

When the component first loads, we set a state object with the variable `name` in the constructor **(❶)** . For data in a React Native application to be dynamic, it either needs to be set in the state or passed down as props. Here, we have set the state in the constructor and can therefore change it if we would like by calling:

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

which would rerender the component. If we did not set the variable in the state, we would not be able to update the variable in the component.

The next thing to happen in the lifecycle is `componentWillMount` is called **(❷)** .`componentWillMount` is called before the rendering of the UI occurs.

`render` is then called **(❸)** , which examines props and state and then must return either a single React Native element, `null`, or `false`. This means that 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` **(❹)** . 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 we can see our result.

1.1.2 React Lifecycle

When a React Native Class is created, there are methods that are instantiated that we can hook into. These methods are called lifecycle methods, and we will cover them in depth in chapter 2. The methods we saw in figure 1.1were `constructor`, `componentWillMount`, `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 we would like. The only lifecycle method that is required is the `render` method, all of the others are optional.

When working with React Native, we are fundamentally working with the same lifecycle methods and specifications as one would use when using React.

1.2 What You Will Learn

In this book, we will cover everything you will 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 will begin by covering and thoroughly explaining how React works in Chapter 2.

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

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

There are many ways to implement navigation, each with their nuances, pros and cons. We will discuss navigation in depth and cover how to build robust navigation using the most important of the navigation APIs. We will be covering not only the native navigation APIs that

come out of the box with React Native, but also a couple of community projects available through npm.

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

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

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

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

1.3 What You Should Know

To get the most out of this book, you should have a beginner to intermediate knowledge of JavaScript. Much of our work will be done with the command line, so a basic understanding of how to use the command line is also needed. You should also understand what npm is and how it works on at least a fundamental level. If you will be building in iOS, a basic understanding of Xcode is beneficial and will speed things along, but is not absolutely necessary. Fundamental knowledge of newer JavaScript features implemented in the es2015 release the JavaScript programming language is beneficial but not necessary. Some conceptual knowledge on MVC frameworks and Single Page Architecture is also good but not absolutely necessary.

1.4 Understanding how React Native works

1.4.1 JSX

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

1.4.2 Threading

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

1.4.3 React

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

1.4.4 Unidirectional data flow

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

1.4.5 Diffing

React takes this idea of diffing and applies it to native components. It takes your UI and sends the smallest amount of data to the main thread to render it with native components. Your UI is declaratively rendered based on the state, and React uses diffing to send the necessary changes over the bridge.

1.4.6 Thinking in components

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

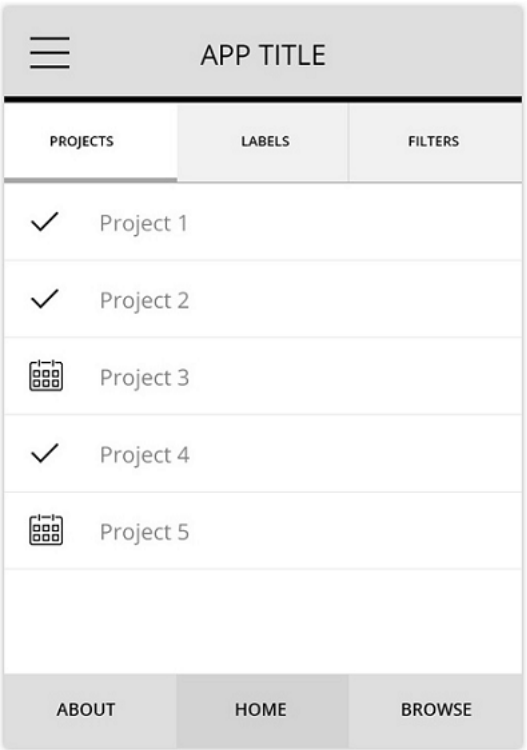


Figure 1.2 Final example app design / structure

The first thing to do is to mentally break the UI elements up into what they actually represent. So, in the above mockup, we have a header bar, and within the header bar we have a title and a menu button. Below the header we have a tab bar, and within the tab bar we have three individual tabs. Go through the rest of the mockup and think of what the rest of the items might be as well. These items that we are identifying will be translated into components. This is the way you should think about composing your UI. When working with React Native, you should break down common elements in your UI into reusable components, and define their interface accordingly. When you need this element any time in the future, it will be available for you to reuse.

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

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

Figure 1.3 App structure broken down into separate components

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

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

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

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

```
TabBar:

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

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

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

1.5 Acknowledging the strengths

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

1.5.1 Developer availability

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

1.5.2 Developer productivity

Traditionally, to build a cross platform mobile application, you needed both an Android team as well as an iOS team. React Native allows you to build your both Android, iOS, and soon Windows applications using only a single programming language, JavaScript, and 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 are no longer tied down to being only an Android or iOS developer, opening the door for a lot of opportunity. This is great news for JavaScript developers as well,

allowing them to spend all of their time in one state of mind when switching between web and mobile projects. It is also a win for teams who were traditionally split between Android and iOS, as they can now work together on a single codebase.

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

1.5.3 Performance

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

1.5.4 One-way data flow

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

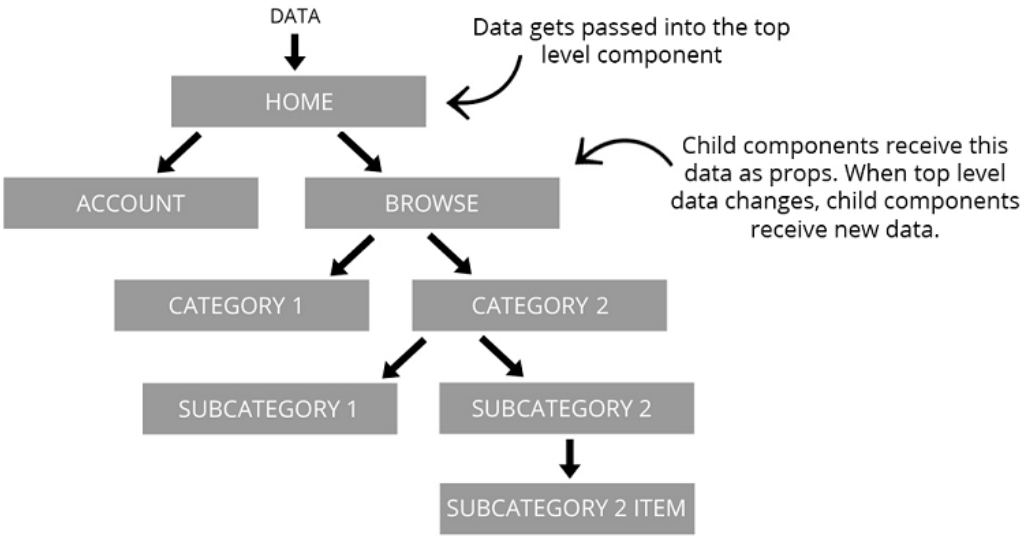


Figure 1.4 Explanation of how one-way data flow works

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

React Native also has something called Hot Reloading built in. What does this mean? Well, while developing an application, imaging having to click a few times into your app to get to the place in your app that you are developing for. While using Hot Reloading, when you make a code change, you do not have to reload and click back through the app to get to the current state. While using this feature in React Native, you simply save the file and the application reloads only the component which you have made changes to, instantly giving you feedback and updating the current state of the ui without having to click back through the app to get to the state you are working in.

1.5.6 Transpilation

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

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

1.5.7 Productivity and efficiency

Native mobile development is becoming more and more expensive, and because of this, engineers who can deliver applications across platforms and stacks will become extremely

valuable and in demand. Once React Native, or something like it if it comes along, makes developing desktop and web as well as mobile applications using a single framework mainstream, there will be a restrucuring and rethinking of how engineering teams are organized. Instead of a developer being specialized in a certain platform, such as iOS or web, they will be in charge of 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 only able to deliver web applications.

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

1.5.8 Community

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

1.5.9 Open source

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

1.5.10 Immediate updates

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

1.5.11 Drawbacks

Now that we've gone over all of the benefits of using React Native, let's take a look at a few reasons and circumstances where you may not want to choose the framework.

First, React Native is still immature when compared with other platforms such as native iOS and Android, as well as Cordova. The feature parity is not there yet with either native or Cordova. While most functionality is now built in, there may be times where you need some functionality that is not yet available, and this means you will either have to dig into the native code to build it yourself, hire someone to do it, or not implement the feature at all.

Another thing to think about is the fact that you and or your team will have to learn a completely new technology if you are not already familiar with React. While most people seem to agree that React is easy to pick up, if you are 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 as well.

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

1.5.12 Conclusion

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

1.6 Creating and using basic components

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

1.6.1 Components

At the core of React Native is the concept of components. React Native has built in components that you will see me describe as Native components, and you will also build components using the framework. Components are a collection of data and UI elements that make up your views and ultimately your application. We will go in depth on how to build, create and use components in this book.

As we mentioned earlier, React Native components are built using JSX. Let's run through a couple of basic examples of what JSX in React Native looks like vs HTML:

Table 1.1 JSX components vs HTML elements

1. Text Component

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

2.View Component

HTML	React Native JSX

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

3. Touchable Highlight

HTML	React Native JSX

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

As you can see in table 1.1, JSX looks very similar to HTML or XML.

1.6.2 Native components

The framework offers native components out of the box, such as View, Text, and Image, among others. We will create our own components using these Native components as building blocks. For example, we may use the following markup to create our own Button component using React Native TouchableHighlight, and Text components (listing 1.5).

Listing 1.5 Creating button component

```
const Button = () => (
   <TouchableHighlight>
      <Text>Hello World</Text>
   </TouchableHighlight>
)
export default Button
```

We can then import and use our new button like this (listing 1.6).

Listing 1.6 Importing and using Button component

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

Next, we will go through the fundamentals of what a component is, how they fit into the workflow, as well as common use cases and design patterns for building them.

1.6.3 Component composition

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

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

We'll be creating this component:

```
<MyComponent />
```

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

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

1. CREATECLASS SYNTAX (ES5, JSX)

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

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

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

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

2. CLASS SYNTAX (ES2015, JSX)

Another way to create React Native components is using ES2015 classes. This is the way we will be creating our *stateful* components for the rest of the book and is now the recommended way to do so by the community and creators of React Native, though we will be using *stateless* components whenever we can.

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

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

3. STATELESS (REUSABLE) COMPONENT (JSX)

Since the release of React 0.14, we have had the ability to create what are called stateless or reusable components. We have yet dived into state, but just remember that these components are basically pure functions of their props, and do not contain their own state, so their state cannot be mutated. 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 <Text>HELLO FROM STATELESS</Text>
}
```

4. CREATEELEMENT (JAVASCRIPT)

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

`React.createElement` takes a few arguments:

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

Let's walk through these arguments:

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

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

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

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

Which is the same as declaring the component like this:

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

1.6.4 Exportable components

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

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

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

export default Home
```

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

IMPORTING

The following code imports and React Native variable declarations:

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

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

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

```
varReact = require('react')
```

The above statement without object destructuring would look like this:

```
import React = from 'react'
constComponent = React.Component
import ReactNative from 'react-native'
constText = ReactNative.Text
constView = 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 we are creating a new instance of a React Native Component class by extending it, and naming it Home. As you can see, before we declared React.Component, we are now just declaring Component. This is because we have imported the Component element in the object destructuring statement, giving us access to Component as opposed to having to call React.Component.

THE RENDER METHOD

Next, take a look at the Render method:

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

The code for the component gets executed in the render method, and the content after the return statement returns what is 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 do not manipulate the state of the component, you can do so here in between the render() method and the return statement.

EXPORTS

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

```
export default Home
```

1.6.5 Combining components

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

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

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

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

      </View>)
    }
  }
```

In the same file, below the Home class declaration, let's start by building out a Header component:

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

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

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

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

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

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

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

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

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

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

As you can see, the code we just wrote is extremely declarative, meaning it's written in such a way that it describes what you want to do, and is easy to understand in isolation.

This is a very high level overview of how we will be creating our components and views in React Native, but should give you a very good idea of how the basics work.

1.7 Creating a starter project

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

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

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

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

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

```
react-native init myProject
```

`myProject` can be any name that you choose.

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

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

- android – This folder contains all of the android platform specific code and

dependencies. You will not need to go into this folder unless you are either implementing a custom bridge into android, or if you install a plugin that calls for some type of deep configuration

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

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

Listing 1.7Index.ios.js / index.android.js

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

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

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

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

```
StyleSheet
AppRegistry
```

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

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

This corresponds directly to:

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

At the bottom of the file, you see

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

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

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

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

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

1.8 Summary

- React Native is a framework for building native mobile apps in JavaScript using the React JavaScript library.
- Some of React Native's strengths are its performance, developer experience, ability to

build cross platform with a single language, one-way data flow, and community. You may consider React Native over hybrid mainly because of its performance, and over Native mainly because of the developer experience and cross platform ability with a single language.

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

2

Understanding React

This chapter covers

- State: how it works and why it is important
- Props: how it works and why it is important
- Understanding the React Component Specification and what it means
- Implementing and understanding React lifecycle methods

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

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

You will see both React and React Native referenced in this chapter. Keep in mind that when I am referencing React, I'll be 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 State

State and props are the way that data is handled and passed down in a React or React Native application. First, we will look at state, and then props.

Table2.1 Props vs State

Props	State
1) external data	1) internal data
2) immutable	2) mutable
3) inherited from parent	3) created in the component
4) can be changed by parent component	4) can only be updated in the component
5) can be passed down as props	5) can be passed down as props
6) cannot change inside component	6) can change inside component

2.1.1 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, React rerenders the component. If any child components are inheriting this state as props, then all of the child components get 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. state also is what allows us to create components that are dynamic and interactive. The main point to understand when differentiating between state and props is that state is mutable, while props are immutable.

SETTING INITIAL STATE

State is initialized when the component is created, either in the getInitialState function (React.createClass) or the constructor (ES2015 class). Once the state is initialized, it is then available in the component as this.state.

The `getInitialState` function is invoked once before the component is mounted (listing 2.1). The returned value will be used as the state value. `getInitialState` will only work when a component is instantiated with `React.createClass`, and not when using a constructor. This is one of the React lifecycle methods. We will learn more about component lifecycle methods and what it means for the component to be mounted in the next section, but for now just understand that this function is called right before the component is created.

Listing 2.1 Setting state with getInitialState()

```
const MyComponent = React.createClass({
  getInitialState() {
    return {
      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 that a JavaScript class is instantiated (listing 2.2). This is not a React lifecycle method, but a regular JavaScript class method.

Listing 2.2 Setting state with a constructor (class syntax)

```
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 takes the place of the getInitialState method when defining the initial state in a React component when defined as a class.

UPDATING STATE

State can be updated in one of two ways: setState and replaceState.

- setState merges the previous state with the current state.

- replaceState will remove the entire previous state and replace it with whatever is provided.

setState will be used unless there is an extreme circumstance where some keys need to be removed from the state. We will not even go into an example of replaceState as the method is not available on ES6 classes and will probably be removed entirely in a future version of React, but it is worth knowing that it exists.

Let's look at how to use setState (listing 2.3). To do so, we will introduce a new method, a touch handler called `onPress`. `onPress` can be called on a few types of 'tappable' React Native components, but here we will be attaching it to a Text component to get us started with this basic example. We will be calling a function called updateYear when the Text component is pressed that will update our state with setState. This function will be defined before our render function, as it is 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 does not affect the actual functionality.

Listing 2.3 Updating state

```
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>
    )
  }
}
```

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, as changing the state variable directly will not 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 what we are doing in listing 2.4 does not work when trying to update state. The state

object is updated, but the UI is not updated because `setState` was not called, and the component does not get rerendered.

Listing 2.4 This does not work when updating state

```
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>
    )
  }
}
```

However, there is a method that is available in React that can force an update once a state variable has been changed as we did above. This method is called `forceUpdate`(listing 2.5). Calling forceUpdate will cause render() to be called on the component, therefore triggering a rerendering of the UI.

Listing 2.5 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 have gone over how to work with state using a basic string, let's look at a few other data types. We will attach a Boolean, array, and object to our state and use it in our component. We will also conditionally show a component based on a Boolean in our state (listing 2.6).

Listing 2.6 State with other data types

```
class MyComponent extends Component {
  constructor(){
    super()
    this.state = {
      year: 2016,
      leapYear: true,
      topics: ['React', 'React', '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 Props

props (short for properties) are a component's inherited values or properties that have been passed down from a parent component. props are immutable, and can only be changed by changing the initial values at the top level where they are declared and passed down. props can be either static or dynamic values when they are declared, but when they are inherited they are immutable. React's Thinking in React documentation says that props are best explained as "a way of passing data from parent to child."

A good way to explain how props work is to simply show an example. In listing 2.7, we will declare a book value and pass it down to a child component as a static prop.

Listing 2.7 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>
    )
  }
}
```

As you can see, we've created two components: `<MyComponent />` and `<BookDisplay />`. When we use `<BookDisplay />`, we have passed in a property called book, and set it to a 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 static properties like you would a dynamic variable, by using curly braces and a string value (listing 2.8).

Listing 2.8 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, let's pass a dynamic property to our component.

In our render method, before the return statement, let's declare a variable "book" and pass that variable in as a prop (listing 2.9).

Listing 2.9 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, let's pass a dynamic property to our component using state (listing 2.10).

Listing 2.10 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 we may go about updating the state and therefore the prop that is passed down. Remember, props are immutable, so we will be changing the state of the parent

component, which will update the props and trigger a rerender of both the component and the child component.

Now, let's break this idea up into individual parts and identify what needs to be done:

1. Declare state variable

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

2. Write function that will update state variable

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

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

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

4. Attach function to touch handler in child component

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

Ok, now that we know the pieces we need, let's write all of the code to put this into action. We will be using the components that we used in the previous examples, and adding the additional functionality there (listing 2.11).

Listing 2.11 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 may try using destructuring here, the same way as we have been doing when we declare our React components. Let's write a component using destructuring (listing 2.12).

Listing 2.12 Destructuring state and props

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

As you can see, we are now not having to refer to `this.state` or `this.props` in our actual component when referencing our book, instead we have taken the book variable out of our

state and our props and can reference the variable itself, which also looks a lot cleaner. This starts to make a lot more sense and will keep your code cleaner as your state and props get larger and more complex.

PROPS WITH STATELESS FUNCTIONAL COMPONENTS

Because stateless components only have to worry about props, and do not 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, simply pass in props as the first argument to the function (listing 2.13).

Listing 2.13 Props with stateless components

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

You can also destructure props in the function argument (listing 2.14).

Listing 2.14 Destructuring props in a stateless component

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

As you can see, that looks a lot nicer and cleans up a lot of unnecessary code! We will be using stateless components wherever we can, simplifying our code base and our logic.

PASSING ARRAYS AND OBJECTS AS PROPS

Other data types work exactly as you might expect. For example, if you would like to pass an array, you simply pass in the array as a prop. If you would like to pass an object, you would simply pass in the object as a prop. Let's look at a basic example (listing 2.15).

Listing 2.15 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', 'JavaScript']] />
     )
   }
 }

 constBookDisplay = (props) => {
    letleapyear
    let { topics } = props
    const { info } = props

    topics = topics.map((topic, i) => {
      return <Text key={ i }>{ 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

As stated before, there are two ways to create a component: createClass syntax (`React.createClass`)and Class syntax (`class MyComponent Extends React.Component`).

When creating React and React Native components, there are several specifications and lifecycle methods that you can hook into to control what is going on in your component. In this section we will discuss all of them and try to get a good understanding of what each one does, and when you should use the method.

First, we will go over the basics of the component specifications. The component spefication basically lays out how your component should react to different things happening in the lifecycle of the component. See figure 2.16 for a list of these specifications. We will then break this apart and go into detail of what each one means and how they are used so you will have a clear understanding of what this all means.

Listing 2.16 Component specifications

```
render()
getInitialState – function // React.createClass only
constructor - function & this.state // ES2015 classes only
getDefaultProps – function // React.createClass only
defaultProps – object // ES2015 classes only
propTypes – object
statics – object
```

2.3.1 render

The render method (figure 2.17) is the only method in the component specification required when creating a component, and must return either a single child element, null, or false. This child element can either be component that you have declared yourself, (such as a `<View />` or `<Text />`), or another component that you have defined.

Listing 2.17 Render method

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

You can use the render method with or without parenthesis. If you don't use parenthesis, then the returned element must of course be on the same line as the return statement, as shown in listing 2.18.

Listing 2.18 render method without parenthesis

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

The render method can also return another component that was defined elsewhere, as shown in listing 2.19.

Listing 2.19 render returning predefined component

```
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, as shown in listing 2.20.

Listing 2.20 render method with conditional

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

2.3.2 getInitialState and constructor

getInitialStateis invoked once before the component is mounted. This function is used to set initial state when using React.createClass to create a component.The value returned is then available in the component as this.state, as shown in listing 2.21.

Listing 2.21 get*InitialState*

```
getInitialState() {
 return {
 someNumber: 1,
     someBoolean: false
   }
 }
```

Use a `constructor` method (listing 2.22)to set the initial state when using classes. The concept of classes, as well as the constructor function, is not specific to React or React Native; it's an ES2015 specification and is really 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 we are in.

Listing 2.22 constructor

```
constructor(){
super()
this.state = {
someOtherNumber: 19,
```

```
someOtherBoolean: true
}
  this.name = 'Hello World'
  this.type = 'class'
  this.loaded = false
}
```

When using a constructor, you must use the super keyword before you can use the `this` keyword. Also, if you need access to any props passed to the component, they must be passed as an argument to the constructor and the super call. Setting the state based on props is usually not good practice unless you are intentionally setting some type of seed data for the component's internal functionality, as the data is no longer going to be consistent across components if it is changed. This is because `getInitialState` is only invoked when the component is first mounted or created, and not again. This means that if you rerender the same component using a different new value as a prop, this already mounted component and any children inheriting this data as props will not receive this new data (listing 2.23).

Listing 2.23 constructor with props

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

2.3.3 Default Props

When creating your component, you may want to set a default value for props being passed into the component. If there are no props passed, the default value will be available to the component. If the prop is passed, the default value will be overwritten. There are different ways to do this depending on how you are writing your component, either with React.createClass or using ES2015 classes.

`getDefaultProps` (React.createClass) This method is invoked and cached once the component is created (listing 2.24). Anything returned here will be available as this.props.

Listing 2.24 Using getDefaultProps with React.createClass

```
const MainComponent = React.createClass({
   render() {
     return <ChildComponent firstName='Sammy' />
   }
 })

const ChildComponent = React.createClass ({
   getDefaultProps(){
     return {
       firstName: 'Nader',
       lastName: 'Dabit',
       favoriteColor: 'blue'
```

```
    }
  },
  render() {
    return (
     <View>
       <Text>{ this.props.firstName }</Text>
       <Text>{ this.props.lastName }</Text>
       <Text>{ this.props.favoriteColor }</Text>
     </View>
    )
  }
})
```

The component in figure 2.24 will return this:

- Sammy // defined in parent component
- Dabit // not defined in parent component, fall back to `getDefaultProps` value
- blue // not defined in parent component, fall back to `getDefaultProps` value

`defaultProps (ES2015 class syntax)` Setting default props when using classes is a little different. `defaultProps` are defined as properties on the component instead of inside the class body, as shown in listing 2.25.

Listing 2.25 Listing 2.6 Using defaultProps with es2015 classes

```
const MainComponent = React.createClass({
  render() {
    return <ChildComponent firstName='Sammy' />
  }
})

const ChildComponent = React.createClass ({
  render() {
    return (
     <View>
       <Text>{ this.props.firstName }</Text>
       <Text>{ this.props.favoriteColor }</Text>
     </View>
    )
  }
})

ChildComponent.defaultProps = {
  firstName: 'Nader',
  favoriteColor: 'blue'
}
```

`defaultProps` can also be set as a property on a stateless component, just like we did with the ES2015 classes, as shown in listing 2.26.

Listing 2.26 defaultProps with stateless components

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

```
    return <ChildComponent firstName='Sammy' />
  }
}

const ChildComponent = (props) => (
   <View>
     <Text>{ props.firstName }</Text>
       <Text>{ props.lastName }</Text>
     <Text>{ props.favoriteColor }</Text>
   </View>
)

ChildComponent.defaultProps = {
  firstName: 'Nader',
  lastName: 'Dabit',
  favoriteColor: 'blue'
}
```

2.3.4 propTypes

Sometimes, you need to ensure that a certain type of prop is passed into your component and whether or not the prop is required. To do this, you can set a `propTypes` object, and check to make sure certain props are passed in, then specifying if they are required. If the props are not passed in and they are required, a warning will be shown in the JavaScript console. You will also get a warning if the incorrect type of prop is passed in. There are two ways to set this up depending on whether you are using `React.createClass` syntax or class syntax. Let's look at how to set this up (listing 2.27).

Listing 2.27 propTypes with react.createClass

```
const MainComponent = React.createClass ({
render() {
 return (
    <ChildComponent
        firstName='Nader'
        favoriteLanguages={['JavaScript', 'Python']}
        about={{age:24, language: 'English'}} />
    )
  }
})

const ChildComponent = React.createClass ({
  propTypes: {
    firstName: React.PropTypes.string,
    favoriteLanguages: React.PropTypes.array,
    about: React.PropTypes.object
  },

  render() {
    return (
     <View>
       <Text>Name: {this.props.firstName}</Text>
       <Text>Fav language: {this.props.favoriteLanguages[0]}</Text>
       <Text>Age: {this.props.about.age}</Text>
```

```
      </View>
    )
  }
})
```

Listing 2.28 Using propTypes with es2015 classes

```
class MainComponent extends Component {
  render() {
    return (
     <ChildComponent
        firstName='Nader'
        favoriteLanguages={['JavaScript', 'Python']}
        about={{age:24, language: 'English'}} />
    )
  }
}

class ChildComponent extends Component {
  render() {
    return (
     <View>
       <Text>Name: {this.props.firstName}</Text>
       <Text>Fav language: {this.props.favoriteLanguages[0]}</Text>
       <Text>Age: {this.props.about.age}</Text>
     </View>
    )
  }
}

ChildComponent.propTypes = {
  firstName: React.PropTypes.string,
  favoriteLanguages: React.PropTypes.array,
  about: React.PropTypes.object
}
```

You can also use `propTypes` with stateless component by defining `propTypes` as a property of the component (listing 2.29).

Listing 2.29 PropTypes with a stateless component

```
SomeStatelessComponent.propTypes = {
  firstName: React.PropTypes.string.isRequired,
  favoriteLanguages: React.PropTypes.array.isRequired,
  about: React.PropTypes.object.isRequired
}
```

If any of the props that are specified are not passed into the component, a warning will be shown in the JavaScript console.

Some props may need to be required for the component to function correctly. This can be solved with the `.isRequired` property. To require a prop, simply add `.isRequired` at the end of the `propType` (listing 2.30).

Listing 2.30 Required PropTypes with React.createClass

```
SomeComponent.propTypes = {
   firstName: React.PropTypes.string.isRequired,
   favoriteLanguages: React.PropTypes.array.isRequired,
   about: React.PropTypes.object.isRequired
 }
```

You can take arrays and objects one step further by specifying what type of values should be in the object or array(listing 2.31).

Listing 2.31 Specifying property values in PropTypes

```
SomeComponent.propTypes = {
   favoriteLanguages: React.PropTypes.arrayOf(React.PropTypes.object),
   about: React.PropTypes.object(React.PropTypes.string)
 }
```

See listing 2.32 for a list of the propTypes that can be used

Listing 2.32 List of available PropTypes

```
React.PropTypes.array,
React.PropTypes.bool,
React.PropTypes.func,
React.PropTypes.number,
React.PropTypes.object,
React.PropTypes.string,
React.PropTypes.node,
React.PropTypes.element,
React.PropTypes.instanceOf(Message),
React.PropTypes.oneOf(['News', 'Photos']),

 // An object that could be one of many types
 React.PropTypes.oneOfType([
   React.PropTypes.string,
   React.PropTypes.number,
   React.PropTypes.instanceOf(Message)
 ]),

React.PropTypes.arrayOf(React.PropTypes.number),
React.PropTypes.objectOf(React.PropTypes.number),

 // An object taking on a particular shape
 React.PropTypes.shape({
   color: React.PropTypes.string,
   fontSize: React.PropTypes.number
 })
```

2.3.5 statics

`statics` statics allow you to define static methods that can be called on the component class. Static methods can be run before component instances are created, but do not have access to the props or state of the component.

There are two ways to define static methods or properties on a component, depending on whether you are using `class` syntax or `React.createClass` syntax.

When using `React.createClass`, you declare a statics object and define the static methods in this object (listing 2.33).

Listing 2.33 statics with React.createClass

```
const MainComponent = React.createClass({

  statics: {
    sayHello() {
      console.log('Hello')
    }
  },

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

MainComponent.sayHello()
```

When using ES2015 classes, you declare statics as a static class method (listing 2.34).

Listing 2.34 statics with ES2015 classes

```
class MainComponent extends Component {
   static sayHello() {
     console.log('Hello!')
   }

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

MainComponent.sayHello()
```

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 these lifecycle methods work is important because they allow us to perform specific actions at different points in the creation and destruction of a component. Think for example if we wanted to make an API call that returned some data, we would probably want to make sure our component was ready to render this data, so we would make the API call once the component mounted in a method called `componentDidMount`. In this section, we will go over all of the lifecycle methods and explain how they work.

2.4.1 ComponentWillMount

`componentWillMount` (listing 2.35)*is invoked* only *once* and is done so *immediately before the initial rendering* of the component occurs. This happens before the render() method is called. At this point the component does not have any access to the UI, and you will also not have any access to child refs as they have not yet been created. *In listing 2.35*, we set the state with an initial value of 0, and we log out both the state and the refs in the render method. The first values logged out for tick is 1 and the refs as an empty object, showing us that the render method runs only after `componentWillMount`, but after `componentDidMount`.

Listing 2.35componentWillMount

```
class MainComponent extends Component {
  constructor() {
    super()
      this.state = { tick: 0 }
   }

   componentWillMount() {
    this.setState({
      tick: this.state.tick + 1
    })
   }

   componentDidMount () {
     this.setState({
      tick: this.state.tick + 1
    })
   }
   render() {
      console.log('state:', this.state)
      console.log('refs:', this.refs)
      debugger
      return <div />
   }
}
```

2.4.2 componentDidMount

`componentDidMount (listing 2.36)`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, or integrate with other JavaScript frameworks.

Listing 2.36 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 componentWillReceiveProps

`componentWillReceiveProps` (listing 2.37) is invoked when a component is receiving new props, and is not called for the initial render. This method enables you to update the state depending on the existing and upcoming props, without triggering another render. A use case for this could be checking a prop change, and setting state based on the value of the new prop vs the existing prop.

Listing 2.37 componentWillReceiveProps

```
class Child extends Component {
  constructor(props) {
    super(props)
    this.state = {
      nameChanged: false
    }
  }

  componentWillReceiveProps(nextProps) {
    if(nextProps.name != this.props.name) {
      this.setState({ nameChanged: true })
    }
  }

  render() {
    return (
      <View>
        <Text>
          { this.props.name }
        </Text>
        { this.state.nameChanged &&<Text>Name has changed</Text> }
```

```
      </View>
    )
  }
}
```

2.4.4 shouldComponentUpdate

`shouldComponentUpdate` (listing 2.38) returns a Boolean, and allows you to decide exactly when a component renders. If you know that the new state or props will not require the component or any of its children to render, you can return false. If you want the component to rerender, return true.

Listing 2.38 shouldComponentUpdate

```
class MainComponent extends Component {

   shouldComponentUpdate(nextProps, nextState) {
     if(nextProps.name !== this.props.name) {
       return true
     }
     return false
   }

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

2.4.5 componentWillUpdate

`componentWillUpdate`(listing 2.39) is invoked immediately before rendering when new props or state are being received. This method is not called for the initial render, and is an opportunity to perform preparation before a render occurs. Here, you can directly manipulate the state of the component without calling `this.setState`. Also note that you cannot call `setState` in this method.

Listing 2.39 componentWillUpdate

```
class MainComponent extends Component {
   componentWillUpdate(nextProps, nextState) {
     if(!nextState.nameChanged) {
        nextState.nameChanged = !nextState.nameChanged
     }
   }
   render() {
     return <SomeComponent />
   }
 }
```

2.4.6 componentDidUpdate

`componentDidUpdate`(listing 2.40) is invoked immediately after the component has been updated and rerendered. You get the previous state and previous props as arguments.

Listing 2.40 componentDidUpdate

```
class MainComponent extends Component {
   componentDidUpdate(nextProps, nextState) {
     if(nextState.showToggled === this.state.showToggled) {
       this.setState({
         showToggled: !showToggled
       })
     }
   }
   render() {
     return <SomeComponent />
   }
 }
```

2.4.7 componentWillUnmount

`componentWillUnmount` (listing 2.41) is called before the component is removed from the application. Here, you can perform any necessary cleanup, remove listeners, or remove timers that were set up in componentDidMount.

Listing 2.41 componentWillUnmount

```
class MainComponent extends Component {

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

2.5 Summary

- State is a way to handle data in React components and updating state rerenders the UI of the component and any child component relying on this data as props.
- Props are how data is passed down through a React Native application to child components and that updating props automatically updates any components receiving the same props.
- The 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, with all other methods and properties being optional.

- React Lifecycle methods are a group of methods available in a React component that are executed at specific points in a component's lifecycle and control how the component functions and updates.

3

Building Your First React Native App

This chapter covers

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

Now that we understand the basics of how React and React Native work, let's put these pieces together to make our first app, a todo app. Going through the process of building a small app and putting to use the knowledge we have gone over so far will be a good way to reinforce your understanding of how to use React Native.

3.1 Building a Todo app

When learning a new framework, technology, language or concept, diving directly into the process by building a real app is a really great way to jump start the learning process. For us, this will mean building a working Todo application.

There will definitely be some concepts being used in our app that we have not gone over yet in depth, and there will be some styling nuances and details we have yet to cover, but do not worry, we will get to all of these details later.

Instead of going over these new ideas one by one now, we will build our basic app and then go over them in detail in later chapters. Take this opportunity to play around with the app as we build it to learn as much as possible in the process, break and fix styles and components to see what happens in the process.

3.1.1 Layout out the Todo app

Let's get started building our todo app. We will be building a todo app similar in style and functionality to the apps on the TodoMVC site. Before we get started, let's take a glance at how our app will look when we are done so that we can conceptualize what components we need and how we may structure these components.

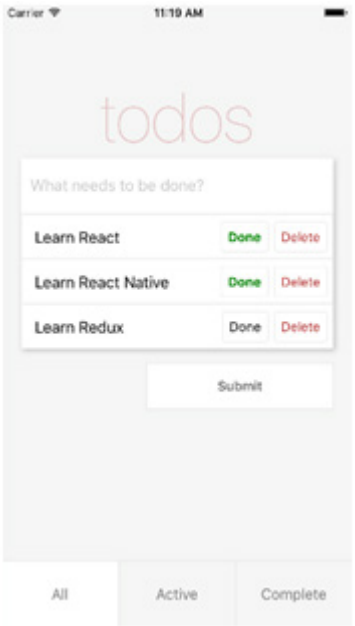


Figure 3.1 Todo app design

As we did in chapter one, let's now visually break this up into components and container components.

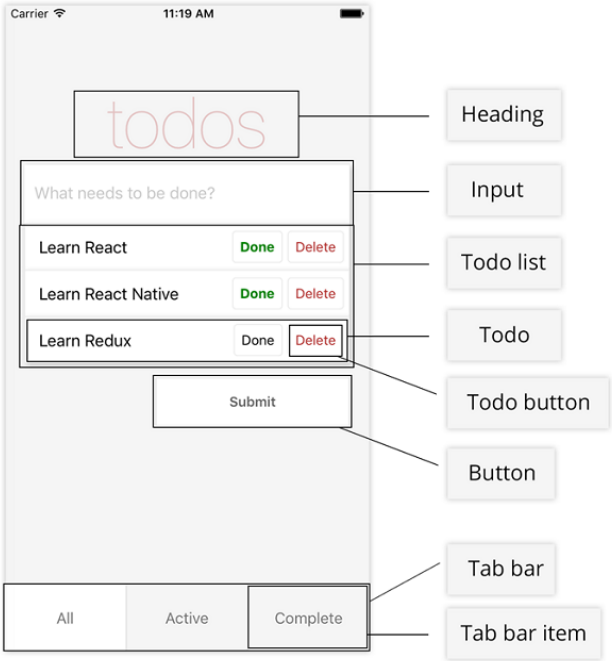


Figure 3.2 Todo app with descriptions

Now let's take a look at how this would look in our app, using React Native components by laying out a basic implementation of these components as shown in listing 3.1.

Listing 3.1 Basic Todo app implementation

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

Our app will start off displaying a heading, tab bar, text input and a button. When we add a Todo to our app, it will add the Todo to our array of Todos and display the new Todo beneath the input. Each Todo will have two buttons: Done and Delete. Our 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 TabBar that will filter the Todos based on whether they are complete or still active.

Let's get started coding the app. To begin, create a new React Native project by typing `react-native init TodoApp` (or, instead of TodoApp whatever you would like your app to be named)in our terminal (figure 3.3).

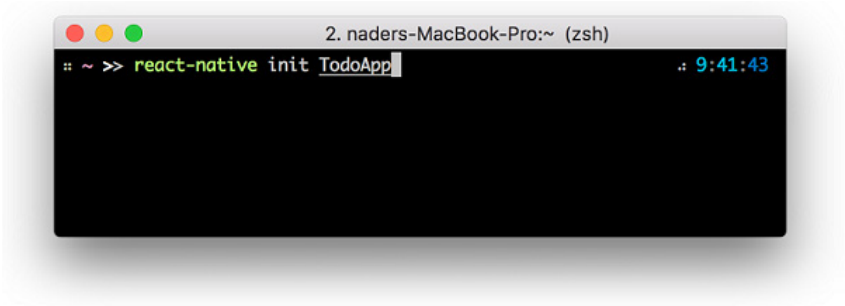


Figure 3.3 Initializing new React Native app

REACT NATIVE VERSION I am using React Native version 0.33 for this example. Newer versions may have api changes, but nothing should be breaking for building our Todo app. You are welcome to use either the most recent version of React Native, or use the specific version I am using here.

Now, go into your index file. If you are developing for iOS, open index.iOS.js, and if developing for Android open index.Android.js. The code for both platforms will be exactly the same.

3.1.2 Coding the Todo app

In the index file, let's import an `App` component that we will create soon, and delete the styling along with any extra components we are no longer using (listing3.2).

Listing 3.2 index.iOS.js / index.Android.js

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

const TodoApp = () =><App />

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

Here, we're only bringing in `AppRegistry` from `react-native`. We're also bringing in our main `App` component, which we will create next.

In the `AppRegistry` method, we're initiating our application. `AppRegistry` is the JS entry point to running all React Native apps. `Appregistry` takes two arguments. The first argument is the `appKey`, or the name of the actual application which we defined when we initialized the app. The second argument is a function that returns the React Native component we would

like to use as the entry point of our app. In this case, we are returning the `TodoApp` component we declared in Listing 3.2

Now, create a folder in the root of the application called app. In the app folder, create a file called App.js. In it, let's add some basic code to get us started (listing 3.3).

Listing 3.3Creating the App component: app/App.js

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

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

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

export default App
```

We've pulled in a new component called `ScrollView`, which wraps the platform `ScrollView` and is basically a scrollable `View` component. We make sure that both the `ScrollView` and the parent `View` of the `ScrollView` both have a `flex:1` value. `flex:1`which will make the component fill the entire space of it's parent container.

Now, let's go ahead and set up an initial state for some of the values we will be needing later on. We will be needing an array to keep our todos that we will name `todos`, a value to hold the current state of the `TextInput` that will be adding the todos which we will name `inputValue`, and a value to store the type of todo that we are currently viewing (All, Current, or Active) which we will name `type`.

In App, before our render function, let's add a constructor and an initial state to the class and initialize these values in our state (listing 3.4).

Listing 3.4Setting the initial state: app/App.js

```
...

class App extends Component {
  constructor() {
    super()
this.state = {
      inputValue: '',
      todos: [],
      type: 'All'
    }
  }
  render() {
...
  }
}

...
```

Next, let's go ahead and create our `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).

Listing 3.5Creating the Heading component app/Heading.js

```
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`, we are passing an `rgba` value to color. If you are not familiar with rgba, the first three values make up the rgb color values, and the last value represents the alpha or the opacity (red, blue, green, alpha). We are passing in an alpha value

of 0.25, or 25%. We are also setting the font weight to be '100', which will give our text a thinner weight and look.

Go back into App.js and bring in the `Header` component and place it in the `ScrollView`, replacing the empty `View` we originally placed there (listing 3.6).

Listing 3.6 Importing and using the Heading component: app/App.js

```
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 style={styles.content}>
<Heading />
</ScrollView>
</View>
    )
  }
}

...
```

Let's go ahead and run our app to see our new Heading and App Layout. This is how our application should now look (figure 3.4).

