Stan Bershadskiy, Crysfel Villa

Foreword by: Christopher Chedeau
Frontend Engineer at Facebook

React Native Cookbook

Take your React Native application development to the next level with this large collection of recipes

Packt>

React Native Cookbook

Take your React Native application development to the next level with this large collection of recipes

Stan Bershadskiy
Crysfel Villa

BIRMINGHAM - MUMBAI

React Native Cookbook

First published: December 2016

Production reference: 1161216

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-255-8

www.packtpub.com

Credits

Authors

Stan Bershadskiy
Crysfel Villa

Reviewer

Brice Mason

Commissioning Editor

Ashwin Nair

Acquisition Editor

Shweta Pant

Content Development Editor

Arun Nadar

Technical Editor

Shivani Kiran Mistry

Copy Editor

Safis Editing

Project Coordinator

Ritika Manoj

Proofreader

Safis Editing

Indexer

Francy Puthiry

Graphics

Jason Monteiro

Production Coordinator

Melwyn Dsa

Foreword

Web development has for a long time been about working around the various browser quirks and use a convoluted subset of what is provided, called "best practices", in order to build dynamic applications on a platform that has originally been designed for static pages.

Jordan Walke, a Facebook engineer, came at this problem with a different mindset: instead of trying to make what exists work, what if we designed an ideal solution and adapt it to make it work under the existing constraints? This is how React was born.

React introduced a lot of concepts borrowed from many fields of computer science to the web development world. The core idea is to declare what your application should look like at any point in time by composing components written using a real programming language.

During a hackathon, four people: Jordan Walke, Ashwin Bharambe, Lin He, and myself, tried to use React to render native iOS components such as UIView, UIImage, and UILabel instead of the web equivalent, , <image>, and . The crazy thing is that it actually worked!

Then, our mantra became "how can we bring the best of both worlds?". From the Web, we borrowed the fast iteration speed and the awesome developer experience with ideas like the element inspector and redbox. Native challenged us to provide an amazing user experience with an acute attention to detail, strong performance, and great gesture support.

As we got to a point where we used it in production and it provided value, we decided to open source it. We knew it was going to be make some buzz, but we had no idea that it would explode as it did. One of our big focuses was to make it a community project, and we worked hard to get core contributors from outside of Facebook. We now have half of the commits contributed by the community!

It's very humbling to see that contributions extended way beyond code. In the digital space, we've seen a huge amount of tutorials and libraries, and it extended to the physical world, with various meetups and conferences all over the world. I'm proud to see that Stan and Crysfel dedicated their time to produce this amazing book about React Native.

React was born out of the frustration that it's way too hard to build software. I hope that this book will help you build awesome and successful user interfaces that will bring real value to your business!

Christopher Chedeau

Frontend Engineer at Facebook

About the Authors

Stan Bershadskiy is an architect at Modus Create and holds a Master's in Computer Science from NYIT. While doing full-stack development, he found working on the frontend most enjoyable because of the speed one can develop and switch focus toward JavaScript. Stan likes to involve himself in anything JavaScript-related, particularly around building rich applications for desktop, web, and mobile. He is located in New York City and can be found co-organizing NYC.JS meetups. More recently, he has focused on promoting React Native by presenting at conferences and publishing blog posts. You can follow him on Twitter at `@stan229`.

First and foremost, I'd like to thank my wife Lika. You have been my strongest supporter and more importantly putting up with me working on the book all hours of the night. You have been at my side from the first word I wrote, to the edits I just completed at 4 AM. To my son, Aiden, you are the center of my world. There's nothing that brightens my day like seeing you. I simply wouldn't be able to write this book without the support of my wife, son, and parents.

I'd like to thank Crysfel for inviting me on-board to write this book and the work we did together. Brice Mason, you are an amazing colleague and editor. You did an incredible job reviewing the book, you are far more thorough than I ever could be. A special thank you goes out to Christopher Chedeau for embodying the spirit of the React Native community and writing an exemplary foreword.

A very special thank you goes to Jay Garcia. You have been my mentor, sounding board, facilitator, and just an excellent friend. You have helped me with this book in far more ways than I can list. Lastly, a thank you goes out to everyone at Modus Create for supporting me not only during the writing of this book but being the absolute best place to work at.

Crysfel Villa is a senior software engineer at Modus Create. He's a passionate JavaScript coder and an accomplished software developer with over 10 years of experience in technical training, consulting, and systems analysis. Crysfel loves to write about emerging technologies and he has deployed several apps to the Apple Store using React Native. He currently lives in NY and can be found attending tech meetups throughout the city. You can follow him on Twitter at `@crysfel`.

It has been a very difficult journey to complete this project. I worked on this book while I was in Japan, Korea, Mexico, and NY. Working overseas, on vacations, visiting family and friends, holidays and what not. I'd like to thank my wife Hazel for the huge help she did, reviewing my scripts, sometimes late at night. I wouldn't be able to accomplish this project without her help.

I would also like to thank my dad Felipe, my two youngest brothers Victor and Jaziel, my in-laws and friends for their support, just by continue asking me about the book and encouraging me to continue working hard on it, many thanks to all of them. Special thanks to Brice, who found some missing steps on recipes, suggested things to improve and overall helping us to complete the book on time.

About the Reviewer

Brice Mason is a husband, father, developer, writer, and speaker with over 15 years of software development experience. As a senior engineer at Modus Create, he's been fortunate to contribute to the delivery of several enterprise-level desktop and mobile applications using JavaScript. When not writing code or writing about code, Brice enjoys spending time with his wife and son. You can reach him via Twitter at `@bricemason`.

I would like to thank Stan, Crysfel, and the terrific folks at Packt Publishing for including me in this project. Special thanks to my wife Heather and son Chase for allowing me the extra time to help contribute to this great book.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us**: `customerreviews@packtpub.com`.

Table of Contents

Preface

React has taken the web development world by storm. It is only natural that its unique architecture and strong third-party support be applied to native application development. Using JavaScript, you can build a truly native application that renders native UI components and accesses native device functionality. This book will take you through the basics of React Native development all the way through some more advanced concepts.

In this book, we will cover topics in React Native ranging from adding basic UI components to successfully deploying for multiple target platforms. Since this is a cookbook, each topic will be described in its own recipe. The book follows a top-down approach, beginning with building rich UIs. These UIs will be created with both built-in and custom components that you will create, style, and animate. You will then learn about different strategies for working with data, including leveraging the popular Redux library. Then you will have the opportunity to step further into exposing native device functionality. Finally, we will discuss how to bring our application to production and maintain its reliability.

The recipes in this book will break down how you can accomplish a particular task. They will also dive into some of the more important concepts supporting the task. After reading the book, you should feel very comfortable tackling any aspect of React Native development.

What this book covers

`Chapter 1`, *Getting Started*, shows you how to perform everyday scenarios using React Native. You will add some basic UI components that come with the framework to your view and style them accordingly.

`Chapter 2`, *Implementing Complex User Interfaces*, steps a bit further into the creation of UIs. In this chapter, you will learn how to create custom view components using JavaScript and JSX, as well as add them to customized layouts.

`Chapter 3`, *Animating the User Interface*, teaches you how to make your application feel more alive with animations. You will start off with basic layout animations and end up having mastered the Animated library and building custom animations.

`Chapter 4`, *Working with Application Logic and Data*, covers what's involved in dealing with data. You will learn everything there is to know about dealing with data whether it is sitting locally or on a remote server. The chapter also covers storing your data in a native database as well as integration with the Facebook SDK.

`Chapter 5`, *Implementing Redux*, explores the extremely popular Redux library that is commonly used with React and React Native. You will learn how to make Redux fit in your application and control all things related to the application logic and data.

`Chapter 6`, *Adding Native Functionality*, shows you how to take your React Native development past the JavaScript layer and leverage the full power of the device. You will learn how to add custom modules and UI components that are implemented using the device's native language. You will then take it on level further by leveraging native functionality that is not available out of the box, such as multithreading, fingerprint detection, and more.

`Chapter 7`, *Architecting for Multiple Platforms*, explains how to deal with the many platforms you can deploy your React Native application to. You will learn how to build a React Native application for different platforms such as Windows 10 and Mac OS desktop. The chapter also teaches you how to deal with coding for specific platforms and the best practices for sharing code across them.