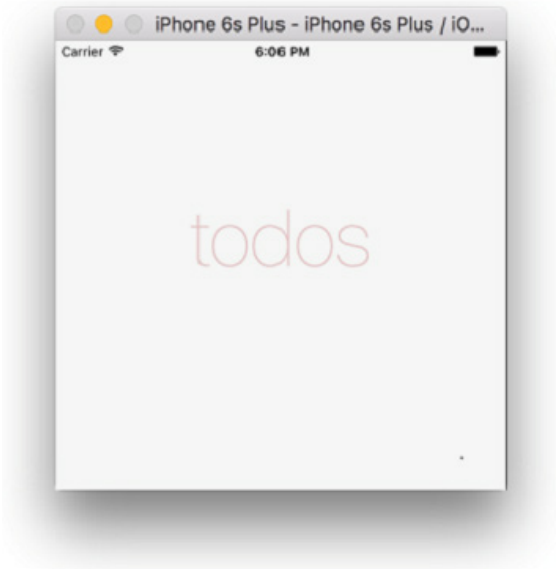


Figure 3.23 Running the app

Next, let's go ahead and create our `TextInput` component and give it some styling. In the app folder, create a file called Input.js (listing 3.7).

Listing 3.7 Creating the Input component: app/Input.js

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

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

conststyles = 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
```

We are using a new React Native component called `TextInput` here. If you are familiar with web development, this is very similar to an html `input`. We are also giving both the `TextInput` and the outer `View` their own styling.

TextInput takes a few other props. Here, we are specifying a `placeholder` to show text before the user starts to type, a `placeholderTextColor` which will style the placeholder text, and a `selectionColor` which styles the cursor for the TextInput.

Next, let's wire up a function that will let us get the value of the `TextInput` and save it to the state of our `App` component. To do this, first 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 us to make sure the function is working.

To view `console.log()` statements in React Native, we first need to open the developer menu. Let's go ahead and take a look at the developer menu and see how it works.

If you are not interested in the Developer Menu or want to skip this section for now, go to section 2.1.6 to continue building out the Todo app.

3.1.3 Opening Developer Menu in iOS Simulator

The developer menu is a built in menu that is available to us as a part of React Native and will give us access to the main debugging tools that we will be using.

While the project is running in the simulator, the developer menu can be opened in one of three ways:

1. Pressing `cmd + D`on the keyboard
2. Pressing `cmd + ctrl + z` on the keyboard
3. Opening the Hardware > Shake Gesture menu in the simulator options (figure 3.5).

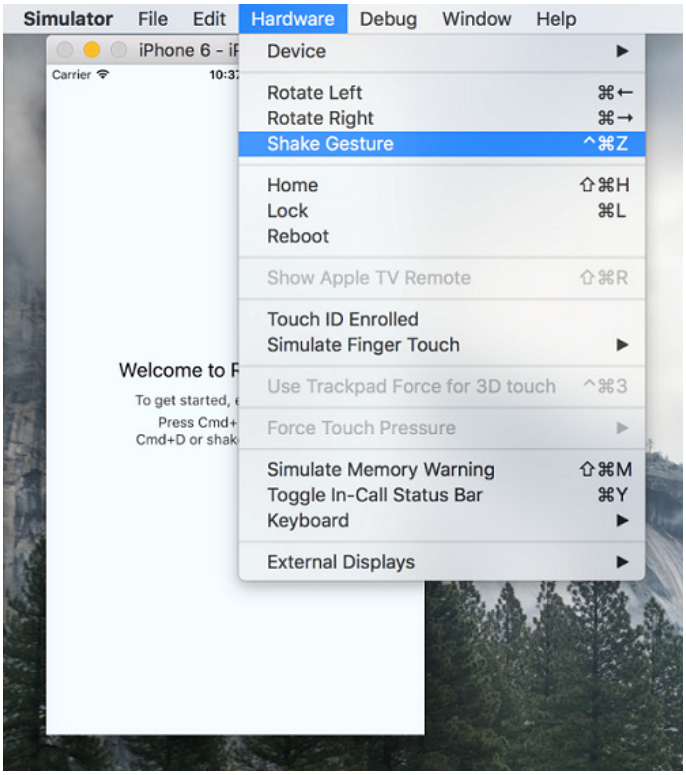


Figure 3.5 Manually opening the developer menu (iOS Simulator)

Now we should see the developer menu shown in figure 3.6.

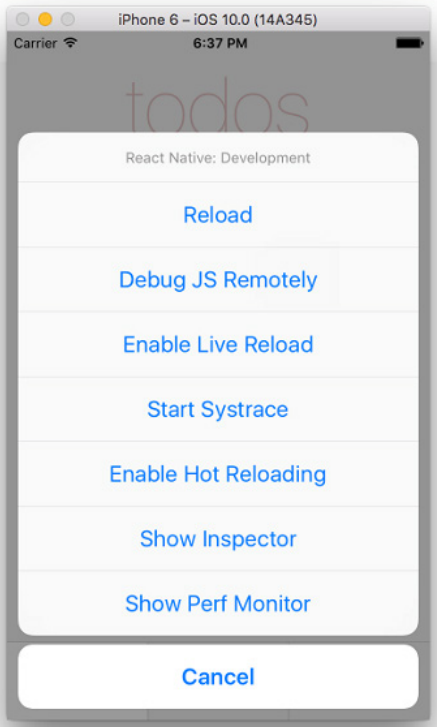


Figure 3.6 React Native Developer Menu (iOS Simulator)

If `cmd + d` or `cmd + ctrl + z`do not open the menu for you, you may need to connect your hardware to the keyboard. To do this, go to Hardware -> Keyboard -> Connect Hardware Keyboard in your simulator menu, as shown in figure 3.7.

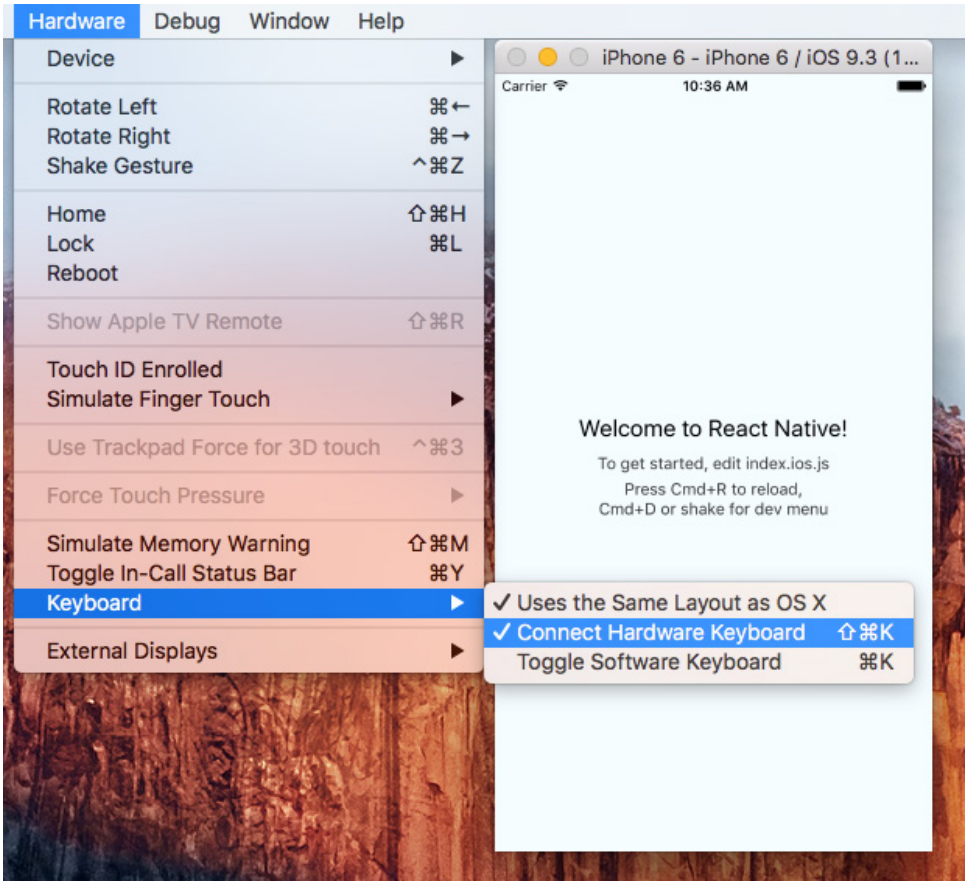


Figure 3.7 Connecting the Hardware Keyboard to iOS Simulator

3.1.4 Opening the developer menu in Android Emulator

To begin debugging for Android, we first need to bring up the developer menu. The developer menu will give us access to the main debugging tools that we will be using. To open the menu, run the project in your simulator. Once the simulator is open and running, the developer menu can be opened in one of two ways:

1. Pressing F2 on the keyboard
2. Press CMD + m on the keyboard
3. Press the hardware button (see figure 3.8).

Figure 3.8 Manually opening hardware menu (Android Emulator)

Now we should see the developer menu shown in figure 3.9.

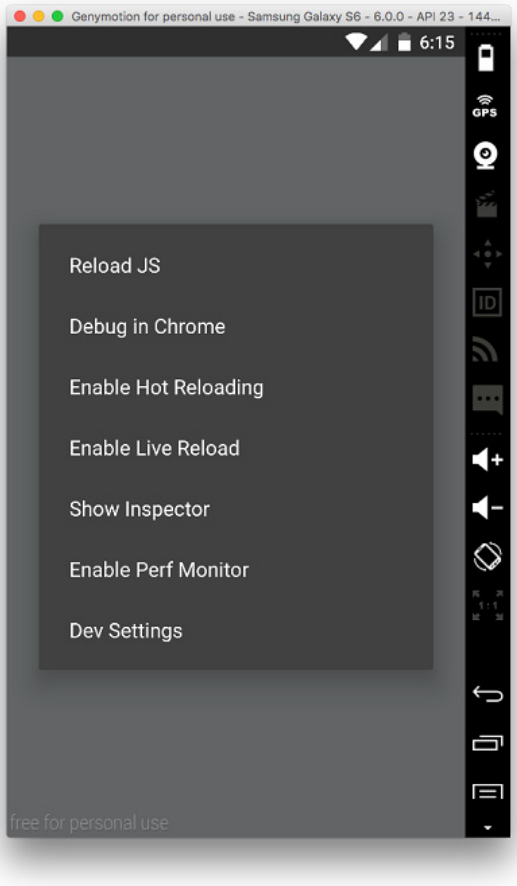


Figure 3.9 React Native developer menu (Android Emulator)

3.1.5 Using the Developer Menu

Once the developer menu is open, you should see the following options:

1. **Reload (iOS) / Reload JS (Android) -** Simply reloads the app. This can also be done by pressing `cmd + r on the keyboard (iOS) or pressing r twice (Android)`
2. **Debug JS Remotely (iOS and Android) -** This opens the chrome dev tools and allows you to have full debugging support through the browser. Here, you have access not only to logging statements within your code, but also to break points and whatever you are used to while debugging web apps with the exception of the DOM (figure 3.6). If you need to log out any information or data in your app, this is usually the place to do so.

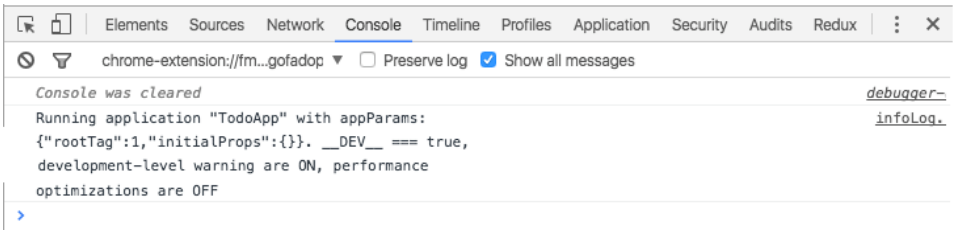


Figure 3.10 Debugging in Chrome

3. **Enable Live Reload (iOS and Android) -** This will enable live reload. When you make changes in your code, the entire app will reload and refresh in the simulator
4. **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 16ms frame while your app is running. Profiled code blocks are surrounded by markers start/end markers which are then visualized in a colorful chart format.To use this, you need to install trace2html by typing `brew install trace2html` into your terminal window.

   Systrace can also be enabled manually from the command line in Android. If you would like to learn more about this, check out the docs for a very comprehensive overview.

   If your trace .html file isn't opening correctly, check your browser console for the following:

   ```
   Uncaught TypeError: Object.observe is not a function
   ```

   Because Object.observe was deprecated in recent browsers, you may have to open the file from the Google Chrome Tracing tool. You can do so by:
   1. Opening tab in chrome chrome://tracing
   2. Selecting load
   3. Selecting the html file generated from the previous command.
5. **Enable hot reloading (iOS and Android) -** This is a really great feature that was added in version .22 of React Native. It offers an amazing developer experience, giving you the ability to see your changes immediately as your files are changed without losing the current state of the app. This is expecially useful when making ui changes deep within your app without losing state. This is different than live reloading as it actually retains the current state of your app, only updating the components / state that has been changed, while the live reloading will reload the entire app therefore losing the current state.
6. **Show inspector (iOS and Android) -** This will bring up a property inspector similar to what you see in the chrome dev tools. You can click on an element and see where it is in the heirarchy of components, as well as any styling applied to the element (figure 3.11).

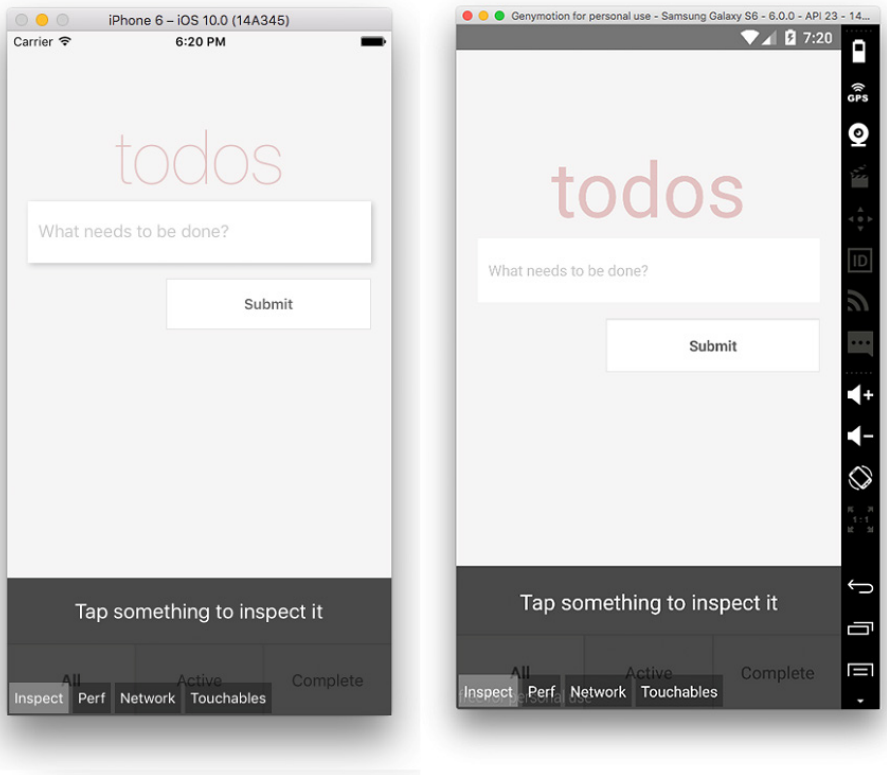


Figure 3.11 Using the inspector (iOS left, Android Right)

7. **Show Perf Monitor (iOS and Android) –** This brings up a small box in the top left corner of your app giving you some information about the performance of your app. Here you will see the amount of RAM being used, and the number of frames per second that the app is currently running at. If you click the box, it will expand to show you even more information about the app (figure 3.12).
8. **Dev Settings (Android Emulator only) –** Brings up additional debugging options, includeing an easy way to toggle between __DEV__ environment variable being true / false.

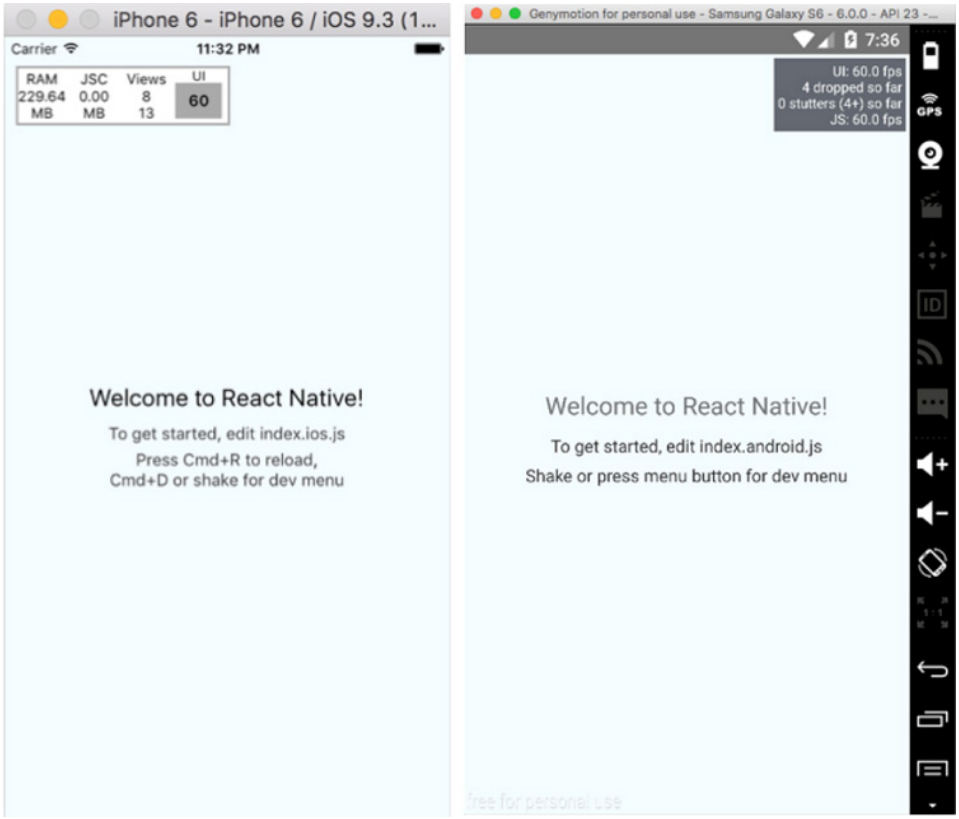


Figure 3.12 Perf Monitor (iOS left, Android right)

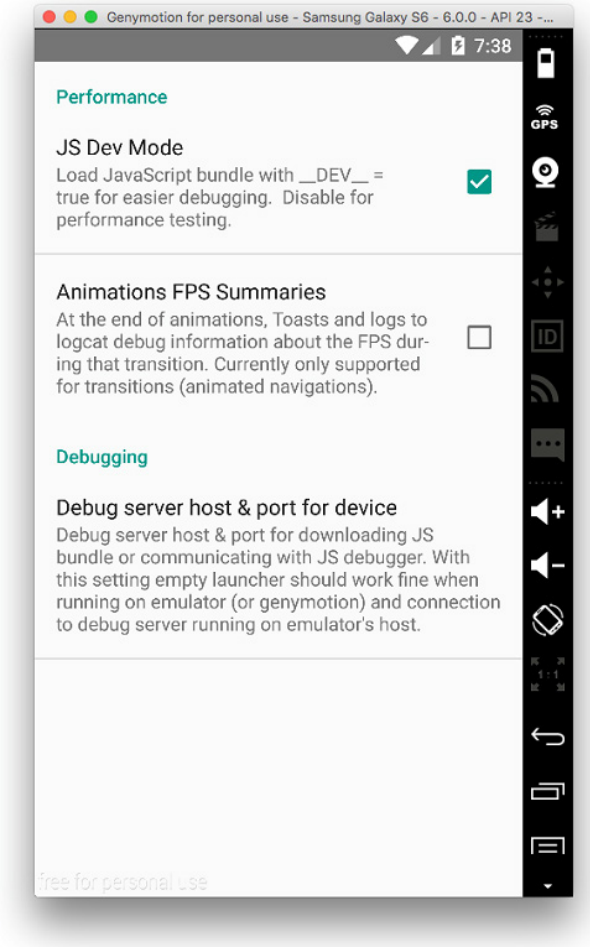


Figure 3.13 Dev Settings (Android Emulator)

3.1.6 Continuing building the Todo app

Now that we know how the developer menu works, let's go ahead and open the developer menu and press **Debug JS Remotely**, which should open the Chrome dev tools and we should be ready to start logging out information to the console.

The next thing we will be doing is attaching a method to `TextInput`which wewill give as a property to the `Input` component. We will also pass the `inputValue` that is stored on the state to `Input`as a property (listing 3.8).

We also make sure to import the `Input` component into app/App.js.

Listing 3.8 Creating the inputChange function: app/App.js

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

class App extends Component {
constructor() {
    …
  }
  inputChange(inputValue) {                    (1)
    console.log('Input Value: ', inputValue)   (2)
    this.setState({ inputValue })              (3)
  }
  render() {
    const{ inputValue } = this.state
    return (
      <View style={styles.container}>
        <ScrollView style={styles.content}>
          <Heading />
          <Input
            inputValue={inputValue}                              (4)
            inputChange={(text) => this.inputChange(text)} />    (5)
        </ScrollView>
      </View>
    )
  }}
```

(1) Here the inputChange method is created, which takes inputValue as an argument.
(2) Log out the inputValuevalue to make sure the method is working
(3) Set the state with the new value (same as this.setState({inputValue: inputValue}))
(4) Pass inputValue down as a property to the Input component
(5) Pass inputChange method down as a property to the Input component

`inputChange` will take one argument, the value of the `TextInput`, and the `inputChange` value in the state to the value passed into the function.

Now, we need to wire the function up with our `TextInput`in the `Input` component. Open Input.js, and update the `TextInpt` component with the new`inputChange` function and the `inputValue` property (listing 3.9).

Listing 3.9Adding inputChange and inputValue to the TextInput: app/Input.js

```
...
constInput = ({ inputValue, inputChange }) => ( (1)
<View style={styles.inputContainer}>
<TextInput
value={inputValue}
      style={styles.input}
      placeholder='What needs to be done?'
      placeholderTextColor='#CACACA'
      selectionColor='#666666'
      onChangeText={inputChange} /> (2)
</View>
```

```
)
...
```

1. **Destructure** `inputValue` **and** `inputChange` **props**
2. **Set** `onChangeText` **method to** `inputChange`

We've destructured the propsinputValue 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`. We've also set the value of the `textInput` to be `inputValue`, so we 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 get passed the value of the `TextInput`.

Now, let's run the project again and see how it looks (figure 3.14).

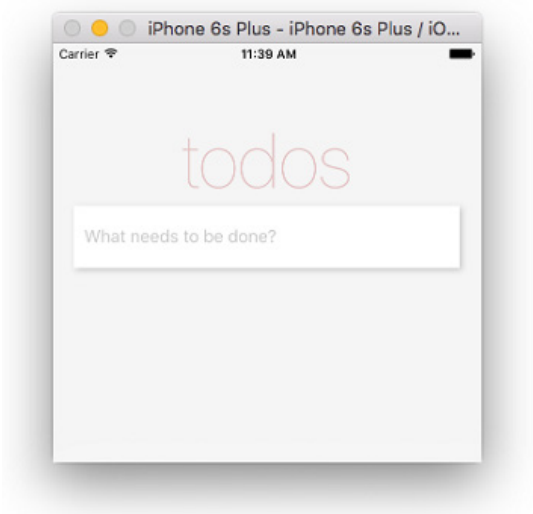


Figure 3.14 Updated view after adding TextInput

We're logging the value of the input, so as you type you should see the value being logged out to the console.

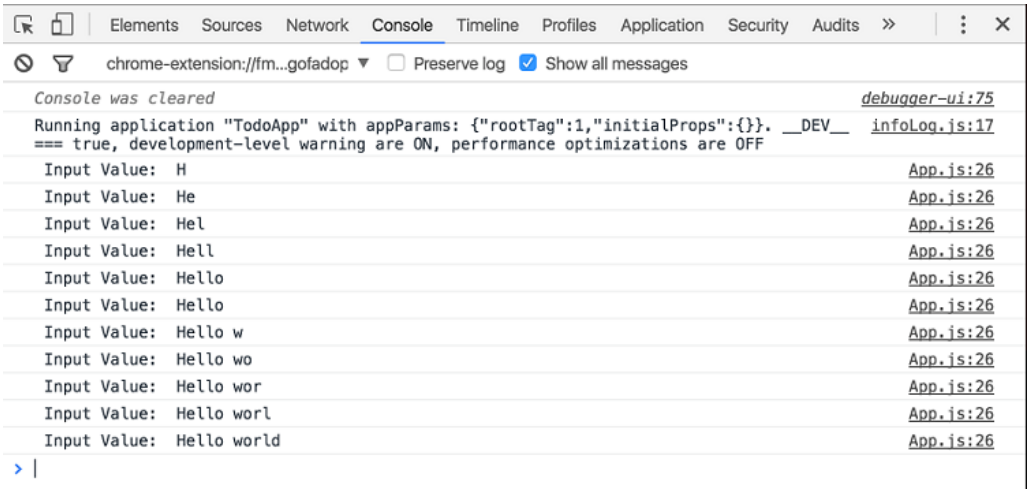


Figure 3.15 Logging out TextInput value with inputChange method

Now that we have our `inputValue`value being stored in the state, let's create a button to add the items to a list of todos.

Before we create the button, let's create a function that we will bind to our button that will add the new todo to our array of todos that we have defined in our constructor.

We will call this function `submitTodo`. Let's place it after our `inputChange` function and before our `render` function (listing 3.10).

Listing 3.10 Adding submitTodo function app/App.js

```
...
submitTodo () {
    if (this.state.inputValue.match(/^\s*$/)) {
    return
  }  ❶
    let todo = {
    title: this.state.inputValue,
    todoIndex: todoIndex,
    complete: false
  }  ❷
    todoIndex++                          ❸
    this.state.todos.push(todo)          ❹
    this.setState({ todos: this.state.todos, inputValue: '' }, () => {
    console.log('State: ', this.state) ❻
  }) ❺
}
...
```

Let's take a look at this function.

❶ **First, we check to see if the `inputValue` is empty, if it is empty we return without doing anything else.**

2. If `inputValue` is not empty, we then create and then assign a `todo` variable an object, we give this object a `title`, a `todoIndex`, and a `complete`boolean (the `todoIndex` has not yet been created, we will do so shortly).
3. After we create our `todo` variable, we increment the `todoIndex`.
4. We then push the new `todo` to the existing array of `todos`.
5. Finally, we set the state of our `todos` to match the updated array of `this.state.todos`, and reset the `inputValue` to an empty string.
6. We also log out the state to make sure that everything is working. This is done in a callback function from `setState`. Once the state is set, you have the option to pass a callback function and that is what we are doing here.

Now, let's create the `todoIndex` at the top of our App.js file, below our last import statement (listing 3.11).

Listing 3.11Creating todoIndex variable app/App.js

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

let todoIndex = 0

class App extends Component {
...
```

Now that our `submitTodo` function has been created, let's create a file called Button.js and wire up this function to work with the button (listing 3.12).

Listing 3.12Creating the Button component app/Button.js

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

constButton = ({ submitTodo }) => ( 1
  <View style={styles.buttonContainer}>
    <TouchableHighlight
      underlayColor='#efefef'
      style={styles.button}
      onPress={submitTodo}>          2
      <Text style={styles.submit}>
        Submit
      </Text>
    </TouchableHighlight>
  </View>
)

conststyles = 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
```

1. **Destructure the submitTodo function**
2. **Attach `submitTodo` to the `onPress` function available to the TouchableHighlight component. This function will get called when the TouchableHighlight is touched or pressed.**

In the above component, we are using `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`, we 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 we will provide as a prop. As you can see, we have specified an `underlayColor` of `'#efefef'` which is a light gray, while the background color is white. This will give the user a good sense of whether or not the touch event has registered. If no underlayColor is defined, it defaults to black.

`TouchableHighlight` only supports one main child component. As you can see, we are passing in a `Text` component. If you would like multiple components to be within a `TouchableHighlight`, simply wrap them in a single `View` and pass this `View` as the child of the `TouchableHighlight`.

We also have quite a bit of styling going on.

> **Do not worry about styling specifics in this chapter as we will cover them in depth in the coming chapters, but do take a look at them and see what we are doing to get an idea of how styling works in each component. This will help out a lot once we start going in depth in the future as you will already be exposed to some styling properties and how they work.**

Now that the `Button` component is created and wired up with the function we defined in App.js, let's bring this component into our app and see if it works! Open app/App.js (listing 3.13).

Listing 3.13 Importing the Button component app/App.js

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

consttodoIndex = 0

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

❶ We import the new Button component
❷ We place the Button below the Input component and give it the
❸ We bind the method to our class in the constructor. Because we are using classes, functions will not be autobound to the class.

We've imported the `Button` component, and then placed it under the `Input` component within our `render` function. `submitTodo` is passed into the `Button` as a property, which will call`this.submitTodo`.

Now, refresh the app. It should look like figure 3.16.

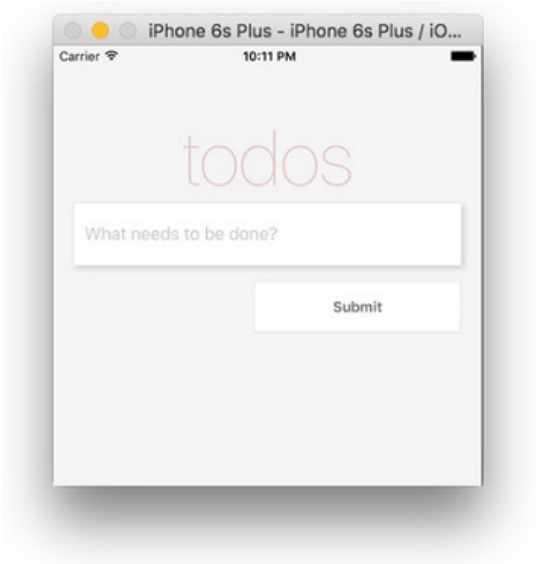


Figure 3.16 Updated app with the Button component

When we add a todo, the `TextInput` should clear and the app state should log out to the console, showing an array of todos with the new todo in the array.

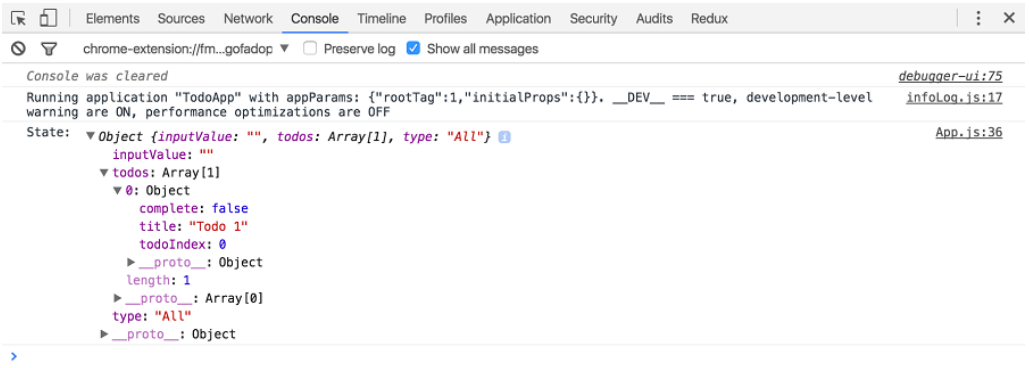


Figure 3.17 Logging out the state

Now that we are adding todos to our array of todos, we need to render these todos to the screen. To get started with this, we need to create 2 new components: `TodoList` and `Todo`. `TodoList` will render our list of `Todos`, and will use the`Todo` component for each individual todo.

We will start by creating a file named Todo.js (listing 3.14).

Listing 3.14 Creating Todo component app/Todo.js

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

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

conststyles = 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 within a Text component. We have also added styling to the View and Text component we are using here.

Now that we have created our `Todo` component, let's create our `TodoList` component (listing 3.15).

Listing 3.15Creating the TodoList component app/TodoList.js

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

constTodoList = ({todos }) => {
  todos = todos.map((todo, i) => {
    return (
```

```
      <Todo
        key={i}
        todo={todo} />
    )
  })
  return (
    <View>
      {todos}
    </View>
  )
}

export default TodoList
```

Our `TodoList` component will take one property for now, an array of todos. We then map over these todos and create a new `Todo`component (which we imported at the top of the file) for each todo, passing in the todo as a property to the `Todo` component. We have also specified a key, and passed in the index (i) as a key to each component.

Now that this is set up, the last thing we need to do is import the `TodoList` component into our App.js file, and pass in the todos as a property (listing 3.16).

Listing 3.16Importing the TodoList component app/App.js

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

Let's run the app. When we add a todo, we should see it pop up in our list of todos (figure 3.18).

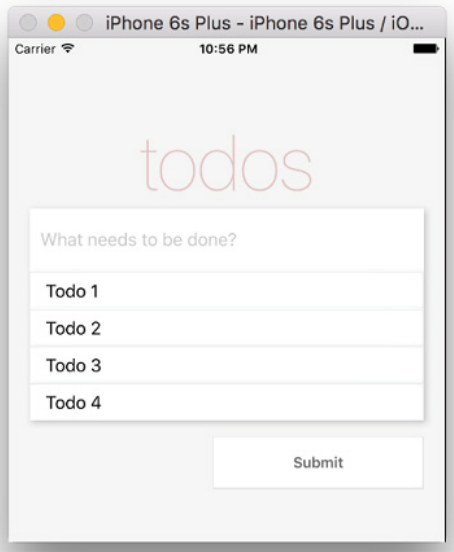


Figure 3.18 Updated app with the TodoList component

Ok, now that we have this working, the next step will be for us to mark a todo as complete, and to delete a todo. Let's open App.js and create a `toggleComplete` and `deleteTodo` function below our `submitTodo` function (listing 3.17).

`toggleComplete` will toggle whether or not the todo is complete, and `deleteTodo` will delete the todo.

Listing 3.17Adding toggleComplete and deleteTodo functions app/App.js

```
constructor () {
  ...
this.toggleComplete = this.toggleComplete.bind(this) (1)
  this.deleteTodo = this.deleteTodo.bind(this)       (2)
}
...
deleteTodo (todoIndex) {     (3)
  let { todos } = this.state
    todos = this.state.todos.filter((todo) => {     return todo.todoIndex !==
      todoIndex
    })
  this.setState({ todos })
}

toggleComplete (todoIndex) { (4)
```

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

1. We bind the toggleComplete method to our class in the constructor
2. We bind the deleteTodo method to our class in the constructor
3. `deleteTodo` takes the `todoIndex` as an argument and then filters our todos to return all but the todo with the index that was passed in. We then reset the state to the remaining todos.
4. `toggleComplete` also takes the `todoIndex` as an argument, and loops through the todos until it finds the todo with the given index. It then changes the complete boolean to be the opposite of what it currently is set to be. After that, it resets the state of the todos.

To hook these in, we need to create a button component we will pass in to the todo. Let's go into our app folder and create a new file called TodoButton.js (listing 3.18).

Listing 3.18Creating TodoButton.js app/TodoButton.js

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

constTodoButttton = ({ onPress, complete, name }) => (    1
  <TouchableHighlight
    onPress={onPress}
    underlayColor='#efefef'
    style={styles.button}>
    <Text style={[
      styles.text,
      complete ? styles.complete : null,                  2
      name === 'Delete' ? styles.deleteButton : null ]} 3
    >
      {name}
    </Text>
  </TouchableHighlight>
)

conststyles = 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 TodoButttton
```

1. This component takes in `onPress`,`complete`, and `name` as props.
2. Here we are checking to see if complete is true and applying a style
3. We are also checking to see if the `name` property equals 'Delete' and also applying a style if it is the case.

Now, let's pass our new functions down as props to the `TodoList` component (listing 3.19).

Listing 3.19 Passing toggleComplete and deleteTodo functions as props to TodoListapp/App.js

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

And then we pass `toggleComplete` and `deleteTodo` as props to the `Todo` component (listing 3.20).

Listing 3.20 Passing toggleComplete and deleteTodo functions as props to ToDo app/TodoList.js

```
...
let 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).

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

```
import TodoButton from './TodoButton'
...
constTodo = ({ 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>
)

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

We've added two `TodoButtons`, one with the name Done and one with the name Delete. We have also passed down `toggleComplete` and `deleteTodo` as functions to be called as the `onPress` we defined in `TodoButton.js`. If we refresh our app and add a todo, we should now see our new buttons.

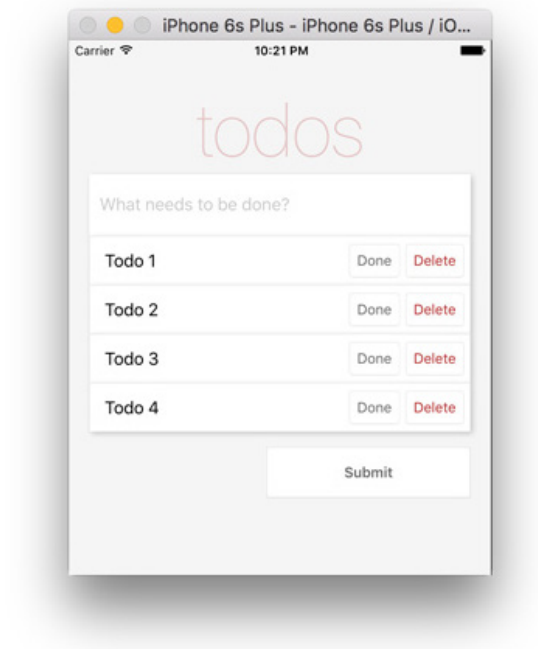


Figure 3.19 App with TodoButton displayed

If we click done, the button text should now be bold and green. If we click delete, the todo should disappear from our list of todos.

We are now almost done with the app. The final step is to build a tab bar filter that will show us either all of our todos, only our complete todos, or only our incomplete todos. To get this started, let's create a new function that will set the type of todos that we will show.

In our constructor, we set a type variable to 'All' when we first created the app. We will now create a function named `setType` that will take in a type as an argument and update the type in our state. Place this function below the `toggleComplete` function (listing 3.22).

Listing 3.22 Adding setType function app/App.js

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

Now, we need to create the `TabBar` and `TabBarItem` components. First, let's go ahead and create the `TabBar` component. Create a file in the app folder named TabBar.js (listing 3.23).

Listing 3.23 Creating TabBar component app/TabBar.js

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

constTabBar = ({ setType, type }) => (
  <View style={styles.container}>
    <TabBarItem  type={type} title='All'
      setType={() => setType('All')} />
    <TabBarItemtype={type}bordertitle='Active'
      setType={() => setType('Active')} />
    <TabBarItemtype={type}bordertitle='Complete'
      setType={() => setType('Complete')} />
  </View>
)

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

export default TabBar
```

This component will take two props: `setType` and `type` which will both be passed down from our main `App` component.

We are importing our yet to be defined `TabBarItem` component. Each `TabBarItem` component takes three props: `title`, `type`, and `setType`. Two of the components also are taking 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).

Listing 3.24 Creating TabBarItem component app/TabBarItem.js

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

constTabBarItem = ({ 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>
)

conststyles = 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, we are checking a few props and setting styles based on the prop. If `selected` is `true`, we give it the style `styles.selected`. If `border` is `true`, we give it the `style styles.border`. If `type` is equal to the `title`, we give it `styles.selected`.

In the `Text` component, we are also checking to see if `type` is equal to `title`, and if so we add a bold style to it.

To implement the `TabBar`, let's open app/App.js and bring in the `TabBar` component and set it up. We will also bring in `type` to our render function as part of our destructuring of `this.state` (listing 3.25).

Listing 3.25 Implementing TabBar component app/TabBar.js

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

```
            todos={todos} />
          <Button submitTodo={() => this.submitTodo()} />
        </ScrollView>
      <TabBar type={type} setType={this.setType.bind(this)} />
    </View>
  )
}
...
```

Here, we bring in the `TabBar` component. We then destructure 'type' from our state, and *pass it not only to our new* `TabBar` *component, but also to our* `TodoList` *component*. We will use this 'type' variable in just a second when filtering our todos based on this type.

We also pass the `setType` function as a prop to the `Tabbar` component.

The last thing we need to do is open our `TodoList` component and add a filter to return only the todos of the type we are currently wanting back based on the tab that is selected. Open TodoList.js and destructure the type out of the props and add the following `getVisibleTodos` function before the return statement (listing 3.26).

Listing 3.26Updating TodoList component app/TodoList.js

```
...
constTodoList = ({ 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) => {
...
```

We are using a switch statement to check and see which type is currently set. If 'All' is set, we return the entire list of todos. If 'Complete' is set, we filter the todos and only return the complete todos. If 'Active' is set, we filter the todos and only return the incomplete todos.

We then set the `todos` variable as the returned value of `getVisibleTodos`.

Now we should be able to run the app and see our new `TabBar`. The `TabBar` should filter based on which type is seleted:

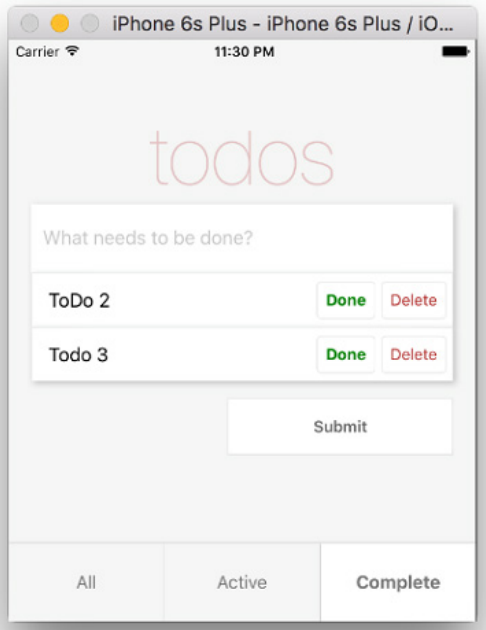


Figure 3.20 Final Todo app

3.2 Summary

In this chapter we learned:

- Using the JavaScript console is a good way to debug your app and log out useful information.
- How to build a complete functioning app by building a todo app.

4

Introduction to styling

This chapter covers

- Styling overview
- Applying styles to View Components
- Applying styles to Text Components

Views are the main building block of a React Native UI, so it is important to go over and explain all of the style properties that a View component can implement.

Next we will walk through styling Text components.

4.1 Styling Overview

Every element has its own set of styles which can be applied to it, but may or may not be applicable to other types of elements. For example, Text elements and View elements both have their own different set of styles that can be applied. `Text` elements can receive a `fontWeight` (`fontWeight` refers to the thickness of the font) property while `View` elements cannot, and `View` elements can receive a `flex`(`flex` refers to the layout of a `View`, something we will cover in depth in the next chapter) properties while `Text` elements cannot, and so on and so forth.

In this chapter, we will go over most of the component's individual style properties and see how they all work.

4.1.1 Applying styles to elements

Applying styles to elements in React Native can be done in a number of ways. We have already gone over inline styling (listing 4.1) and styling using a StyleSheet (listing 4.2) in chapters 1 and 3.

Listing 4.1 Styling inline

```
<View style={{marginLeft: 20}}>
  <Text style={{fontSize: 18}}>Some Text</Text>
</View>
```

Listing 4.2 Styling with a StyleSheet

```
<View style={style.container}>
    <Text>Some Text</Text>
  </View>

const style = StyleSheet.create({
  container: {
    padding: 10
  }
})
```

4.1.2 Creating a StyleSheet

A StyleSheet is an abstraction similar to CSS StylesSheets. StyleSheet allows us to create a style object and refer to each style individually, away from the render method, making the code easier to understand. StyleSheet also allows us to create, reuse and reference repeated styles across an application. This way we can create external StyleSheets, and import these StyleSheets into components whenever we need to use them.

A StyleSheet is created by importing StyleSheet from React Native, and then calling StyleSheet.create, passing in an object with the style properties we would like to define (listing 4.3).

Listing 4.3 Importing StyleSheet and creating a StyleSheet

```
import { StyleSheet } from 'react-native'

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'red'
  }
})
```

The created styles are then available on the styles object (listing 4.4).

Listing 4.4 Referencing and using styles within a StyleSheet

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

class App extends Component {
  render () {
    return (
      <View style={styles.container} />
    )
  }
```

```
}

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'red'
  }
})
```

In addition to applying predefined styles one at a time, you can also pass multiple styles in an array (listing 4.5).

Listing 4.5 Passing an array of style properties

```
<View style={[style.container, style.leftContainer]}>
    <Text style{[style.text, style.bold]}>Some Text</Text>
  </View>

  const style = StyleSheet.create({
  // style definitions below
})
```

Remember when doing this, that the last style passed in will override the previous style if there is a duplicate property. For example, in the above component, if we had the following StyleSheet defined, the `Text` component would render red text even though `Text` was originally set to a color of black (listing 4.6).

Listing 4.6 Passing an array of style properties

```
<View style={[style.container, style.leftContainer]}>
    <Text style{[style.text, style.bold]}>Some Text</Text>
  </View>

  const style = StyleSheet.create({
    container: {
      width: 100
    },
    leftContainer: {
      paddingRight: 10
    },
    text: {
      color: 'black'
    },
    bold: {
      color: 'red',
      fontWeight: 'bold
    }
  })
```

You can also pass in both inline styles and StyleSheet styles into an array. To do this, you need to make sure that the inline styles are defined in a separate object (listing 4.7).

Listing 4.7 Passing an array of style properties and inline styles

```
<View style={[style.container, {marginLeft: 20, marginTop: 20}]}>
  <Text style{[style.text, style.bold]}>Some Text</Text>
</View>
```

4.1.3 Styling View Components

Now that we have gone over how to apply styles to a component, let's go through all of the ways you can style a View element.

PIXELS AND NUMBERS

When dealing with numbers that are applied to styling, React Native uses logical pixels (also known as "points" on iOS), as opposed to device pixels, at the JavaScript level. When working at the native level, you occasionally may need to work with device pixels by multiplying the logical pixels by the screen scale (e.g. 2x, 3x). We will discuss methods of manipulating pixels based on screen scale later in the book, but for we do not need to worry about this.

The `View` is the main building block of your UI, and is one of the most important component to understand if you want to get your styling right. Remember, a `View` element is very similar to a `div` in the sense that you can use it to wrap other elements and build blocks of UI code within it.

We will be going through each style property available to the `View` element one by one and looking closely at how itis implemented. Some of these properties overlap into other elements as mentioned before. For example, `Text` elements can also receive margin and padding properties among others. If we cover these style properties once we will not cover them again in detail when we go over other elements.

BACKFACEVISIBILITY

The `backfaceVisibility` *property* dictates whether or not an element is visible when the element is rotated more than 90 degrees. This property can be set to either 'visible' or `'hidden '`.

This property is useful when transforming elements by rotating their position, or flipping them backwards, and wanting to control whether or not they are still visible. For example, we will take a View and rotate it 180 degrees, and see how this property works (listing 4.8).

Listing 4.8 backfaceVisibility

```
<View style={styles.container}>
  <Text>Hello from App.js</Text>
</View>
 ...

const styles = StyleSheet.create({
    container: {
    padding: 20,
    backgroundColor: '#ededed',
```

```
    backfaceVisibility: 'visible',
    transform: [{
      rotateY: '180deg'
    }]
  }
})
```

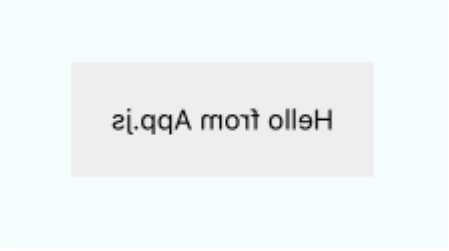


Figure 4.1 backfaceVisibility property

If we changed `backfaceVisibility` to `'hidden'`, then you would not see the element at all. If we did not set a property at all, which will be the case almost all of the time, then it would still show, as `backfaceVisibility` defaults to `'visible'`.

> **TRANSFORMS** We have not yet covered transforms, so don't worry about fully understanding them if you have not worked with them before. We'll be covering them in more depth later on in this chapter. If you have worked with CSS transforms before, just remember that they are similar to those.

BACKGROUNDCOLOR

The `backgroundColor` property sets the background color of an element. This property takes a string of one of the following properties. These same color properties are available anywhere colors are used in a React Native application:

- rgb stand for red, green, and blue. The values for red, green, and blue may be specified using a scale from 0–255. Higher numbers mean more of each color.
- alpha is similar to opacity (0 is transparent, 1 is solid)

```
'#06f'- #rgb
'#06fc'- #rgba
'#0066ff'- #rrggbb
'#0066ff00' - #rrggbb
'rgb(0, 102, 255)' - rgb(number, number, number)
'rgba(0, 102, 255, .5)' - rgb(number, number, number, alpha)
'hsl(216, 100%, 50%)' -hsl(hue, saturation, lightness)
'hsla(216, 100%, 50%, .5)' -hsl(hue, saturation, lightness, alpha)
'transparent' – transparent background
'dodgerblue' – any css3 specified named color (black, red, blue, etc...)
```

Let's create a basic `View` with a `backgroundColor` of red (listing 4.9).

Listing 4.9 backgroundColor

```
class App extends Component {
  render () {
    return (
      <View style={style.container}>
        <Text>Hello!</Text>
      </View>
    )
  }
}

const style = StyleSheet.create({
  container: {
    backgroundColor: 'red'
  }
})
```

Figure 4.2 backgroundColor property

When working with `backgroundColor`, as with all styles, you can also use variables (listing 4.10).

Listing 4.10 variables in styles

```
<View style={style.container}>
<Text>Hello!</Text>
</View>
...
const opacity = 0.5
const blue = '0, 102, 255'
const style = StyleSheet.create({
  container: {
    backgroundColor: `rgba(${blue}, ${opacity})`
  }
})
```

Figure 4.3 Using variables for styling

*Above, we are using template literals to process our variables. Template literals were introduced with the es2015 specification and are a great way to embed expressions into strings. To use them, just add back ticks around the statement and wrap any variable or

expressions in a dollar sign and two curly braces (`${someVariable}`). The two following expressions are exactly the same (listing 4.11).

Listing 4.11 Template literals

```
const name = 'Nader Dabit'
const greeting = 'Hello, ' + name
const greeting2 = `Hello, ${name}`
```

BORDER PROPERTIES

There are quite a few `border` properties, and they all work together, so we will go over them here all together:

- `borderColor`
- `borderWidth`
- `borderStyle`
- `borderLeftColor`
- `borderLeftWidth`
- `borderBottomColor`
- `borderBottomLeftRadius`
- `borderBottomRightRadius`
- `borderTopLeftRadius`
- `borderTopRightRadius`
- `borderBottomWidth`
- `borderTopColor`
- `borderTopWidth`
- `borderRightColor`
- `borderRightWidth`

To set a `border`, we must first set a `borderWidth`. The `borderWidth` is the size of the border, and it will always be a number. This can be done in a few ways. We can either set a `borderWidth` that applies to the entire component, or choose which `borderWidth` we would like to set specifically. First, let's set a `borderWidth` of 1 to a `View` element (listing 4.12).

Listing 4.12 borderWidth

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>
...
const style = StyleSheet.create({
  border: {
    borderWidth: 1
  }
})
```

Figure 4.4 borderWidth property

Notice that the border color defaults to black, and it is applied to the entire component on all four sides. Let's specify only a border left color and increase the width of the border so we can see it better (listing 4.13).

Listing 4.13 borderWidth with borderLeftColor

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>
...
const style = StyleSheet.create({
  border: {
    borderWidth: 3,
    borderLeftColor: 'red'
  }
})
```

Figure 4.5 border property

Now, let's declare a main `borderColor` property. Notice the results, specifically that the red color (`borderLeftColor`) does not get overwritten even though we have it declared before the green color (`borderColor`). This is because specificity takes precedence over generality (listing 4.14).

Listing 4.14 borderWidth with borderColor

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>

 ...

const style = StyleSheet.create({
   border: {
     borderWidth: 3,
     borderLeftColor: 'red',
     borderColor: 'green'
   }
 })
```

Figure 4.6 borderColor property

Now, let's see how the `borderWidth` property will respond with specificity. We'll do this by setting a `borderTopWidth` of 0 before we declare the `borderWidth` (listing 4.15).

Listing 4.15 borderTopWidth property

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>
...
const style = StyleSheet.create({
  border: {
    borderTopWidth: 0,
    borderWidth: 3,
    borderLeftColor: 'red',
    borderColor: 'green'
  }
})
```

As you can see, the `borderTopWidth` property also takes precedence over the `borderWidth` property:

Figure 4.7 borderTopWidth property

BORDERRADIUS

Now, let's look at `borderRadius`. `borderRadius` will allow us to define how rounded border corners are on our elements. Let's use `borderRadius` to make our view into a circle. To do this, we'll be setting a `width` and `height` value on our `View`, and calculating our `borderRadius` value as ½ of that value (listing 4.16).

* When specifying `borderRadius` without a position (for example, `borderTopRadius`), it will set the radius to all 4 corners of the element.

Listing 4.16 borderRadius property

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>

const style = StyleSheet.create({
  border: {
    borderWidth: 3,
    borderColor: 'green',
    width: 100,
    height: 100,
    borderRadius: 50
  }
})
```

Figure 4.8 borderRadius property

Notice that the `Text` element's background is covering up the circle! **This is because `Text` elements will always inherit the background color of the parent element**. Let's fix this. To do so, let's set a `backgroundColor` of `transparent` to the parent `View` (listing 4.17).

Listing 4.17 Setting background color to transparent

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>

 ...

const style = StyleSheet.create({
  border: {
    borderWidth: 3,
    borderColor: 'green',
    width: 100,
    height: 100,
    borderRadius: 50,
    backgroundColor: 'transparent'
  }
})
```

Figure 4.9 Setting parent of Text to transparent background

The fact that that `Text` elements inherit their `backgroundColor` will definitely be something to keep in mind when working with the `borderRadius` property, as most of the time there will be a parent `backgroundColor` set somewhere in the app, and you will need to override it.

Now, let's only set a `borderBottomLeftRadius` and `borderBottomRightRadius` (listing 4.18).

Listing 4.18 borderBottomRightRadius and borderBottomLeftRadius

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>
...
const style = StyleSheet.create({
  border: {
    borderWidth: 3,
    borderColor: 'green',
    width: 100,
    height: 100,
    borderBottomLeftRadius: 50,
    borderBottomRightRadius: 50
  }
})
```

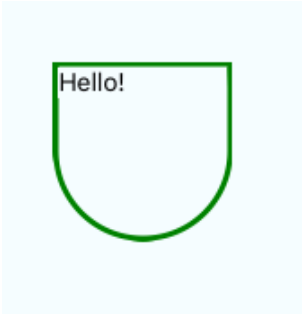


Figure 4.10 BorderBottomLeftRadius and borderBottomRightRadius

BORDERSTYLE

`borderStyle` is the last of the `border` properties we have yet to cover. The `borderStyle` property defaults to solid, which is what we have seen already. The other two options are dotted and dashed. Let's apply dashed to a `View` component and see how it looks (listing 4.19).

Listing 4.19 borderStyle property

```
<View style={style.border}>
  <Text>Hello!</Text>
</View>
...
const style = StyleSheet.create({
  border: {
    borderWidth: 3,
    borderColor: 'green',
    borderStyle: 'dashed'
  }
})
```

Figure 4.11 borderStyle property

MARGIN

Next, let's take a look at `margin`. `margin` defines how far away an element is to the previous or parent component.

The margin properties available are

- `margin`
- `marginLeft`
- `marginRight`
- `marginTop`
- `marginBottom`

If only the general `margin` property is set without another more specific value such as `marginLeft` or `marginTop`, then that value is passed to all sides of the component (top, right, bottom, and left). If both `margin` and a more specific `margin` property are specified (for example, `marginLeft`), then the more specific `margin` property takes precedence.

First, let's look at how `margin` effects an element. If you are familiar with CSS, you will notice the similarities.

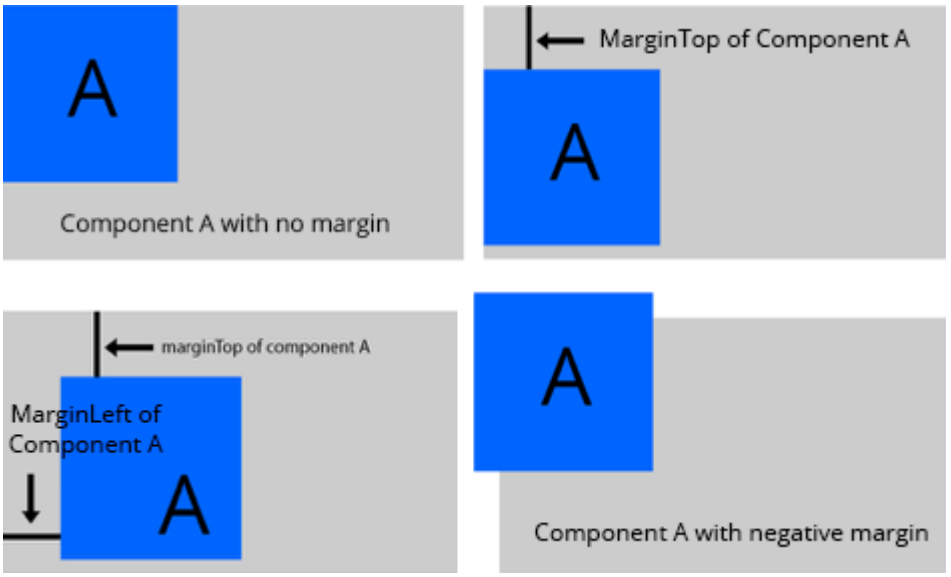


Figure 4.12 Components with different margins

Now, let's recreate `marginLeft` and `marginRight` functionality in a component (listing 4.20).

Listing 4.20 margin property

```
<View style={style.parent}>
  <View style={style.child}>
  <Text style={style.text}>A</Text>
  </View>
</View>
...
const style = StyleSheet.create({
  parent: {
    width: 300,
    height: 200,
    backgroundColor: '#cccccc'
  },
  child: {
    width: 150,
    height: 150,
    backgroundColor: '#0066ff',
    marginLeft: 40,
    marginTop: 40
  },
  text: {
    textAlign: 'center',
    fontSize: 80,
    marginTop: 25,
    backgroundColor: 'transparent'
  }
})
```

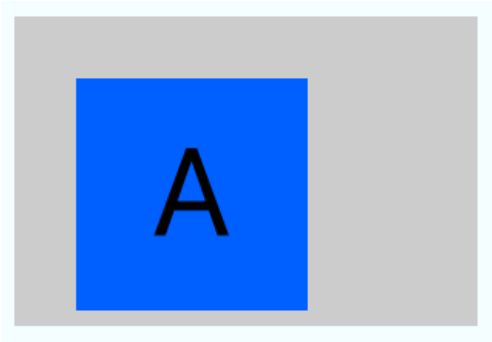


Figure 4.13 margin property applied to a component

PADDING

`padding` sets the space between the content of the element and the border of the same element.

The available properties available for `padding` are:

- `padding`
- `paddingLeft`
- `paddingRight`
- `paddingTop`
- `paddingBottom`

If only the main `padding` property is set without another more specific value such as `paddingLeft` or `paddingTop`, then that value is passed to all sides of the component (top, right, bottom, and left). If both `padding` and a more specific `padding` property are specified, for example, `paddingLeft`, then the more specific `padding` property takes precedence.

Let's recreate similar design as above, this time using `padding` instead of `margin`. *In figure 4.16, take a look* at components A and B, with no padding.

Figure 4.14 Component B with no padding

In figure 4.15, we add `paddingLeft` to component B.

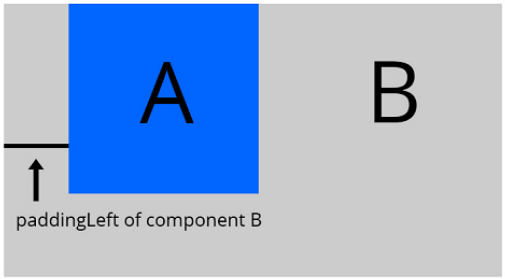


Figure 4.15 Component B with paddingLeft

Now, we'll recreate this `paddingLeft` styling in a component (listing 4.21).

Listing 4.21 paddingLeft property

```
<View style={style.parent}>
  <View style={style.child}>
    <Text style={style.text}>A</Text>
  </View>
</View>
...
const style = StyleSheet.create({
  parent: {
    width: 300,
    height: 200,
    backgroundColor: '#cccccc',
    paddingLeft: 40
  },
...
})
```

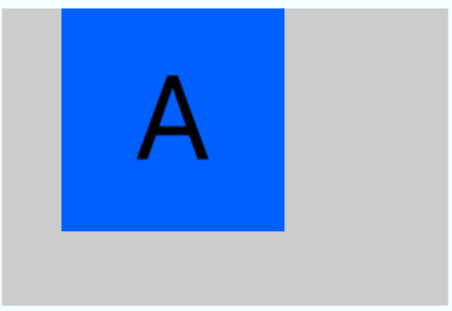


Figure 4.16 paddingLeft property applied to a component

POSITION

`position` defines how the component should be laid out in relation to the other components relative to it. The default position is '`relative`'

The available properties available for position are:

- `relative`
- `absolute`

When using `absolute` positioning, the following properties are also available:

- `top`
- `left`
- `bottom`
- `right`

As stated above, relative is the default position of all elements. `relative` position states that all items be laid out relative to the sibling or parent. This means that if there is a component that has a height of 100 and a `marginBottom` of 20, then the next component will be 20 pixels below the previous component. The only way to have two `relative` components overlap is to apply a negative margin to one of them

With `absolute` positioning, the component will be positioned absolutely relative to the parent, and whose margins relative to their parent can be controlled with top, bottom, left, and right properties. What does this mean? Well, think of it like this: If we have three nested components: A, B, and C, and give element C an `absolute` position, then C's position will be `absolute` *relative* to B, because B is the parent of C. If we want C to be positioned absolutely relative to A, then we need to move C to be a child of A.

If what we just described doesn't make sense, let's take a look at this in action to give us a clearer understanding of how absolute positioning works. We will have three components: container, parent, and child. We will give child a position of '`absolute`', left of 0, and bottom of 0 and see how this looks, taking into consideration the styling that parent has (specifically the `paddingLeft` property) (listing 4.22).

Listing 4.22 absolute positioning property

```
<View style={style.container}>
   <View style={style.parent}>
    <View style={style.child} />
  </View>
</View>
const style = StyleSheet.create({
  container: {
    paddingTop: 200,
    backgroundColor: '#f4fcff',
    paddingLeft: 40,
    flex: 1
  },
  parent: {
```

```
    width: 300,
    height: 200,
    backgroundColor: '#cccccc',
    paddingLeft: 40
  },
  child: {
    width: 150,
    height: 150,
    position: 'absolute',
    backgroundColor: 'red',
    bottom: 0,
    left: 0
  }
})
```

Figure 4.17 absolute positioning

Notice that the `paddingLeft` had no effect on the element, and we were able to position the element exactly where we wanted it by giving it a `left`value of 0 and a `bottom` value of 0. Also notice that the element is still within its parent, which is what we meant when we said relative to its parent. It did not go to the bottom of the entire screen because it still has to stay relative to its parent. Next, let's take the child component out of the parent and place it directly into the container, keeping the same styling (listing 4.23).

Listing 4.23 absolute positioning property

```
<View style={style.container}>
<View style={style.parent} />
<View style={style.child} />
</View>
```

Figure 4.18 absolute positioning

Now that the child component is relative to the entire container, we see that it drops down to the bottom left of the screen.

SHADOWPROPTYPESIOS & ELEVATION

If you are looking to add a drop shadow to a `View` element, there are separate ways to do this depending on what platform you are on.

If you are on Android, you use `elevation` which uses Android's underlying elevation API. This adds a drop shadow to the item and affects z-order (z-index) for overlapping views.

If you are on iOS you use `ShadowPropTypesIOS` for drop shadows, which will only add a shadow and will not affect the z-order.

The available properties available for `ShadowPropTypesIOS` are

- `shadowColor`
- `shadowOffset`
- `shadowOpacity`
- `shadowRadius`

The available properties available for elevation are

- `number`**`(0 to infinity)`**

Let's take the previous component we were using and add a drop shadow to the child using `elevation` (Android only) (listing 4.24).

Listing 4.24 absolute positioning property

```
<View style={style.parent}>
  <View style={style.child}>
    <Text style={style.text}>A</Text>
  </View>
</View>
...
const style = StyleSheet.create({
...
  child: {
    width: 150,
    height: 150,
    backgroundColor: '#0066ff',
    elevation: 11
  },
...
})
```

Figure 4.19 elevation

As we stated earlier, `elevation`also effects the z-index of the item. This means that if there are two or more items occupying the same space, we can decide which one needs to be in front by giving it the larger elevation and therefore the larger z-index.

Let's see this in practice. To do so, we will create a `View` with three boxes, each of which are positioned absolutely. We will give them three different elevations: 1, 2, and 3. Though they will be in the order of 2, 1, and 3 in our code, the styling will apply an elevation of 1 to child 1, 2 to child 2, and 3 to child three, making them appear in correct order though they are not laid out the correct order in our code (listing 4.25).

Listing 4.25 Elevation and z-index

```
<View style={style.parent}>
  <View style={[style.child, style.child2]} />
    <View style={[style.child, style.child1]} />
  <View style={[style.child, style.child3]} />
</View>

const style = StyleSheet.create({
  parent: {
```

```
    width: 300,
    height: 200,
    backgroundColor: '#cccccc',
    paddingLeft: 40
  },
  child: {
    width: 150,
    height: 150,
    position: 'absolute'
  },
  child1: {
    backgroundColor: 'red',
    top: 0,
    left: 0,
    elevation: 1
  },
  child2: {
    backgroundColor: 'orange',
    top: 20,
    left: 20,
    elevation: 2
  },
  child3: {
    backgroundColor: 'blue',
    top: 40,
    left: 40,
    elevation: 3
  }
})
```

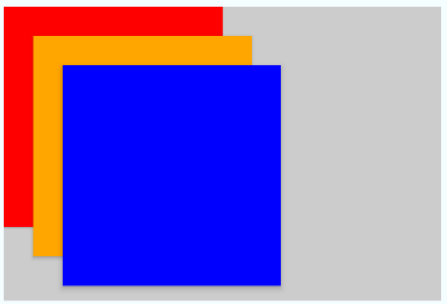


Figure 4.20 Layered elevation

As you can see in figure 4.22 or when you run this code, even though child2 comes before child1 in the code, child1 is behind child2 when the component is rendered.

Next, let's create a shadow on an iOS element. Before we do so, let's go over the four available properties for adding a shadow (usually all used together to get the right effect):

```
shadowColor
shadowOffset
shadowOpacity
shadowRadius
```

SHADOWCOLOR

`shadowColor` is the color of the shadow (for example, 'red', 'rgba(0,0,0,.3), and so on).

SHADOWOFFSET

`shadowOffset`is the distance from the element that the shadow should appear. It takes the following arguments:

```
shadowOffset : {
  width: number,
  height: number
}
```

SHADOWOPACITY

`shadowOpacity` is the opacity of the shadow (number)

SHADOWRADIUS

`shadowRadius` allows us to describe how spread out we would like our shadow to be. The smaller the number, the less spread out and denser the shadow is. The larger the number, the more spread out and the less dense it will be.

Let's set up a basic component and use the above properties to add a shadow (listing 4.26).

Listing 4.26 ShadowPropTypesIOS

```
<View style={style.container}>
  <View style={style.child} />
</View>

  container: {
    paddingTop: 200,
    backgroundColor: '#f4fcff',
    paddingLeft: 40,
    flex: 1
  },
  child: {
    width: 150,
    height: 150,
    backgroundColor: 'red',
    marginLeft: 90,
    shadowColor: 'black',
    shadowOffset: {
      height: 2,
      width: 2
  },
  shadowOpacity: 0.4,
  shadowRadius: 10
}
```

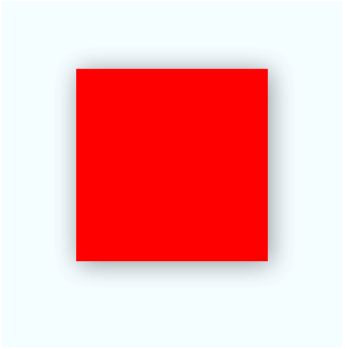


Figure 4.21 ShadowPropTypesIOS

As you can see above, we have set the `shadowColor` to black, given the `shadowOffset` a `height` of 2 and a `width` of 2 (which means it will be pushed 2 pixels to the right and down 2 pixels. These can also be negative numbers for the shadow to go up and left), a `shadowOpacity` of .4, and a `shadowRadius` of 10.

TRANSFORMS

Transforms allow you to modify the shape and position of an element in 3d space. What this means it that we can use this property to do things like rotate, scale, and skew components. These transform properties are especially useful when working with animations. Transform takes an array of transform properties, for example:

```
transform: [{rotate: '90deg ', scale: .5}]
```

The properties available for Transforms are:

- `perspective`
- `rotate`
- `rotateX`
- `rotateY`
- `rotateZ`
- `scale`
- `scaleX`
- `scaleY`
- `translateX`
- `translateY`
- `skewX`
- `skewY`

We will go over these one by one and see how they work.

PERSPECTIVE

`perspective` gives an element a 3D-space by affecting the distance between the Z plane and the user. This is used with other properties to give a 3d effect.

ROTATE

```
transform: [{ rotate:  '45deg' }]
```

rotate does just what it sounds like it would, it rotates an element. Let's take our red square from earlier and rotate it 45 degrees (listing 4.27).

Listing 4.27 ShadowPropTypes IOS

```
<View style={style.container}>
  <View style={style.child} />
</View>
container: {
 ...
},
child: {
  width: 150,
  height: 150,
  backgroundColor: 'red',
  marginLeft: 90,
  transform: [
    {rotate: '45deg' }
  ]
}
```

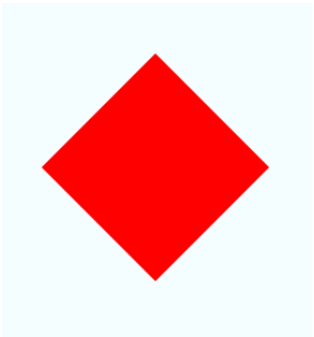


Figure 4.22 rotate

ROTATEX

```
transform: [{ rotateX:  '50deg' }]
```

`rotateX` rotates an element on its x axis. It's not too apparent what is going on if we use our previous red square, let's instead use some large text and apply this property (listing 4.28).

Listing 4.28 rotateX

```
<View style={style.container}>
  <View style={style.child}>
    <Text style={style.text}>rotateX</Text>
  </View>
</View>
container: {
 ...
},
child: {
  width: 150,
  height: 150,
  marginLeft: 90,
  transform: [
    {rotateX: '45deg' }
  ]
},
test: {
  fontSize: 24,
  textAlign: 'center'
}
```

Our text should now be rotated 45 degrees on the x axis, skewing the way it looks:

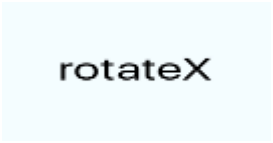


Figure 4.23 rotateX

ROTATEY

```
transform: [{ rotateY:  '50deg' }]
```

`rotateY` rotates an element on its Y axis. We will use the same example from last time, but switching the rotateX for rotateY (listing 4.29).

Listing 4.29 rotateY

```
...
},
child: {
  ...
  transform: [
    {rotateX: '45deg' }
  ]
}
```

Our text should now be rotated 45 degrees on the y axis, skewing the way it looks:

Figure 4.24 rotateY

ROTATEZ

```
transform: [{ rotateZ:  '50deg' }]
```

`rotateZ` rotates an element on its Z axis. We will use the same example from last time, but switching the `rotateY` for `rotateZ`. Let's also add the `backgroundColor` back to give us a better idea of what is going on (listing 4.30).

Listing 4.30 rotateZ

```
...
},
child: {
  width: 150,
  height: 150,
  marginLeft: 90,
  backgroundColor: 'red',
  transform: [
    {rotateZ: '50deg' }
  ]
},
text: {
  fontSize: 24,
  textAlign: 'center'
}
```

Our text should now be rotated 45 degrees on the Z axis, rotating it to the right:

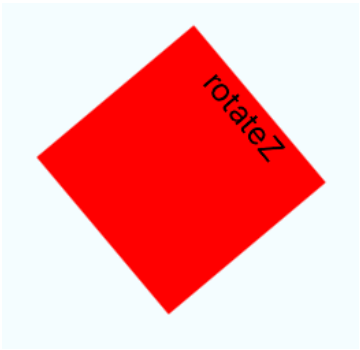


Figure 4.25 rotateZ

SCALE

```
transform: [{ scale: .3 }]
```

`scale` multiplies the size of the element by the number passed to it, the default being 1. If we want an element to appear larger, we can pass a value larger than 1, and if we want it to be smaller, a value smaller than 1. Let's create three squares, and scale two of them (listing 4.31).

Listing 4.31 scale

```
<View style={style.container}>
  <View style={style.child} />
  <View style={[style.child, style.scale1]} />
  <View style={[style.child, style.scale2]} />
</View>

container: {
  paddingTop: 100,
  backgroundColor: '#f4fcff',
  paddingLeft: 40,
  flex: 1
},
child: {
  width: 50,
  height: 50,
  marginLeft: 90,
  marginTop: 30,
  backgroundColor: 'red'
},
scale1: {
  transform: [
    {scale: .5 }
  ]
},
scale2: {
  transform: [
    {scale: 2 }
  ]
}
```

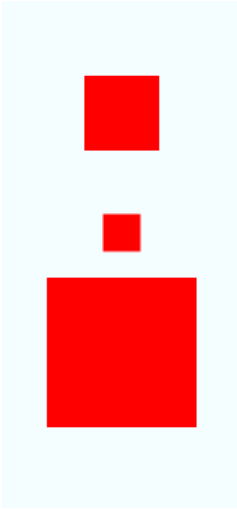


Figure 4.26 scale

TRANSLATEX AND TRANSLATEY

```
transform: [{ translateX: 150 }]
```

`translate` moves an element along the x (`translateX`) or y (`translateY`) axis from the current position. This is not very useful in normal development as we already have margin, padding, and other position properties available. This is something that becomes very useful though when we get into animations. To demonstrate this, we will have two components, and we will add a `translateX` to one of them (listing 4.32).

Listing 4.32 translateX

```
<View style={style.container}>
  <View style={style.child} />
  <View style={[style.child, style.childX]} />
</View>

child: {
  width: 150,
  height: 150,
  marginLeft: 90,
  marginTop: 30,
  backgroundColor: 'red'
},
childX: {
  transform: [{ translateX: 50 }]
}
```

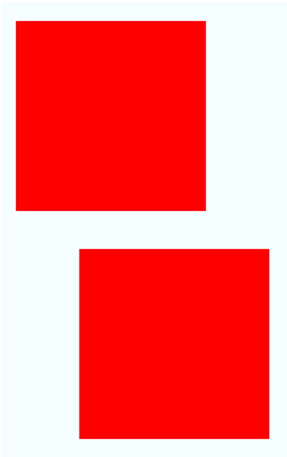


Figure 4.27 translateX

SKEW

```
transform: [{ skewY: '50deg' }]
```

The `skew` property will skew an element across wither the X or Y axis. Let's apply these properties to two square components (listing 4.33).

Listing 4.33 skew

```
<View style={style.container}>
  <View style={[style.child, style.childY]} />
  <View style={[style.child, style.childX]} />
</View>

child: {
  width: 150,
  height: 150,
  marginLeft: 90,
  marginTop: 30,
  backgroundColor: 'red'
},
childY: {
  transform: [{ skewY: '45deg' }]
},
childX: {
  transform: [{ skewX: '45deg' }]
}
```

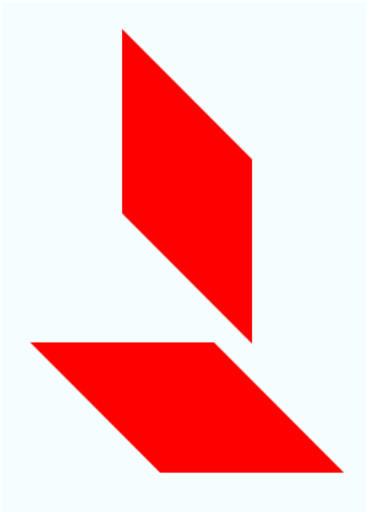


Figure 4.28 skew

4.2 Styling Text Components

With the exception of Flex properties which we have yet to go over, most of the styles applicable to `View` elements will also work as expected with `Text` elements. Most of the styles that `Text` elements can utilize will not work for `View` elements.

Here is a list of styles that can be applied to Text components:

- `color`
- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `lineHeight`
- `textAlign`
- `textDecorationLine`
- `textShadowColor`
- `textShadowOffset`
- `textShadowRadius`

Android only:

- `textAlignVertical`

iOS only:

- `letterSpacing`
- `textDecorationColor`
- `textDecorationStyle`
- `writingDirection`

COLOR

```
color: 'red'
```

This property specifies the color of the text in a `Text` element.

All of the color properties we covered earlier will also work here, but just as an overview let's take a look at them again:

```
'#06f'- #rgb
'#06fc'- #rgba
'#0066ff'- #rrggbb
'#0066ff00' - #rrggbb
'rgb(0, 102, 255)' - rgb(number, number, number)
'rgba(0, 102, 255, .5)' - rgb(number, number, number, alpha)
'hsl(216, 100%, 50%)' - hsl(hue, saturation, lightness)
'hsla(216, 100%, 50%, .5)' - hsl(hue, saturation, lightness, alpha)
'transparent' – transparent background
'dodgerblue' – any css3 specified named color (black, red, blue, etc...)
```

Let's create a couple of `Text` elements and pass them some different colors (figure 4.29 and the listing).

Figure 4.29 color

Listing 4.34 color

```
<View>
  <Text style={style.text1}>Text1</Text>
  <Text style={style.text2}>Text2</Text>
  <Text style={style.text3}>Text3</Text>
  <Text style={style.text4}>Text4</Text>
</View>

text1: {
```

```
  color: '#06f'
},
text2: {
  color: 'rgba(0, 0, 0, .7)'
},
text3: {
  color: '#666'
},
text4: {
  color: 'red'
}
```

FONTFAMILY

```
fontFamily: 'string'
```

For iOS, there are a large number of available fonts that can be implemented out of the box. For android, there are only three: (`normal`(Droid Sans), `serif`(Droid Serif), and `monospace` (Droid Sans Mono)). For a full list of iOS fonts available out of the box in React Native, go to: https://github.com/dabit3/react-native-fonts.

We can also add custom fonts to our project using font files (ttf, otf, and so on). We will go over how to add new and custom fonts to our project in a later chapter. For now, let's just talk about how to use existing fonts in our project.

Let's look at how to implement a custom font in iOS using the `fontFamily` property (listing 4.35).

Listing 4.35 iOS fontFamily

```
<Text style={style.text}>ChalkboardSE-Regular</Text>

 const style = StyleSheet.create({
    text: {
    fontFamily: 'ChalkboardSE-Regular',
    fontSize: 25
  }
})
```

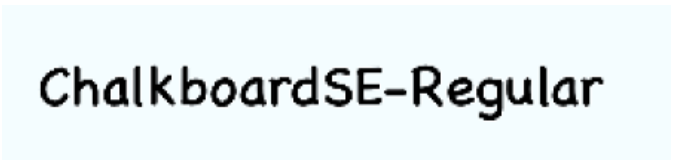


Figure 4.30 iOS fontFamily

Custom font in Android (listing 4.36).

Listing 4.36 Android fontFamily

```
<Text style={style.text}>monospace</Text>

const style = StyleSheet.create({
  text: {
    fontFamily: 'monospace',
    fontSize: 25
  }
})
```

Figure 4.31 Android fontFamily

FONTSIZE

```
fontSize: 18
```

`fontSize` is pretty simple, it just adjusts the size of the text in a `Text` element. We've used this already quite a bit, so we won't go into too much detail other than the fact that the default `fontSize` is 14.

FONTSTYLE

```
fontStyle: 'italic'
```

This is to change the font style to italic. The default is 'normal '. The only two options at this moment are 'normal ' and 'italic.'

FONTWEIGHT

```
fontWeight: 'bold'
```

`fontWeight` refers to the thickness of the font. The default is `'normal' or '400'`.

The options for fontWeightare one of the following: `'normal', 'bold', '100', '200', '300', '400', '500', '600', '700', '800', '900'`. The smaller you go, the lighter / thinner the text gets. The larger you go, the thicker / bolder the text gets.

LINEHEIGHT

```
lineHeight: 20
```

`lineHeight` specifies the height of the text element. When using `lineHeight`, the default functionality is that the text will be aligned at the bottom.

Let's take a look at an example and then view it in our inspector to see this in action. We will set up three `Text` elements, and give the middle element a `lineHeight` of 50 (listing 4.37).

Listing 4.37 lineHeight