`Chapter 8`, *Integration with Applications*, teaches you how to work with existing iOS and Android applications that you may have. You will learn not only how to render a React Native application inside a larger existing native app, but also how to communicate bi-directionally between them.

`Chapter 9`, *Deploying Our App*, guides you through the one of the most critical process of application development, deploying. You will learn how to deploy your application at different stages of development, including deploying to the app stores. You will also learn how to perform over-the-air updates to your already deployed React Native apps.

`Chapter 10`, *Automated Testing*, shows you how to set up automated testing for your React Native application. You will learn how to create functional tests using Appium and Mocha. You will also be able to take your tests and run them on an iOS device for a real-world experience.

`Chapter 11`, *Optimizing the Performance of Our App*, explores how to get the most out of React Native applications. This chapter covers everything related to performance and memory optimization. You will be provided with various tips, tricks, and best practices on how to get the most out of your app.

What you need for this book

For this book, you will require the following:

- Node.js (v4+), Watchman (latest)
- For iOS development, you should be running Mac OS and have XCode 8.0+ installed
- For Android development, you should have Android Studio 2.2+ installed

Who this book is for

This book is intended for developers who may be either just starting out or already experienced with React Native development. Existing knowledge of JavaScript ES2015 is highly recommended. A basic understanding of iOS development with Objective-C or Swift and Java for Android development is optional, but may help readers when adding custom native functionality.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a new user for JIRA in the database and grant the user access to the `jiradb` database we just created using the following command:"

A block of code is set as follows:

```
<Contextpath="/jira"docBase="${catalina.home}
/atlassian- jira" reloadable="false" useHttpOnly="true">
```

Any command-line input or output is written as follows:

```
mysql -u root -p
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select **System info** from the **Administration** panel."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors` .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/React-Native-Cookbook`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started

In this chapter, we will cover the following recipes:

- Adding styles to text and containers
- Using images to mimic a video player
- Creating a toggle button
- Displaying a list of items
- Adding tabs to the viewport
- Using flexbox to create a profile page
- Setting up a navigator

Introduction

React Native is a fast-growing library. Over the last year it has become very popular among the open source community. There's a new release every other week that improves performance, adds new components, or provides access to new APIs on the device.

While this is great for the most part, it comes with a drawback. Sometimes, things on our project will break with a new release; we need to be very careful when updating our projects. As a rule of thumb, never update to the latest version without reading the release notes, which usually describe the breaking changes. On top of that, we need to make sure that any of the third-party libraries we are using in our projects are up to date with the new release.

In this chapter, we will learn about the most common components within the library. This book aims to reach out to developers who have already completed the *Getting Started Guide* on the React Native documentation, which means we will jump right to the good stuff and avoid things such as *installing the environment* and H*ello World* examples.

To step through all the recipes in this chapter, we will have to create a new app, so make sure you have your environment up and running. I recommend you follow the documentation on the React Native official website, then run the following commands on your terminal for each recipe, where `AnAppName` is the name of your app:

```
$ react-native init --verbose AnAppName
$ cd AnAppName
$ react-native run-android
$ react-native run-ios
```

Adding styles to text and containers

We have several components at our disposal, but containers and texts are the most common and useful components to create layouts or other components. In this recipe, we will see how to use containers and text, but most importantly we will see how styles work in React Native.

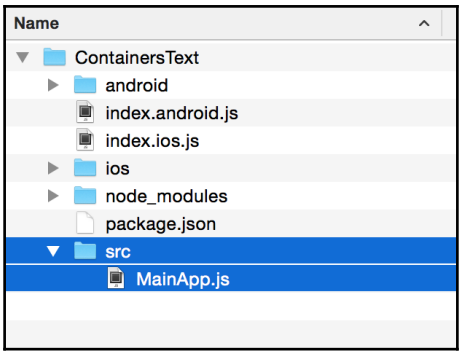
We will create a UI for a simple music player; we won't be using icons for now, but we will add them later.

Getting ready

Please follow the instructions in the introduction in order to create a new app with the name `ContainersText`.

How to do it…

1. Let's start by creating an `src` folder in the root of the project. This is where all our JavaScript code will be placed.
2. Create a new JavaScript called `MainApp.js`:



3. In the `MainApp.js` file, we are going to create a stateless component; this component will mimic a small music player. For now, it will only display the name of the song and a bar to show the progress:

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

4. Once we have imported the dependencies, we can write the component as follows:

```
const MainApp = () => {
  const name = '01 - Hey, this is my life';

  return (
    <View style={styles.container}>
      <View style={styles.innerContainer} />
      <Text style={styles.title}>
        <Text style={styles.subtitle}>Playing:</Text> {name}
      </Text>
    </View>
  );
};
```

5. We have our component ready, so now we need to add some styles in order to add colors and font styles:

```
const styles = StyleSheet.create({
  container: {
    margin: 10,
    marginTop: 100,
    backgroundColor: '#e67e22',
    borderRadius: 5,
  },
  innerContainer: {
```

```
    backgroundColor: '#d35400',
    height: 50,
    width: 150,
    borderTopLeftRadius: 5,
    borderBottomLeftRadius: 5,
  },
  title: {
    fontSize: 18,
    fontWeight: '200',
    color: '#fff',
    position: 'absolute',
    backgroundColor: 'transparent',
    top: 12,
    left: 10,
  },
  subtitle: {
    fontWeight: 'bold',
  },
});
```

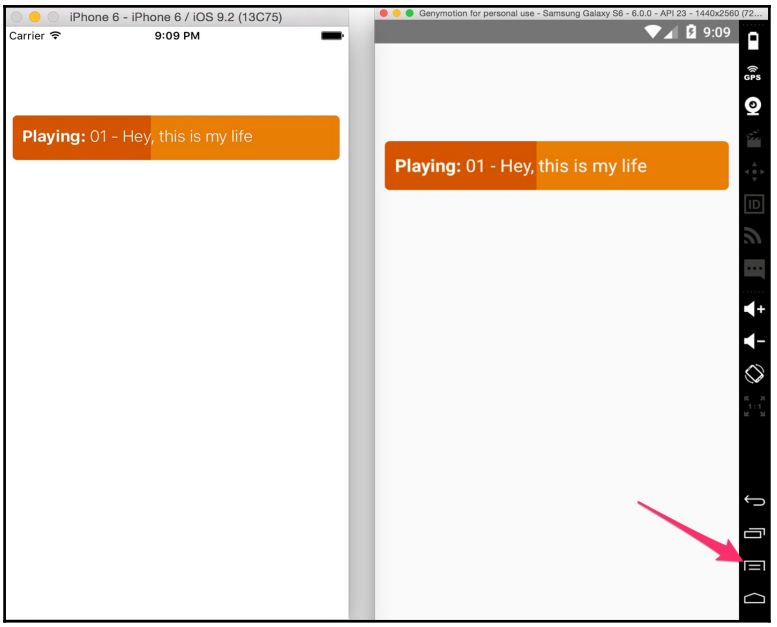
6. In order to use this component outside of this file, we need to export it, as follows:

```
export default MainApp;
```

7. The next step is to import our new component in the `index.ios.js` and `index.android.js` files. The code will be the same for both platforms:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('ContainersText', () => MainApp);
```

8. In order to see our changes in the simulators, we need to reload the app. For iOS, you can press *cmd + R*; for Android, click on the menu button and then the refresh button:



How it works…

Let's take a look at what we did in the previous recipe. In steps 3 to 6 we created our component with the necessary styles. There are several things going on in these steps, so let's dig deeper.

In step *3*, we included the dependencies of our component. In this case, we will use a `View`, which is a container; if you are familiar with web development, a `View` is similar to a `div`. We can add more `Views` inside other `Views`, `Texts`, `Lists`, and any other custom component that we create or import from a third-party library.

In step *4*, we defined the name of our component. In this case, it will be `MainApp`. As a convention, we should always use the same name for the file and for the component. As you can see, this is a stateless component, which means it doesn't have any state; it's a pure function and doesn't support any of the life cycle methods.

We are defining a `name` constant, but in real-world applications this data should come from the props. In the return we are defining the JSX that we are going to need to render our component, along with a reference to the styles.