```
<View style={style.container}>
  <Text style={style.text}>Text</Text>
  <Text style={style.text2}>Text</Text>
  <Text style={style.text3}>Text</Text>
</View>

const style = StyleSheet.create({
  text: {
    fontSize: 20
  },
  text2: {
      fontSize: 20,
    lineHeight: 50
  },
  text3: {
      fontSize: 20
  }
})
```

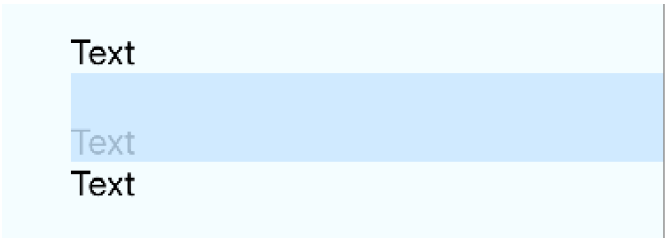


Figure 4.32 lineHeight (iOS debugger)

As you can see above (iOS simulator and debugger) highlighted in the darker blue, the height of the middle text element is larger than the others. Also notice that all of the height has been added to the top of the element.

This is only true in iOS. In Android, the opposite is true. The text will be aligned to the top in Android, not the bottom.

TEXTALIGN

```
textAlign: 'center'
```

`textAlign` refers to how the text in the element will be horizontally aligned.

The options for `textAlign` are the following:

```
'auto', 'center', 'right', 'left', 'justify' ('justify' is iOS only).
```

Setting the `textAlign` property will change the horizontal alignment of the text in the element.

TEXTDECORATIONLINE

```
textDecorationLine: 'underline '
```

The options for `textDecorationLine` are `'none'`, `'underline'`, `'line-through'`, and `'underline line-through'`. The default value is `'none'`.

This property adds either an underline or line through the given text.

TEXTSHADOW

`textShadow` encompasses the following three properties:

```
textShadowColor: 'red '
textShadowOffset: {width: -2, height: -2}
textShadowRadius: 4
```

This property allows us to add a shadow to a `Text` element. Let's incorporate this into a component to see how it works (listing 4.38).

Listing 4.38 textShadow

```
<Text style={style.text}>Text Shadow</Text>

 const style = StyleSheet.create({
   text: {
     marginLeft: 20,
     fontSize: 25,
     textShadowColor: 'red',
     textShadowOffset: {width: -2, height: -2},
     textShadowRadius: 3
   }
 })
```

Figure 4.33 textShadow

As you can see, the text shadow is starting from the left and the top of the text. This is because we declared width and height of -2.

TEXTALIGNVERTICAL (ANDROID ONLY)

```
textAlignVertical: 'center'
```

The options for `textAlignVertical` are `'auto'`, `'top'`, `'bottom'`, and `'center'`.

This will allow us to choose the alignment position of our text The default value for this property in Android is ' `top` '. This is especially useful when using the `lineHeight` property (listing 4.39).

Listing 4.39 textAlignVertical

```
<Text style={style.text}>Text1</Text>
<Text style={[style.text, style.alignCenter]}>Text2</Text>
<Text style={style.text}>Text3</Text>

 const style = StyleSheet.create({
   text: {
     fontSize: 25
   },
   alignCenter: {
     textAlignVertical: 'center',
     lineHeight: 100
   }
 })
```

Figure 4.34 textAlignVertical

LETTERSPACING(IOS ONLY)

```
letterSpacing: 2
```

`letterSpacing` specifies spacing between text characters (listing 4.40).

Listing 4.40 letterSpacing

```
<Text>No letter spacing</Text>
  <Text style={style. text }>With letter spacing</Text>

  const style = StyleSheet.create({
    text: {
      letterSpacing: 3
    }
  })
```

Figure 4.35 letterSpacing

TEXTDECORATION STYLES(IOS ONLY)

```
textDecorationLine: 'underline'
textDecorationColor: 'red'
textDecorationStyle: 'double'
```

`textDecorationColor` and `textDecorationStyle` are used with `textDecorationLine` to give custom control over the styling of the `textDecorationLine` property.

`textDocorationStyle` can take any of the following properties: 'solid', 'double', 'dotted', 'dashed'

Listing 4.41 textDecoration styles

```
<Text>No letter decoration</Text>

const style = StyleSheet.create({
  text: {
    textDecorationColor: 'red',
    textDecorationStyle: 'dotted',
    textDecorationLine: 'underline'
  }
})
```

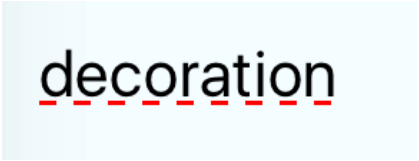


Figure 4.36 textDecoration styles

WRITINGDIRECTION

```
writingDirection: 'rtl'
```

`writingDirection`gives us control over the direction that the text is displayed (right to left, left to right). This is especially useful when implementing internationalization.

`writingDirection` can take any of the following properties: `'auto', 'ltr', 'rtl'`

4.3 Summary

In this chapter, we learned:

- Styling can be applied inline or by using a StyleSheet and referencing the style variable used when creating the StyleSheet.
- Styling View components
- Styling Text Components

5

Styling in depth

This chapter covers

- Flexbox
- Dynamic Styles
- Organizing styles
- Summary

Now that we have an understanding of styling elements in React Native, as well as how to use them, we'll take a look at the React Native layout system using Flexbox. FlexBox is a fundamental concept that needs to be properly understood to create layouts and UIs in React Native.

We'll put all of the styling knowledge we have together and harness props and state to dynamically style our components, styling and updating styles based on these properties.

Next, we'll talk about a few ways to define styles that we will be reusing by exporting and importing them into other files and components.

Finally, we'll talk about a few different strategies and best practices that make for practical code organization.

5.1 Flexbox

Flexbox is a layout implementation that React Native uses to provide an efficient way for users to create UIs and control positioning in React Native. The React Native Flexbox implementation is based on the W3C Flexbox web specification, but does not share 100% of the APIthat the W3C implementation carries with it. It aims to give us an easy way to reason about, align and distribute space among items in our layout, even when their size is not known or even when their size is dynamic. Flexbox layout is only available for use on View

components. To better understand how Flexbox works, let's go ahead and take a look at some code.

5.1.1 Flexbox Properties

Here are the alignment properties specific to flexBox layout in a View component:

```
flex
flexDirection
alignItems
justifyContent
alignSelf
flexWrap
```

5.1.2 flex Property

The first property we need to talk about is the flex property. The flex property is used with a key of flex, and a value of a number or a variable that holds a number:

Listing 5.1 flex property Using the flex property in your styling

```
flex: 1
```

The flex number specifies the ability of the item to alter its dimensions to fill the space of the container that it is within. This value is relative to the rest of the items within the same container.

This means that if we have a View element with a height of 300 and a width of 300, and a child View with a property of flex: 1, then the child view will completely fill the parent view.

If we decide to add another child element with a flex property of flex: 1, they will each take up equal space within the parent container.

Another way to look at this is to think of the flex properties as being percentages. For example, if you would like your child components to take up 66.6% and 33.3% respectively, you could use flex:66 and flex:33, which would also work, with the first item occupying 2/3 and the second item occupying 1/3 of the parent container. The flex number is only important relative to the other flex items occupying the same space. Thinking about flex numbers in percentages sometimes makes it much easier to reason about.

To better understand how this all works, let's take a look at a couple of diagrams depicting Views with a few different flex values:

Listing 5.2 flex property flex:1 example in code flex property

```
<View style={styles.container}>
  <View style={styles.flex1} />
</View>

container: {
  width: 300,
  height: 300,
```

```
  marginTop: 150,
  backgroundColor: 'white'
},
flex1: {
  flex: 1,
  backgroundColor: '#666666'
}
```

As you can see, this filled the entire 150 width and height with the child box. Next, let's add another box next to it with a lighter shade of gray:

Listing 5.3 flex property Example with two flex items in code

```
<View style={styles.container}>
  <View style={styles.flexBox1} />
  <View style={styles.flexBox2} />
</View>

container: {
  width: 300,
  height: 300,
  marginTop: 150,
  backgroundColor: 'white'
},
flexBox1: {
  flex: 1,
  backgroundColor: '#666666'
},
flexBox2: {
  flex: 1,
  backgroundColor: '#ededed
}
```

Figure 5.2 Example with two flex items on device

Now, we see that there is still a single square container but our two boxes now share equally the space of the parent container. Next, let's change the flex property of the second box to flex:2 and see what happens:

Listing 5.4 flex property Example of two flex items with different flex properties in code

```
<View style={styles.container}>
  <View style={styles.flexBox1} />
  <View style={styles.flexBox2} />
</View>

container: {
...
},
flexBox1: {
  flex: 1,
  backgroundColor: '#666666'
},
flexBox2: {
  flex: 2,
  backgroundColor: '#ededed
}
```

Figure 5.3 flex:1 example in with two flex items on device

Now, the second flex item takes up twice as much space as the first flex item.

5.1.3 flexDirection Property

You may notice that the items in our flex container are laying out in a column. Using the flexDirection property, we canchange the direction of the layout. flexDirection is applied to the parent view that contains child flex views.

Listing 5.5 flexDirection property flexDirection in code

```
flexDirection: 'row'
```

There are two options for this property: row and column. The default setting is column. If you do not specify a flexDirection property, your content will lay out in a column layout as we saw

before because column is the default layout. Let's change this to flexDirection: row and see how it looks. Everything else in our code will stay the same:

Listing 5.6 flexDirection property Implementation of flexDirection in code

```
<View style={styles.container}>
  <View style={styles.flexBox1} />
  <View style={styles.flexBox2} />
</View>

container: {
  width: 150,
  height: 150,
  marginTop: 150,
  backgroundColor: 'white',
  marginLeft: 20,
  flexDirection: 'row'
},
flexBox1: {
  ...
},
flexBox2: {
  ...
}
```

Figure 5.4 Implementation of flexDirection on device

As you can see, the child elements now lay out left to right. This property is something that you will use a lot when developing apps in React Native so it is important to grasp it and understand how it works.

5.1.4 alignItems Property

alignItems allows us to specify the alignment of the flex items. This property is declared on the parent view and affects the child flex items.

Listing 5.7 alignItems property alignItems in code

```
alignItems: 'flex-start'
```

There are four options for this property: flex-start, flex-end, center, and stretch. The default value is stretch, so if no other value is declared, stretch is the behaviour you will get out of the box.

Let's create a component to test out this property:

Listing 5.8 alignItems property Base styles without alignItems implemented

```
<View style={styles.container}>
  <View style={styles.flexBox1}>
    <Text>AlignItems</Text>
  </View>
</View>

const styles = StyleSheet.create({
  container: {
    width: 150,
    height: 150,
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20
  },
  flexBox1: {
    backgroundColor: '#666666'
  }
})
```

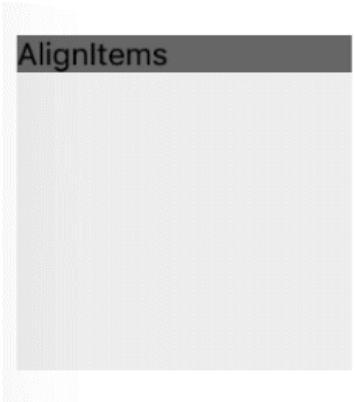


Figure 5.5 Above code rendered on device

As you can see, the flexBox1 style and child take up the entire width of the container. Everything that happens concerning alignItems is relative to the flexDirection. As you can see, we have not declared a flexDirection, so we are getting the flexDirection column behaviour as

it is the default. Let's add flexDirection:'row' to the parent container and see what happens. Keep in mind that the behaviour we are seeing is the same as if we declared alignItems: 'stretch' on the parent container:

Listing 5.9 Adding flexDirection to the container Adding flexDirection to container in code

```
container: {
  ...
  flexDirection: 'row'
}
```

Figure 5.6 flexDirection:row on device

Let's change only alignItems property of the container by adding alignItems: 'center' to the styling. We'll be keeping the flexDirection: 'row' property for now:

Listing 5.10 alignItems added to container alignItems:'center' implemented in code

```
container: {
  ...
  flexDirection: 'row',
  alignItems: 'center'
}
```

Figure 5.7 alignItems:'center' on device.

Now, the content is centered vertically. Next, let's remove the flexDirection: 'row'property:

Figure 5.8 alignItems:'center' on device, withouth flexDirection: 'row'

Because the flexDirection is now defaulted to column, the content is centered horizontally.

Changing between the row and column flexDirection and understanding alignItems along with the soon to be covered justifyContent can sometimes get confusing as they change based on flexDirection. Don't worry about understanding this exactly right away. Just remember that being aware of how these properties affect each other should allow you to debug and troubleshoot much easier.

Next, let's implement the same view with alignItems: 'flex-start'. flex-start will align the items with the beginning of the parent container:

Listing 5.11 alignItems: 'flex-start' added to container alignItems: 'flex-start' implemented in code

```
container: {
  width: 150,
  height: 150,
  marginTop: 150,
  backgroundColor: '#ededed',
  marginLeft: 20,
  alignItems: 'flex-start'
}
```

Figure 5.9 alignItems:'flex-start' on device

Because both alignItems properties begin their axes at the top left of the container, they both will render as shown above.

Finally, let's implement alignItems: 'flex-end', with no flexDirection property:

Listing 5.12 alignItems: 'flex-end' added to container alignItems 'flex-end' implemented in code

```
container: {
  ...
  alignItems: 'flex-end'
}
```

Figure 5.10 alignItems 'flex-end' on device

The items now vertically align flush with the end of the container. This is because our flexDirection is defaulted to column.

Now, we will change the flexDirection property to 'row':

Listing 5.13 alignItems: 'flex-end' with flexDirection: 'row' alignItems: 'flex-end' with flexDirection: 'row' in code

```
container: {
  ...
  alignItems: 'flex-end',
  flexDirection: 'row'
}
```

Figure 5.11 alignItems: 'flex-end' with flexDirection: 'row' on device

Now that our flexDirection has changed to row, the end of the flex container is rendered as being the bottom vertically. Remember that the columns stack vertically, and the rows stack horizontally. This means the the beginning and end of the column will be left and right, while the beginning and end of the row will be top and bottom.

5.1.5 justifyContent Property

The justifyContent property defines how space is distributed between and around flex items along the main-axis of their container.

The alignment is done after the lengths and auto margins are applied, meaning that, if there is at least one flexible element, with a flex property different from `0`, it will have no effect as there won't be any available space.

Listing 5.14 alignItems property justifyContent property in code

```
justifyContent: 'flex-end'
```

There are five options for this property: flex-start, flex-end, center, space-around, and space-between. The default value is flex-start, so if no other value is declared, flex-start is the behaviour you will get out of the box.

Let’s take another look at the component we have been working with so far:

Listing 5.15 justifyContent default behaviour justifyContent default behaviour in code

```
<View style={styles.container}>
  <View style={styles.flexBox1}>
   Text>AlignItems</Text>
  </View>
</View>

container: {
  width: 150,
  height: 150,
  marginTop: 150,
  backgroundColor: '#ededed',
  marginLeft: 20
},
  flexBox1: {
  backgroundColor: '#666666'
}
```

Figure 5.12 justifyContent default behaviour on device

As you can see, our content displays at the top of the parent container. This is the behavior we will get if we applied justifyContent: 'flex-start' as well. Now, let's append the justifyContent: 'flex-end' property to our container style:

Listing 5.16 justifyContent'flex-end' justifyContent: 'flex-end' in code

```
container: {
...
  justifyContent: 'flex-end'
}
```

Figure 5.13 justifyContent: 'flex-end' in on device

Now we see the content move to the end of the container. If we were using flexDirection: 'row' as our layout, the content would instead move to the right side of the container, taking up the entire height.

Next, let's implement justifyContent: 'center'. This will center our content in the parent container:

Listing 5.17 justifyContent: 'center' justifyContent: 'center' implemented in code.

```
container: {
  ...
  justifyContent: 'center'
}
```

Figure 5.14 justifyContent: 'center' on device

Our content is now rendered in the center of our flex container.

Most of the examples we have looked at so far have only been implemented with one child element. We looked at them in this way to make understanding how they worked a little easier, but feel free to experiment with the properties we have covered so far by adding multiple elements and seeing what happens, and if it is what you expected. For the next two justifyContent properties, we will need to display multiple child elements in order to understand how they work.

The next property we will implement will be the space-between property. space-between will basically distribute the items evenly between the first item at the start and the last at the end.

To get started with this and see space-around in action, we will need to create a reusable Box component. To do so, let's change code to the below:

Listing 5.18 justifyContent: 'space-between' justifyContent: 'space-between' implemented in code.

```
class App extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Box />
```

```
        <Box />
        <Box />
      </View>
    )
  }
}

const Box = () => (
  <View style={styles.flexBox1}>
    <Text>AlignItems</Text>
  </View>
)

const styles = StyleSheet.create({
  container: {
    width: 150,
    height: 150,
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    justifyContent: 'space-between'
  },
  flexBox1: {
    backgroundColor: '#666666'
  }
})
```

Figure 5.15 justifyContent: 'space-between' on device

The last property we will implement will be the space-around property. space-around will basically distribute items evenly with all items having equal space around them.

Listing 5.19 justifyContent: 'space-around' justifyContent: 'space-around in code'

```
  container: {
    ...
    justifyContent: 'space-around'
  }
```

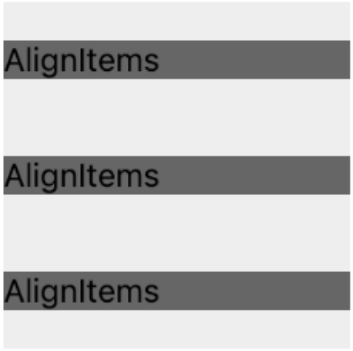


Figure 5.16 justifyContent: 'space-around on device

5.1.6 alignSelf Property

So far, all of the properties have been implemented by applying them to the parent container. With alignSelf, we can get access to the alignItems property for individual elements with a container. This means that if we have multiple items in a container and we do not them all to have the same alignItems property passed down from the parent, we can implement alignSelf and control each one individually.

There are five options for this property: flex-start, flex-end, center, auto, and stretch. The default value is auto, so if no other value is declared, auto is the behaviour you will get out of the box.

To show how this works, let's add a new property to our Box component and pass down alignSelf as a prop. This way, we can see all of the properties side by side:

Listing 5.20 alignSelf alignSelf property in code

```
class App extends Component {
    render () {
      return (
      <View style={styles.container}>
        <Box align='auto' />
        <Box align='stretch' />
        <Box align='flex-start' />
        <Box align='center' />
        <Box align='flex-end' />
      </View>
    )
  }
}

const Box = ({align}) => (
  <View style={[styles.flexBox1, {alignSelf: align} ]}>
    <Text>AlignItems</Text>
  </View>
```

```
)

const styles = StyleSheet.create({
  container: {
    width: 150,
    height: 150,
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20
  },
  flexBox1: {
    backgroundColor: '#666666'
  }
})
```

Figure 5.33 alignSelf property in code

Figure 5.17 alignSelf property on device

Now we have independent control of the alignment of the child items. If there is an alignItems property on the parent container, alignSelf will override the alignItems property.\

5.1.7 flexWrap Property

When having multiple child elements within a parent component with a flexDirection of row, these elements will not wrap to stay in the view, but will instead keep going off of the view. To fix this, you may need to use the flexWrap property.

There are two options for this property: no-wrap and wrap. The default value is no-wrap, so if no other value is declared, no-wrap is the behaviour you will get out of the box.

To understand this better, let's look at an example. Consider the following code and layout:

Listing 5.21 flexWrap property flexWrap not defined (defaults to no-wrap) in code

```
class App extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Box />
        <Box />
        <Box />
        <Box />
        <Box />
      </View>
    )
  }
}

const Box = () => (
  <View style={styles.flexBox1} />
)

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flex: 1,
    flexDirection: 'row'
  },
  flexBox1: {
    width: 150,
    height: 150,
    marginLeft: 10,
    marginBottom: 10,
    backgroundColor: 'red'
  }
})
```

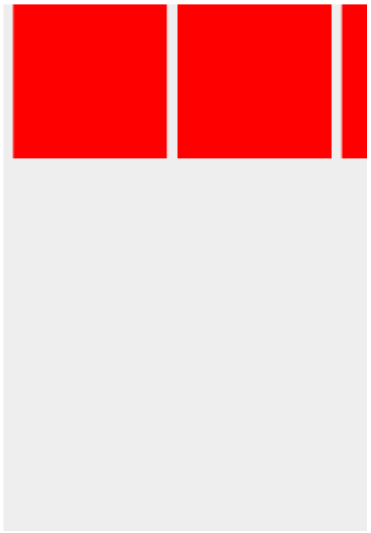


Figure 5.18 flexWrap not defined on device

We have declared five boxes in our layout, but as you can see only two of them are shown with the third overflowing off of the screen. Now, let's add the flexWrap: 'wrap' property to the container:

Listing 5.22 flexWrap property flexWrap: 'wrap' in code.

```
container: {
...
  flexWrap: 'wrap'
}
```

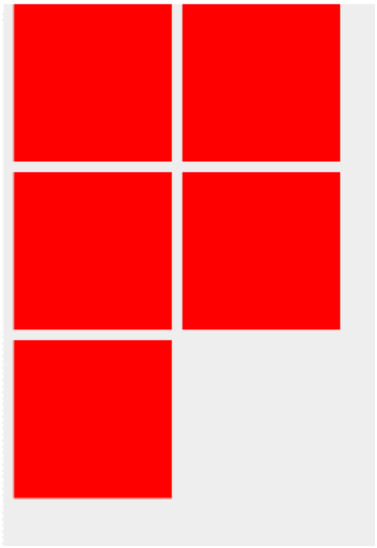


Figure 5.19 flexWrap: 'wrap' on device.

With flexWrap: 'wrap' declared, we see that any items that would be rendered off of the screen instead now wrap nicely into our view.

5.2 Dynamic Styles

Now that we have covered most of the styles available to us in a React Native application, let's talk about how to use them dynamically. Dynamic styling is a very powerful way for us to manipulate styling based on variables and conditions in our app. There are many ways to implement dynamic styling, but we will go over the ways that myself and the community have found to be valuable and best practice.

5.2.1 Dynamic Styles Using State and Props

Let's take a look at a few different ways to manipulate styles based on a prop value:

Listing 5.23 Dynamic styling with props Dynamic styling with boolean prop value in code

```
class App extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Box border />
        <Box />
      </View>
    )
  }
```

```
}

const Box = ({border}) => (
  <View style={[styles.flexBox1, border && styles.border ]} />
)

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flexDirection: 'row'
  },
  flexBox1: {
    width: 150,
    height: 150,
    margin: 10,
    backgroundColor: 'red'
  },
  border: {
    borderWidth: 5
  }
})
```

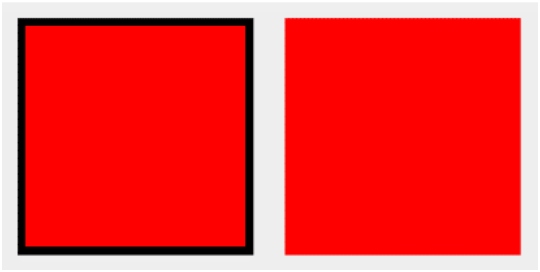


Figure 5.20 Dynamic styling with boolean prop value on device

What we've done is passed down a boolean value of border to the Box component. We check to see if the border prop is passed, if it is then we apply the border style. Any type of logic can be used here, and we will look at a few other ways of going about doing this. How you use dynamic styles in your components will entirely depend on circumstance and what your component is trying to do.

Another common styling use case is passing down color properties as props. Let's take a glance at how that would work:

Listing 5.24 Dynamic colors Dynamic color properties

```
<BoxbackgroundColor='yellow' />

const Box = ({backgroundColor}) => (
  <View style={[ styles.flexBox1, backgroundColor &&{backgroundColor} ]} />
)
```

We pass down a backgroundColor of yellow to the Box component. If it is defined, we set the backgroundColor to the backgroundColor that was passed using an es6 shorthand property name.

Working with dynamic styles based on state values is very similar in concept to working with dynamic styles based on prop value. Let's take a look at an example of how this would work:

Listing 5.25 Dynamic styling based on state Dynamic styling with state value

```
class App extends Component {
  constructor () {
    super()
    this.state = {loaded: false}
  }
  componentDidMount () {
    setTimeout(() => {
      this.setState({
        loaded: true
      })
    }, 1000)
  }
  render () {
    const { loaded } = this.state
    return (
      <View style={styles.container}>
        <Box backgroundColor={loaded && 'yellow'} />
      </View>
    )
  }
}

const Box = ({backgroundColor}) => (
  <View style={[ styles.flexBox1, backgroundColor && {backgroundColor} ]} />)

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flexDirection: 'row'
  },
  flexBox1: {
    width: 150,
    height: 150,
    margin: 10,
    backgroundColor: 'red'
  }
})
```

We set an initial state with a value of loaded: false. In componentDidMount, we simulate an api call with a setTimeout and update the loading boolean to true. In our Box component, we check the value of this.state.loaded and update the backgroundColor to yellow when it is true.

5.2.2 Dynamic Styles Using Functions and Class Methods

Sometimes we will need to calculate something and render a style based on the output. This can be implemented either with class methods or regular functions in our component. Here, we will check out a few ways to implement this functionality.

Say for example we have an array of colors and would like to apply a random color as a style property. We could possibly set up a helper method that would return a random color and them use that as a style in our component. We will set this up outside of our component since we do not need access to the state or props:

Listing 5.26 Dynamic with helper function Dynamic styling with helper function

```
const colors = ['red', 'blue', 'yellow', 'green']
function getRandomNum () {
  const index = Math.floor(Math.random() * (4))
  console.log('index:', index)
  return colors[index]
}
class App extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Box backgroundColor={getRandomNum()} />
        <Box backgroundColor={getRandomNum()} />
        <Box backgroundColor={getRandomNum()} />
        <Box backgroundColor={getRandomNum()} />
      </View>
    )
  }
}

const Box = ({backgroundColor}) => (
  <View style={[ styles.flexBox1, backgroundColor && {backgroundColor} ]} />
)

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flexDirection: 'row',
    flexWrap: 'wrap'
  },
  flexBox1: {
    width: 150,
    height: 150,
    margin: 10
  }
})
```

Here we are calling getRandomNum() from within our component and returning a random color from our array.

Another common scenario is styling components based on their index in a mapping function. We do this to see whether the item is even or odd, and apply styling accordingly. This is often done in list views where a lot of rows are rendered and the rows need to stand out from one another:

Listing 5.27 Dynamic with helper function Dynamic styling of rows based in index

```
let people = ['Jennifer', 'Chris', 'Emily', 'Becky', 'Mark']

class App extends Component {
  render () {
    people = people.map((p, i) => {
      return (
        View key={i} backgroundColor={i % 2 === 0 &&'yellow'}>
          <Text>{p}</Text>
        </View>
      )
    })
    return (
<View style={styles.container}>
      {people}
</View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flexWrap: 'wrap'
  }
})
```

Figure 5.21

Here we are simply mapping through all of the items in our people array, and setting the backgroundColor to yellow based on whether the index is even or odd.

5.3 Organizing Styles

There are a few main ways that styles can be organized within a react-native, with no exact declared convention or best practice. How this is done entirely depends on you and or your team's preference. That being said, let's take a look at a few options we have at our disposal.

5.3.1 Declaring styles in component

As we have done so far in this book, a very popular way to declare styles is within the component that will be using them. The major benefit of this is that the styles of the component then becomes entirely encapsulated. This component can be moved or used anywhere in the app and it does not have to worry about it's styling changing based on something else in the app changing. One of the drawbacks of this could be if you have a style that is being used elsewhere you will be writing the same code in multiple places. If a style changes in your app, you may have to go to every component implementing that style and update it to match the new styling convention being adopted.

5.3.2 Creating reusable stylesheets

If you are used to writing css, this may seem like a better approach and also feel more familiar. To do this, simply create a new file called styles.js in which to place the new stylesheet:

Listing 5.28 Creating reusable stylesheets – styles.js Creating external reusable stylesheet

```
import { StyleSheet } from 'react-native'

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    marginLeft: 20,
    flexWrap: 'wrap'
  }
})

export default styles
```

Then, we can import and reuse these styles whenever necessary as we would if they were declared within our component:

Listing 5.29 Importing reusable stylesheet Creating external reusable stylesheet

```
Import styles from './styles'

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

Another technique to keep in mind is that we can also create multiple stylesheets within the same file. This may come in handy if we are also wanting to create a separate group of ui elements that we may want to have access to without having to import the entire stylesheet:

Listing 5.30 Exporting multiple stylesheets from single file

```
import { StyleSheet } from 'react-native'

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    flexWrap: 'wrap'
  }
})

const buttons = StyleSheet.create({
  primary: {
    flex: 1,
    height: 70,
    backgroundColor: 'red',
    justifyContent: 'center',
    alignItems: 'center',
    marginLeft: 20,
    marginRight: 20
  }
})

export { styles, buttons }
```

We could then import and use these styles like so:

Listing 5.31 Importing multiple stylesheets from single file

```
import { styles, buttons } from './app/styles'

<View style={styles.container}>
  <TouchableHighlight style={buttons.primary} />
    ...
  </TouchableHighlight>
</View>
```

6

Building a Star Wars app using cross-platform components

This chapter covers

- The basics of fetching data using the fetch API
- Using a Modal component to show and hide views
- Creating a list using the FlatList component
- Using the ActivityIndicator to show loading state
- Using the Picker component to create a data filter
- Using react-navigation in a real-world project to handle navigation

React Native ships with many components that are ready for you to use in your app. Some of these components work cross-platform; that is, they work regardless of whether you are running your app on iOS or Android. Other components are platform specific; for example, ActionSheetIOS only runs on iOS, and ToolbarAndroid only runs on the Android platform.

In this chapter, we will be covering some of the most used cross-platform components and how to implement each one as we build a demo application.

The main difference between a cross-platform component and one that is not, is that a cross-platform component will work on both iOS and Android, while a platform specific component will not.

An example of a *non*-cross--platform component would be something like ToolbarAndroid, which only runs on Android, or ActionSheetIOS which would only be for iOS.

This code for this chapter is located at:
https://github.com/dabit3/react-native-in-action/tree/chapter6/StarWars

In this section, we will be implementing these cross-platform components by building a cross-platform Star Wars information app.

This app will access the SWAPI (https://swapi.co/) and return information about all the Star Wars characters, starships, home planets, and more (listing 6.1)!

Figure 6.1 The completed Star Wars app that we will be building.

In this tutorial, we'll focus on the first link, People. When a user clicks People, the app fetches the main cast from https://swapi.co/api/people/ and displays their information. In the process, the app uses several React Native cross-platform components. In this chapter, you'll learn how to use these components as you

1. Set up a new React Native application and install dependencies.
2. Import the People component and create the Container component.
3. Create the Navigation component and register routes.
4. Create the main class for the view.
5. Create the People component and use the cross-platform components FlatList, Modal, and Picker to
6. Create the state and set up a fetch call to retrieve data.

CODE LOCATION You can find this chapter's code at https://github.com/dabit3/react-native-in-action/tree/chapter6/StarWars

6.1 Creating a new React Native app and install dependencies

The first thing we need to do is set up a new React Native application and install any dependencies we will need to build this app.

First, go to the command line and create a new React Native app by typing in the following:

```
react-native init StarWarsApp
```

The only thing we will be needing to kick this off is react-navigation, so let's go ahead and install it using either npm or yarn:

- **using npm**: npm i react-navigation
- **using yarn**: yarn add react-navigation

Now that the project is created, let's open index.ios.js (if developing for iOS) or index.android.js (if developing for Android) and create the components we will need for the first screen (figure 6.2).

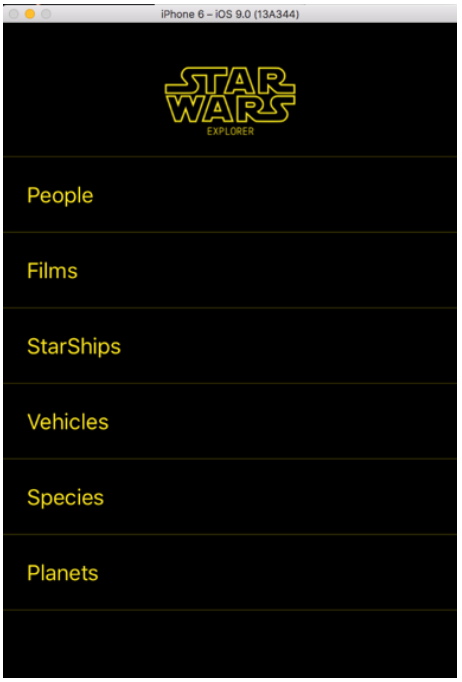


Figure 6.2 The initial view of the app

At the top of the file, let's import the following components (listing 6.1).

Listing 6.1 Importing initial component for application – index.ios.js / index.android.js

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  Image,
  FlatList,
  TouchableHighlight
} from 'react-native';
import { StackNavigator } from 'react-navigation';
```

In listing 6.1, we import the React Native components we need for this file, as well as `StackNavigator` from `react-navigation`. `StackNavigator` is a navigator from react-navigation that provides an easy way to navigate between scenes, and each scene is pushed on top of a route stack. All the animations are configured for you and give the default iOS and Android feel / transitions.

6.2 Importing the People component and creating the Container component

Next, we need to import the two views we will be using in this file. Let's take another look at the first screen (figure 6.2). As you can see, we have People, Films, and so on as links. We need to make it so that when the user clicks on People, we navigate to a component that lists the people (main actors / actresses) in the Star Wars films.

To do so, we will need to create a People component. We will do so in section 6.5, but for now let's go ahead and import this component and then later we will create it. Below our last import, let's import the yet to be created People component:

```
import People from './People'
```

Because the design uses a black color background and we do not want to be repeating styling code across components, let's create a Container component that we will use as a wrapper for our views. This component will be strictly used for styling. In the root of the app create Container.js, and in it write the following code (listing 6.2).

Listing 6.2 Creating a reusable Container component – Container.js

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

let styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'black',
  },
})
```

```
const Container = ({ children }) => ( ❶
  <View style={styles.container}>
    {children}                        ❷
  </View>
)

export default Container
```

❶ Container takes a single property, children, which will in our case be the component that is wrapped in the Container

❷ We wrap the children component in a View and give it a style of container, which will give the component a basic black background and a flex property of 1.

Now that Container is created, let's go ahead and import it into our index.ios / index.android.js file, and do so below the last import of the People component.

```
import Container from './Container'
```

Below that import, we need to go ahead and create an array of data that we will be using for the links. This array should contain objects, and each object should contain a title key. We need the title key for displaying the name of the link.

```
const links = [
  { title: 'People' },
  { title: 'Films' },
  { title: 'StarShips' },
  { title: 'Vehicles' },
  { title: 'Species' },
  { title: 'Planets' }
]
```

Now at the bottom of the file, let's create our main Navigation component and pass it to the `AppRegistry`.

We are StackNavigator as our navigation component, and we now need to register the routes we will be using in our application.

At the bottom of the file, let's create the StackNavigator and the pass the navigator to the `AppRegistry` method, replacing the default StarWars component with the Navigation component (listing 6.3).

Listing 6.3 Creating Navigation component and registering component

```
const Navigation = StackNavigator({
  Home: {
    screen: blankRNApp,
  },
  People: {
    screen: People
  }
})
AppRegistry.registerComponent('StarWars', () => Navigation)
```

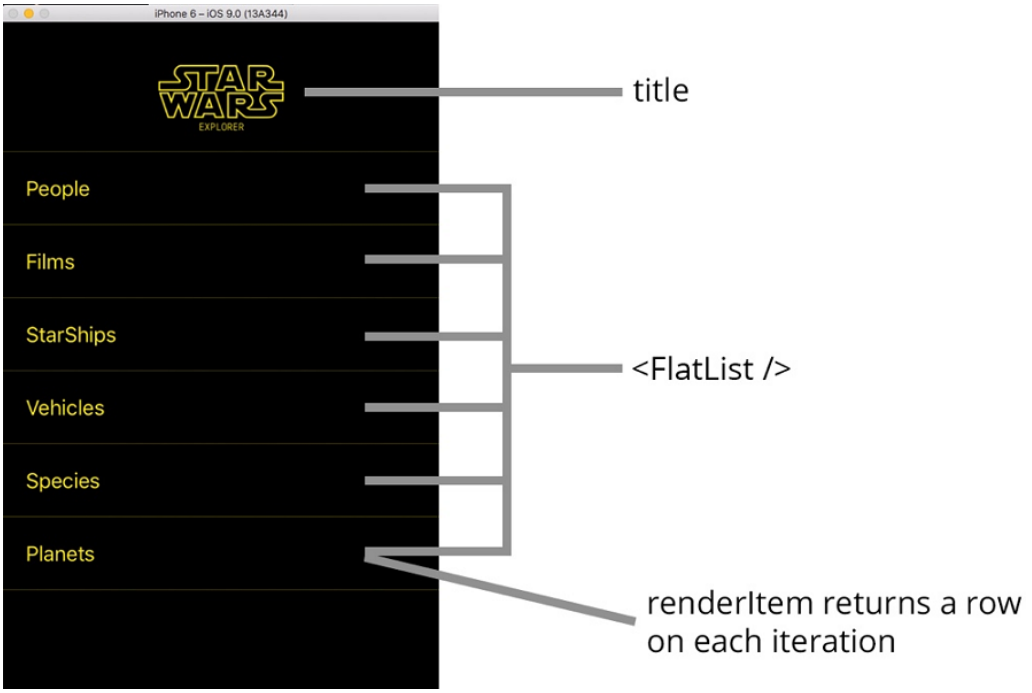


Figure 6.3 Components and title descriptions for index file

Next, below the links array, let's go ahead and create the main class for the view in listing 6.4.

This class will return a list that will render all the actors and actresses that come back from the API.

We will render this list using the `FlatList` component from React Native.

FlatList is a built-in interface for rendering simple lists in a React Native app.

Listing 6.4 Creating the main StarWars component – index.ios.js / index.android.js

```
class StarWars extends Component {

  static navigationOptions = {          ❶
    header: {
      title: <Image style={{ width: 110, height: 64 }} source={{ uri:
      'https://raw.githubusercontent.com/dabit3/react-native-in-
      action/chapter6/sw.jpg' }} />,
      style: { backgroundColor: 'black', height: 110 }
    }
  }

  navigate = (link) => {                ❷
    const { navigate } = this.props.navigation
    navigate(link)
```

```
  }

  renderItem = ({ item, index }) => { ❸
    return (
      <TouchableHighlight
        onPress={() => this.navigate(item.title)}
        style={[ styles.item, { borderTopWidth: index === 0 ? 1 : null} ]}>
        <Text style={styles.text}>{item.title}</Text>
      </TouchableHighlight>
    )
  }

  render() {
    return (
      <Container> ❹
        <FlatList ❺
          data={links}
          keyExtractor={(item) => item.title}
          renderItem={this.renderItem}
        />
      </Container>
    )
  }
}

const styles = StyleSheet.create({
  item: {
    padding: 20,
    justifyContent: 'center',
    borderColor: 'rgba(255,232,31, .2)',
    borderBottomWidth: 1
  },
  text: {
    color: '#ffe81f',
    fontSize: 18
  }
});
```

❶ Because we are using StackNavigation from react-navigation, we have the option to pass in some configuration for each route that we will be working with. In this route, we want to change the default header configuration and styling to suit our design. To do so, we pass in a header object with a title, and a style. The title is the image we will be using as our logo, and the styling sets the background color to black, and gives a set height to accommodate the image.

❷ We create a navigate method that will take in a link as an argument. Any component that is rendered by StackNavigation automatically receives the navigation object as a prop. We use this navigation prop to destructure the navigate method, and then call navigate to the link that is passed in. The link, in our case the title property in the links array, will correlate directly with the keys passed into the StackNavigator.

❸ FlatList takes a renderItem method. RenderItem will loop through the array of data that is passed in as the data property, and will return an object with an item and an index for each item in the array. The item will be the actual item with all its properties, and the index is the index of the item. We destrucuture these as arguments and use the item to pass as an argument to navigate as well as to display the title, and use index to apply a borderTop style if it is the first item in the array.

❹ The render method returns the Container, and in it we wrap the FlatList component, passing in links as our data, and the renderItem method we created above to renderItem. We also pass in a keyExtractor method. If there is no item labeled key in our array, we specifically have to tell the FlatList which item to use as its key or else it will throw an error.

❺ For the final code for this file, check out https://github.com/dabit3/react-native-in-action/blob/chapter6/StarWars/index.ios.js

This is all we need to do for this file.

6.5 Creating the People component using FlatList, Modal, and Picker

The next thing we need to do is create a file called People.js. We will use this People component to fetch and display information about the Star Wars cast that we get from the Star Wars API.

Before we build this page out, let's take another look at it (figure 6.3).

Figure 6.4 This component will display the Loading (left) and Loaded (middle) state of People.js screen. It will also allow us to view the actor / actress's homeworld information (right).

The People component will fetch and display information about the Star Wars cast that we get from the Star Wars API. As part of this component, we'll use the React Native cross-platform components Modal and Picker.

Modal is a component that will allow us to display an element on top of whatever view we are currently working in.

The Picker is a component that will display a scrollable list of options or values, and when a value is selected we can use the value within our class.

When this component loads, it will start off with a loading state of true, along with a few other pieces of state:

```
state = {
    data: [],
    loading: true,
    modalVisible: false,
    gender: 'all',
    pickerVisible: false
  }
```

We start off with an empty data array, a loading Boolean of true, a `modalVisible` Boolean of true, a gender (string) of all, and a `pickerVisible` Boolean of false.

When the component mounts, we will fetch the data we need from the Star Wars API at https://swapi.co/api/people/, and when this data returns, we will populate the data array with the returned data, and set the loading Boolean to false.

We will use the `modalVisble` Boolean to show and hide the Modal that will fetch information about the home world of the person.

We will use `pickerVisible` to show and hide a Picker component. this Picker component will allow us to choose between which gender of person we want to view, and will pass the result to a filter that will filter the results for us based on this gender.

Let's go ahead and start by creating this new file, People.js, and begin coding (listing 6.5).

Listing 6.5 People.js – imports – People.js

```
import React, { Component } from 'react'
import {                                ❶
  StyleSheet,
  Text,
  View,
  Image,
  TouchableHighlight,
  ActivityIndicator,
  FlatList,
  Modal,
  Picker
} from 'react-native'
import _ from 'lodash'

import Container from './Container'     ❷
import HomeWorld from './Homeworld'     ❸
```

❶ We import the components from React Native that we will need for this component. Notable components that we have not used before are FlatList, Modal, and Picker

❷ We import the Container component that we used in index.ios.js / index.android.js, as we will need the same styling here

❸ We import the HomeWorld component that we have yet to create. HomeWorld will populate with the actor's / actress's HomeWorld information when we click on View Homeworld for that actor / actress.

Now that we have the imports set up, the next step is to create the main Class for the component, and set up the `navigationOptions` to give the header a title as well as some styling.

Under the last import, create a People class (listing 6.6).

Listing 6.6 Creating the People class and setting up the page title – People.js

```
export default class People extends Component {
  static navigationOptions = { (1)
    header: {
      title: 'People',
      style: {
        borderBottomWidth: 1,
        borderBottomColor: '#ffe81f',
        backgroundColor: 'black'
      },
      titleStyle: {
        color: '#ffe81f',
      },
    }
  }
}
```

(1) **The static `navigationOptions` property is created here, like in our index file, but instead of passing in a component, we pass in a string of People as the title. We also add some styling.**

Next, we'll create the state and set up a fetch call on componentDidMount.

`fetch` is an API that is a newer interface for fetching network resources that is still not 100% compatible with all internet browsers, but React Native provides a polyfill and it is an easy to use out of the box way to work with network requests, including GET, POST, PUT, and DELETE. `fetch` returns a promise and makes it easy to work with in an asynchronous manner.

The fetch call will hit the Star Wars API at https://swapi.co/api/people/ which will return an object containing a results array. This results array contains the actors / actresses we will want to display on this page. If you would like to view this dataset, all you need to do is open the URL in a browser to check out the data structure.

The data set looks like the following, with results being the array of actors / actresses we are interested in using:

```
{
  "count": 87,
  "next": "http://swapi.co/api/people/?page=2",
  "previous": null,
  "results": [
    {   "name": "Luke Skywalker",
        "height": "172",
        "mass": "77",
        ...
    },
    ...
}
```

Once the data is returned from the API, we reset the data array in the state with the results array that comes back from the API.

Below the `navigationOptions` object, let's create the state and the `componentDidMount` fetch call (listing 6.7).

Listing 6.7 Setting up the initial state and fetching data – People.js

```
state = {
    data: [],
    loading: true,
    modalVisible: false,
    gender: 'all',
    pickerVisible: false
  }
  componentDidMount() { (1)
    fetch('https://swapi.co/api/people/')
      .then(res => res.json())
      .then(json => this.setState({ data: json.results, loading: false }))
      .catch((err) => console.log('err:', err))
  }
```

(1) **In componentDidMount, we fetch the data from the API, then use `the .json()` method to read the response to completion. `.json()` returns a promise containing the `JSON data`. We then set the state again, updating the data and loading variables.**

At this point in the app, if we load this page, we should have the data loaded into the state and ready for us to use. Next up, we need to create the rest of the functionality that we will be using to display this data, as well as a render method that will display the data. The next step for us will be to create the rest of the methods that we will be using in this component (listing 6.8).

Listing 6.8 Remaining methods for component functionality – People.js

```
renderItem = ({ item }) => {  (1)
  return (
    <View style={styles.itemContainer}>
      <Text style={styles.name}>{item.name}</Text>
      <Text style={styles.info}>Height: {item.height}</Text>
      <Text style={styles.info}>Birth Year: {item.birth_year}</Text>
      <Text style={styles.info}>Gender: {item.gender}</Text>
      <TouchableHighlight
        style={styles.button}
        onPress={() => this.openHomeWorld(item.homeworld)}
      >
        <Text style={styles.info}>View Homeworld</Text>
      </TouchableHighlight>
    </View>
  )
}

openHomeWorld = (url) => {    (2)
  this.setState({
```

```
    url,
    modalVisible: true
  })
}

closeModal = () => {    ❸
  this.setState({ modalVisible: false })
}

togglePicker = () => { ❹
  this.setState({ pickerVisible: !this.state.pickerVisible })
}

filter = (gender) => { ❺
  this.setState({ gender })
}
```

❶ The `renderItem` method is what we will pass to FlatList to render the data in our state. Every time an item is passed through this method, we get an object with two keys: item and kay. We destructure the item when the method is called, and use the item properties to display the data for the user (i.e. item.name, item.height, etc...). Note the `onPress` method passed to the `TouchableHighlight` component. This method will pass the item.homeWorld property to the `openHomeWorld` method. Item.HomeWorld is a URL that we will use to fetch the actor / actress's homeworld information.

❷ `openHomeWorld` is the method that will update the `url` and the `modalVisible` Boolean in the state. When this is called, the HomeWorld modal will open.

❸ `closeModal` will close the modal by setting the `modalVisible` Boolean in the state to `false`.

❹ `togglePicker` will toggle the `pickerVisible` Boolean. We will use this Boolean to show and hide a picker that will allow us to choose a filter. The filters will filter the actors / actresses by gender, and show either all, female, male, or other (robots / etc.)

❺ `filter` will update the filter value in the state to the passed in value. This value will be the value that we will use to filter our data in the render method.

Now that we have all the methods set up, the last thing we need to do is implement the UI in the render method.

After the filter method, let's implement the render method (listing 6.9).

Listing 6.9 render method of People.js

```
render() {
  let { data } = this.state           ❶
  if (this.state.gender !== 'all') {  ❷
    data = data.filter(f => f.gender === this.state.gender)
  }

  return (
    <Container>
      <TouchableHighlight style={styles.pickerToggleContainer}
      onPress={this.togglePicker}>    ❸
        <Text style={styles.pickerToggle}>{this.state.pickerVisible ? 'Close Filter'
      : 'Open Filter'}</Text>
      </TouchableHighlight>
      {                               ❹
        this.state.loading ? <ActivityIndicator color='#ffe81f' /> : (
          <FlatList
            data={data}
```

```
            keyExtractor={(item) => item.name}
            renderItem={this.renderItem}
          />
        )
      }
      <Modal                                    ❺
        onRequestClose={() => console.log('onrequest close called')}
        animationType="slide"
        visible={this.state.modalVisible}>
        <HomeWorld closeModal={this.closeModal} url={this.state.url} />
      </Modal>
      {
        this.state.pickerVisible && (  ❻
          <View style={styles.pickerContainer}>
            <Picker                             ❼
              style={{ backgroundColor: '#ffe81f' }}
              selectedValue={this.state.gender}
              onValueChange={(item) => this.filter(item)}>

              <Picker.Item itemStyle={{ color: 'yellow' }} label="All" value="all" />
              <Picker.Item label="Males" value="male" />
              <Picker.Item label="Females" value="female" />
              <Picker.Item label="Other" value="n/a" />
            </Picker>
          </View>
        )
      }
    </Container>
  );
}
```

❶ We destructure the data array from the state so we can access this easier

❷ This is where the filtering happens. We check to see if the filter is set to all, and if it is not we perform a filter based on the gender of the actor / actress vs the gender set in the state. If it is set to all, we skip this function.

❸ We create a button here that will show "Close Filter" or "Open Filter" based on the value of `this.state.pickerVisible`. When this button is clicked, the `togglePicker` method is called and the Picker is shown or hidden.

❹ We check to see if the data is loading by evaluating `this.state.loading`. If it is loading, we show an `ActivityIndicator` which will indicate that the loading is taking place. If it is not, then we will render the FlatList component, passing in the data as the data source, `this.renderItem` as the `renderItem` function, and a `keyExtractor` method.

❺ The Modal component will stay hidden until the `modalVisible` value is set to true. When it is set to true, then the Modal will slide up into view. We pass a few props to this component:
5a) `animationType="slide"` indicates the animation type of the modal. This could also be none, or fade
5b) `visible={this.state.modalVisible}` tells the modal whether to show
5c) `onRequestClose` is a required property. Since we do not need to do anything here, we are just logging to the console when this is called.

❻ Here, we check to see if the value if `this.state.pickerVisible` is set to `true`, and if it is, we render the `Picker` component.

❼ We render the `Picker`, passing in a value, a style, and an `onValueChange` method. `onValueChange` will fire every time the picker value is updated, which will then update the state, triggering a rerender of the component, and then updating the filtered list of items in the view.

The last thing we need is the styling for this component (listing 6.10). This goes below the class definition.

Listing 6.10 Styling for People.js

```
const styles = StyleSheet.create({
  pickerToggleContainer: {
    padding: 25,
    justifyContent: 'center',
    alignItems: 'center'
  },
  pickerToggle: {
    color: '#ffe81f'
  },
  pickerContainer: {
    position: 'absolute',
    bottom: 0,
    right: 0,
    left: 0
  },
  itemContainer: {
    padding: 15,
    borderBottomWidth: 1, borderBottomColor: '#ffe81f'
  },
  name: {
    color: '#ffe81f',
    fontSize: 18
  },
  info: {
    color: '#ffe81f',
    fontSize: 14,
    marginTop: 5
  }
});
```

// **The final code for this component can be found at https://github.com/dabit3/react-native-in-action/blob/chapter6/StarWars/People.js**

The next, and last, component we will need to create to finish our app is going to be the HomeWorld component. In People.js, we created a Modal, and in the modal we used this HomeWorld component as the content of the modal.

```
<Modal
  onRequestClose={() => console.log('onrequest close called')}
  animationType="slide"
  visible={this.state.modalVisible}>
  <HomeWorld closeModal={this.closeModal} url={this.state.url} />
</Modal>
```

We will use the HomeWorld component to fetch data about the actor / actress's home planet and display this information to our user in the modal (figure 6.4).

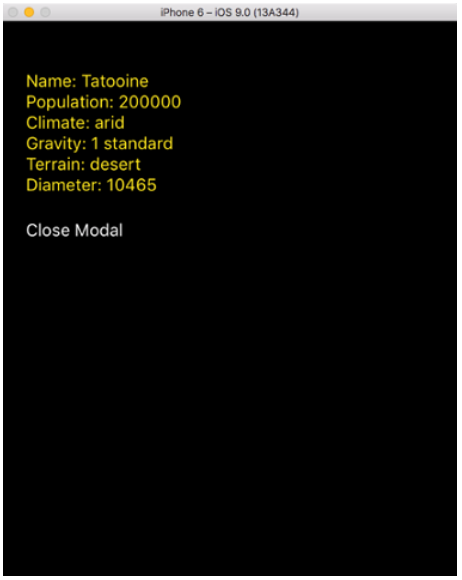


Figure 6.5 HomeWorld component displaying data after fetching from the API.

This component will fetch the url prop that was passed in when the modal opens in a fetch call placed in `componentDidMount`. This happens because `componentDidMount` is called every time the visible property of the modal is set to true, as it is basically loading the component when the modal is shown.

To get started, let's import the components we will need, create the class definition, and create our initial state (listing 6.11).

Listing 6.11 HomeWorld component class, imports, and initial state – HomeWorld.js

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

export default class HomeWorld extends React.Component {
  state = { ❶
    data: {},
    loading: true
  }
}
```

❶ Our initial state will hold only two things: an empty data object and a loading Boolean set to true. When the component loads, we will show a loading indicator while we wait for the data to come back from the API. Once the data loads, we will update the loading Boolean to false and render the data that came back from the API.

The next thing we need to do is call the API using the url property. We will do this in `componentDidMount`, which will be called once the component loads. Below the state declaration, create the following `componentDidMount` method. (listing 6.12).

Listing 6.12 Fetching data and uploading state in componentDidMount – HomeWorld.js

```
componentDidMount() {
  if (!this.props.url) return                                    ❶
  const url = this.props.url.replace(/^http:\/\//i, 'https://') ❷
  fetch(url)                                                     ❸
    .then(res => res.json())
    .then(json => {
      this.setState({ data: json, loading: false })
    })
    .catch((err) => console.log('err:', err))
}
```

❶ The first thing we do is to check to make sure there is a url. If not, then we return out of the function as to not cause an error

❷ The next thing we need to do is to update the API url to use https. We do this because React Native does not allow unsecure http requests out of the box, though it can be configured to work if necessary.

❸ Here we call fetch on the url that was passed in as a prop. When the response comes back, we transform the data into JSON and then update the state to set loading to false, as well as add the data to the state by updating the data value of state with the returned JSON.

The last things we need to do are to create the render method and our styling.

In our render method, we will be displaying some properties relating to the homeworld such as the name, population, climate, etc. These styles will be repetitive. In React and React Native, it is best to create a component and reuse the component vs creating styling and reusing the styling if it is something we will be doing more than a handful of times.

In this case, it will make sense to create our own custom Text Container component to use in the render method to display our data. Let's create this component. Above the class declaration, create the following component called TextContainer (listing 6.13).

Listing 6.13 Creating a reusable TextContainer component – HomeWorld.js

```
const TextContainer = ({ label, info }) => (
  <Text style={styles.text}>{label}: {info}</Text>
)
```

// In this component, we return a basic Text component, and receive two props that we will use: label and info. Label will be the description of the field and will be static, while info will be the information we get when the API returns the homeworld data. We will use it like so:

```
<TextContainer label="Name" info={this.props.data.name} />
```

Now that we have our `TextContainer` ready to go, let's finish the component by creating the render method and the styling (listing 6.14).

Listing 6.14 render and styling – HomeWorld.js

```
export default class HomeWorld extends React.Component {
  ...
  render() {
    const { data } = this.state  (1)
    return (
      <View style={styles.container}>
        {
          this.state.loading ? ( (2)
            <ActivityIndicator color='#ffe81f' />
          ) : (
              <View style={styles.HomeworldInfoContainer}> (3)
                <TextContainer label="Name" info={data.name} />
                <TextContainer label="Population" info={data.population} />
                <TextContainer label="Climate" info={data.climate} />
                <TextContainer label="Gravity" info={data.gravity} />
                <TextContainer label="Terrain" info={data.terrain} />
                <TextContainer label="Diameter" info={data.diameter} />
                <Text             (3)
                  style={styles.closeButton}
                  onPress={this.props.closeModal}>
                  Close Modal
              </Text>
              </View>
            )
        }
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#000000',
    paddingTop: 20
  },
  HomeworldInfoContainer: {
    padding: 20
  },
  text: {
    color: '#ffe81f',
  },
  closeButton: {
    paddingTop: 20,
    color: 'white',
    fontSize: 14
  }
})
```

(1) **We destructure the `data` object from the state**
(2) **We check to see if `loading` is true. If so, we show an `ActivityIndicator` to indicate loading**
(3) **If loading is not true, we return the main View component that wraps our `TextContainers` and displays the data that was returned from the API, now stored as the `data` object in our state.**

4 **We create a button that will `call this.props.closeModal` to give the user the ability to close the modal.**

7

Navigation

This chapter covers

- React Native Navigation Overview
- Using NavigatorIOS
- Using Navigator
- Using Navigation Experimental

Navigation in React Native is a very important concept to understand when building your application.

There are many options out there for creating and managing navigation state for a React Native application, but only three come out of the box, so we will be covering those. They are NavigatorIOS, Navigator, and Navigation Experimental.

Navigation state is usually held in an array and routes are pushed and popped to the array, rendering the current chosen item in the array. We navigate through our routes by pushing and popping to the array, rendering the chosen route (listing 7.1).

Listing 7.1 Route stack example (pseudo code)

```
import Home from './Home'
import About from './About'
import Contact from 'Contact'

const routes = [{ component: Home, title: 'Home' }, { component: About, title:
      'About' }]

// push to route
routes.push({
  component: Contact,
  title: 'Contact'
})
```

```
// go back one route
routes.pop()
```

This is a very basic implementation of how a route array works, and what it looks like to push and pop routes to the route array. This concept will be applicable to all the navigation options we look at in this chapter, and also carry on to most navigation options in general when dealing with React Native.

7.1 NavigatorIOS

The first navigation option we will be covering in this chapter is NavigatorIOS. This navigator is great for beginners, prototypes, or applications that do not require a complex navigation state, and applications that only need to run on the iOS platform. To be clear, this navigator will not work on Android.

The main benefits of NavigatorIOS are its simple API and smooth transitions. While many of the other options available for navigation use JavaScript for animating the transitions between routes, NavigatorIOS uses real native transitions, giving it a slightly smoother feel and truly native performance. It is also very easy to get up and running.

In the context of NavigatorIOS, a route is an object with at least one property: a component.

```
{component: MyComponent}
```

For our basic example, we will set up a navigator that pushes and pops a couple of routes. We will specify a `ref`, `tintColor`, `titleTextColor`, and `initialRoute` on the original instance of the NavigatorIOS, and have three components that we will navigate between (Home, About, and Contact).

To set up an instance of NavigatorIOS, there are really only two things that must be given as props: a style of `flex:1`, to fill the view with the navigator, and an `initialRoute` object with a title and component property.

```
class App extends Component {
  render () {
    return (
<NavigatorIOS
        style={{ flex: 1 }}
        initialRoute={{ title: 'Home', component: Home }}
      />
    )
  }
}
```

With that in mind, let's code our first application using NavigatorIOS. Our application will consist of three routes: Home, About, and Contact. The initial route will be a component we will call Home (listing 7.2, figure 7.1).

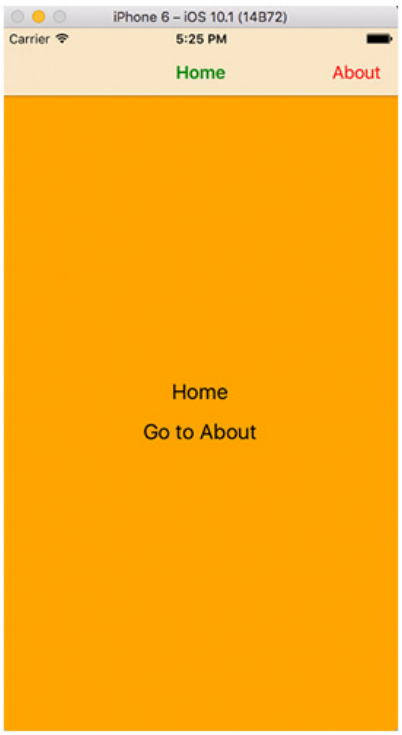


Figure 7.1 NavigatorIOS with three routes

Listing 7.2 NavigatorIOS

```
import React, { Component }  from 'react'
import { AppRegistry, NavigatorIOS, View, Text, StyleSheet } from 'react-native' ❶
let styles = {}

const Home = ({ navigator }) => ( ❷
<View style={styles.container}>
<Text style={styles.text}>Home</Text>
<Text
      style={styles.text}
      onPress={() => navigator.push({ ❸
          title: 'About',
          component: About,
          leftButtonTitle: 'Back',
          onLeftButtonPress: () => navigator.pop()
        })
      }
>Go to About</Text>
</View>
)

const About = ({ navigator }) => (
```

```
<View style={styles.container}>
<Text style={styles.text}>About</Text>
<Text
      style={styles.text}
      onPress={() => navigator.pop()} ❹
>Go Back</Text>
<Text
      style={styles.text}
      onPress={() => navigator.replace({
          component: Contact,
          onLeftButtonPress: () => navigator.pop() ❺
        })
      }
>Go to Contact</Text>
</View>
)

const Contact = ({ navigator }) => (
<View style={styles.container}>
<Text style={styles.text}>Contact</Text>
<Text style={styles.text} onPress={() => navigator.pop()}>Go Back</Text>
</View>
)

class App extends Component {
  render () {
    return (
<NavigatorIOS ❻
        ref='navigator'
        style={{ flex: 1 }}
        tintColor='red'
        titleTextColor='green'
        initialRoute={{ rightButtonTitle: 'About', onRightButtonPress: () =>
      this.refs.navigator.push({ component: About, title: 'About' }), title: 'Home',
      component: Home }}
      />
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 60,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'orange'
  },
  text: {
    fontSize: 20,
    padding: 7
  }
})
AppRegistry.registerComponent('App', () => App)
```

❶ **import** `NavigatorIOS` **from 'react-native'**

2. create a component called Home. Notice that we get access to navigator as props, this is because any component that gets rendered by NavigatorIOS anywhere in the routes stack automatically gets passed the navigator reference as a prop.
3. create an `onPress` method that calls `navigator.push`, passing in a `title`, `component`, `leftButtTitle`, and `onLeftButtonPress` method. If we do not pass in a `leftButtonTitle` or `onLeftButtonPress` method, the default that is rendered is a Back button that will call `navigator.pop()` when pressed.
4. in About, we have a back button that calls `navigator.pop()` to go back
5. in About, we also have a button that calls `navigator.replace`, replacing the current route with the Contact component. Using replace will not render any animations at all, instead it will just replace the current scene without any transitions or animations.
6. in the render method, we return the instance of NavigatorIOS, passing in all of the configuration we will need to wire everything together. Notice we also have a `ref= 'navigator'` which gives us access to the navigator methods within the initial component. We do this so we can give the `initailRoute` object access to the navigator push method in the `onRightButtonPress` function by `calling this.refs.navigator.push`.

NavigatorIOS has the props and methods shown in tables 7.1 and 7.2.

Table 7.1 NavigatorIOS props

prop	type	description(some from docs)
barTintColor	string	default navigation bar background color
initialRoute	object	initial route to be rendered
interactivePopGestureEnabled	Boolean	Boolean value that indicates whether the interactive pop gesture is enabled. This is useful for enabling/disabling the back-swipe navigation gesture.
itemWrapperStyle	object (style)	The default wrapper style for components in the navigator. A common use case is to set the backgroundColor for every scene.
style	object (style)	container style of the navigator
navigationBarHidden	Boolean	hides or shows navigation bar
shadowHidden	Boolean	Boolean value that indicates whether to hide the 1px hairline shadow by default.
tintColor	string	text and icon color for buttons in the navigation bar
titleTextColor	string	text color for the navigation bar title
translucent	Boolean	indicates whether the navigation bar is translucent by default, default is true

Table 7.2 NavigatorIOS methods

method	arguments	description (some from docs)
push	push({route})	Navigate to a new route
pop	none – pop()	go back one route
popN	popN(number)	go back specified number of scenes at once
replaceAtIndex	replaceAtIndex({route}, index)	replaces a route in the navigation stack.
replace	replace({route})	replaces the route for the current scene and immediately load the view for the new route.
replacePrevious	replacePrevious({route})	replace the route/view for the previous scene.
popToTop	none – popToTop()	go back to the topmost item in the navigation stack.
popToRoute	popToRoute({route})	go back to the item for a particular route object.
replacePreviousAndPop	replacePreviousAndPop({route})	replaces the previous route/view and transitions back to it.
resetTo	resetTo({route})	replaces the top item and pop to it.

7.2 Using Navigator to create cross platform navigation

As of this writing, Navigator is currently the only stable cross platform navigation solution that comes out of the box with React Native. Though it has been the recommendation for cross platform navigation up until now, keep in mind that Navigation Experimental will one day take the place of this navigator as the recommended way to implement navigation once it is stable, though there has yet to be a solid timeline laid out for this transition, so you are safe in learning and implementing both APIs as of the time of this writing.

Navigator is a good choice for many use cases. The implementation has been around since the release of the framework, so there is a lot of documentation and information available for how to implement different navigation state using the Navigator.

That being said, remember that Navigator will one day be replaced by Navigation Experimental for a few reasons that may be hard to understand coming into this as a beginner, including the fact that Navigator has an imperative API that does not jive with the single-directional data flow of the React philosophy and includes somewhat difficult to customize scene animations.

Navigator has a somewhat similar API to NavigatorIOS. The main difference starting off that you will notice is that we also need to add a `renderScene` method to the Navigator instance which describes how to render a scene for a given route, while NavigatorIOS did this rendering for us (listing 7.3).

Listing 7.3 Basic Navigator implementation (not full implementation, will not run)

```
import Home from './pathtohomecomponent/'

<Navigator
  initialRoute={{ name: 'Home', component: Home }}
  renderScene={(route, navigator) =><route.component />}
/>
```

One thing to remember is that any route properties, such as the `name` or the `component` values in listing 7.3, will be available as properties of the `route` argument of `renderScene`. That is why we can use `<route.component />` to render our scene, as we are returning the `Home` component as the `initialRoute` component value. The`initialRoute` object has both a component and name key, which we then access in renderScene as route.component.

Navigator also has a way to specify scene animations and custom header that calls methods, all of which we will cover shortly, but first let's set up an instance of Navigator so we can begin working with it.

To get started, we will need to create a component for our initial route and a component that will render the navigator (Listing 7.4, figure 7.2).

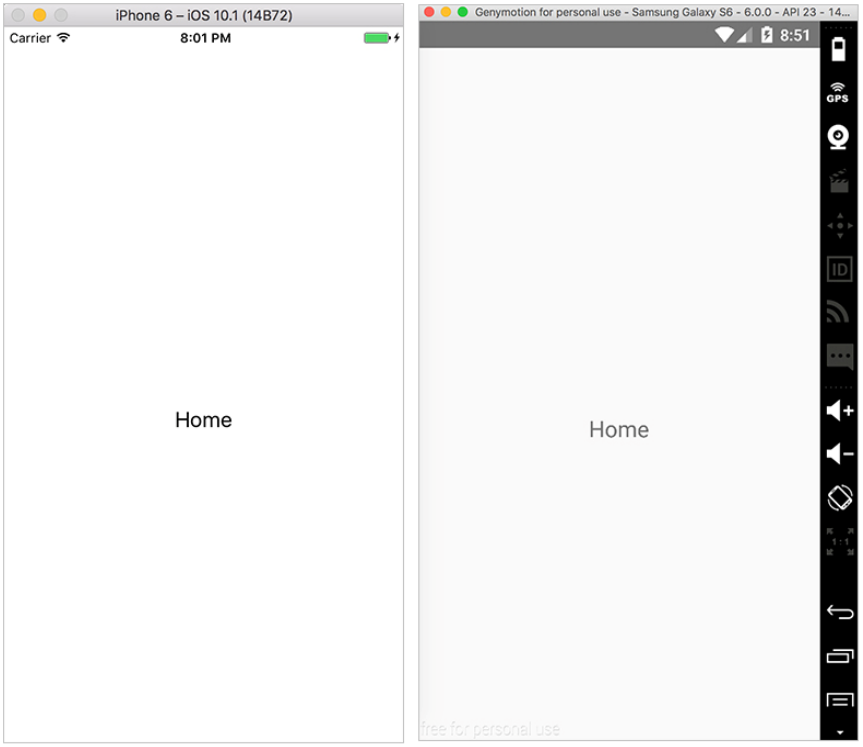


Figure 7.2 Navigator with initial route loaded

Listing 7.4 Navigator with a single route

```
import React, { Component }  from 'react'
import { Navigator, View, Text, StyleSheet } from 'react-native' (1)
let styles = {}

const Home = () => ( (2)
<View style={styles.container}>
<Text style={styles.text}>Home</Text>
</View>
)

class App extends Component {
  constructor() {
    super()
    this.renderScene = this.renderScene.bind(this)
  }
  renderScene(route, navigator) { (3)
    return <route.component navigator={navigator} />
  }
  render () {
    return (
<Navigator (4)
        ref='navigator'
        renderScene={this.renderScene}
        initialRoute={{ title: 'Home', component: Home }}
      />
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 60,
    justifyContent: 'center',
    alignItems: 'center'
  },
  text: {
    fontSize: 20,
    padding: 7
  }
})

export default App
```

(1) import Navigator along with other needed components from 'react-native'
(2) create a Home component to render as out initial route component
(3) create a renderScene method that returns the component property of the route, and passes the navigator reference as a prop to the rendered scene
(4) create Navigator instance, passing the initialRoute and the renderScene as props

Now, when we run our app, we should get the Home component rendered when the app starts (figure 7.2).

Now that we have our Navigator working, let's add a couple of routes that we can navigate between. In the same file, below the Home declaration, let's create two more components, Contact and About. We will also update the Home component to take the navigator props and link to the two new components (Listing 7.5).

Listing 7.5 Adding two routes to

```
const Home = ({ navigator }) => (     ❶
<View style={styles.container}>
<Text style={styles.text}>Home</Text>
<Text onPress={() => navigator.push({component: About, title: 'About'})}>About</Text>
<Text onPress={() => navigator.push({component: Contact, title:
      'Contact'})}>Contact</Text>
</View>
)

const About = ({ navigator }) => (    ❷
<View style={styles.container}>
<Text style={styles.text}>About</Text>
<Text onPress={() => navigator.pop()}>Back</Text>
</View>
)

const Contact = ({ navigator }) => ( ❷
<View style={styles.container}>
<Text style={styles.text}>Contact</Text>
<Text onPress={() => navigator.pop()}>Back</Text>
</View>
)
```

❶ **Home is updated to take the navigator prop that was passed as a prop in the `renderScene` method, and use it to call navigator.push to navigate to the <About /> and <Contact /> components we created. We push to a new route by calling `navigator.push`, passing an object with both component and title values specified (title will be used shortly when we create a navigation header).**

❷ **About and Contact also take the navigator prop that was passed in renderScene and return a basic component with a back button that calls `navigator.pop()`.**

Now we should be able to click on About and Contact from the Home screen to navigate to our new components.

With our new components working, let's now add a navigation bar. To enable a `navigationBar`, we need to destructure the `NavigationBar`from the `Navigator`, create a `NavigationBar` component and pass it as a prop to the `navigationBar` in Navigator. Destructuring `NavigationBar`from Navigator is the same as calling `Navigator.NavigationBar` but in a more concise manner (Listing 7.6, figure 7.3).

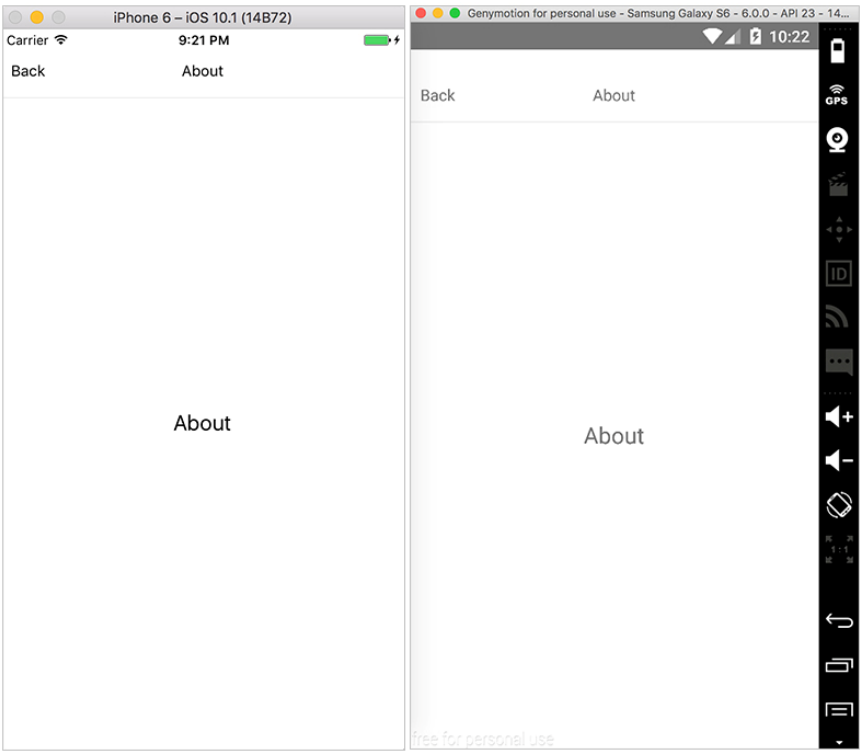


Figure 7.3 NavigationBar on the About route

Listing 7.6 Navigation Bar Creation

```
const { NavigationBar } = Navigator (1)

const NavBar = ( (2)
<NavigationBa  r (3)
    style={{ borderBottomWidth: 1, borderBottomColor: '#ededed' }}
    routeMapper={{ (4)
      LeftButton: (route, navigator, index, navState) => {
        if (index === 0) return null
        return (<Text style={styles.backButton} onPress={navigator.pop}>Back</Text>)
      },
      RightButton: (route, navigator, index, navState) => {
        return null
      },
      Title: (route, navigator, index, navState) => {
        return (<Text style={styles.title}>{route.title}</Text>)
      },
    }}
  />
)
```

1. destructure the `NavigationBar` from the Navigator
2. create a variable called `NavBar` to hold the `NavigationBar`
3. create the `NavigationBar`
4. `NavigationBar` takes a prop called `routeMapper` that should contain three functions: `LeftButton`, `RightButton`, and Title. They should all return either a React Native element or null. Each function is invoked with route, navigator, index, and `navState`. `navState` is an object that contains all of the navigation state information including the route stack.

   The Title component will return the route title, the `LeftButton` will call `navigator.pop`,

   The `LeftButton` component will check to see if we are on the initial route (index === 0). If we are on the initial route, it will return null. If we are not on the initial route, it will return a back button that calls `navigator.pop`

   The `RightButton` will always return null

Now that we have created a `NavBar` component, the last thing we need to do is pass it in as a prop to the Navigator. In the render method, when we return the Navigator, pass the `NavBar` in as a `navigationBar` prop. (listing 7.7).

Listing 7.7 Passing the NavBar component as a prop to the Navigator

```
render () {
    return (
<Navigator
        navigationBar={NavBar} (1)
        ref='navigator'
        renderScene={this.renderScene}
        initialRoute={{ title: 'Home', component: Home }}
      />
    )
  }
```

1. pass in the `NavBar` to the `navigationBar` prop of Navigator

Now when we run our app we should get a navigation bar (figure 7.3).

With the current setup, we see that when we navigate to a new page the transitions happen from right to left, and when we go back the go from left to right. What if we want to specify a certain type of transition? Doing so with this component is very straightforward. Navigator also has a prop called `configureScene` which lets us specify how we want the scene to animate (Listing 7.8).

Listing 7.8 Basic configureScene implementation

```
const { SceneConfigs } = Navigator
return (
<Navigator
    renderScene={this.renderScene}
    configureScene={(route, routeStack) =>SceneConfigs.FloatFromRight }
    initialRoute={initialRouteObject} />
```

The default configuration is `PushFromRight`, so nothing different will happen when we specify `PushFromRight`, but there are quite a few options to choose from (listing 7.9).

Listing 7.9 Navigator scene configuration options

```
Navigator.SceneConfigs.PushFromRight (default)
Navigator.SceneConfigs.FloatFromRight
Navigator.SceneConfigs.FloatFromLeft
Navigator.SceneConfigs.FloatFromBottom
Navigator.SceneConfigs.FloatFromBottomAndroid
Navigator.SceneConfigs.FadeAndroid
Navigator.SceneConfigs.SwipeFromLeft
Navigator.SceneConfigs.HorizontalSwipeJump
Navigator.SceneConfigs.HorizontalSwipeJumpFromRight
Navigator.SceneConfigs.HorizontalSwipeJumpFromLeft
Navigator.SceneConfigs.VerticalUpSwipeJump
Navigator.SceneConfigs.VerticalDownSwipeJump
```

configureScene is invoked with two parameters, the route and the routeStack.

```
(route, routeStack) =>SceneConfigs.FloatFromBottom
```

One way to dynamically implement one of these scene configurations is to create a method that can be attached to the `configureScene` prop and do logic on the route. That is, in the route object we can pass parameters to check for in the `configureScene` method.

In our example, we will check to see if there is a `type: 'modal'` in the route, and if so we will float the scene from the bottom like a Modal window.

After the `renderScene` method, we will create a `configureScene` method. We will also create a new component called Modal and update the Home component to have a new link called Modal. (listing 7.10).

Listing 7.10 ConfigureScene implementation

```
❶
const Home = ({ navigator }) => (
<View style={styles.container}>
<Text style={styles.text}>Home</Text>
<Text onPress={() => navigator.push({component: About, title: 'About'})}>About</Text>
<Text onPress={() => navigator.push({component: Contact, title:
      'Contact'})}>Contact</Text>
<Text onPress={() => navigator.push({component: Modal, title: 'Modal', type:
      'modal'})}>Modal</Text>
</View>
)

❷
const Modal = ({ navigator }) => (
<View style={[styles.container, { backgroundColor: 'red' } ]}>
<Text style={styles.text}>Modal</Text>
</View>
)

...
❸
constructor() {
  super()
  this.renderScene = this.renderScene.bind(this)
```

```
  this.configureScene = this.configureScene.bind(this)
}
...

❹
configureScene (route) {
  const { SceneConfigs } = Navigator
  if (route.type === 'modal') {
    return SceneConfigs.FloatFromBottom
  } else {
    return SceneConfigs.FloatFromRight
  }
}
...