Each component has a property called `style`; this property receives an object with all the styles that we want to apply to the given component. Styles are not inherited (except for the `Text` component) to the children components, which means we need to set individual styles for each component.

In step *5*, we defined the styles for our component. We are using the `StyleSheet` API to create all our styles. As previously mentioned, all we need is an object containing the styles; however, by using the `StyleSheet` API instead of a simple plain object, we gain some performance optimizations, as the styles will be reused for every renderer as opposed to creating an object every time the render method gets executed.

The properties in the `styles` object are very straightforward. If you are a web developer it, should be really easy to get used to this, because it's similar to CSS; however, it's not the same. We have `margins`, `paddings`, `width`, `height`, `border width`, `border color`, `border radius`, and many more properties. I recommend you take a look at the documentation to find out all the available properties.

In step *7*, we only imported our new component and used it as the component that will bootstrap our app.

There's more…

I'd like to call to your attention to the definition of the `title` style in step *5*. Here, we have defined a property called `backgroundColor` and set `transparent` as its value. As a good exercise, let's comment this line of code and see the result:

On iOS, the text will have an orange background color and it might not be what we really want to accomplish in our UI. In order to fix this, we need to set the background color of the text as transparent. But the question is, why is this happening? The reason is that React Native adds some optimizations to the text by setting the color from the parent's background color. This will improve the rendering performance because the rendering engine will not have to calculate the pixels around each letter of the text and the rendering will be executed faster.

Think carefully when setting the background color as `transparent`. If the component is going to be updating the content very frequently, there might be some performance issues with text, especially if the text is too long.

Using images to mimic a video player

Images are an important part of any UI. Whether we use them to display icons, avatars, or pictures, with React Native you can do all of that. In this recipe, we will use images to create a video player. We will also display the icons from the local device, and a large image from a remote server on Amazon S3.

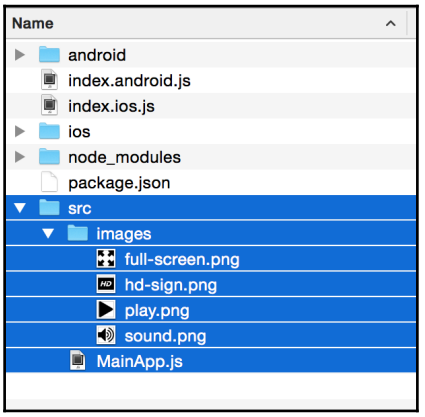
Getting ready

In order to follow the steps in this recipe, its necessary to create an empty app using the **React Native CLI**. Follow the instructions in the introduction of this chapter if you don't know to create an empty app. We are going to name it `LoadingImages`.

We are going to display a few images in our app to mimic a video player, so make sure to download the assets for this recipe.

How to do it…

1. The first thing we are going to do is to create a new folder called `src`. Inside this folder, we need to create a file called `MainApp.js`, and an `images` folder to store our icons. Our project should look as follows:



2. In the `MainApp.js` file, we are going to include all the dependencies we'll need for this component:

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

3. We need to require the `images` that will be displayed in our component. By defining a constant we can use the same image in different places. We might need to restart the package server to successfully load the images, especially if we are using Windows:

```
const playIcon = require('./images/play.png');
const volumeIcon = require('./images/sound.png');
const hdIcon = require('./images/hd-sign.png');
const fullScreenIcon = require('./images/full-screen.png');
const remoteImage = {
uri:
  'https://s3.amazonaws.com/crysfel/
   public/book/new-york.jpg' };
```

4. We are going to use a stateless component to render the JSX. We'll use all the images we have declared in the previous step:

```
const MainApp = () => {
 return (
    <Image source={remoteImage} style={styles.fullscreen}>
      <View style={styles.container}>
        <Image source={playIcon} style={styles.icon} />
        <Image source={volumeIcon} style={styles.icon} />
        <View style={styles.progress}>
          <View style={styles.progressBar} />
        </View>
        <Image source={hdIcon} style={styles.icon} />
        <Image source={fullScreenIcon} style={styles.icon} />
      </View>
    </Image>
  );
};
```

5. Once we have the elements that we are going to render, we need to define the styles for each element:

```
const styles = StyleSheet.create({
  fullscreen: {
    flex: 1,
  },
  container: {
    position: 'absolute',
    backgroundColor: '#202020',
    borderRadius: 5,
    flexDirection: 'row',
    height: 50,
    padding: 5,
    paddingTop: 16,
    bottom: 30,
    right: 10,
    left: 10,
    borderWidth: 1,
    borderColor: '#303030',
  },
  icon: {
    tintColor: '#fff',
    height: 16,
    width: 16,
    marginLeft: 5,
    marginRight: 5,
  },
  progress: {
    backgroundColor: '#000',
    borderRadius: 7,
    flex: 1,
    height: 14,
    margin: 10,
    marginTop: 2,
  },
  progressBar: {
    backgroundColor: '#bf161c',
    borderRadius: 5,
    height: 10,
    margin: 2,
    width: 80,
  },
});
```

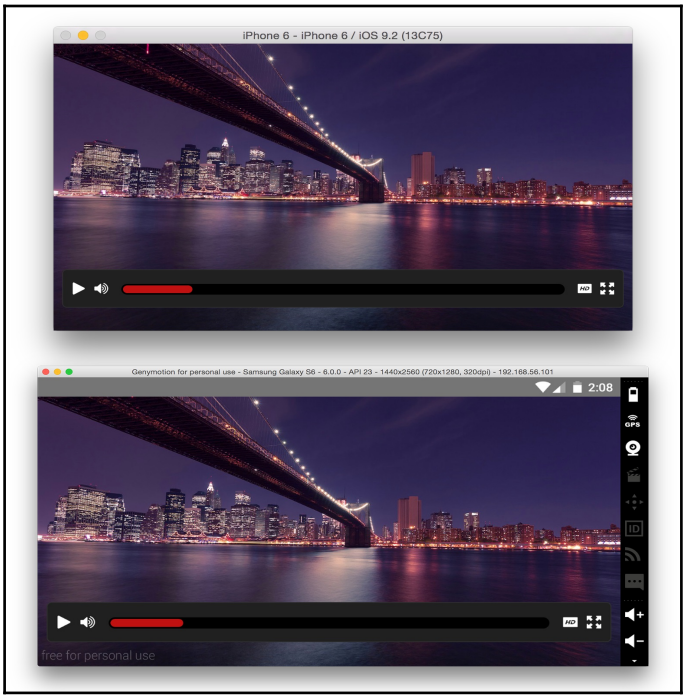
6. In order to use our component, we need to export it. This is a very simple step that requires only one line of code:

```
export default MainApp;
```

7. Finally, we need to import our new component inside `index.ios.js` and `index.android.js`:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';
AppRegistry.registerComponent('LoadingImages', () => MainApp);
```

8. We are done! Just refresh the app on the simulators and you should see something like this:



How it works…

In step *2*, we required the `Image` component; this is the component responsible for rendering images from the local filesystem on the device or from a remote server.

In step *3*, we required all the images. It's a good practice to require the images outside of the component in order to only require them once, and then we can use them in our component. On every renderer, React Native will use the same image; if we are dealing with dynamic images from a remote server, then we should require them on every renderer.

The `require` function accepts the path of the `image` as a parameter; the path is relative to the folder that our class is. For remote images, we need to use an object defining the `uri` where our file is.

In step *4*, a stateless component was declared. We are using the `remoteImage` as the background of our app. In order to set an image in the background, we need to define all the other elements as children of the image. There's not a `backgroundUrl` property, such as in CSS.

The `source` property of the `Image` accepts an object to load remote images or a reference to the required file. It's very important to explicitly require every image that we want to use, because, when we prepare our app for distribution, images will be added to the bundle automatically. This is the reason we should avoid doing things like the following:

```
const iconName = playing ? 'pause' : 'play';
const icon = require(iconName);
```

The preceding code will not include the images in the final bundle. As a result, we will have errors when trying to access these images. Instead, we should refactor our code as follows:

```
const pause = require('pause');
const play = require('playing');
const icon = playing ? pause : play;
```

This way, the bundle will include both images when preparing our app for distribution, and we will decide dynamically which image to display.

In step *5*, we defined the styles. Most of the properties are self-explanatory; the `tintColor` property might be a bit confusing, but this property is basically setting the color of the image, in this case, to white. I will talk more about flex in separate recipes, but for now let's just say that `flexDirection: 'row'` is allowing us to align the icons horizontally.

In step *7*, we included the `MainApp` class in the iOS and Android index files. After this, we should be able to run our application on the simulators.

There's more…

In this recipe, we have used flexbox to horizontally arrange the controls of the player. If you want to learn more about flexbox, take a look at *Using flexbox to create a profile page.* For more advanced content, you should go to `Chapter 2`, *Implementing Complex User Interfaces.*

Creating a toggle button

We all know that buttons are an essential UI component in every application. We use buttons for navigation, to trigger API calls, and so on. In this recipe, we will create a toggle button, which by default is going to be unselected. When the user taps on it, we will change the styles of the button to make it look selected.

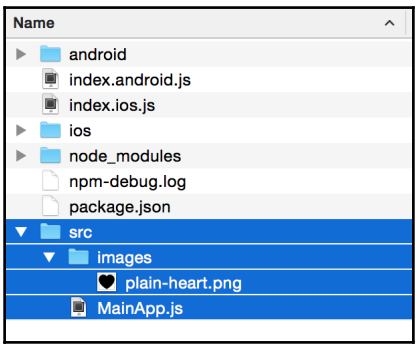
We will learn how to detect the tap event, use an image as the UI, keep the state of the button, and add styles based on the component state.

Getting ready

Let's create a new app using the React Native CLI. We are going to name it `ButtonsAndEvents`. We are going to use one image in this recipe; make sure to download the assets for this recipe or feel free to use your own image.

How to do it…

1. We need to create an `src` folder where our source will be stored; inside this folder we will create an `images` folder and a `MainApp.js` file:



2. Let's import the dependencies for this class:

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

const heartIcon = require('./images/plain-heart.png');
```

3. For this particular recipe, we need to keep the `state` of the button when pressed; therefore, we need to create a class that extends from `Component`, as follows:

```
class MainApp extends Component {
  state = {
    liked: false,
  };

  _onPressBtn = () => {
    // We will define the content on step 6
  }

  render() {
    // We will define the content on step 4
  }
}
```

4. We need to define the content of our new component inside the `render` method; here, we are going to define the `Image` button and a `Text` underneath:

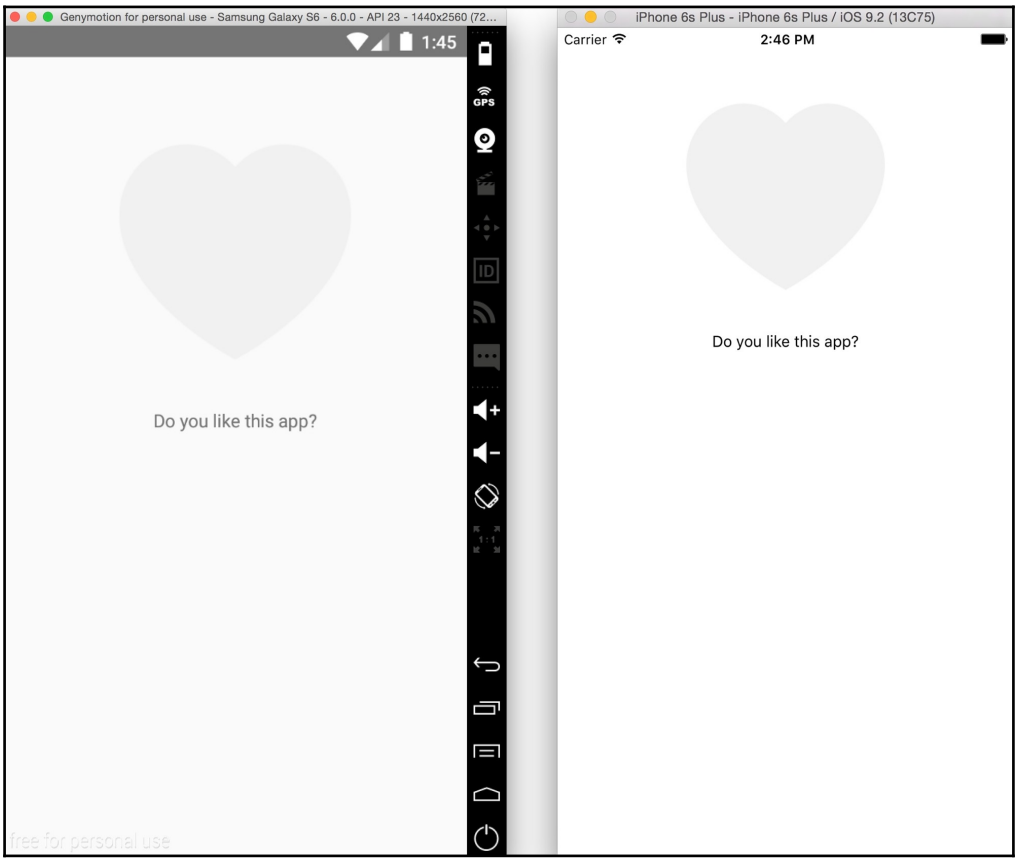
```
render() {
  return (
   <View style={styles.container}>
      <TouchableHighlight
        style={styles.btn}
        underlayColor="#fefefe"
      >
        <Image
    source={heartIcon}
    style={styles.icon}
      />
      </TouchableHighlight>
      <Text style={styles.text}>Do you like this app?</Text>
    </View>
```

```
    );
  }
```

5. Let's define some styles to set dimensions, position, margins, colors, and so on:

```
const styles = StyleSheet.create({
  container: {
    marginTop: 50,
    alignItems: 'center',
  },
  btn: {
    borderRadius: 5,
    padding: 10,
  },
  icon: {
    width: 180,
    height: 180,
    tintColor: '#f1f1f1',
  },
  liked: {
    tintColor: '#e74c3c',
  },
  text: {
    marginTop: 20,
  },
});
```

6. If we run the project on the simulators, we should have something similar to the following screenshot:



7. In order to respond to the tap event, we need to define the content of the `_onPressBtn` function and assign it as a callback to the `onPress` property:

```
class MainApp extends Component {
  state = {
    liked: false,
  };

  _onPressBtn = () => {
    this.setState({
      liked: !this.state.liked,
    });
  }
```

```
  render() {
    return (
      <View style={styles.container}>
        <TouchableHighlight
          onPress={this._onPressBtn}
          style={styles.btn}
          underlayColor="#fefefe"
        >
          <Image source={heartIcon} style={styles.icon} />
        </TouchableHighlight>
        <Text style={styles.text}>Do you like this app?</Text>
      </View>
    );
  }
}
```

8. If we test our code, we won't see anything changing on the UI, even though the state on the component is changing when we press the button. Let's add a different color to the image when the state changes; that way, we will be able to see some response from the UI:

```
render() {
 const likedStyles = this.state.liked ? styles.liked : null;
 return (
   <View style={styles.container}>
     <TouchableHighlight
       onPress={this._onPressBtn}
       style={styles.btn}
       underlayColor="#fefefe"
     >
       <Image
         source={heartIcon}
         style={[styles.icon, likedStyles]}
       />
     </TouchableHighlight>
     <Text style={styles.text}>Do you like this  app?</Text>
   </View>
 );
}
```

9. We are almost done with this class; the only thing that is missing is the `export`. At the bottom of the file we can add the following line:

```
export default MainApp;
```