❺
render () {
  return (
<Navigator
      navigationBar={NavBar}
      configureScene={this.configureScene}
      ref='navigator'
      renderScene={this.renderScene}
      initialRoute={{ title: 'Home', component: Home }}
    />
  )
}
```

❶ **Add a new route called Home, and add a new type property to the route object.**
❷ **Create a new component called Modal, and give it a background color of red so we can see the exact transition and how it differs from the others.**
❸ **In the existing constructor, we bind the new `configureScene` method to the class.**
❹ **In the newly added `configureScene method`, we check to see if the `route.type` is modal. If it is, we float the scene from the bottom. If it is not, we float it in from the right.**
❺ **In the existing Navigator, we add a new prop called `configureScene` and pass in our new `configureScene` method.**

Now when we press Modal, we see the nice bottom to top transition, while all our other transitions stay the same!

What if we need to pass properties to another scene? With Navigator, we can do this pretty easily.

The first thing we need to do is go back to our `renderScene` method and tell it to pass along any properties to each component being returned (Listing 7.11).

Listing 7.11 Passing Props

```
renderScene(route, navigator) {
  return <route.component navigator={navigator} {...route.props} />
}
```

To pass props, we can now just add a props key to the route object (listing 7.12).

Listing 7.12 Passing Props

```
<Text
onPress={() => navigator.push({component: Modal, title: 'Modal', type: 'modal',
      props: { author: 'Nader Dabit' } })}>Modal</Text>
```

Now, the author prop will be available in the Modal component (listing 7.13).

Listing 7.13 Rendering passed props from Navigator

```
const Modal = ({ navigator, author }) => (
<View style={[styles.container, { backgroundColor: 'red' } ]}>
<Text style={styles.text}>Modal</Text>
<Text>{author}</Text>
</View>
)
```

Navigator also has the props and methods shown in tables 7.3 and 7.4.

Table 7.3 Navigator props

prop	type	description (some from docs)
configureScene	function	describes the animations and gestures of the current scene being rendered.
initialRoute	object	object containing the initial scene being rendered.
initialRouteStack	array	Pass this in to provide a set of routes to initially mount. This prop is required if initialRoute is not provided to the navigator. If this prop is not passed in, it will default internally to an array containing only initialRoute.
navigationBar	node	Use this to provide an optional component representing a navigation bar that is persisted across scene transitions. This component will receive two props: navigator and navState representing the navigator component and its state. The component is re-rendered when the route changes.
navigator	object	Optionally pass in the navigator object from a parent Navigator.
onDidFocus	function	Required function which renders the scene for a given route. Will be invoked with the route and the navigator object.
sceneStyle	style (object)	Styles to apply to the container of each scene.

Table 7.4 Navigator methods

method	arguments	description (some from docs)
immediatelyResetRouteStack(nextRouteStack)	nextRouteStack (array of objects)	Resets every scene with a new array of routes.
jumpTo(route)	route (object)	Transitions to an existing scene without unmounting
jumpForward()	none	Jump forward to the next scene in the route stack.
jumpBack()	none	Jump backward without unmounting the current scene.
push(route)	route (object)	Navigate forward to a new scene, squashing any scenes that you could jump forward to.
popN(number)	number	Go back N scenes at once. When N=1, behavior matches pop(). When N is invalid(negative or bigger than current routes count), do nothing.
pop()	none	Transition back and unmount the current scene.
replaceAtIndex(route, index, callback)	(route, index, callback)	Replace a scene as specified by an index.
replace(route)	route (object)	Replace the current scene with a new route.
replacePrevious(route)	route (object)	Replace the previous scene.
popToTop()	none	Pop to the first scene in the stack, unmounting every other scene.
popToRoute(route)	route (object)	Pop to a particular scene, as specified by its route. All scenes after it will be unmounted.
replacePreviousAndPop(route)	route (object)	Replace the previous scene and pop to it.
resetTo(route)	route (object)	Navigate to a new scene and reset route stack.
getCurrentRoutes()	none	Returns the current list of routes.

7.3 Using NavigationExperimental to create cross platform navigation

Navigation Experimental is the newest navigation API from the React Native core project. It aims to be more in line with the React and Redux philosophy of one way data flow, and unlike Navigator and NavigatorIOS, gives us complete control over the state of our Navigation.

The following statement is from the original proposal for this API (https://github.com/ericvicenti/navigation-rfc):

> A new Navigation system for react native. Focuses on the following improvements over <Navigator />:

Single-directional data flow, using reducers to manipulate top-level state object

Navigation logic and routing must be independent from view logic to allow for a variety of native and js-based navigation views

Improved modularity of scene animations, gestures, and navigation bars.

What does this mean for us? Well, at this point we have yet to run into any pitfalls with either Navigator or NavigatorIOS, but in many larger applications, navigation state can become complex and hard to reason about. While using Navigator, we also have to keep up with our navigator instance, which can be tiresome, as it would have to be passed down as a prop to child components wherever we need it.

NavigationExperimental offers a central location that we can keep up with and update our navigation, and it is the way going forward as far as the React Native project is concerned.

Navigator will also one day be deprecated and Navigation Experimental will be the main Navigator as well as the one being maintained, so it is important to understand this API and understand how to use it.

A basic concept to understand when working with NavigationExperimental is the idea of a reducer function. A reducer is a function that takes in an initial state or value as an argument and returns a new state or value based on all previous and current actions. Reducers must be pure, meaning the reducer function should:

- Always return the same output given the same input
- Not rely on any external state, only the arguments
- produce no side effects

Something else to remember when writing reducers and working with the state of the reducer is that the data structures should be immutable, meaning that we should not mutate the state, but instead return a completely new data structure.

This means, for example, that instead of adding and removing items from an array stored in the state when the time comes to update the array, we should create a copy of the array, push or pop the item to or from the copied array, and then return the newly copied array along with the existing state.

If you are new to the idea of immutable data, and you would like to have this idea reinforced for you, there are a couple of open source projects that you can use in your application. Check out ImmutableJS or Seamless Immutable as they are both great libraries and are open source.

Let's look at an example reducer function (listing 7.14).

Listing 7.14 Reducer function

```
const initialState = { (1)
name: 'Nader Dabit',
  age: 22
```

```
}
function personReducer(state = initialState, action) { ❷
  switch(action.type) { ❸
    case 'increment_age':
      return { ❹
        ...state,
        age: state.age + 1
      }
    case 'change_name':
  return { ❺
        ...state,
        name: action.name,
      }
  }
}
```

❶ create an `initialState` object with two keys, name and age.
❷ `personReducer` takes two arguments, state (an object) and action (an object)
❸ switch on the `action.type`
❹ in the case of 'increment_age', return a new state object with the existing state, and we replace the age key with the existing age plus 1
❺ in the case of 'change_name', return a new state object with the existing state, and we replace the name key with the action.name value

We will be making use of this reducer pattern to manage navigation state in NavigationExperimental, so now that we've gone over what this looks like, let's implement this into a NavigationExperimental implementation.

When working with NavigationExperimental, we can apply some of the same concepts we've learned so far from Navigator and NavigatorIOS, the main one being the idea of returning a main navigation component in the render method of the main application class (listing 7.15).

Listing 7.15 Returning NavigationExperimental NavigationCardStack

```
class App extends Component {
  render() {
    return (
<NavigationCardStack />
    )
  }
}
```

`NavigationCardStack` is very similar to `Navigator` and `NavigatorIOS` in that it takes a `renderScene` method, but instead of an `initialRoute` object, it takes a `navigationState` object (listing 7.16).

Listing 7.16 NavigationCardStack with renderScene and navigationState

```
const navState = { ❶
  index: 0,
  routes: [{ key: 'Home' }]
```

```
}

class App extends Component {
  renderScene() {
    switch(props.scene.route.key) { ❷
      case 'Home':
        return <Home />
    }
  }
  render() {
    return (
<NavigationCardStack
        navigationState={navState}
        renderScene={this.renderScene}
      />
    )
  }
}
```

❶ `navState` **is an object with at least two properties, an index and an array of routes**
❷ `renderScene` **is called when the** `navigationState` **is changed, and returns the route in the** `navigationState` **routes array that corresponds with the current index of the** `navigationState`

Again, `renderScene` will return the route in the `navigationState` routes array that corresponds with the current index of the `navigationState`, therefore the following `navigationState` will return the About route because the index is 2 and route in the corresponding position in the routes array is 'About'(listing 7.17).

Listing 7.17navigationState with an index set to 2

```
const navState = {
  index: 2,
  routes: [{ key: 'Home' }, { key: 'Contact' }, { key: 'About' }]
}
```

Now that we've gone over some of the basics, we can start building navigation using NavigationExperimental.

To get started, we will create a basic app that will only render an initial route using NavigationExperimental (listing 7.18, figure 7.4).

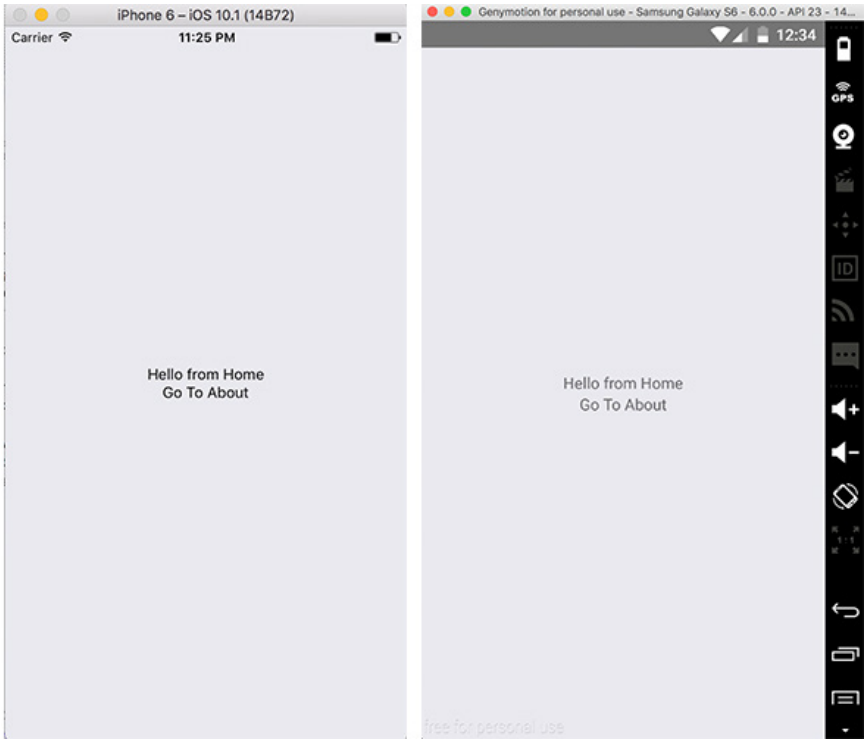


Figure 7.4 NavigationExperimental NavigationCardStack with 1 route

Listing 7.18 Setting up NavigatorExperimental

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

const {
  CardStack: NavigationCardStack, ❷
} = NavigationExperimental

let styles = {}

const Home = () => ( ❸
<View style={styles.container}>
<Text>Hello from Home</Text>
</View>
)

function reducer(state) { ❹
  if (!state) {
    return {
      index: 0,
      routes: [{key: 'Home'}],
```

```
    };
  }
}

class NavigationCardStackExample extends Component {
  constructor() {
    super()
   this.state = {
      navState: reducer() ❺
}
    this.renderScene = this._renderScene.bind(this)
  }

  _renderScene(props) { ❻
    switch(props.scene.route.key) {
      case 'Home':
        return <Home />
    }
  }

  render() {
    const { navState } = this.state
    return (
<NavigationCardStack ❼
        navigationState={navState}
        renderScene={this._renderScene}
      />
    )
  }

}

styles = {
  container: {
    justifyContent: 'center',
    alignItems: 'center',
    flex: 1
  },
}

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

❶ **import** `NavigationExperimental` **from React Native**
❷ **destructure** `NavigationCardStack` **from** `NavigationExperimentalCardStack`
❸ **create initial route of Home**
❹ **create a reducer that for now will always return the initial state object created inside of the reducer**
❺ **set the initial state of the class to be the returned value of the reducer**
❻ `renderScene` **will be invoked with the navigation props object that we will call props. The current scene will be available as** `props.scene`**, and we will switch on the scene key to check for the scene we want, and will return a component based on the key value**
❼ **the main class will return** `NavigationCardStack` **with the** `navigationState` **and** `renderScene` **method passed as props**

Now when we run this, we should just get the main Home route rendered to the scene as it is the route that is in position 0 of the routes array in the `navigationState` object (figure 7.4).

Now that we have the initial route set up, we can add in a navigate function along with another route to navigate to and from, we will call this route About. We will also set up a navigate method and update the reducer to take an action object (listing 7.19, figure 7.5).

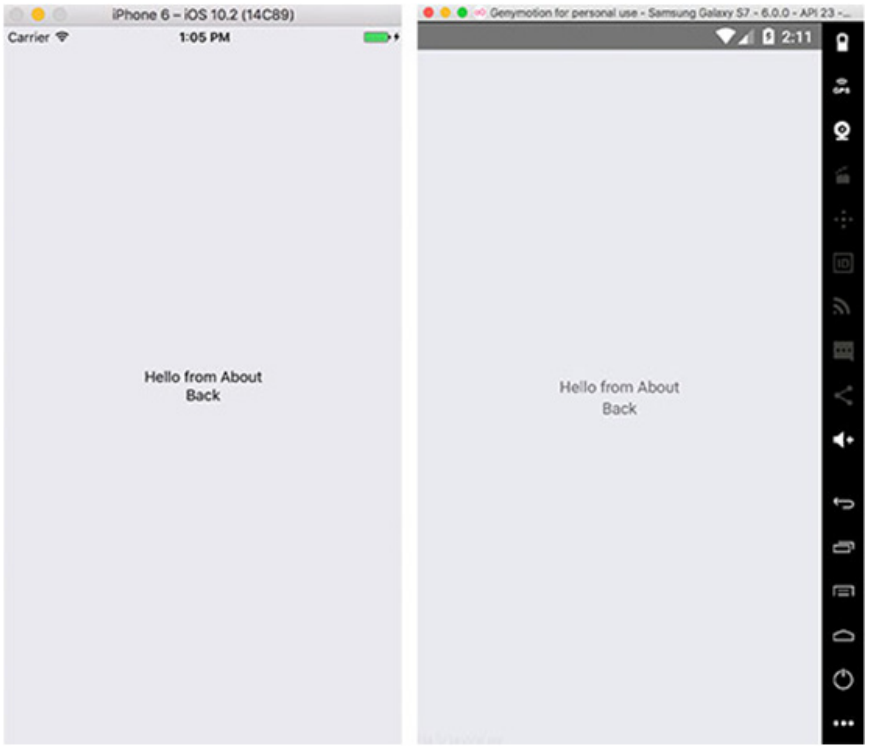


Figure 7.5 NavigationExperimental NavigationCardStack with 2 routes, showing About route

Listing 7.19 NavigationExperimental NavigationCardStack with 2 routes

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

const {
  CardStack: NavigationCardStack,
} = NavigationExperimental

let styles = {}

const Home = ({ navigate }) => { (1)
  return (
<View style={styles.container}>
<Text>Hello from Home</Text>
<Text onPress={() => navigate({ type: 'push', route: { key: 'About' }})}>Go To
      About</Text>
```

```
</View>
  )
}

const About = ({ navigate }) => (
<View style={styles.container}> ❷
<Text>Hello from About</Text>
<Text onPress={() => navigate({ type: 'pop' })}>Back</Text>
</View>
)

function reducer(state, action) {
  if (!state) {
    return {
      index: 0,
      routes: [{ key: 'Home' }],
    };
  }
  switch (action.type) {
    case 'push': { ❸
      const routes = state.routes.slice();
      routes.push(action.route);
      return {
        ...state,
        index: routes.length - 1,
        routes,
      }
    }
    case 'pop': { ❹
      if (state.index <= 0) return state
      const routes = state.routes.slice(0, -1);
      return {
        ...state,
        index: routes.length - 1,
        routes,
      }
    }
    default:
      return state
    }
}

class NavigationCardStackExample extends Component {
  constructor() {
    super()
    this.state = { navState: reducer() }
    this._renderScene = this._renderScene.bind(this)
    this._navigate = this._navigate.bind(this)
  }
_renderScene(props) { ❺
    switch(props.scene.route.key) {
      case 'Home':
        return <Home navigate={this._navigate} />
      case 'About':
        return <About navigate={this._navigate} />
    }
  }
```

```
  _navigate(action) { ❻
    const navState = reducer(this.state.navState, action)
    this.setState({
      navState
    })
  }
  render() {
    const { navState } = this.state
    return (
<NavigationCardStack
        navigationState={navState}
        renderScene={this._renderScene}
      />
    )
  }
}

styles = {
  container: {
    justifyContent: 'center',
    alignItems: 'center',
    flex: 1
  },
}

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

❶ Home now takes a single prop, navigate, and we use navigate to move to the About route by calling navigate and passing in a type of 'push', and a route object with a key of 'About'.
❷ About also takes the navigate method as a prop, and we use it to go back to the previous route by calling navigate with the type of 'pop'
❸ the reducer function now takes an action, and we switch on the action type to either push or pop routes to our `navState`. If push is called, we create a shallow copy of the routes array by calling .slice() on the routes array, push the new route to the array, and then return a new object with the existing state, an index with a value of the routes length minus 1, and the new routes array
❹ if pop is called, we call .slice on the route array, removing the last item in the array, and then return a new object with the existing state, an index with a value of the routes length minus 1, and the new routes array
❺ `renderScene` now checks for the 'About' key and returns the About route if it matches
❻ the `_navigate` method calls the reducer method with the first argument being the `navState`, and the second argument being the action, and resets the `navState` with the returned value of the reducer

Now, we should be able to press the 'Go To About' button and navigate to the About page.

Now that we have the basic navigation working, let's add a header that will display the current route title and a back button when not on the initial route.

To get started, we will need to import the Header component from NavigationExperimental as well as create the header we will be using (listing 7.20, figure 7.6).

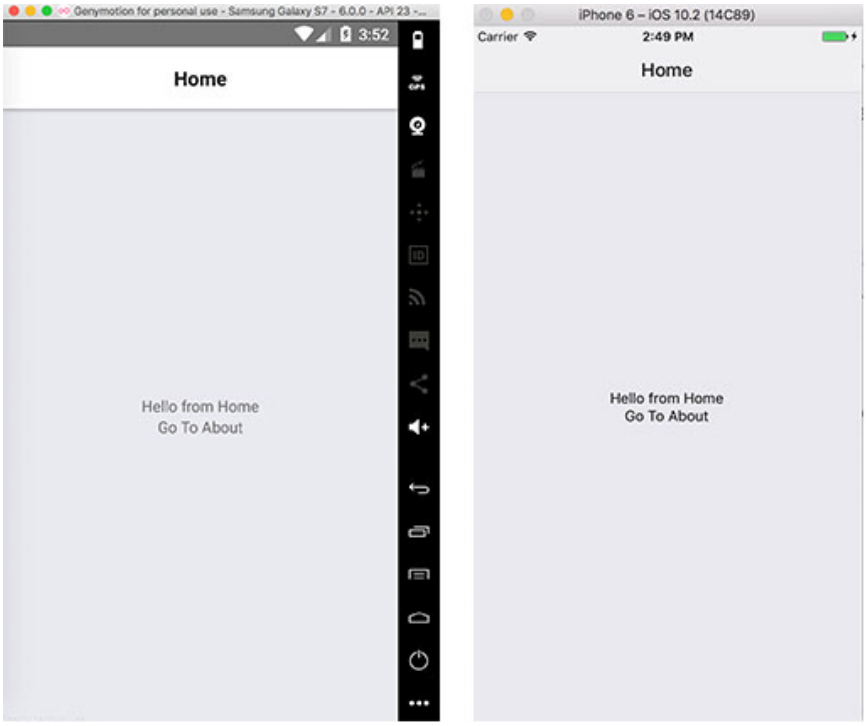


Figure 7.6 NavigationExperimenatal with Header

Listing 7.20Creating a Header component for Navigation Experimental

```
...

const {
  CardStack: NavigationCardStack,
  Header: NavigationHeader, (1)
} = NavigationExperimental

let styles = {}

class Header extends Component {
  _back = () => { (2)
    this.props.navigate({ type: 'pop' });
  }
  _renderTitleComponent = (props) => { (3)
    return (
<NavigationHeader.TitletextStyle={{ textAlign: 'center' }}>
        {props.scene.route.key}
</NavigationHeader.Title>
    );
  }
  render() {
```

```
    return (
<NavigationHeader ❹
        {...this.props}
        renderTitleComponent={this._renderTitleComponent}
        onNavigateBack={this._back}
      />
    );
  }
}

...
```

❶ **import and destructure Header from** `NavigationHeader`
❷ **create back method that will navigate back when the back button is pressed because we attach this method to** `onNavigateBack` **in the** `NavigationHeader`
❸ `renderTitleComponent` **will return the title of the component (in our case the key, or route.key), declared in the route, and display it in the middle of the navigation bar**
❹ `NavigationHeader` **returns the header, and we pass in the** `onNavigateBack` **method and** `renderTitleComponent` **methods as props, as well as a spread object containing all props.**

The next thing we need to do to is to return this new header component in our `NavigationCardStack` component. To do this, `NavigationCardStack` has a method called `renderHeader` that will return a Header component (listing 7.21).

Listing 7.21 Navigation Header

```
class NavigationCardStackExample extends Component {
  ...
  _renderHeader = (props) => { ❶
    return (
<Header
        navigate={this._navigate}
        {...props}
      />
    )
  }
  ...
  render() {
    const { navState } = this.state
    return (
<NavigationCardStack
        renderHeader={this._renderHeader} ❷
        navigationState={navState}
renderScene={this._renderScene}
      />
    )
  }
}
```

❶ `_renderHeader` **will return the new Header component we created earlier. We pass in the** `_navigate` **method as well as all the props from the navigator.**
❷ **add** `renderHeader` **method as a prop to the** `NavigationCardStack` **component.**

If you are interested in further exploring the Navigation options available via open source packages, I would especially take a close look at ExNavigation by Exponent, as the maintainers work closely with the React Native team and are core contributors to React Native.

7.4 Summary

- NavigatorIOS is a good basic navigator to begin using, but only works on iOS and may not scale well to a large project.
- Navigator is currently the most stable navigation component. Navigator works cross platform and there are a good amount of documentation and resources on the web for this navigation implementation.
- The NavigationExperimental API is currently being finalized and is the next iteration of navigation in the React Native project.

8

Cross-platform APIs

This chapter covers

- How to implement cross-platform APIs

One of the key benefits of using React Native is the ease in which native APIs can be accessed and used with JavaScript. In this chapter, we will cover most of the cross-platform APIs available in the framework. When accessing these APIs, you will be able to use a single codebase to implement platform specific behavior on both iOS and Android.

The main difference between the native APIs in this chapter and the native components in chapter 6 is that native components usually have something to do with the UI, such as showing a specific UI element, whereas the APIs are more about accessing native features and component within the phone such as interacting with or accessing data held within the device (geolocation, application state, etc…).

In addition to cross platform APIs, there are also platform specific APIs (i.e. APIs that only work on either Android or iOS). We will be covering iOS specific APIs in Chapter 9, and Android specific APIs in chapter 10.

A few examples of what APIs we will be covering in this chapter are the PanResponder (tracking touch event locations), Alert (triggering and using the native alert dialog), and Geolocation (get location of user) among others.

8.1 Implementing Cross Platform APIs

8.1.1 Alert

Alert launches a platform specific alert dialog with a title, message, and optional methods that can be called when these buttons are pressed. Alert can be triggered by calling the alert method (`Alert.alert`)

Alert takes four arguments (listing 8.1). Alert.alert(title, message, buttons, options)

Table 8.1 Alert methods and arguments

Argument	Type	Description
title	string	main message of Alert button
message	string	secondary message of Alert
buttons	array	array of buttons, each being an object with two keys: title (string) and onPress (function)
options	object	object containing a cancelable Boolean

You can trigger an alert by calling the Alert.alert() method, passing in one or more arguments.

In our example, we will create an Alert with two options: 'Cancel' & 'Show Message'. If cancel is pressed, we will dismiss the Alert, and if Show Message is clicked, we will update the state to show our message (listing 8.2)

Listing 8.1 Alert

```
import React, { Component } from 'react'
import { TouchableHighlight, View, Text, StyleSheet, Alert } from 'react-native' (1)
let styles = {}

class App extends Component {
  constructor () {
    super()
    this.state = { (2)
      showMessage: false
    }
    this.showAlert = this.showAlert.bind(this)
  }
  showAlert () { (3)
    Alert.alert(
      'Title',
      'Message!',
      [
        {
          text: 'Cancel',
          onPress: () => console.log('Dismiss called...'),
          style: 'destructive'
        },
        {
          text: 'Show Message',
          onPress: () => this.setState({ showMessage: true })
        }
      ]
    )
  }

  render () {
    const { showMessage } = this.state
```

```
    return (
      <View style={styles.container}>
        <TouchableHighlight onPress={this.showAlert} style={styles.button}>
          <Text>SHOW ALERT</Text>
        </TouchableHighlight>
        { ❹
          showMessage &&<Text>Showing message - success</Text>
        }
</View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    flex: 1
  },
  button: {
    height: 70,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#ededed'
  }
})
```

❶ Alert is imported from React Native
❷ The state is instantiated with showMessage set to false
❸ the showAlert method is defined, passing in a title of 'Title', a message of 'Message', and two buttons. If 'Show Message' is pressed, the state is updated to showMessage being true.
❹ We hide a message unless showMessage is set to true

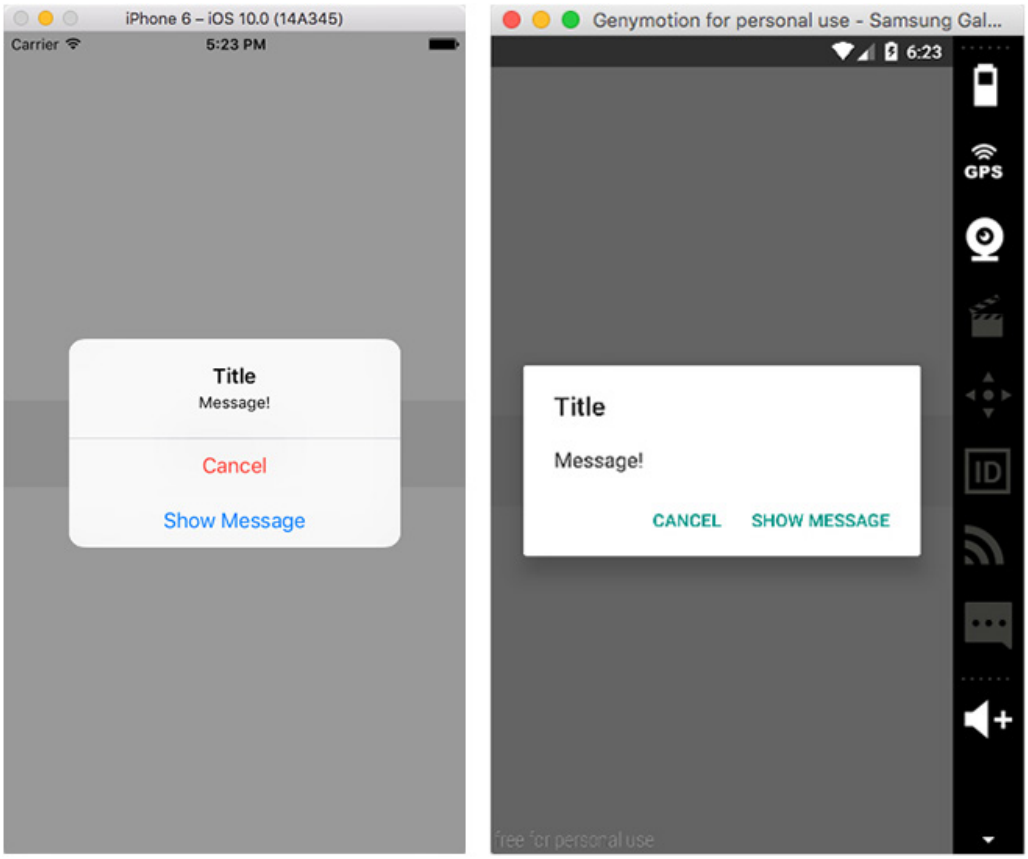


Figure 8.1 Alert

8.1.2 AppState

AppState will tell you whether the app is active, inactive, or in the background. This will basically call a method whenever the app state changes, allowing you to perform actions or call other methods based on the state of the app. If you would like to log a user out for example when they minimize the app, this would be where you would probably do so.

AppState will return either active, inactive or background when it is called. To set up this method, you would add an eventlistener, and call a method when the event when it is fired. The events that AppState uses to respond are either 'change', or 'memorywarning'. We will create an example using 'change' because it is what you will be mainly using in a real world scenario.

Listing 8.2 AppState

```
import React, { Component } from 'react'
import { AppState, View, Text, StyleSheet } from 'react-native' ❶
let styles = {}

class App extends Component {
  componentDidMount () { ❷
    AppState.addEventListener('change', this.handleAppStateChange)
  }
  handleAppStateChange (currentAppState) { ❸
    console.log('currentAppState:', currentAppState)
  }
  render () {
    return (
      <View style={styles.container}>
         <Text>Testing App State</Text>
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    flex: 1
  }
})

export default App
```

❶ import the AppState API from React Native
❷ in `componentDidMount`, we call the AppState.addEventListener, passing in the type of event to listen for (change), and a callback function, in this case it is `handleAppStateChange`.
❸ in handleAppStateChange, we log out the currentAppState

Now, when we run the project, we can test this by either pressing CMD + shift + H in iOS simulator or pressing the home button in the Android emulator. We should not see the console logging out the current app state (either active, inactive, or background).

8.1.3 AsyncStorage

Next up is the AsyncStorage API. AsyncStorage is a great way to persist and store data. It is asynchronous, meaning that you can retrieve data either using a promise or async await, and uses a key-value system to store and retrieve the data.

Table 8.2 AsyncStorage methods and arguments

Method	Arguments	Description
setItem	setItem(key, value, callback)	stores item in AsyncStorage

getItem	getItem(key, callback)	retrieves item from AsyncStorage
removeItem	removeItem(key, callback)	removes item from AsyncStorage
mergeItem	mergeItem(key, value, callback)	merges existing value with another existing value. (both values must be stringified JSON)
clear	clear(callback)	erases all values in AsyncStorage
getAllKeys	getAllKeys(callback)	gets all keys stored in your app
flushGetRequests	none - flushGetRequests()	flushes any pending requests
multiGet	multiGet([keys], callback)	allows you to get multiple values using an array of keys
multiSet	multiSet([keyValuePairs], callback)	allows you to set multiple key value pairs at once.
multiRemove	multiRemove([keys], callback)	allows you to delete multiple values using an array of keys
multiMerge	multiMerge([keyValuePars], callback)	allows you to merge multiple key value pairs into one method

In our example, we will take a user object and store it into the AsyncStorage in componentDidMount. We will then use a button to extract the data from AsyncStorage, populate our state with the data and render it to our View.

Listing 8.3 AsyncStorage

```
import React, { Component } from 'react'
import { TouchableHighlight, AsyncStorage, View, Text, StyleSheet } from 'react-
      native'    ❶
let styles = {}

const person = { ❷
  name: 'James Garfield',
  age: 50,
  occupation: 'President of the United States'
}

const key = 'president' ❸

class App extends Component {
  constructor () {
    super()
    this.state = {
      person: {}        ❹
    }
    this.getPerson= this.getPerson.bind(this)
  }
  componentDidMount () {
    AsyncStorage.setItem(key, JSON.stringify(person)) ❺
      .then(() => console.log('item stored...'))
      .catch((err) => console.log('err: ', err))
```

```
  }
getPerson() { ❻
    AsyncStorage.getItem(key)
      .then((res) => this.setState({ person: JSON.parse(res) }))
      .catch((err) => console.log('err: ', err))
  }
  render () {
    const { person } = this.state
    return (
      <View style={styles.container}>
      <Text style={{textAlign: 'center'}}>Testing AsyncStorage</Text>
      <TouchableHighlight onPress={this.getPerson} style={styles.button}> ❼
        <Text>Get President</Text>
      </TouchableHighlight>
      <Text>{person.name}</Text>
      <Text>{person.age}</Text>
      <Text>{person.occupation}</Text>
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    flex: 1,
    margin: 20
  },
  button: {
    justifyContent: 'center',
    marginTop: 20,
    marginBottom: 20,
    alignItems: 'center',
    height: 55,
    backgroundColor: '#dddddd'
  }
})
```

❶ import AsyncStorage from 'react-native '
❷ create a person object and store our information in this object
❸ create a key that we will be using to add and remove data from AsyncStorage
❹ create a person object in our state
❺ in componentDidMount, we call AsyncStorage.setItem, passing in the key as well as a the person. You will notice that we are calling JSON.stringify. We do this because we need the value that we store in AsyncStorage to be a string. JSON.stringify simply will take objects and arrays and turn them into strings. As you will see next, we will simply call JSON.parse when we retrieve from AsyncStorage, which will turn this data back into a JavaScript object.
❻ create getPerson method, which will call AsyncStorage.getItem, passing in the key we created earlier. When this is called, we receive a callback function with the data that is retrieved from AsyncStorage. We then call JSON.parse on the returned data and populate the state using setState.
❼ finally, we wire up the getPerson method to a TouchableHighlight in our view. When the TouchableHighlight is pressed, the data from AsyncStorage is rendered to our View.

As you can see, we used promises to set and return the values from AsyncStorage. There's also another way to do this, so let's take a look at async await.

Listing 8.4 async await

```
async componentDidMount () {
  try {
    await AsyncStorage.setItem(key, JSON.stringify(person))
    console.log('item stored')
  } catch (err) {
    console.log('err:', err)
  }
}
async getPerson () {
  try {
    var data = await AsyncStorage.getItem(key)
    var person = await data
    this.setState({ person: JSON.parse(person) })
  } catch (err) {
    console.log('err: ', err)
  }
}
```

async await first requires you to mark the function as async by adding the async keyword before the function name. We are then able to use the await keyword to wait for the, allowingus to write promise-based code as if it were synchronous. When we await a promise, the function waits until the promise settles, but does so in a non-blocking way, then assigns the value to the variable.

8.1.4 ClipBoard

Clipboard lets you save and retrieve content from the clipboard on both iOS and Android. ClipBoard has two methods (Listing 8.8).

Table 8.3 Clipboard methods

Method	Arguments	Description
getString	n/a	get contents of clipboard
setString	setString(content)	set content of clipboard

In our example, we will set an initial Clipboard value of 'Hello World' in componentDidMount, then use a method attached to the TextInput to update the clipboard, and add a button that pushes the current ClipboardValue to an array, and renders it to our View.

Listing 8.5 Clipboard

```
import React, { Component } from 'react'
import { TextInput, Clipboard, TouchableHighlight, getString, View, Text, StyleSheet
      } from 'react-native' (1)
let styles = {}

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

```
    super()
    this.state = {
      clipboardData: [] ❷
    }
    this.pushClipboardToArray = this.pushClipboardToArray.bind(this)
  }
  componentDidMount () {
    Clipboard.setString('Hello World! '); ❸
  }
  updateClipboard (string) {
    Clipboard.setString(string);          ❹
  }
  async pushClipboardToArray() {          ❺
    const { clipboardData } = this.state
    var content = await Clipboard.getString();
    clipboardData.push(content)
    this.setState({clipboardData})
  }
  render () {
    const { clipboardData } = this.state
    return (
        <View style={styles.container}>
          <Text style={{textAlign: 'center'}}>Testing Clipboard</Text>
        <TextInput style={styles.input} onChangeText={(text) =>
      this.updateClipboard(text)} />       ❻
        <TouchableHighlight onPress={this.pushClipboardToArray}
      style={styles.button}>               ❼
        <Text>Click to Add to Array</Text>
        </TouchableHighlight>
        {
          clipboardData.map((d, i) => {    ❽
            return <Text key={i}>{d}</Text>
         })
       }
    </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    flex: 1,
    margin: 20
  },
  input: {
    padding: 10,
    marginTop: 15,
    height: 60,
    backgroundColor: '#dddddd'
  },
  button: {
    backgroundColor: '#dddddd',
    justifyContent: 'center',
    alignItems: 'center',
    height: 60,
    marginTop: 15,
```

```
  }
})
```

1. Import ClipBoard from React Native
2. Set an empty array called clipboardData in our state
3. In componentDidMount, update the Clipboard value to 'Hello World'
4. Add a updateClipboard method that will replace the existing Clipboard value
5. add async method named pushClipBoardToArray, using the async await syntax we covered in listing 8.6. This method will take the value of the clipboard and store it in a variable we named content, then push to the clipboardData array and reset the state of the clipboardData array.
6. Attach the TextInput with the updateClipboardText method
7. Attach the pushClipboardToArray method to be called when the TouchableHighlight it pressed
8. Map through the items in the clipboardData array and render them to the screen.

8.1.5 Dimensions

Dimensions gives us a way to get the device screen height and width. This is a good way to set width and height in your device, or to do calculations that are based on the width and height of the device.

To use Dimensions, you need to import the Dimensions api from React Native, then call the get() method, and return either the width, the height, or both (listing 8.9).

Listing 8.6 Dimensions

```
import React, { Component } from 'react'
import { View, Text, Dimensions, StyleSheet } from 'react-native' (1)
let styles = {}

const { width, height } = Dimensions.get('window') (2)
const windowWidth = Dimensions.get('window').width (3)

const App = () => (
  <View style={styles.container}> (4)
    <Text>{width}</Text>
    <Text>{height}</Text>
    <Text>{windowWidth}</Text>
  </View>
)

styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  }
})
```

1. import Dimensions from React Native
2. one way to access the Dimensions is to destructure what is returned from calling Dimensions.get on the window, which in this case is width and height. You can also get the scale of the window.
3. another way is to call Dimensions.get and access the object property directly, calling .width on Dimensions.get.
4. In our View, we render the dimensions that were stored in the variables we retrieved off of the Dimensions.get method.

8.1.6 Geolocation

Geolocation is achieved in React Native using the same API that is used in the browser, with the navigator.geolocation global variable. You do not need to import anything to begin using this, as it is again available as a global.

To get started with geolocation, you must enable geolocation to be used in the app if developing for Android (iOS is enabled by default) (listing 8.10).

Listing 8.7 Enabling Geolocation– Android

In AndroidManifest.xml, add the following line of code:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Now, we can start using the geolocation api.

Let's take a look at the available methods available to the api (listing 8.11).

Table 8.4 Geolocation methods

Method	Arguments	Description
getCurrentPosition	getCurrentPosition(successcallback, errcallback, optionsobject{enableHighAccuracy: Boolean, timeout: number, maximumAge: number})	gets the current position, success returns an object with a coords object and a timestamp
watchPosition	watchPosition (successcallback, errcallback, optionsobject{enableHighAccuracy: Boolean, timeout: number, maximumAge: number})	will get the current position and will automatically be called when the device position changes
clearWatch	clearWatch(watchid)	will cancel a watch. store the watchPosition method in a variable when created to have access to the watchId
stopObserving	none - stopObserving()	will cancel the all geolocation watches that have been set up

The coordinates returned from getCurrentPosition and watchPosition is an object with information about the current user's location (figure 8.2)

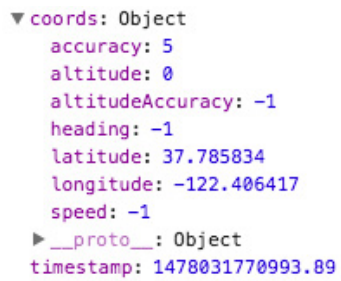


Figure 8.2 Coordinates object returned from geolocation

To see this in action, we will set up an instance of both geolocation getCurrentPosition and watchPosition, and have a button that will enable us to clear the watch. We will then display both the original coordinates as well as the updated coordinates (latitude and longitude) (Listing 8.12).

Listing 8.8

```
import React, { Component } from 'react'
import { TouchableHighlight, View, Text, StyleSheet } from 'react-native'
let styles = {}

class App extends Component {
  constructor () {
    super()
    this.state = { ❶
      originalCoords: {},
      updatedCoords: {},
      id: ''
    }
    this.clearWatch = this.clearWatch.bind(this)
  }
  componentDidMount() {
    navigator.geolocation.getCurrentPosition( ❷
      (success) => {
        this.setState({originalCoords: success.coords})
      },
      (err) => console.log('err:', err)
    )
    let id = navigator.geolocation.watchPosition( ❸
      (success) => {
        this.setState({
          id,
          updatedCoords: success.coords
        })
      },
      (err) => console.log('err:', err)
    )
  }
  clearWatch () { ❹
    navigator.geolocation.clearWatch(this.state.id)
```

```
  }
  render () {
    const { originalCoords, updatedCoords } = this.state
    return (
    <View style={styles.container}> ❺
      <Text>Original Coordinates</Text>
      <Text>Latitude: {originalCoords.latitude}</Text>
      <Text>Longitude: {originalCoords.longitude}</Text>
      <Text>Updated Coordinates</Text>
      <Text>Latitude: {updatedCoords.latitude}</Text>
      <Text>Longitude: {updatedCoords.longitude}</Text>
      <TouchableHighlight ❻
          onPress={this.clearWatch}
          style={styles.button}>
      <Text>Clear Watch</Text>
      </TouchableHighlight>
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    padding: 20,
  },
  button: {
    height: 60,
    marginTop: 15,
    backgroundColor: '#ededed',
    justifyContent: 'center',
    alignItems: 'center'
  }
})
```

❶ create an initial state with both an originalCoords and updatedCoords set as an empty object as well as id set as an empty string
❷ in componentDidMount, call getCurrentPosition on navigator.geolocation and set the state of coriginalCoords to success.coords
❸ call watchPosition and store the result of the function in a variable named id that we will use later to clear the watch, and reset the state with the id.
❹ create clearWatch method to clear the watch that was created in step C
❺ in our render method, we display the latitude and longitude from both the original coordinates as well as the updated coordinates

8.1.7 Keyboard

The Keyboard API allows us to have access to the native keyboard. We can use this to either dismiss the keyboard, or to listen to keyboard events and call methods based on these events.

Table 8.5

Method	Arguments	Description
addListener	addListener(event, callback)	connects a method to be called based on native keyboard events such as keyboardWillShow, keyboardDidShow, keyboardWillHide, keyboardDidHide, keyboardWillChangeFrame, and keyboardDidChangeFrame
removeAllListeners	removeAllListeners(eventType)	removes all listeners of the type specified
dismiss	none – dismiss()	dismisses the keyboard

In our example, we will set up a TextInput and have listeners set up for all available events. When the event is fired, we will log the event to the console. We will also have two buttons, one to dismiss the keyboard and another to remove all event listeners that were set up in componentWillMount (Listing 8.13).