10. Finally, we need to update the `index.ios.js` and `index.android.js` files to import and use our new class:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent
('ButtonsAndEvents', () => MainApp);
```

How it works…

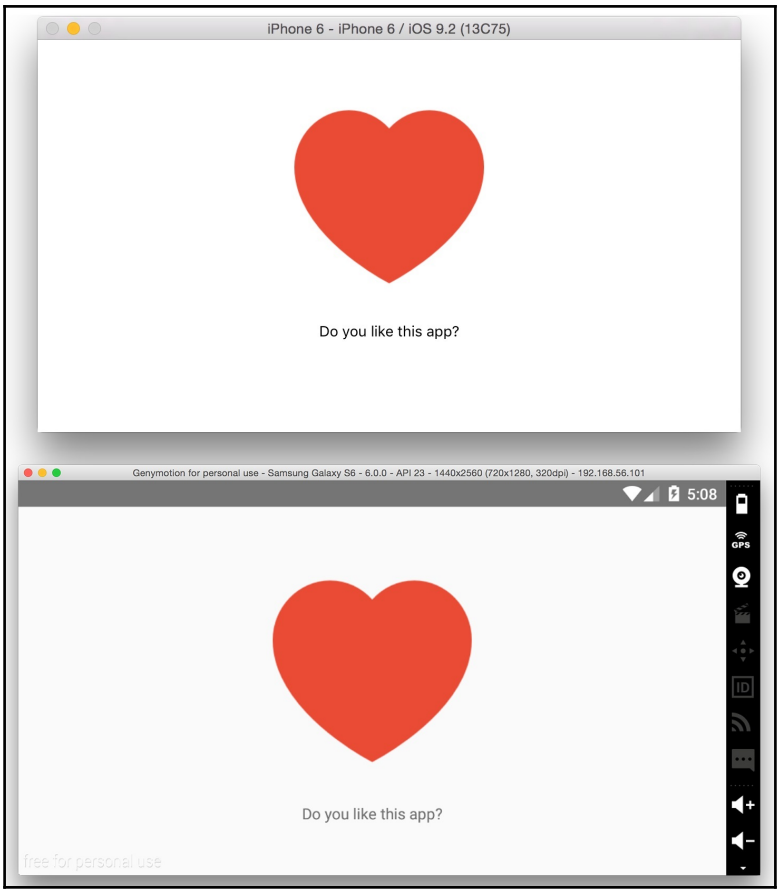
In step *2*, we imported the `TouchableHighlight` component. This is the component responsible for handling the touch event.

When the user touches the active area, the content will be highlighted based on the `underlayColor` value we have set.

In step *3*, we defined the state of the `Component`. In this case, there's only one property on the state, but we can add as many as needed. In `Chapter 2`, *Implementing Complex User Interfaces*, we will see more recipes about handling the state in more complex scenarios.

In step *6*, we used the `setState` method to change the value of the `liked` property. This method is inherited from the `Component` class that we are extending.

In step *7*, based on the current state of the `liked` property, we used the styles to set the color of the image to red, or we returned a `null` to avoid applying any styles. When assigning the styles to the `Image` component, we used an array to assign many objects; this is very handy because, internally, the component will merge all the styles into one single object. The objects with the highest index will overwrite the properties from the lowest object index in the array:



There's more…

In a real application, we are going to use several buttons, sometimes with an icon aligned to the left, a label, different sizes, and colors, and so on. It's highly recommended to create a reusable component to avoid duplicating code all over our app. In `Chapter 2`, *Implementing Complex User Interfaces*, we will create a button component to handle some of these scenarios.

Displaying a list of items

Lists are everywhere: A list of orders on the user's history, a list of available items in a store, a list of songs to play; basically, any application will need to display information in a list.

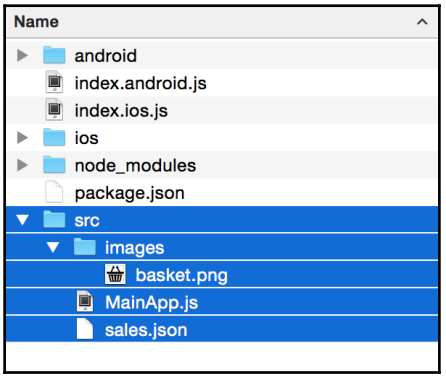
For this recipe, we are going to display several items in a `list` component. We are going to define a JSON file with some data, then we are going to load this file using a simple `require` to finally render each item with a nice but simple layout.

Getting ready

Let's start by creating an empty app–in this case we will name it `ListItems`. We are going to need an icon to display on each item, so please download the assets for this recipe or use your own `.png` image.

How to do it…

1. We will start by creating an `src` folder. Inside this folder we will have the `MainApp.js` file and the `sales.json` file:



2. Inside the `sales.json` file, we will define the data that we are going to display on the list. Here's some sample data:

```
[
  {"items": 5, "address": "140 Broadway, New York, NY 11101",
  "total": 38, "date": "May 15, 2016"}
]
```

3. To avoid cluttering the pages of this book, I've only defined one record, but go ahead and add more content to the array. Copying and pasting the same object multiple times will do the trick. In addition, you could change some values on the data.
4. Let's open the `index.ios.js` and `index.android.js` files, remove the existing code, and add the following to import dependencies and register the app:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('ListItems', () => MainApp);
```

5. In the previous step, we imported the `MainApp` component, but it's not defined yet. Let's open the `src/MainApp.js` file and import the dependencies:

```
import React, { Component } from 'react';
import {
  StyleSheet,
  View,
  ListView,
  Image,
  Text,
} from 'react-native';
import data from './sales.json';

const basketIcon = require('./images/basket.png');
```

6. Now we need to create the class to render the list of items. We are going to keep the sales data on the state; that way, we could insert or remove elements easily:

```
class MainApp extends Component {
  constructor(props) {
    super(props);
    var ds = new ListView.DataSource({
      rowHasChanged: (r1, r2) => r1 !== r2
    });

    this.state = {
      dataSource: ds.cloneWithRows(data),
    };
  }

  renderRow(record) {
    // Defined on step 8
  }
```

```
  render() {
    // Defined on step 7
  }
}

export default MainApp;
```

7. In the `render` method, we need to define the `ListView` component and we will use the `renderRow` method to render each item. The `dataSource` property defines the array of elements that we are going to render on the list:

```
render() {
  return (
    <View style={styles.mainContainer}>
      <Text style={styles.title}>Sales</Text>
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderRow}
      />
    </View>
  );
}
```

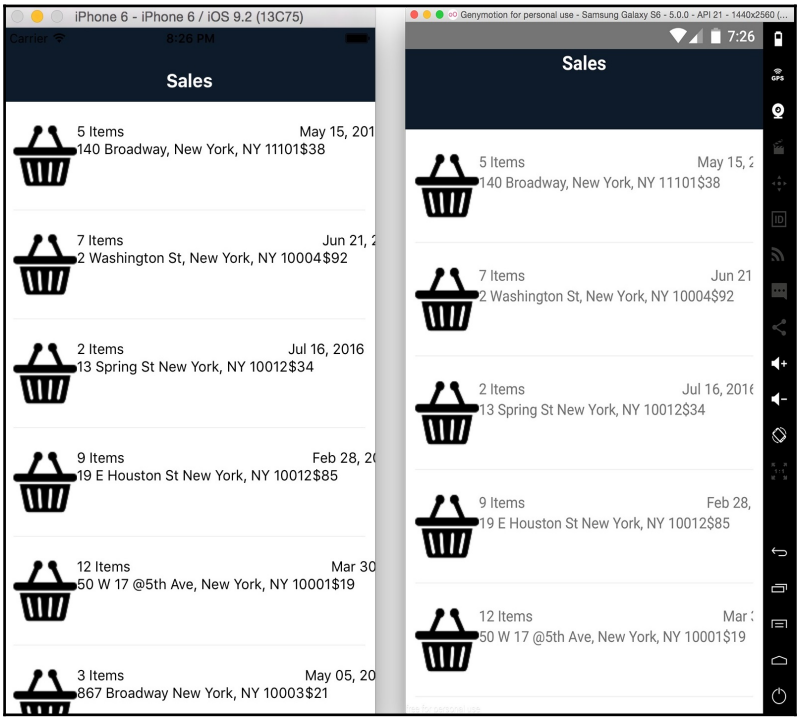
8. Now we can define the content of `renderRow`. This method receives each object containing all the information we need. We are going to display the data in three columns. In the first column we will show an icon, in the second column we will show the number of items for each sale and the address where this order will ship, and the third column will display the date and the total:

```
renderRow(record) {
  return (
    <View style={styles.row}>
      <View style={styles.iconContainer}>
        <Image source={basketIcon} style={styles.icon} />
      </View>
      <View style={styles.info}>
        <Text style={styles.items}>{record.items} Items</Text>
        <Text style={styles.address}>{record.address}</Text>
      </View>
      <View style={styles.total}>
        <Text style={styles.date}>{record.date}</Text>
        <Text style={styles.price}>${record.total}</Text>
      </View>
    </View>
  );
}
```