Listing 8.9 Keyboard

```
import React, { Component }  from 'react'
import { TouchableHighlight, Keyboard, TextInput, View, Text, StyleSheet } from
      'react-native' ❶

let styles = {}

class App extends Component {
  componentWillMount () { ❷
    this.keyboardWillShowListener = Keyboard.addListener('keyboardWillShow', () =>
      this.logEvent('keyboardWillShow'))
    this.keyboardDidShowListener = Keyboard.addListener('keyboardDidShow', () =>
      this.logEvent('keyboardDidShow'))
    this.keyboardWillHideListener = Keyboard.addListener('keyboardWillHide', () =>
      this.logEvent('keyboardWillHide'))
    this.keyboardDidHideListener = Keyboard.addListener('keyboardDidHide', () =>
      this.logEvent('keyboardDidHide'))
    this.keyboardWillChangeFrameListener =
      Keyboard.addListener('keyboardWillChangeFrame', () =>
      this.logEvent('keyboardWillChangeFrame'))
    this.keyboardDidChangeFrameListener =
      Keyboard.addListener('keyboardDidChangeFrame', () =>
      this.logEvent('keyboardDidChangeFrame'))
  }
  logEvent(event) { ❸
    console.log('event: ', event)
  }
  dismissKeyboard () { ❹
    Keyboard.dismiss()
  }
  removeListeners () { ❺
    Keyboard.removeAllListeners('keyboardWillShow')
    Keyboard.removeAllListeners('keyboardDidShow')
```

```
    Keyboard.removeAllListeners('keyboardWillHide')
    Keyboard.removeAllListeners('keyboardDidHide')
    Keyboard.removeAllListeners('keyboardWillChangeFrame')
    Keyboard.removeAllListeners('keyboardDidChangeFrame')
  }
  render () {
    return (
      <View style={styles.container}>
      <TextInput style={styles.input} />
      <TouchableHighlight (6)
          onPress={this.dismissKeyboard}
          style={styles.button}>
      <Text>Dismiss Keyboard</Text>
      </TouchableHighlight>
      <TouchableHighlight (7)
          onPress={this.removeListeners}
          style={styles.button}>
      <Text>Remove Listeners</Text>
      </TouchableHighlight>
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 150,
  },
  input: {
    margin: 10,
    backgroundColor: '#ededed',
    height: 50,
    padding: 10
  },
  button: {
    height: 50,
    backgroundColor: '#dddddd',
    margin: 10,
    justifyContent: 'center',
    alignItems: 'center'
  }
})
```

1. import the Keyboard API from React Native
2. in componentWillMount, set up event listeners for all available keyboard events, then call the logEvent method to log out the event name
3. logEvent takes in the event name, and logs out the name of the event
4. dismissKeyboard will dismiss the keyboard if it is in view
5. in removeListeners, we call Keyboard.removeAllListeners passing in each of the listeners declared in componentWillMount
6. wire up the dismissKeyboard method to a button in the UI
7. wire up the removeListeners method to a button in the UI

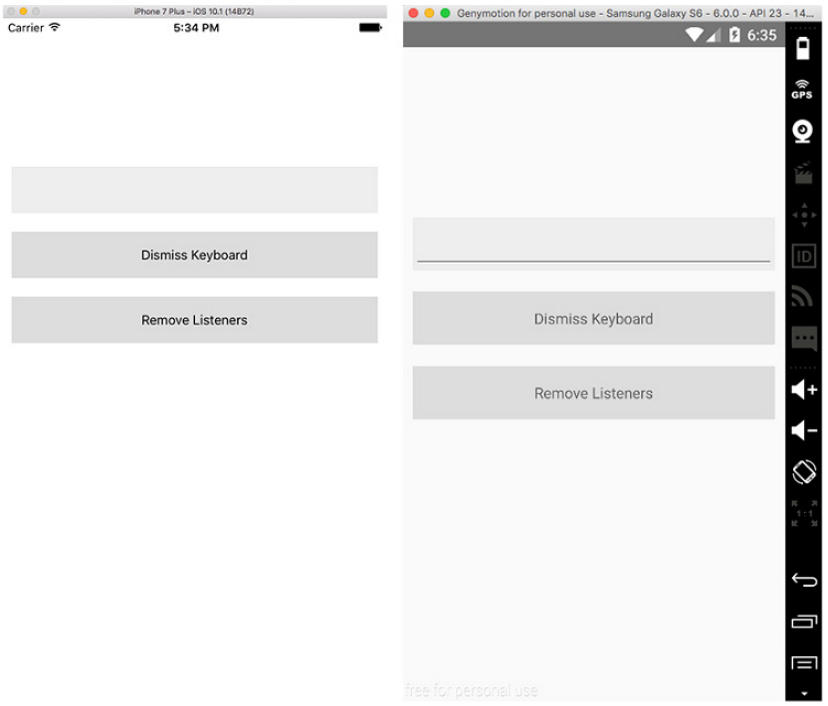


Figure 8.3 Keyboard

8.1.8 NetInfo

NetInfo is an API that allows us to access data describing whether the device is online of offline.

Both iOS and Android have different connectivity types (listing 8.14).

Table 8.6

iOS	Android
none	NONE
Wifi	BLUETOOTH
cell	DUMMY
unknown	ETHERNET
	MOBILE
	MOBILE_DUN

	MOBILE_HIPRI
	MOBILE_MMS
	MOBILE_SUPL
	VPN
	WIFI
	WIMAX
	UNKNOWN

To determine the connection, there are a few methods we can use (listing 8.15).

Table 8.7 NetInfo methods

Method	Arguments	Description
fetch	none – fetch().done(callback)	returns the reach of the connection
isConnectionExpensive	none -isConnectionExpensive()	returns promise which returns a Boolean, whether the connections is or is not expensive
isConnected	none – isConnected.fetch()	returns a promise which returns a Boolean, whether the device is or is not connected
addEventListener	removeEventListener(eventName, callback)	adds an event listener for the specified event
removeEventListener	removeEventListener(eventName, callback)	removes an event listener for the specified event

In our example, we will set up a NetInfo.fetch method to get the initial connection information, and then set up a listener to log out the current NetInfo if and when it changes (Listing 8.16).

Listing 8.10 NetInfo

```
import React, { Component }  from 'react'
import { TouchableHighlight, NetInfo, TextInput, View, Text, StyleSheet } from
      'react-native' (1)

let styles = {}

class App extends Component {
  constructor () {
    super()
    this.state = {     (2)
      connection: ''
    }
    this.handleConnectivityChange = this.handleConnectivityChange.bind(this)
  }
  componentDidMount () {
```

```
    NetInfo.fetch().done((connection) => {    ❸
      console.log('connection: ' + connection)
      this.setState({connection})
    })
    NetInfo.addEventListener('change', this.handleConnectivityChange) ❹
  }
  handleConnectivityChange (connection) {     ❺
    console.log('new connection:', connection)
    this.setState({connection})
  }
  render () {
    return (
      <View style={styles.container}>
        <Text>{this.state.connection}</Text> ❻
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  }
})
```

❶ import NetInfo from React Native
❷ set the initial state of connection to an empty string
❸ in componentDidMount, get the initial connection and reset the state
in componentDidMount, also create an event listener to call handleConnectivityChange when the connection changes
❺ in handleConnectivityChange, update the state with the new connection information
❻ render the connection information to the view

8.1.9 PanResponder

PanResponder offers a way to make use of data from touch events. With it, we can granularly respond to and manipulate our application state based on single and multiple touch events, such as swiping, tapping, pinching, scrolling, and more.

For example, let's take a look at a basic gesture event and see what type of data is available to us using `onPanResponderMove(event, gestureState)`, which gives us data about the current position of the touch event, including current position, accumulated difference between current position and original position among other things (listing 8.17).

Listing 8.11 PanResponder onPanResponderMove

```
onPanResponderMove(evt, gestureState) {
  console.log(evt.nativeEvent)
  console.log(gestureState)
}
```

Table 8.8 evt and gestureState properties

evt.nativeEvent	
changedTouches	Array of all touch events that have changed since the last event
identifier	The ID of the touch
locationX	The X position of the touch, relative to the element
locationY	The Y position of the touch, relative to the element
pageX	The X position of the touch, relative to the root element
pageY	The Y position of the touch, relative to the root element
target	The node id of the element receiving the touch event
timestamp	A time identifier for the touch, useful for velocity calculation
touches	Array of all current touches on the screen

Table 8.9

gestureState	
stateID	ID of the gestureState- persisted as long as there at least one touch on screen
moveX	the latest screen coordinates of the recently-moved touch
moveY	the latest screen coordinates of the recently-moved touch
x0	the screen coordinates of the responder grant
y0	the screen coordinates of the responder grant
dx	accumulated distance of the gesture since the touch started
dy	accumulated distance of the gesture since the touch started
vx	current velocity of the gesture
vy	current velocity of the gesture
numberActiveTouches	Number of touches currently on screen

To use the panResponder, we first need to create an instance of PanResponder in the componentWillMount method. In this instance we can then set all of our configuration and callback methods for the PanResponder, using the different methods to manipulate the state and the View.

Let's take a look at the `create` method, which is the only available method for PanResponder, and creates the configuration for the PanResponder (listing 8.19).

Listing 8.12 PanResponder create method

```
this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: (evt, gestureState) => {
        // determines whether to enable the PanResponder
        // gets called after the element is touched
        // returns boolean
      },
      onMoveShouldSetPanResponder: (evt, gestureState) => {
        // determines whether to enable the PanResponder
        // gets called after the initial touch has first move
        // returns boolean
      },
      onPanResponderReject: (evt, gestureState) => {
        // gets called if the PanResponder does not register
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderGrant: (evt, gestureState) => {
        // gets called if the PanResponder does regiser
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderStart: (evt, gestureState) => {
       // gets called after the PanResponder registers
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderEnd: (evt, gestureState) => {
        // gets called after the PanResponder has finished
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderMove: (evt, gestureState) => {
      // gets called when the PanResponder moves
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderTerminationRequest: (evt, gestureState) => {
        // gets called when something else wants to become responder
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderRelease: (evt, gestureState) => {
        // gets called when the touch has been released
        // do stuff with evt.nativeEvent or gesturestate
      },
      onPanResponderTerminate: (evt, gestureState) => {
        // this responder has been taken by another one
        // do stuff with evt.nativeEvent or gesturestate
      }
    })
```

For our example, we will create a draggable square, and will display the x and y coordinates of the square to our view (Listing 8.20).

Listing 8.13 PanResponder

```
import React, { Component }  from 'react'
import { Dimensions, TouchableHighlight, PanResponder, TextInput, View, Text,
      StyleSheet } from 'react-native' ❶
const { width, height } = Dimensions.get('window') ❷
```

```
let styles = {}

class App extends Component {
  constructor () {
    super()
    this.state = {
      oPosition: { ❸
        x: (width / 2) - 100,
        y: (height / 2) - 100,
      },
      position: { ❹
        x: (width / 2) - 100,
        y: (height / 2) - 100,
      },
    }
    this._handlePanResponderMove = this._handlePanResponderMove.bind(this)
    this._handlePanResponderRelease = this._handlePanResponderRelease.bind(this)
  }
  componentWillMount () { ❺
    this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: () => true,
      onPanResponderMove: this._handlePanResponderMove,
      onPanResponderRelease: this._handlePanResponderRelease
    })
  }
  _handlePanResponderMove (evt, gestureState) { ❻
    let ydiff = gestureState.y0 - gestureState.moveY
    let xdiff = gestureState.x0 - gestureState.moveX
    this.setState({
      position: {
        y: this.state.oPosition.y - ydiff,
        x: this.state.oPosition.x - xdiff
      }
    })
  }
  _handlePanResponderRelease () { ❼
    this.setState({
      oPosition: this.state.position
    })
  }
  render () {
    return (
      <View  style={styles.container}>
      <Text style={styles.positionDisplay}>x: {this.state.position.x}
      y:{this.state.position.y}</Text> ❽
      <View
          {...this._panResponder.panHandlers} ❾
          style={[styles.box, { marginLeft: this.state.position.x, marginTop:
      this.state.position.y } ]} />     ❿
      </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
```

```
  },
  positionDisplay: {
    textAlign: 'center',
    marginTop: 50,
    zIndex: 1,
    position: 'absolute',
    width
  },
  box: {
    position: 'absolute',
    width: 200,
    height: 200,
    backgroundColor: 'red'
  }
})
```

1. import Dimensions, PanResponder, and everything else we will be needing for this component
2. store the window width and height in variables for later use
3. create an object to store our original square position x and y axes to center the square, called oPosition, and store into the state
4. create an object to store our actual square position x and y axes to center the square, called position, and store into the state
5. create a new PanResponder, returning true for onStartShouldSetPanResponder, and setting up onPanResponderMove and onPanResponderRelease methods
6. in onPanResponderMove, we calculate the total movements of both x and y by calculating the difference between the location of the panResponderGrant and the current total of movement since the grant. We then update the state position with these values
7. In handlePanResponderRelease, we set the state of oPosition with the updated position we are using in our view
8. We display the current position values in our view
9. We attach the created PanResponder to our view by passing in {...this._panResponder.panHandlers} as props
10. We attach the position x and y values to our view to update the margins, making the item draggable

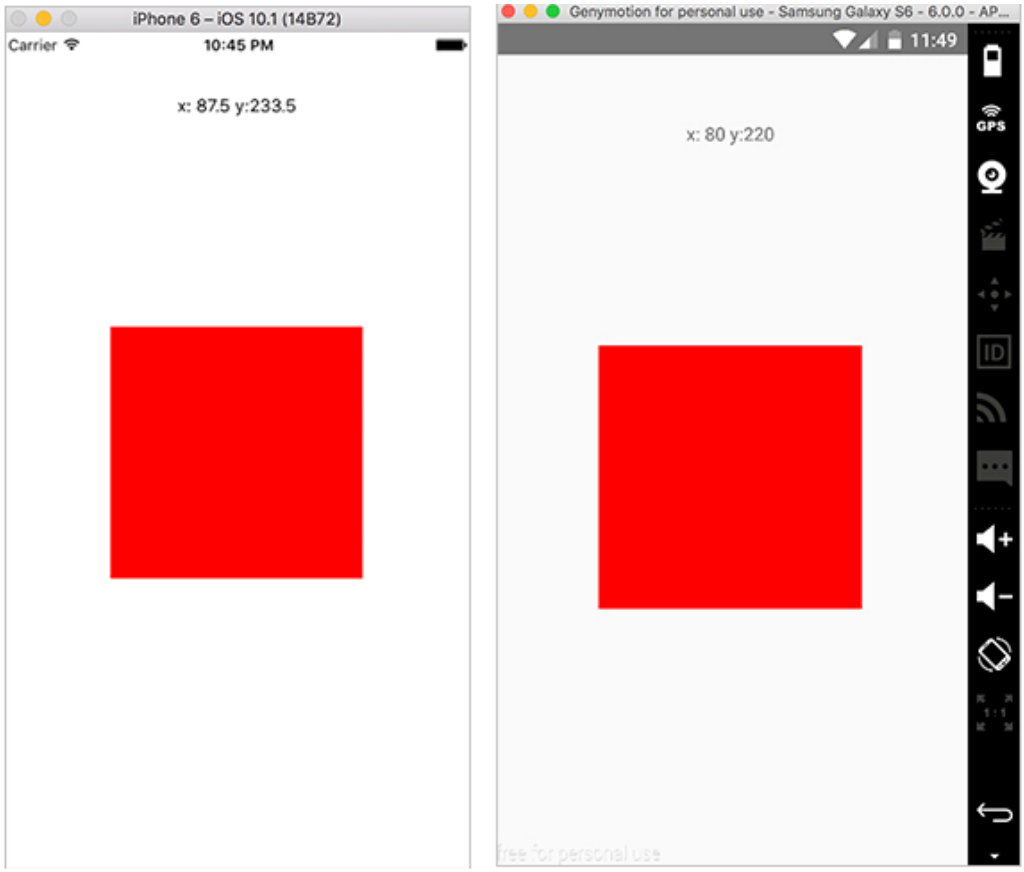


Figure 8.4 PanResponder

9

iOS-specific Components and APIs

This chapter covers

- How to implement and use iOS-specific APIs
- Strategies for how to effectively target platform-specific code
- Using DatePickerIOS to choose and save dates in your app
- How to use PickerIOS to choose from a list of data
- Showing loading progress using ProgressViewIOS
- Choosing between options using SegmentedControlIIOS
- Creating and switching between tabs using TabBarIOS
- Calling and choosing items in an action sheet using ActionSheetIOS

9.1 Implementing iOS-specific APIs and Components

One of the end goals of the React Native project is to have a minimal amount of platform specific logic and code. Most APIs can be built so that the platform-specific code is abstracted away by the framework, giving us a single way to interact with them and easily create cross-platform functionality.

Unfortunately, there will always be platform-specific APIs that cannot be completely abstracted away in a way that makes sense cross-platform. Therefore, we will have at least a handful of platform specific APIs and components we will need to use.

In this chapter, we will cover iOS specific APIs and Components, discuss their props and methods, and create examples that will mimic functionality and logic that will get you up to speed quickly.

9.2 Targeting platform-specific code

The main idea of platform-specific code is writing components and files in a way that render iOS- or Android-specific code based on the platform we are on.

There are a few techniques that can be implemented to show components based on what platform the app is running, and we will cover the most useful of those techniques here.

9.2.1 iOS and Android file extension

The first way to target platform-specific code is by simply naming the file with the correct file extension depending on the platform you wish to target.

For example, one component that differs quite a bit between iOS and Android is the DatePicker. If we wanted to have specific styling around our DatePicker, writing all this code in our main component may become verbose and difficult to maintain. Instead, we will create two files: DatePicker.ios.js and DatePicker.android.js and import them into our main component. When we run the project, React Native will automatically choose the correct file and render it based on the platform we are using.

Let's look at a basic example. Note, these are just examples (listings 9.1 and 9.2) and will throw an error as is because `DatePicker` requires both props and methods to function correctly, so this is more of a broad overview of how to implement this type of functionality, we will be going over the `DatePicker` soon in this chapter (listings 9.1, 9.2, and 9.3).

Listing 9.1 iOS platform specific code

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

export default () => (
  <View>
    <Text>This is an iOS specific component</Text>
    <DatePickerIOS />
  </View>
)
```

Listing 9.2 Android platform specific code

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

export default () => (
  <View>
    <Text>This is an Android specific component</Text>
    <DatePickerAndroid />
  </View>
)
```

Listing 9.3 Rendering the cross-platform component

```
import React from 'react'
import DatePicker from './DatePicker'
```

```
const MainComponent = () => (
  <View>
    ...
    <DatePicker />
    ...
  </View>
)
```

In listing 9.3, we import the Datepicker without giving a specific file extension. React Native knows which component to import depending on the platform. From there, we are then able to use it in our application without having to worry about which platform we are on.

9.2.2 Detecting platform using the Platform API

Another way to detect and perform logic based on the platform is to use the Platform API. Platform has two properties. The first is an OS key (standing for Operating System) that reads either `ios` or `android`, depending on the platform (listing 9.4).

Listing 9.4 Platform module detecting using Platform.OS property

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

const PlatformExample = () => (
  <Text style={{ marginTop: 100, color: Platform.OS === 'ios' ? 'blue' : 'green' }}
  >
     Hello { Platform.OS }
  </Text>
)
```

In listing 9.4, we check to see if the value of Platform.OS is equal to the string 'ios', and if it is, we return a color of blue. If it is not, we return green.

The second is a method called `select`. `select` takes in an object containing the platform.OS strings as keys (either ios or android), and returns the value for the platform you are running (listing 9.5).

Listing 9.5 Using Platform.select to render components based on Platform

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

const ComponentIOS = () => (
  <Text>Hello from IOS</Text>
)

const ComponentAndroid = () => (
  <Text>Hello from IOS</Text>
)

const Component = Platform.select({
  ios: () => ComponentIOS,
```

```
  android: () => ComponentAndroid,
})();


const PlatformExample = () => (
  <View style={{ marginTop: 100 }}>
    <Text>Hello from my App</Text>
    <Component />
  </View>
)
```

We can also use the ES2015 spread syntax to return objects, and use those objects to apply styling (listing 9.6).

Listing 9.6 Using Platform.select to render styles based on Platform

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

let styles = {}

const PlatformExample = () => (
  <View style={styles.container}>
    <Text>
       Hello { Platform.OS }
    </Text>
  </View>
)

styles = {
  container: {
    marginTop: 100,
    ...Platform.select({
      ios: {
        backgroundColor: 'red'
      }
    })
  }
}
```

9.3 DatePickerIOS

DatePickerIOS provides an easy way to implement a native date picker component on iOS.

`DatePickerIOS` has three modes that come in handy when working with dates and times: `date`, `time`, and `dateTime`.

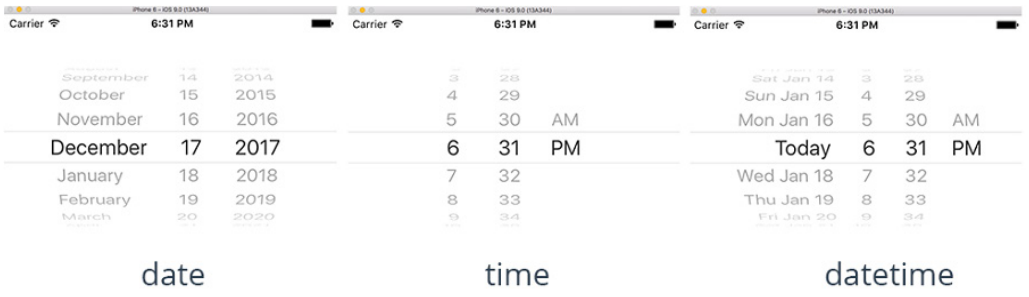


Figure 9.1 DatePicker with date mode, time mode, and datetime mode

The minimum props that need to be passed to the `DatePickerIOS` component are date (the date that is the beginning or current date choice), and an `onDateChange` method.

When any of the date values are changed, `onDateChange` is called, passing the function the new date value.

In our example, we will set up a `DatePickerIOS` component and display the time in the View. We will not pass in a mode prop, as the mode defaults to `datetime` (Listing 9.7).

Listing 9.7 Using DatePicker to show and update time values.

```
import React, { Component } from 'react'
import { Text, View, DatePickerIOS } from 'react-native' (1)

class App extends Component {

  constructor() {
    super()
    this.state = {
      date: new Date(), (2)
    }
    this.onDateChange = this.onDateChange.bind(this)
  }

  onDateChange(date) {  (3)
    this.setState({date: date});
  };

  render() {
    return (
      <View style={{ marginTop: 50 }}>
        <DatePickerIOS  (4)
          date={this.state.date}
          onDateChange={this.onDateChange}
        />
        <Text style={{ marginTop: 40, textAlign: 'center' }}>
          { this.state.date.toLocaleDateString() } {
       this.state.date.toLocaleTimeString() } (5)
        </Text>
```

```
      </View>)
  }
}
```

1. import DatePickerIOS from React Native
2. create a date value and store it in the state
3. create a method called onDateChange that updates the state with the new date value
4. return the DatePickerIOS component and pass In both the date and the onDateChange method as props
5. render the date value as text

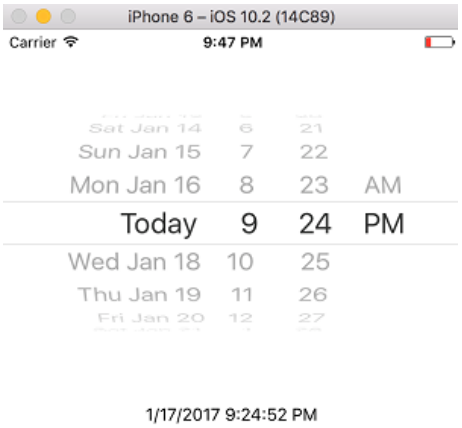


Figure9.2 DatePickerIOS rendering chosen date and time

Table 9.1 DatePickerIOS methods and arguments

prop	Type	Description (some from docs)
date	Date	currently selected date
maximumDate	Date	maximum allowed date
minimumDate	Date	minimum allowed date
minuteInterval	enum	the interval at which minutes can be selected
mode	string (date, time, or datetime)	date picker mode
onDateChange	function `onDateChange(date) { }`	function called when date changes
timeZoneOffsetInMinutes	number	timezone offset in minutes. Will override the default which is the device timezone.

9.4 PickerIOS

`PickerIOS` allows us to access the native IOS Picker component, which basically allows us to scroll through and choose from a list of values using the Native UI.

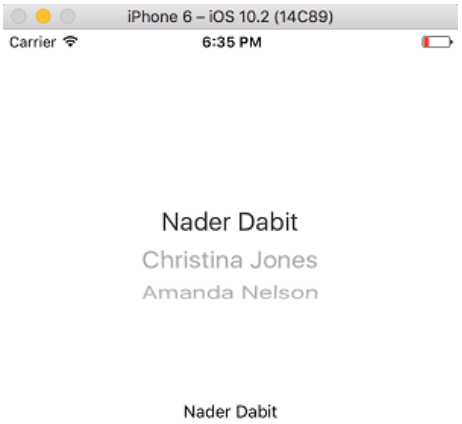


Figure 9.3 PickerIOS rendering list of people

`PickerIOS` wraps a list of items to be rendered as children. Each child item must be a `PickerIOS.Item`.

```
import { PickerIOS } from 'react-native'
const PickerItem = PickerIOS.Item

<PickerIOS>
  <PickerItem />
  <PickerItem />
  <PickerItem />
</PickerIOS>
```

It is possible to declare each `PickerIOS.Item` individually as we have done above, but most of the time you will be mapping over elements in an array and returning a `PickerIOS.Item` for each item in the array (listing 9.8).

Listing 9.8 Example of using PickerIOS with an array of PickerIOS.Items

```
const people = [ // an array of people ];

render() {
  <PickerIOS>
    {
      people.map((p, i) =>(
```

```
        <PickerItem />
      ))
    }
  </PickerIOS>
}
```

Both `PickerIOS` and `PickerIOS.Item` receive their own props.

For `PickerIOS`, the main props are `onValueChange` and `selectedValue`. The `onValueChange` method is called whenever the picker is changed. The `selectedValue` is the value that the picker shows as selected in the UI.

For `PickerIOS.Item`, the main props are `key`, `value`, and `label`. `key` is a unique identifier that is required as it is in an array, `value` is what will be passed to the `onValueChange` method of the `PickerIOS` component, and `label` is what is displayed in the ui as the label for the `PickerIOS.Item`.

In our example, we will render an array of people in the `PickerIOS`, and when the value changes, update the UI to show the new value (Listing 9.9).

Listing 9.9 Using PickerIOS to render an array of People

```
import React, { Component } from 'react'
import { Text, View, PickerIOS } from 'react-native' (1)

const people = [ (2)
  {
    name: 'Nader Dabit',
    age: 36
  },
  {
    name: 'Christina Jones',
    age: 39
  },
  {
    name: 'Amanda Nelson',
    age: 22
  }
];

const PickerItem = PickerIOS.Item

class App extends Component {

  constructor() {
    super()
    this.state = { (3)
      value: 'Christina Jones'
    }
    this.onValueChange = this.onValueChange.bind(this)
  }

  onValueChange(value) { (4)
    this.setState({ value });
  };
```

```
  render() {
    return (
      <View style={{ marginTop: 50 }}>
        <PickerIOS ❺
          onValueChange={this.onValueChange}
          selectedValue={this.state.value}
        >
          {
            people.map((p, i) => { ❻
              return (
                <PickerItem
                  key={i}
                  value={p.name}
                  label={p.name}
                />
              )
            })
          }
        </PickerIOS>
        <Text style={{ marginTop: 40, textAlign: 'center' }}>
          {this.state.value} ❼
        </Text>
    </View>)
  }
}
```

❶ **import** `PickerIOS` **from React Native**
❷ **create an array of people. We will use this to populate our** `PickerItem` **values**
❸ **create an initial value in the state to hold the chosen picker value**
❹ **create an** `onValueChange` **method that will update the state value with the new value from our** `PickerIOS`
❺ **render the** `PickerIOS` **and pass the** `onValueChange` **method and the** `selectedValue` **as props**
❻ **render a** `PickerIOS.Item` **for every person in the people array**
❼ **render the** `value of this.state.value` **in the UI**

Table 9.2 PickerIOS methods and props

prop	Type	Description (some from docs)
itemStyle	object (style)	style DatePickerIOS container
onValueChange	function(value)	this method is called when the PickerIOS value changes
selectedValue	number or string	currently selected PickerIOS value

9.5 ProgressViewIOS

ProgressViewIOS is a way for us to render the native UIProgressView in our UI. ProgressViewIOS basically is a native way for us to show loading percentage indication, download percentage indication, or for any time we need to show an indication of a task that is being completed.

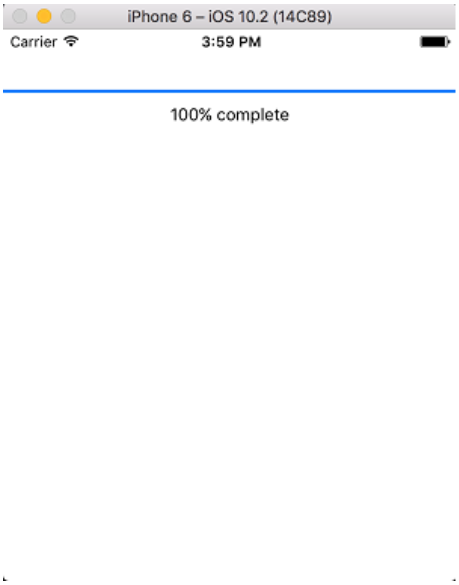


Figure 9.4 Rendering ProgressViewIOS in the UI

The main prop that we need to know about to create this functionality is the `progress` prop. `progress` takes a number between 0 and 1 and fills the `ProgressViewIOS` with a percentage fill between 0% and 100%.

In our example, we will simulate some data loading by setting a `setInterval` method that gets called in `componentDidMount`, and increment the state value by 0.01 every 0.01 seconds until we are at 1, starting the initial value at 0 (listing 9.10).

Listing 9.10

```
import React, { Component } from 'react'
import { Text, View, ProgressViewIOS } from 'react-native' (1)

class App extends Component {

  constructor() {
    super()
    this.state = { (2)
      progress: 0,
    }
  }

  componentDidMount() {
    this.interval = setInterval(() => { (3)
      if (this.state.progress >= 1) {
        return clearInterval(this.interval)
      }
```

```
      this.setState({
        progress: this.state.progress + .01
      })
    }, 10)
  }

  render() {
    return (
      <View style={{ marginTop: 50 }}>
        <ProgressViewIOS ❹
          progress={this.state.progress}
        />
      <Text style={{ marginTop: 10, textAlign: 'center' }}>
          {Math.floor(this.state.progress * 100)}% complete ❺
      </Text>
    </View>)
  }
}
```

❶ **import** `ProgressVIewIOS` **from React Native**
❷ **create initial state value of progress, set to zero**
❸ **in** `componentDidMount`**, store a** `setInterval` **method in a variable, and increment the state value of progress every 1/100 of a second by .01. If** `this.state.progress` **is greater than or equal to 1, we clear and cancel the interval by calling** `clearInterval` **and return.**
❹ **render the** `ProgressViewIOS`**, passing in** `this.state.progress` **as the** `progress` **prop**
❺ **round and render the value** `of this.state.progress` **in the UI**

Table 9.3 ProgressViewIOS methods and props

prop	Type	Description (some from docs)
progress	number	the progress value (between 0 and 1).
progressImage	image source	a stretchable image to display as the progress bar.
progressTintColor	string (color)	the tint color of the progress bar itself
progressViewStyle	enum (default or bar)	the progress bar style
trackImage	image source	a stretchable image to display behind the progress bar
trackTintColor	string	the tint color of the progress bar track

9.6 SegmentedControlIOS

SegmentedControlIOS allows us to access the native iOS UISegmentedControl component.

SegmentedControlIOS is basically a horizontal tab bar that is made up of individual buttons (figure 9.5).

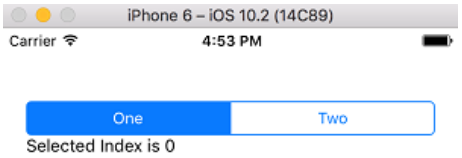


Figure 9.5 Basic SegmentedControlIOS implementation with 2 values (one and two).

At a minimum, `SegmentedControlIOS` takes an array of values to render the control values, a `selectedIndex` as the index of the control that is selected, and an `onChange` method that will be called when a control is pressed.

In our example, we will take an array of three items and render them as a SegmentedControlIOS. We will also show a value in our UI based on which item is selected (listing 9.11).

Listing 9.11 SegmentedControlIOS rendering three values

```
import React, { Component } from 'react'
import { Text, View, SegmentedControlIOS } from 'react-native' (1)

const values = ['One', 'Two', 'Three']       (2)

class App extends Component {

  constructor() {
    super()
    this.state = {
      selectedIndex: 0,      (3)
    }
  }

  render() {
    const { selectedIndex } = this.state
    let selectedItem = values[selectedIndex] (4)
    return (
      <View style={{ marginTop: 40, padding: 20 }}>
        <SegmentedControlIOS (5)
          values={values}
          selectedIndex={this.state.selectedIndex}
          onChange={(event) => {
            this.setState({selectedIndex: event.nativeEvent.selectedSegmentIndex});
          }}
```

```
        />
      <Text>{selectedItem}</Text> ❻
    </View>)
  }
}
```

❶ import `SegmentedControlIOS` from React Native
❷ create an array of values to use in the `SegmentedControlIOS`
❸ create a state value of `selectedIndex` set to zero
❹ create a variable called `selectedItem`, set to the value of the `selectedIndex` of the values array
❺ render the `SegmentedControlIOS` component, passing in the values array as the values prop, `this.state.seletedIndex` as the `selectedIndex`, and an `onChange` method that updates the `selectedIndex` state value with the index of the pressed item.
❻ render the value of `selectedItem` in the UI

Table 9.4 SegmentedControlIOS methods and props

prop	Type	Description (some from docs)
enabled	Boolean	If false the user won't be able to interact with the control. Default value is true.
momentary	Boolean	if true, then selecting a segment won't persist visually. The `onValueChange` callback will still work as expected.
onChange	function (event)	Callback that is called when the user taps a segment; passes the event as an argument
onValueChange	function (value)	Callback that is called when the user taps a segment; passes the segment's value as an argument
selectedIndex	number	The index in props.values of the segment to be (pre)selected.
tintColor	string (color)	Accent color of the control.
values	array of strings	The labels for the control's segment buttons, in order.

9.7 TabBarIOS

TabBarIOS allows us to access the native iOS Tab Bar. TabBarIOS renders tabs at the bottom of the UI, allowing a nice and easy way to separate your application into sections.

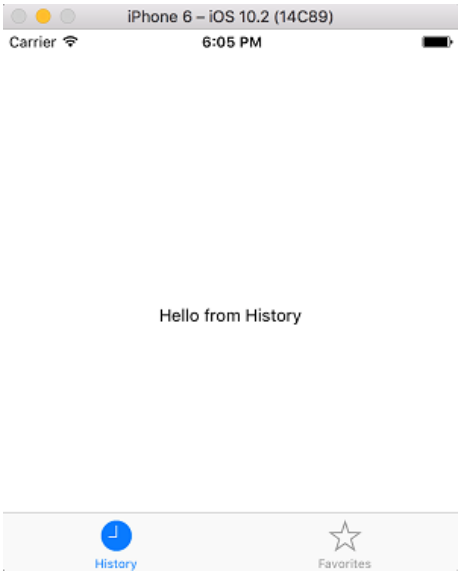


Figure 9.6 TabBarIOS with two tabs: History and Favorites

`TabBarIOS` takes a list of `TabBarIOS.Item` components as children.

```
const Item = TabBarIOS.Item

<TabBarIOS>
  <Item>
    <View> // some content here </View>
  </Item>
  <Item>
    <View> // some other content here </View>
  </Item>=
</TabBarIOS>
```

To show the content within the TabBarIOS.Item, the selected prop of the TabBarIOS.Item must be true.

```
<Item
  selected={this.state.selectedComponent === 'home'}
>
  // your content here
</Item>
```

In our example, we will create an app with two views: History and Favorites. When the `TabBarIOS.Item` is clicked, we will switch between views by calling an `onPress` method to update the state (Listing 9.12).

Listing 9.12 Rendering tabs using TabBarIOS

```
import React, { Component } from 'react'
import { Text, View, TabBarIOS } from 'react-native' (1)

const Item = TabBarIOS.Item   (2)

class App extends Component {

  constructor() {
    super()
    this.state = {
      selectedTab: 'history', (3)
    }
    this.renderView = this.renderView.bind(this)
  }

  renderView(tab) {           (4)
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <Text>Hello from {tab}</Text>
      </View>
    )
  }

  render() {
    return (
      <TabBarIOS> (5)
        <Item
          systemIcon="history" (6)
          onPress={() => this.setState({ selectedTab: 'history' })} (7)
          selected={this.state.selectedTab === 'history'}
          >
          {this.renderView('History')} (8)
        </Item>
        <Item
          systemIcon='favorites'
          onPress={() => this.setState({ selectedTab: 'favorites' })}
          selected={this.state.selectedTab === 'favorites'}
        >
          {this.renderView('Favorites')}
        </Item>
      </TabBarIOS>
    )
  }
```

(1) **import** `TabBarIOS` **from React Native**
(2) **create a variable called Item to hold the** `TabBarIOS.Item` **component**
(3) **create an initial state value of** `selectedTab` **and set it to history**
(4) **create a reusable** `renderView` **method that takes in a tab as an argument**
(5) **render a** `TabBarIOS` **in our UI, passing in two Item components as children**
(6) **set the systemIcon prop to history (see https://developer.apple.com/ios/human-interface-guidelines/graphics/system-icons/ for all system icons). Icons either can be set with a system icon or by passing in an icon prop and requiring a local image.**
(7) **attach an** `onPress` **method to the item, updating the** `selectedTab` **value in the state with the value passed in to** `this.setState({})`

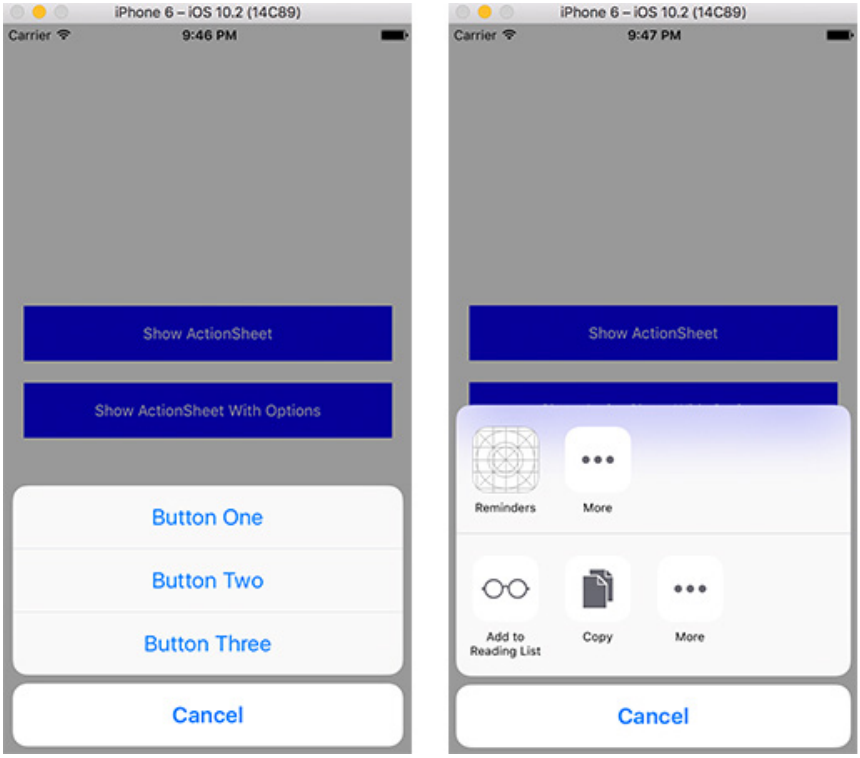
❽ render the view by calling `this.renderView` method

Table 9.5 TabBarIOS methods and props

Prop	Type	Description (some from docs)
barTintColor	String (color)	Background color of the tab bar
itemPositioning	enum('fill', 'center', 'auto')	Specifies tab bar item positioning. Available values are: - fill - distributes items across the entire width of the tab bar - center - centers item in the available tab bar space - auto (default) - distributes items dynamically according to the user interface idiom. In a horizontally compact environment (e.g. iPhone 5) this value defaults to fill, in a horizontally regular one (e.g. iPad) it defaults to center.
style	object (style)	style of the TabBarIOS
tintColor	string (color)	Color of the currently selected tab icon
translucent	Boolean	A Boolean value that indicates whether the tab bar is translucent
unselectedItemTintColor	string (color)	Color of unselected tab icons. Available since iOS 10.
unselectedTintColor	string (color)	Color of text on unselected tabs

9.8 ActionSheetIOS

ActionSheetIOS allows us to access the native iOS UIAlertController to show a native iOS action sheet or share sheet.



Listing 9.7 ActionSheetIOS rendering an action sheet (left) and a share sheet (right)

The two main methods that we can call on `ActionSheetIOS` are `showActionSheetWithOptions` or `showShareActionSheetWithOptions`.

`showActionSheetWithOptions`lets us pass an array of buttons, and attach methods to each of the buttons. `showActionSheetWithOptions` gets called with two arguments: `options` object and a callback function.

```
showActionSheetWithOptions(options, callback);
```

`showShareActionSheetWithOptions` will allow us to display the native iOS share sheet, passing in a url, message, and subject to share. `showShareActionSheetWithOptions` gets called with three arguments: `options` object, a failure callback function, and a success callback function.

```
showShareActionSheetWithOptions(options, failureCallback, successCallback)
```

In our example, we will create a View with two buttons. One button will call `showActionSheetWithOptions` and the other will call `showShareActionSheetWithOptions`(listing 9.13).

Listing 9.13 Using ActionSheetIOS to create action sheets and share sheets.