9. Once we have the JSX defined, it's time to add the styles. First, we will define colors, margins, paddings, and so on for the main container, and the title and the row container. In order to create the three columns for each row, we need to use the `flexDirection: 'row'` property. We will learn more about this property in another recipe in this chapter:

```
const styles = StyleSheet.create({
  mainContainer: {
    flex: 1,
    backgroundColor: '#fff',
  },
  title: {
    backgroundColor: '#0f1b29',
    color: '#fff',
    fontSize: 18,
    fontWeight: 'bold',
    padding: 10,
    paddingTop: 40,
    textAlign: 'center',
  },
  row: {
    borderColor: '#f1f1f1',
    borderBottomWidth: 1,
    flexDirection: 'row',
    marginLeft: 10,
    marginRight: 10,
    paddingTop: 20,
    paddingBottom: 20,
  },
});
```

10. If we refresh the simulators, we should see something similar to the following screenshot:

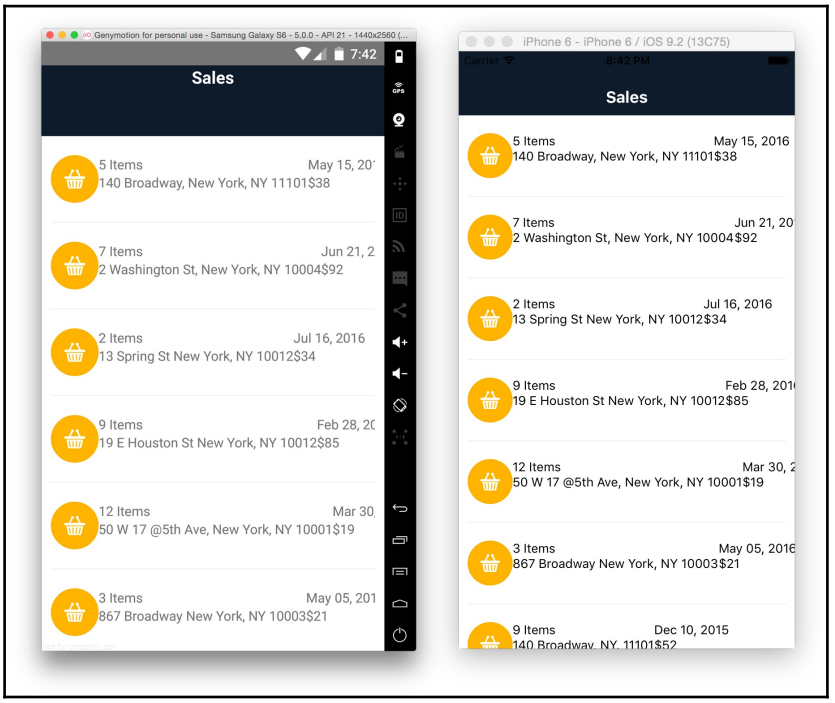


11. Now, inside the `StyleSheet` definition, let's add styles to the icon. We are going to add a yellow circle as the background and change the color of the icon to white:

```
iconContainer: {
  alignItems: 'center',
  backgroundColor: '#feb401',
  borderColor: '#feaf12',
  borderRadius: 25,
  borderWidth: 1,
  justifyContent: 'center',
  height: 50,
  width: 50,
},
icon: {
  tintColor: '#fff',
  height: 22,
  width: 22,
```

```
},
```

12. After this change, we will see a nice icon on the left side of each row, as shown in the following screenshot:

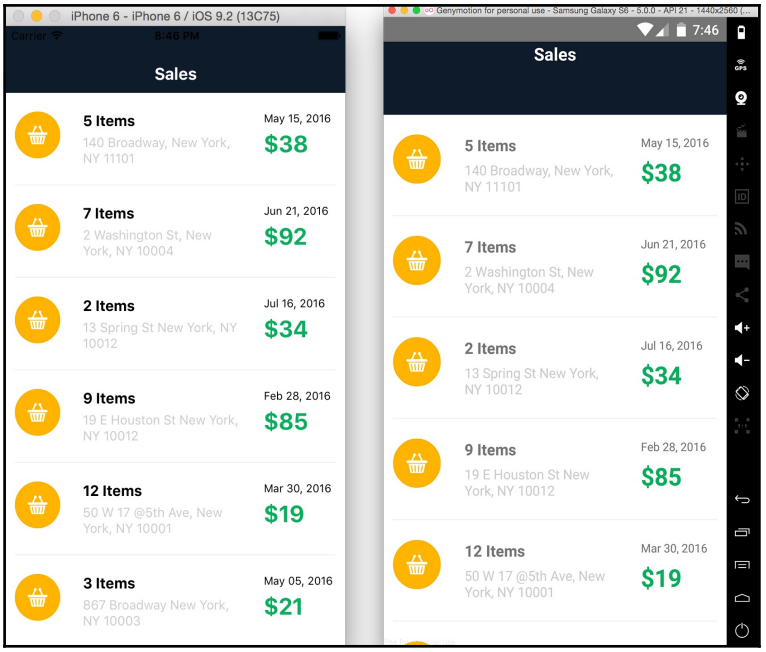


13. Finally, the styles for the text. We need to set the `color`, `size`, `fontWeight`, `padding` and a few other things:

```
info: {
  flex: 1,
  paddingLeft: 25,
  paddingRight: 25,
},
items: {
  fontWeight: 'bold',
  fontSize: 16,
  marginBottom: 5,
},
address: {
  color: '#ccc',
  fontSize: 14,
},
```

```
total: {
  width: 80,
},
date: {
  fontSize: 12,
  marginBottom: 5,
},
price: {
  color: '#1cad61',
  fontSize: 25,
  fontWeight: 'bold',
},
```

14. The end result should look similar to the following screenshot:



How it works…

In step *6*, we create the data source and added data to the state. The `ListView.DataSource` class implements performance data processing for the `ListView` component. The `rowhHasChanged` property is required, and it should be a function to compare the next element.

When filling up the data source with data, we need to call the `cloneWithRows` method and send an array of records.

If we want to add more data, we should call again the `cloneWithRows` method with an array containing the previous and new data. The data source will make sure to compute the differences and re-render the list if necessary.

In step *7*, we define the JSX to render the list. Only two properties are required for the list, the data source we already have from step *6*, and the `renderRow`.

The `renderRow` property accepts a function as a value; this function needs to return the JSX for each row.

There's more…

We have created a simple layout using flexbox; however, there's another recipe in this chapter where we will dive into more detail about using flexbox.

Once we have our list, the chances are that we are going to need to see the detail of each order. We will have to use the `TouchableHighlight` component as the main container for each row, so go ahead and give it a try. If you are not sure how to use the `TouchableHighlight` component, take a look at the *Creating a toggle button* recipe in this chapter.

Adding tabs to the viewport

Tabs are a very common component, especially in iOS apps. In this recipe, we will learn how to use the tabs component on iOS devices only. As of now, we don't have support for Android, and if we really want to use tabs, we would have to use a third-party library to add similar functionality.

Getting ready

We need to create an empty app using the React Native CLI. We will name it `TabsComponent`. Then we will create an `src` folder in the root of the project where we are going to define all our JavaScript code. We are going to use four different icons, but feel free to use your own or make sure to download the assets for this recipe.

How to do it…

1. Let's start by importing all the dependencies for this component, as well as the images we are going to use for the icons:

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

const homeIcon = require('./images/home.png');
const favIcon = require('./images/star.png');
const blogIcon = require('./images/notebook.png');
const profileIcon = require('./images/user.png');
```

2. In order to select a tab, we need to keep the current selection on the state, therefore we need to use a class, which will look something like this:

```
class MainApp extends Component {
  state = {
    selected: 'home',
  };

  selectTab(id) {
    // Defined on step 5
  }

  renderTab(options) {
    // Defined on step 4
  }

  render() {
    // Defined on step 3
  }
}
```

3. Inside of the `render` method, we need to define the tab component, along with each tab that we want to show. In this case, we are going to use the `renderTab` method to build the JSX, which will allow us to reduce our code base by calling a single function with different options:

```
render() {
  return (
    <TabBarIOS
      tintColor="#42b49a"
    >
      {this.renderTab(
        {title: 'Home', id: 'home', icon: homeIcon})}
      {this.renderTab(
        {title: 'Favorites', id: 'favorites', icon: favIcon})}
      {this.renderTab(
        {title: 'Blog', id: 'blog', icon: blogIcon})}
      {this.renderTab(
        {title: 'Profile', id: 'profile', icon: profileIcon})}
    </TabBarIOS>
  );
}
```

4. For the `renderTab` method, we need to define a few properties, such as the title of the tab, the icon, weather it is selected or not, and a callback function to define the actual selection. For now, we will use the same content for each tab, but in real-world applications we would pass the main content as a parameter as well. One of the most important properties here is the `selected` property. We can only have one tab selected at a time, and we will use the state to keep the current selection:

```
renderTab(options) {
  return (
    <TabBarIOS.Item
      title={options.title}
      selected={this.state.selected === options.id}
      onPress={() => this.selectTab(options.id)}
      icon={options.icon}
    >
      <View style={styles.container}>
        <Image source={options.icon} style={styles.icon} />
        <Text style={styles.title}>{options.title}</Text>
      </View>
    </TabBarIOS.Item>
  );
}
```

5. In the previous step, we are calling the `selectTab` function when the tab item is pressed. The idea here is to call this function when the user presses any of the tabs. We will send the `id` as a parameter and then we will set the current selection on the state:

```
selectTab(id) {
  this.setState({
    selected: id,
  });
}
```

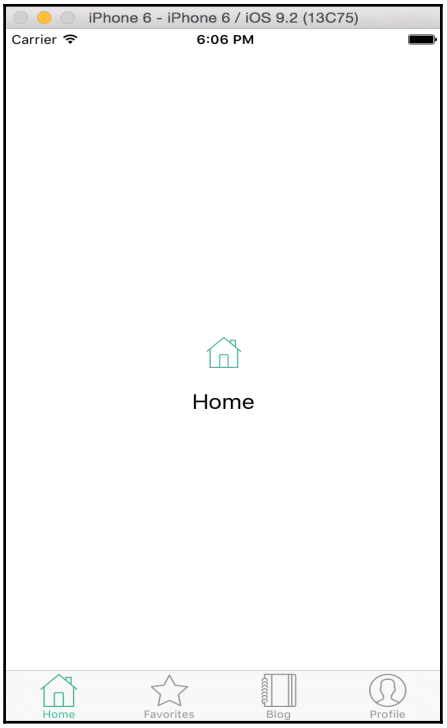
6. Let's add some styles to center the content on the screen and set a nice color to the image of each tab. We will also export the component in order to be able to import it anywhere else:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  title: {
    fontSize: 20,
    marginTop: 20,
  },
  icon: {
    width: 30,
    height: 30,
    tintColor: '#42b49a'
  },
});

export default MainApp;
```

7. Finally, we need to update the `index.ios.js` file to import our new class:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('TabsComponent', () => MainApp);
```

8. The final result should look similar to the following screenshot:



Using flexbox to create a profile page

In this recipe, we will learn about flexbox. In the previous recipes in this chapter we've been using flexbox to create some layouts, but in this recipe we will focus on the properties we have at our disposal to create a profile page.

Getting ready

We need to create an empty app using the React Native CLI–let's name it `SimpleLayout`. We are going to use a few icons and an image to show in the profile page. You can use your own or download the assets for this recipe.

How to do it…

1. Let's start by creating an `src` folder in the root of the project. Then we need to create the `MainApp.js` file where we are going to define the code for this recipe.
2. In the new file, let's import the dependencies of our class, just a few basic components, the `profileImage` and some `icons`. Make sure to create the `images` folder and add some icons there, as well as the user profile image:

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

const profileImage = require('./images/user-profile.jpg');
const friendsIcon = require('./images/profile.png');
const favIcon = require('./images/plain-heart.png');
const msgIcon = require('./images/chat.png');
```

3. We are going to create a class. It will have the user's data on the state and two methods to render the JSX. As a rule of thumb, we should always try to split the JSX into reusable methods, which will allow us to have a more readable code base:

```
class MainApp extends Component {
  state = {
    name: 'Crysfel',
    lastName: 'Villa Roman',
    occupation: 'Software Engineer',
    friends: '1,200',
    favorites: '2,491',
    comments: '4,832',
  };

  renderStat(options) {
    // Defined on step 5
  }

  render() {
    // Defined on step 4
  }
}
```

4. Inside the `render` method, we are going to set the profile image as the background of the app. Then we need a container to move the user's information to the bottom of the screen. Inside the container, we will define two more containers, one for the basic information and one to show the number of `friends`, `favorites`, and `comments`:

```
render() {
  const {
    name,
    lastName,
    occupation,
    friends,
    favorites,
    comments,
  } = this.state;

  return (
    <Image source={profileImage} style={styles.container}>
      <View style={styles.info}>
        <View style={styles.personal}>
          <Text style={styles.name}>{name}
          {lastName}
          </Text>
          <Text style={styles.occupation}>
            {occupation.toUpperCase()}
          </Text>
        </View>
        <View style={styles.stats}>
          {this.renderStat(
          { icon: friendsIcon,value: friends, selected: true })}
          {this.renderStat({ icon: favIcon,value: favorites })}
          {this.renderStat({ icon: msgIcon,value: comments })}
        </View>
      </View>
    </Image>
  );
}
```

5. In the `renderStat` method, we are going to define just a container with an `Image` and `Text`. This method receives an object of properties with the values to use; if the `selected` property is `true`, we are going to add specific styles for this:

```
renderStat(options) {
 return (
   <View style={styles.stat}>
     <Image
```

```
          source={options.icon}
          style={[styles.icon, options.selected ?
          styles.selected :                null]}
        />
        <Text style={styles.counter}>{options.value}</Text>
      </View>
    );
  }
```

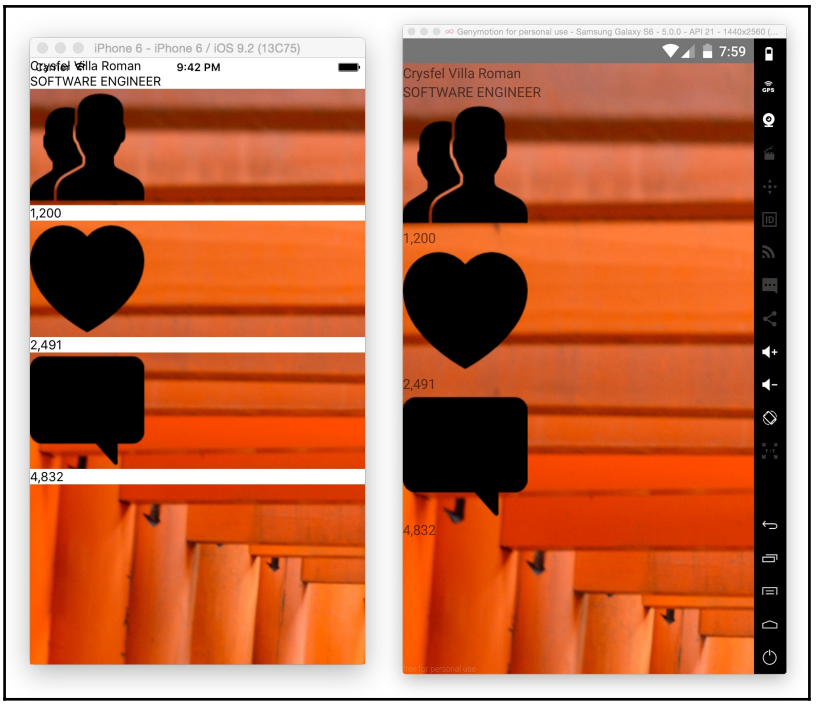
6. Now we should `export` our new class, which will allow us to use it anywhere else:

```
export default MainApp;
```