```
import React, { Component } from 'react'
import {Text, View, ActionSheetIOS, TouchableHighlight } from 'react-native' (1)

const BUTTONS = ['Cancel', 'Button One', 'Button Two', 'Button Three']       (2)

class App extends Component {
  constructor() {
    super()
    this.state = {     (3)
      clicked: null
    }
    this.showActionSheet = this.showActionSheet.bind(this)
    this.showShareActionSheetWithOptions =
      this.showShareActionSheetWithOptions.bind(this)
  }

  showActionSheet() { (4)
    ActionSheetIOS.showActionSheetWithOptions({
      options: BUTTONS,
      cancelButtonIndex: 0,
    },
    (buttonIndex) => {
      if (buttonIndex > 0) {
        this.setState({ clicked: BUTTONS[buttonIndex] });
      }
    });
  }

  showShareActionSheetWithOptions() { (5)
    ActionSheetIOS.showShareActionSheetWithOptions({
      url: 'http://www.reactnative.training',
      message: 'React Native Training',
    },
    (error) => console.log('error:', error),
    (success, method) => {
      if (success) {
        console.log('successfully shared!', success)
      }
    });
  };
  render() {
    return (
      <View style={styles.container}> (6)
        <TouchableHighlight onPress={this.showActionSheet} style={styles.button}>
          <Text style={styles.buttonText}>Show ActionSheet</Text>
        </TouchableHighlight>
        <TouchableHighlight onPress={this.showShareActionSheetWithOptions}
      style={styles.button}>
        <Text style={styles.buttonText}>Show ActionSheet With Options</Text>
          </TouchableHighlight>
        <Text>
          {this.state.clicked}
        </Text>
      </View>
    )
```

```
  }
}

styles = {
  container: {
    flex: 1,
    justifyContent: 'center',
    padding: 20,
  },
  button: {
    height: 50,
    marginBottom: 20,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'blue'
  },
  buttonText: {
    color: 'white'
  }
}
```

1. import `ActionSheetIOS` from React Native
2. create an array of buttons for us to use in the action sheet
3. create a variable clicked and set to null
4. create `showActionSheet` method.
   - pass in buttons as the options
   - set the `cancelButtonIndex` to zero (this will position Cancel at the bottom of the action sheet)
   - create callback method that takes in the button index as an argument.
   - If the button index is greater than zero, we set the clicked state value with the new button value
5. create `showShareActionSheetWithOptions` method
   - pass in `url` and `message` to share
   - in first callback function, check to see if there is an error. If so, log out the error
   - in second callback method, check to see if success is true, and if so log out 'successfully shared!'
6. create two buttons in our View, and attach the `showActionSheet` and `showShareActionSheetWithOptions` to them

Table 9.5 ActionSheetIOS showActionSheetWithOptionsoptions

Option	Type	Description (some from docs)
options	array of strings	a list of button titles (required)
cancelButtonIndex	integer	index of cancel button in options
destructiveButtonIndex	integer	index of destructive button in options
title	string	a title to show above the action sheet
message	string	a message to show below the title

Table 9.6 ActionSheetIOS showShareActionSheetWithOptions options

Option	Type	Description (some from docs)
url	string	a URL to share
message	string	a message to share
subject	string	a subject for the message
excludedActivityTypes	array	the activities to exclude from the ActionSheet

9.9 Summary

- Using platform android.js and ios.js extensions to import cross-platform files
- Using the Platform API to render platform specific code
- Implementing DatePickerIOS to choose and save dates in your app
- Using PickerIOS to render and save values from a list
- Using ProgressViewIOS to show loading progress
- Using SegmentedControlIOS to choose from an array of options
- Using TabBarIOS to create and switch between tabs in your app
- How ActionSheetIOS allows us to call either a native iOS action sheet or share sheet in an app

10

Android-specific components and APIs

This chapter covers

- How to implement and use Android-specific APIs
- Using DrawerLayoutAndroid to create a side menu
- Creating a native toolbar with ToolbarAndroid
- Create paging views using ViewPagerAndroid
- Using DatePickerAndroid to create and manage dates in your application
- Managing time with TimePickerAndroid
- Creating toasts using ToastAndroid

In this chapter, we will cover Android specific APIs and components, discuss their props and methods, and create examples that will mimic functionality and logic that will get you up to speed quickly.

We will do so by creating a demo app that will show off these Android specific APIs and components.

10.1 Creating the menu using DrawerLayoutAndroid

To get started, we will first create a slide out menu that will link to each of these pieces of functionality. We will create this menu using the `DrawerLayoutAndroid` component.

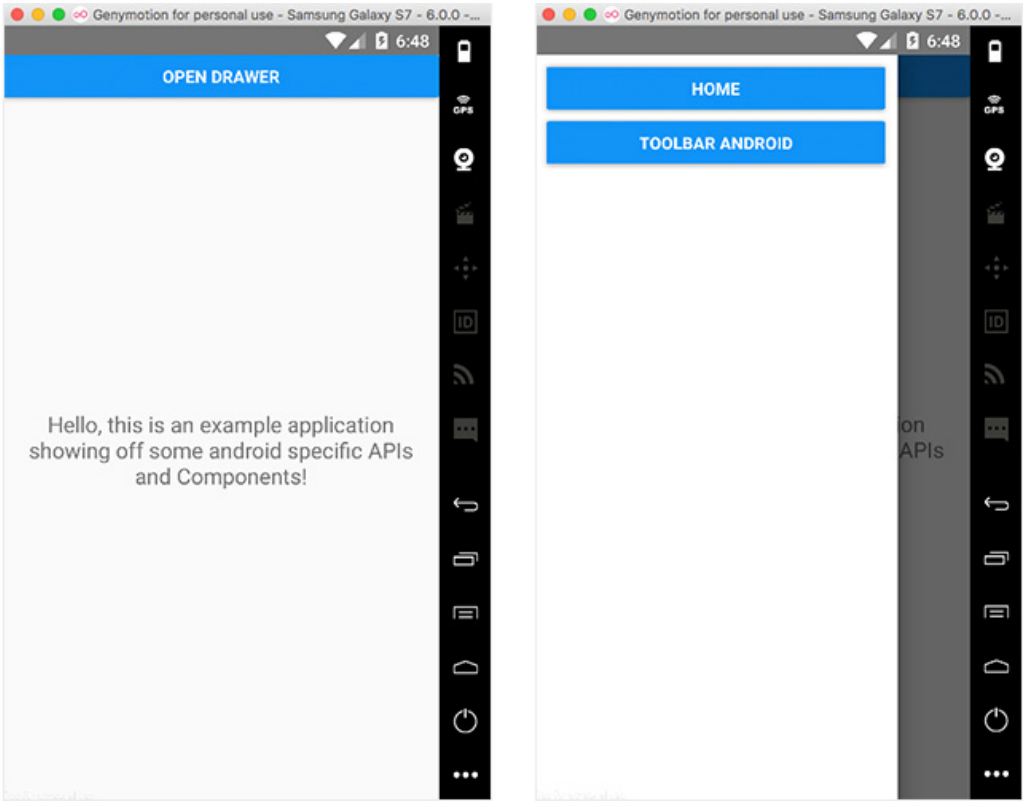


Figure 10.1 Initial layout of our application using DrawerLayoutAndroid

The first thing we need to do is create a new android application. From your command line in the folder that you will be working in, create a new application with `YourApplication` being the name of the application you are choosing.

```
react-native init YourApplication
```

The next thing we need to do is create the files we will be using. In the root of the application, create a folder named `app` and three files in this folder named `App.js`, `Home.js`, `Menu.js`, and `Toolbar.js`.

The first thing we need to do is update index.android.js to use our first Android specific API, `DrawerLayoutAndroid`, which is the sliding toolbar from the left (figure 10.1).

To get started, let's edit index.android.js to include and implement this component.

Listing 10.1 index.android.js

```
import React from 'react'
import {
```

```
  AppRegistry,
  DrawerLayoutAndroid,          ❶
  Button,
  View
} from 'react-native'

import Menu from './app/Menu'   ❸
import App from './app/App'     ❷

class chapter10 extends React.Component {

  constructor () {
    super()
    this.state = {
      scene: 'Home'             ❸
    }
    this.jump = this.jump.bind(this)
    this.openDrawer = this.openDrawer.bind(this)
  }

  openDrawer () {
    this.drawer.openDrawer()    ❹
  }

  jump (scene) {                ❺
    this.setState({
      scene
    })
    this.drawer.closeDrawer()
  }

  render () {
    return (
      <DrawerLayoutAndroid      ❻
        ref={drawer => this.drawer = drawer}                        ①
        drawerWidth={300}                                           ②
        drawerPosition={DrawerLayoutAndroid.positions.Left}         ③
        renderNavigationView={() =><Menu onPress={this.jump} />}> ④
      <View style={{ margin: 15 }}>                                 ⑤
      <Button onPress={() => this.openDrawer()} title='Open Drawer' />
      </View>
      <App                                                          ⑤
          openDrawer={this.openDrawer}
          jump={this.jump
          scene={this.state.scene} />
      </DrawerLayoutAndroid>
    )
  }
}

AppRegistry.registerComponent('chapter10', () => chapter10)
```

❶ **import `DrawerLayoutAndroid` from React Native**
❷ **import the yet to be created App component**
❸ **create a component state setting scene to 'Home'**
❹ **create a method to open the Drawer**
❺ **create a method to update the scene state, and then call `closeDrawer()`**

6. implement the `DrawerLayoutAndroid` component
1. create a reference to the drawer to call methods on the component
2. give the drawer a width of 300
3. position the drawer to the left
4. render the navigation view, which is a Menu component we have yet to create
5. pass in a button as a child and attach the jump method to it. We will use this across the application to open the drawer. We also pass in the App component as a child, giving the `openDrawer`, `jump`, and `scene` as props.

Next, we will need to create the menu we will be using in the drawer. In app/Menu.js, create the following component (listing 10.2).

Listing 10.2 Menu.js - Creating the DrawerLayoutAndroid menu

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

let styles

const Menu = ({onPress }) => {
  const {
    button
  } = styles

  return (
    <View style={{ flex: 1 }}>
      <View style={button} >
        <Button onPress={() => onPress('Home')} title='Home' />
      </View>
      <View style={button} >
        <Button onPress={() => onPress('Toolbar')} title='Toolbar Android' />
      </View>
    </View>
  )
}

styles = StyleSheet.create({
  button: {
    margin: 10,
    marginBottom: 0
  }
})

export default Menu

Next, let's create the App component that will render based on the scene prop that is
      passed to it.
<App
    openDrawer={this.openDrawer}
    jump={this.jump
    scene={this.state.scene} />
```

In app/App.js, create the following component (listing 10.3).

Listing 10.3 app/App.js

```
import React from 'react'

import Home from './Home'                ❶
import Toolbar from './Toolbar'          ❷

function getScene (scene) {              ❸
  switch (scene) {
    case 'Home':
      return Home
    case 'Toolbar':
      return Toolbar
    default:
      return Home
  }
}

const App = (props) => {
  const Scene = getScene(props.scene) ❹
  return (
    <Scene openDrawer={props.openDrawer} jump={props.jump} /> ❺
  )
}

export default App
```

❶ import the Home component that we have yet to create
❷ import the Toolbar component that we have yet to create
❸ create a `getScene` method that will check the scene and return the correct component
❹ create a component based on the current scene prop
❺ render the component, passing in openDrawer and jump as props

Now that everything is pretty much all set up, we can start creating components to interact with the menu. For our current setup to work, we need to create a Home and a Toolbar component.

In app/Home.js, create the following component (listing 10.4).

Listing 10.4 app/Home.js

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

let styles

class Home extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Text
          style={styles.text}>
```

```
          Hello, this is an example application showing off some android specific
      APIs and Components!
      </Text>
     </View>
    )
  }
}

styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  text: {
    margin: 20,
    textAlign: 'center',
    fontSize: 18
  }
})

export default Home
```

And in app/Toolbar.js, create the following component (listing 10.5).

Listing 10.5 app/Toolbar.js

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

class ToolBar extends React.Component {
  render () {
    return (
      <View style={{ flex: 1 }}>
        <Text>Hello from Toolbar</Text>
      </View>
    )
  }
}

export default ToolBar
```

Now, we should be able to start the application and see the toolbar as seen in figure 10.1.

10.2 ToolbarAndroid

Now that everything is set up, let's add a new component, ToolbarAndroid. ToolbarAndroid is a React Native component that wraps the native Android Toolbar. This component can display a variety of things, including a title, subtitle, log, and navigation icon.

In our example, we will implement `ToolbarAndroid` with a title, subtitle, and two actions (Options and Menu). When Menu is clicked, we will trigger the `openDrawer` method that we have available as a prop.

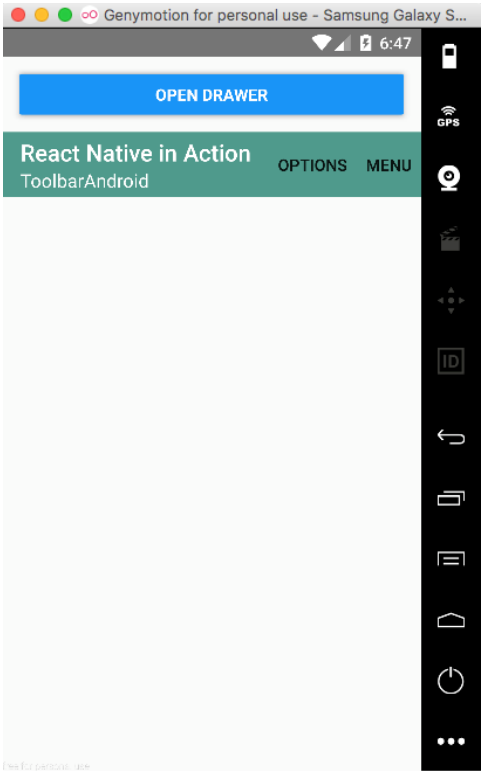


Figure 10.2 ToolbarAndroid with title, subtitle, and two actions

In app/Toolbar.js, update our code to the following (listing 10.6).

Listing 10.6 app/Toolbar.js – ToolbarAndroid implementation.

```
import React from 'react'
import {
  ToolbarAndroid, (1)
  View
} from 'react-native'

class Toolbar extends React.Component {
  render () {
    const onActionSelected = (index) => { (2)
      if (index === 1) {
        this.props.openDrawer()
```

```
      }
    }

    return (
      <View style={{ flex: 1 }}>
        <ToolbarAndroid          ❸
          subtitleColor='white'                  ①
          titleColor='white'                     ②
          style={{ height: 56, backgroundColor: '#12998c' }} ③
          title='React Native in Action'         ④
          subtitle='ToolbarAndroid'              ⑤
          actions={[ { title: 'Options', show: 'always' }, { title: 'Menu', show:
      'always' } ]}              ⑥
          onActionSelected={onActionSelected}  ⑦
      />
      </View>
    )
  }
}

export default Toolbar
```

❶ **import the ToolbarAndroid component**
❷ **create an `onActionSelected` method. This method takes in an index, and will call `this.props.openDrawer` if the index is one. We will later have an array of actions, each action will call this method when clicked, passing in its own index.**
❸ **return ToolbarAndroid**

① **pass in white as the `subtitleColor` prop**
② **pass in white as the `titleColor` prop**
③ **give the component a height and `backgroundColor`**
④ **pass in a title prop of 'React Native in Action'**
⑤ **pass in a subtitle prop of 'ToolbarAndroid'**
⑥ **pass in an array of actions. When these actions are clicked, they will be called with their index in the array as an argument**
⑦ **pass in `onActionSelected` as the `onActionSelected` method**

Now, we should not only see the ToolbarAndroid when we refresh our device, but we should also be able to open the DrawerLayoutAndroid menu by clicking on menu.

10.3 ViewPagerAndroid

Next, let's create an example using ViewPagerAndroid. ViewPagerAndroid is a component that easily allows you to swipe left and right between views. Every child of ViewPagerAndroid will be treated as its own separate swipeable view (figure 10.3).

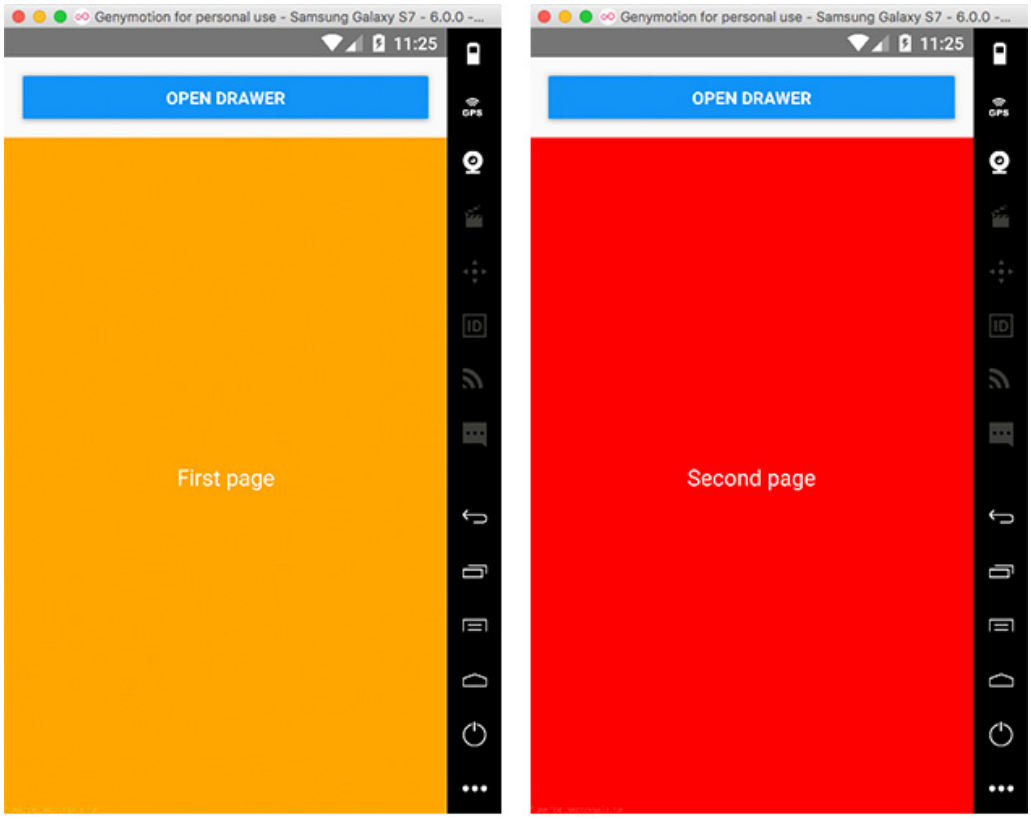


Figure 10.3 ViewPagerAndroid with two child views

In app/ViewPager.js, create the following component (listing 10.7).

Listing 10.7 ViewPagerAndroid

```
import React, { Component } from 'react'
import {
  ViewPagerAndroid, (1)
  View,
  Text
} from 'react-native'

let styles

class ViewPager extends Component {
  render () {
    const {
      pageStyle,
      page1Style,
      page2Style,
```

```
      textStyle
    } = styles
    return (
      <ViewPagerAndroid ❷
        style={{ flex: 1 }}
        initialPage={0}>
        <View style={[ pageStyle, page1Style ]}>
          <Text style={textStyle}>First page</Text>
        </View>
        <View style={[ pageStyle, page2Style ]}>
          <Text style={textStyle}>Second page</Text>
        </View>
      </ViewPagerAndroid>
    )
  }
}

styles = {
  pageStyle: {
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
    flex: 1,
  },
  page1Style: {
    backgroundColor: 'orange'
  },
  page2Style: {
    backgroundColor: 'red'
  },
  textStyle: {
    fontSize: 18,
    color: 'white'
  }
}

export default ViewPager
```

❶ import ViewPagerAndroid from React Native
❷ return ViewPagerAndroid with two child views, one of them with an orange background and one with a red background

Next we need to update Menu.js to add the button to view the new component. In Menu.js, add this button below the Toolbar Android button.

```
<View style={button} >
  <Button onPress={() => onPress('ViewPager')} title='ViewPager Android' />
</View>
```

Finally, we need to import the new component and update the switch statement in App.js to render the new component (listing 10.8).

Listing 10.8 App.js with new ViewPager component

```
import React from 'react'

import Home from './Home'
import Toolbar from './Toolbar'
import ViewPager from './ViewPager'

function getScene (scene) {
  switch (scene) {
    case 'Home':
      return Home
    case 'Toolbar':
      return Toolbar
    case 'ViewPager':
      return ViewPager
    default:
      return Home
  }
}

const App = (props) => {
  const Scene = getScene(props.scene)
  return (
    <Scene openDrawer={props.openDrawer} jump={props.jump} />
  )
}

export default App
```

Now, we should be able to run the app and see the new ViewPager Android Button in the side menu, and can view and interact with the new component.

10.4 DatePickerAndroid

DatePickerAndroid lets us open and interact with the native Android date picker dialog (figure 10.4).

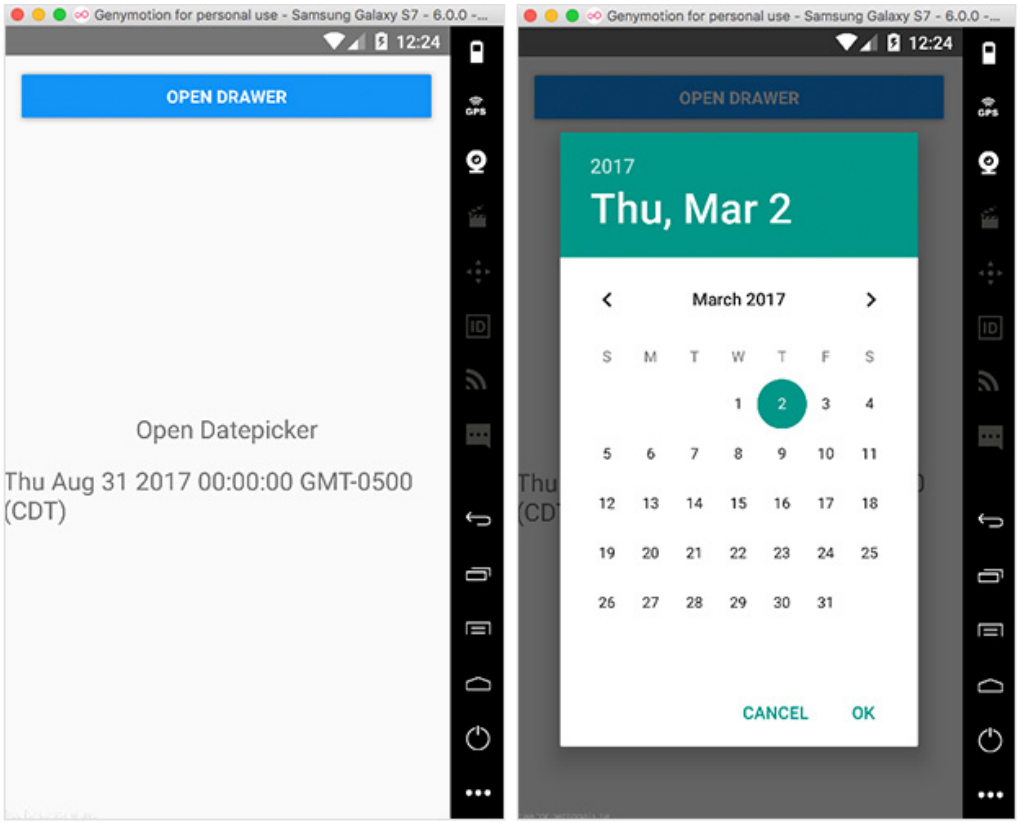


Figure 10.4 DatePickerAndroid

To open and use the `DatePickerAndroid` component, we import `DatePickerAndroid` and call `DatePickerAndroid.open()`.

To get started, create app/DatePicker.js and create the following component (listing10.9).

Listing 10.9 DatePicker.js

```
import React, { Component } from 'react'
import { DatePickerAndroid, View, Text } from 'react-native' (1)

let styles

class DatePicker extends Component {

  constructor() {
    super()
    this.state = { (2)
      date: new Date()
    }
```

```
    this.openDatePicker = this.openDatePicker.bind(this)
  }

  openDatePicker () { ❸
    DatePickerAndroid.open({
      date: this.state.date
    })
      .then((date) => {
      const { year, month, day, action } = date
      if (action === 'dateSetAction') {
        this.setState({ date: new Date(year, month, day) })
      }
    })  }

  render() {
    const {
      container,
      text
    } = styles

    return (
      <View style={container}> ❹
        <Text onPress={this.openDatePicker} style={text}>Open Datepicker</Text>
        <Text style={text}>{this.state.date.toString()}</Text>
      </View>
    )
  }
}

styles = {
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  text: {
    marginBottom: 15,
    fontSize: 20
  }
}

export default DatePicker
```

❶ **import** `DatePickerAndroid` **from React Native**
❷ **create the state, setting the date as a** `new Date()`
❸ **create openDatePicker method, passing in the current date as the date to show on when the datepicker opens. The open method returns a promise, giving us an object with the chosen day, month, year, and the action that was chosen. If you choose a date, then the action is** `dateSetAction`**. If the modal is dismissed, then the action is** `dismissedAction`**.**
❹ **We create a button that will call the** `openDatePicker` **method, and display the date in our View.**

Now that we have the component created, let's update app/App.js to include the new component (listing 10.10).

Listing 10.10 app/App.js with new DatePicker component

```
import React from 'react'

import Home from './Home'
import Toolbar from './Toolbar'
import ViewPager from './ViewPager'
import DatePicker from './DatePicker'

function getScene (scene) {
  switch (scene) {
    case 'Home':
      return Home
    case 'Toolbar':
      return Toolbar
    case 'ViewPager':
      return ViewPager
    case 'DatePicker':
      return DatePicker
    default:
      return Home
  }
}

const App = (props) => {
  const Scene = getScene(props.scene)
  return (
    <Scene openDrawer={props.openDrawer} jump={props.jump} />
  )
}

export default App
```

Finally, we can update the Menu to add the new button that will open our new DatePicker component. In app/Menu.js, add the following button below the ViewPager Android button.

```
<View style={button} >
  <Button onPress={() => onPress('DatePicker')} title='DatePicker Android' />
</View>
```

10.5 TimePickerAndroid

Next up is TimePickerAndroid. TimePickerAndroid is like DatePickerAndroid in that you import it and call the open method to interact with it. This component brings up a TimePicker dialog that allows you to choose a time and use it in your application (figure 10.5).

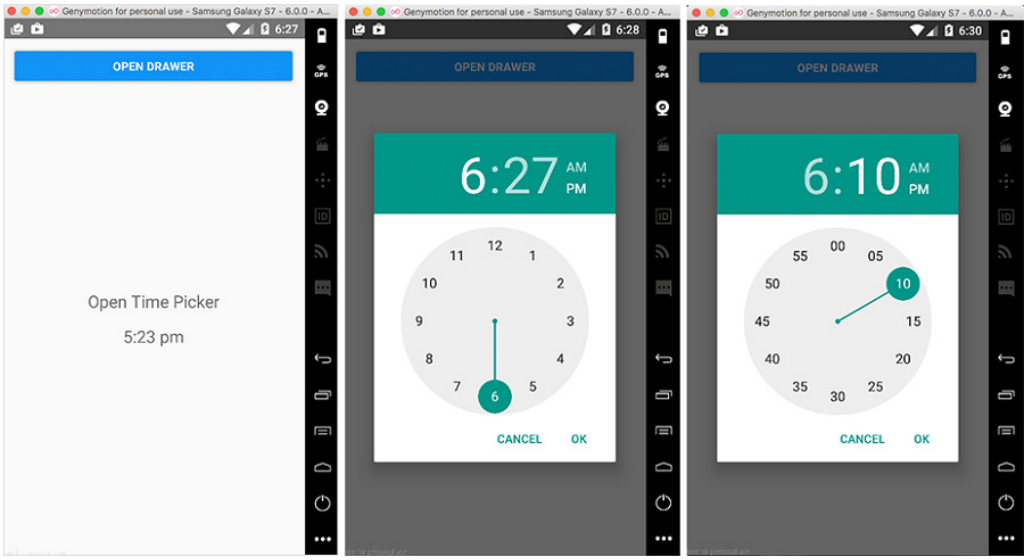


Figure 10.5 TimePickerAndroid with both hour and minute views

To standardize our time formats, we will be using a library called momentjs. To get started with this library, let's install moment. In the root directory of the project, install moment using npm or yarn.

```
npm install moment –save
```

Next, let's create the TimePicker component. In app/TimePicker.js, create the following component (figure 10.11).

Figure 10.11 app/TimePicker.js – TimePickerAndroid using moment.js

```
import React, { Component } from 'react'
import { TimePickerAndroid, View, Text } from 'react-native' ❶
import moment from 'moment' ❷

let styles

class TimePicker extends Component {

  constructor () {
    super()
    this.state = {
      time: moment().format('h:mm a') ❸
    }
    this.openTimePicker = this.openTimePicker.bind(this)
  }

  openTimePicker () {                    ❹
```

```
    TimePickerAndroid.open({
      time: this.state.time
    })
    .then((time) => {
      const { hour, minute, action } = time
      if (action === 'timeSetAction') {
        const time = moment().minute(minute).hour(hour).format('h:mm a')
        this.setState({ time })
      }
    })
  }

  render () {
    const {
      container,
      text
    } = styles

    return (
      <View style={container}>    ❺
        <Text onPress={this.openTimePicker} style={text}>Open Time Picker</Text>
        <Text style={text}>{this.state.time.toString()}</Text>
      </View>
    )
  }
}

styles = {
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  text: {
    marginBottom: 15,
    fontSize: 20
  }
}

export default TimePicker
```

❶ **import TimePickerAndroid from React Native**
❷ **import moment from moment**
❸ **create an initial time and store it in the state. We call `moment().format('h:mm a')` to format the date. The h:mm a that was passed in tells moment that we only want the hour, minute and whether the time is am or pm.**
❹ **create `openTimePicker` method. Again, like `DatePickerAndroid`, the open method returns a promise, with a time object that contains hour, minute, and action. We check to see if the action is `timeSetAction`, and if so we update the state to reflect the new time.**
❺ **create a button in the view to call the `openTimePicker` method and display the time in the view.**

Now that we have the component created, let's update app/App.js to include the new component (listing 10.12).

Listing 10.12 app/App.js with added TimePicker component

```
import React from 'react'

import Home from './Home'
import Toolbar from './Toolbar'
import ViewPager from './ViewPager'
import DatePicker from './DatePicker'
import TimePicker from './TimePicker'

function getScene (scene) {
  switch (scene) {
    case 'Home':
      return Home
    case 'Toolbar':
      return Toolbar
    case 'ViewPager':
      return ViewPager
    case 'DatePicker':
      return DatePicker
    case 'TimePicker':
      return TimePicker
    default:
      return Home
  }
}

const App = (props) => {
  const Scene = getScene(props.scene)
  return (
    <Scene openDrawer={props.openDrawer} jump={props.jump} />
  )
}

export default App
```

Finally, we can update the Menu to add the new button that will open our new TimePicker component. In app/Menu.js, add the following button below the DatePicker Android button.

```
<View style={button} >
  <Button onPress={() => onPress('TimePicker')} title='TimePicker Android' />
</View>
```

10.6 ToastAndroid

ToastAndroid allows us to easily call native Android toasts from within our React Native application. An android toast is just a popup with a message that goes away after a given period (figure 10.6).

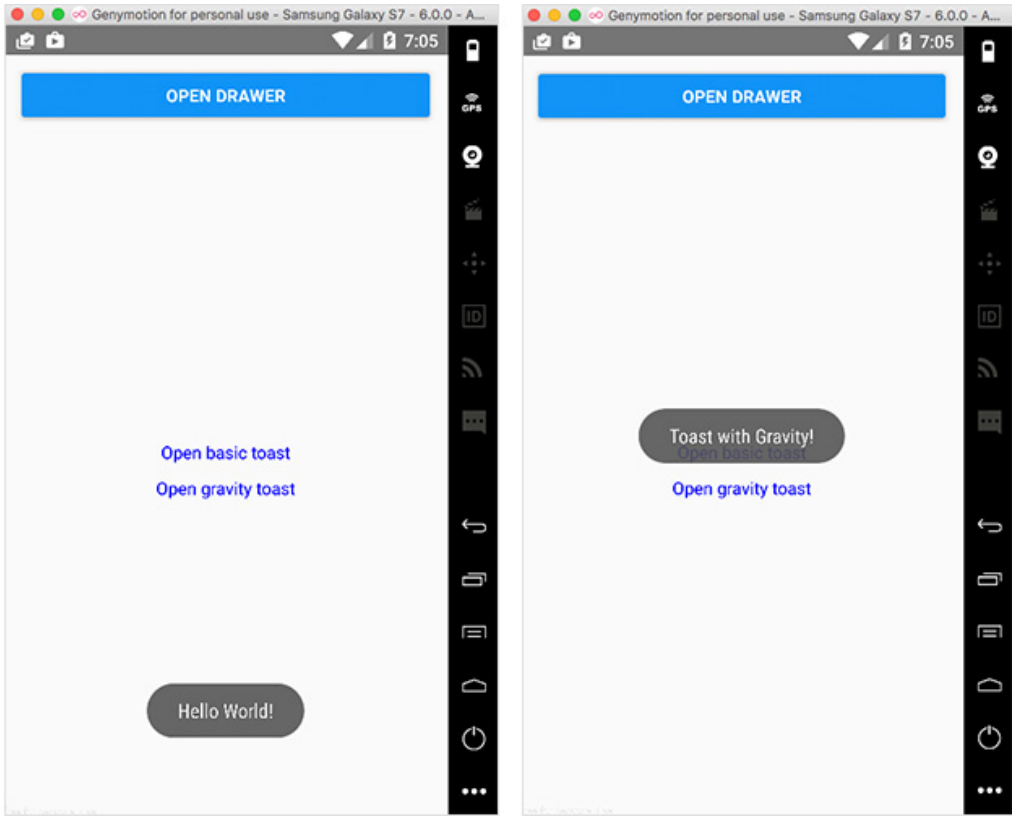


Figure 10.6 ToastAndroid with default and middle positioned toasts.

To get started building out this component, create app/Toast.js with the following component (listing 10.13).

Listing 10.13 app/Toast.js - ToastAndroid

```
import React from 'react'

import { View, Text, ToastAndroid } from 'react-native' (1)

let styles

const Toast = () => {
  let {
    container,
    button
  } = styles

  const basicToast = () => { (2)
```

```
    ToastAndroid.show('Hello World!', ToastAndroid.LONG)
  }

  const gravityToast = () => { ❸
    ToastAndroid.showWithGravity('Toast with Gravity!', ToastAndroid.LONG,
      ToastAndroid.CENTER)
  }

  return (
    <View style={container}>   ❹
      <Text style={button} onPress={basicToast}>Open basic toast</Text>
      <Text style={button} onPress={gravityToast}>Open gravity toast</Text>
    </View>
  )
}

styles = {
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  button: {
    marginBottom: 10,
    color: 'blue'
  }
}

export default Toast
```

❶ **import `ToastAndroid` from React Native**
❷ **create a `basicToast` method that will `call ToastAndroid.show()`, passing in two arguments: 1. A message and 2. A length of time to show the toast. Can be either SHORT (about 2 seconds) or LONG (about 4 seconds)**
❸ **create `gravityToast` method that will call `ToastAndroid.showWithGravity()`. This method is like `ToastAndroid.show()`, but it allows for a third argument to be passed, allowing us to position the toast either at the top, bottom, or center of the view. We pass in `ToastAndroid.CENTER` as the third argument, centering the toast in the middle of the screen.**
❹ **create two buttons in the view, attaching our methods to these buttons**

Now that we have the component created, let's update app/App.js to include the new component (listing 10.14).

Listing 10.14 app/Menu.js – Adding Toast component to app

```
import React from 'react'

import Home from './Home'
import Toolbar from './Toolbar'
import ViewPager from './ViewPager'
import DatePicker from './DatePicker'
import TimePicker from './TimePicker'
import Toast from './Toast'
```

```
function getScene (scene) {
  switch (scene) {
    case 'Home':
      return Home
    case 'Toolbar':
      return Toolbar
    case 'ViewPager':
      return ViewPager
    case 'DatePicker':
      return DatePicker
    case 'TimePicker':
      return TimePicker
    case 'Toast':
      return Toast
    default:
      return Home
  }
}

const App = (props) => {
  const Scene = getScene(props.scene)
  return (
    <Scene openDrawer={props.openDrawer} jump={props.jump} />
  )
}

export default App
```

Finally, we can update the Menu to add the new button that will open our new Toast component. In app/Menu.js, add the following button below the TimePicker Android button.

```
<View style={button} >
  <Button onPress={() => onPress('Toast')} title='Toast Android' />
</View>
```

10.7 Summary

- Implementing a side menu using DrawerLayoutAndroid
- Using ToolbarAndroid to create an interactive app Toolbar
- How to use ViewPagerAndroid to create swipeable views
- Using DatePickerAndroid to create and manipulate dates in your application
- Using TimePickerAndroid to create and manipulate time in your application
- Using ToastAndroid to create native android Toast notifications

A

Installing and running React Native

This appendix covers

- InstallingReact Native for iOS and Android
- Creating and running a new React Native application

A.1 Developing for iOS Devices

At the time of this writing, if you would like to develop for iOS you must have a Mac, as Linux and Windows are not supported for developing for the iOS platform.

A.1.1 Getting Started

To get started with iOS, you need the following installed on your machine and be using a Mac.

> **NOTE** If you do not have homebrew installed, go to http://brew.sh/ and install homebrew before following the next steps.

1. Xcode – Xcode is available through the app store.
2. Node.js – The React Native team recommends installing node.js through homebrew on the command line:

```
brew install node
```

3. Wathman – This is also recommended to be installed through homebrew on the command line:

```
brew install watchman
```

4. React Native command line interface – this should be installed via the command line

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

NOTE If you get a permission error, try with sudo:

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

TESTING THE INSTALLATION ON IOS

Next, we will check to see if we have React Native properly installed by creating a new project. In the terminal or on your command line of choice, run the following commands:

```
react-native init MyProject
cd MyProject
```

Now that we've created the new project and changed into the new directory, we can run the project in a couple of different ways.

- Run the project from the command line: To do this, execute `react-native run-ios` from within the MyProject directory.
- Open the project in Xcode: To do this, open the MyApp.xcodeproj file located at MyProject/ios/MyApp.xcodeproj .

A.2 Developing for Android devices

Developing React Native for Android can be done with either a Mac, Linux, or Windows environment.

A.2.1 Mac and Android

To get started, you need the following to be installed on your machine.

- Node.js
- React Native command line tools
- Watchman
- Android Studio

The React Native docs, as well as myself, recommend installing node and watchman via homebrew.

1. If you do not already have Homebrew installed, go to http://brew.sh/ and install it on your machine.
2. Next, open a command line and install node and watchman using Homebrew:

```
brew install node
brew install watchman
```

3. Once node.js is installed, then install the React Native command line tools by running the following command from your command line:

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

4. Next, install Android Studio at https://developer.android.com/studio/install.html.

Once everything is installed, go to section A.3 of the appendix to create your first project.

A.2.2 Windows and Android

To get started, you need the following to be installed on your machine.

- node.js
- React Native command line tools
- Watchman
- Android Studio

Watchman is in the alpha stage for Windows, but is working fine so far in my experience.

1. To install watchman, go to https://github.com/facebook/watchman/issues/19 and download the alpha build via the link in the first comment.
2. React Native recommends installing node.js and Python2 via Chocolatey, a package manager for Windows. To do so, install Chocolatey, then open a commandline as admin, then run:

```
choco install nodejs.install
choco install python2
```

3. Then, install the React Native command line interface:

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

4. Then, download and install Android Studio at https://developer.android.com/studio/install.html.

Once everything is installed, go to section A.3 of the appendix to create your first project.

A.2.3 Linux and Android

To get started, you need the following to be installed on your machine.

- Node.js
- React Native command line tools
- Watchman
- Android Studio

1. First, if you do not already have node.js installed, go to https://nodejs.org/en/download/package-manager/ and follow the installation instructions for your Linux distribution.
2. Once node.js is installed, run the following command to install the React Native command line tools:

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

3. Then, download Android Studio at https://developer.android.com/studio/install.html .
4. Then, download and install Watchman at https://facebook.github.io/watchman/docs/install.html#installing-from-source .

Once everything is installed, go to section A.3 to create your first project.

A.3 Creating a new project

Once your development environment is set up, and react-native-cli is installed, creating a new project is done from the command line.

To create a new React Native application, navigate to the folder in which you would like to create your project and issue the following command:

```
react-native init MyProjectName
```

NOTE ***MyProjectName*** **can be whatever you want to name your project.**

A.4 Running the project

Once you have successfully created a new project, change directories into the project from your command line and run the following commands:

RUN PROJECT FOR IOS

```
react-native run-ios
```

RUN PROJECT FOR ANDROID

```
react-native run-android
```