7. In order to test our new component, we need to import it into the `index.ios.js` and `index.android.js` files:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('SimpleLayout', () => MainApp);
```

8. We are now ready to start creating our layout! Without any styles defined, our app should look something like the following screenshot. It doesn't look good at all, but we have the JSX in place:



9. First, we will fix the background image. You can see that the image is huge and doesn't fit on the screen. In order to fix this issue, we need to set the `width` and `height` to `null`. We also need to set `flex:1`, which will automatically get the height of the parent:

```
const styles = StyleSheet.create({
  container: {
   flex: 1,
    height: null,
    width: null,
  },
});
```

10. Now we are going to move the information container to the bottom of the screen. To accomplish this, we need to set the position to `absolute` and the `bottom`, `left`, and `right` to `0`:

```
info: {
  backgroundColor: 'rgba(0,0,0,0.5)',
  bottom: 0,
  left: 0,
  position: 'absolute',
  right: 0,
},
```

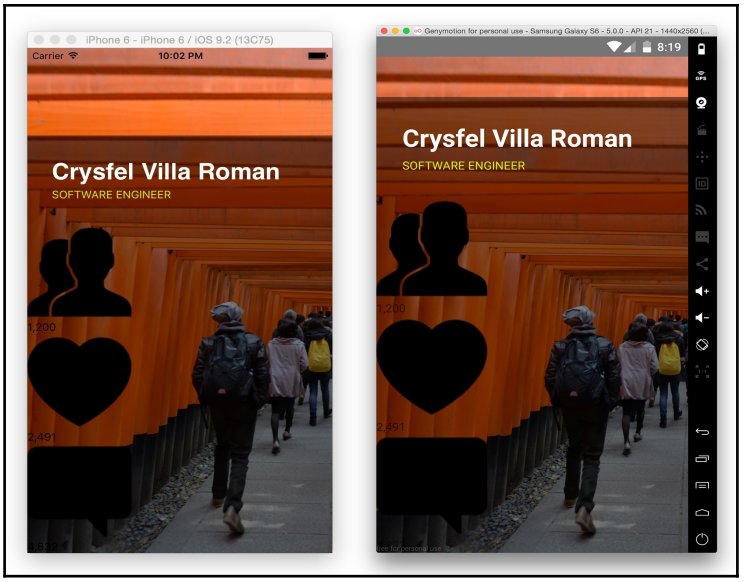
11. While this works as expected, we could reduce our code by using the `StyleSheet.absoluteFillObject` object, which basically contains the absolute position and all four sides to set `0`. In our case, we need the `top` to be `null`. The following code is exactly the same as the previous one:

```
info: {
  ...StyleSheet.absoluteFillObject,
  backgroundColor: 'rgba(0,0,0,0.5)',
  top: null,
},
```

12. We are going to set the `fontSize`, `color`, and some `padding` for the personal information data:

```
personal: {
  padding: 30,
},
name: {
  color: '#fff',
  fontFamily: 'Helvetica',
  fontSize: 30,
  fontWeight: 'bold',
},
occupation: {
  color: '#d6ec1b',
  marginTop: 5,
},
```

13. At this point we should have something similar to the following screenshot:



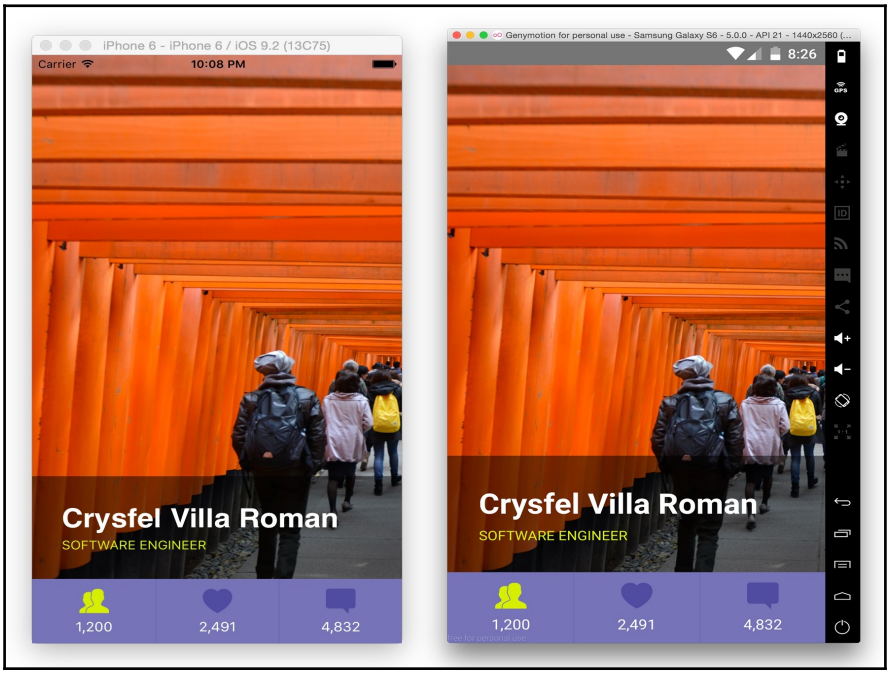
14. We are getting there! Now let's style the icons for friends, favorites, and comments. We are going to define the `tintColor`, `width`, `height`, `color`, and `marginTop`:

```
selected: {
  tintColor: '#d6ec1b',
},
icon: {
  tintColor: '#504f9f',
  height: 30,
  width: 30,
},
counter: {
  color: '#fff',
  fontSize: 15,
  marginTop: 5,
},
```

15. Finally, we are going to arrange the icons horizontally by changing the `flexDirection` to `row`. We are also adding some additional styles to set the `backgroundColor`, `padding`, and a `borderLeftWidth`:

```
stats: {
  flexDirection: 'row',
},
stat: {
  alignItems: 'center',
  backgroundColor: '#7675b7',
  borderColor: '#6e6db1',
  borderLeftWidth: 1,
  flex: 1,
  padding: 10,
},
```

16. After all these styles have been applied, let's reload the simulator and see what we have so far:



How it works…

This profile is looking really good, and it was really simple to accomplish it just by using flexbox. We have arranged the icons horizontally; we have used an image as a background, we set the position of the container to absolute so we could move it around the screen. We can use many of these techniques in real-world applications to create amazing layouts.

When using flexbox, we have two directions, `row` and `column`:

- `row`: Allows us to arrange the children of the container horizontally
- `column`: This is the default direction, and arranges the children of the container vertically

When setting `flex:1` in any of the children of the container, we are making that child flexible.

If we have three children (as in this recipe) and each of the elements has `flex:1`, the layout engine will render all of them to have the same width. If we change the orientation of the device from portrait to landscape, the render engine will render the three children with the new width according to the orientation.

Flexbox is great to support different screen resolutions as well. As you can see in the previous image, the iOS and Android simulators have different resolutions, and the layout looks good on both devices.

There's more…

There's a lot more to talk about flexbox, but for now let's just touch the surface. In `Chapter 2`, *Implementing Complex User Interfaces*, we will learn more about layouts. We will create a complex layout to use all the other available properties.

Setting up a navigator

One of the most popular features in React Native is the **navigator**. This component allows us to add or remove views easily. When we add a new view, the navigator will transition with a nice animation to the new view, and when removing the latest view, the navigator will go to the previous view with a nice animation as well.

Getting ready

Before we start working on this recipe, we need to create a new app using the React Native CLI–let's name it `UsingNavigator`. We are going to create a simple music app similar to **Spotify**. At the home page, we will display a list of songs, and we are going to display the details, whenever any of these songs are tapped.

For this recipe, we are going to use three classes, so let's create an `src` folder where all our code will be. Then we'll create three files, `MainApp.js`, `Home.js`, and `Detail.js`.

How to do it…

1. This time, we are going to start by updating the `index.ios.js` and `index.android.js` files. We are going to require the `MainApp` class, which, for now, is empty, because we want to test our app while adding the new features:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('UsingNavigator', () => MainApp);
```

2. Let's create the main class. Here, we are going to import all the views that we want to display in the navigator, as well as the dependencies for this component:

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Navigator,
} from 'react-native';
import Home from './Home';
import Detail from './Detail';
```

3. Once we have the dependencies, we need to create the class. Inside the `render` method, we will only define the `Navigator`. This component will be responsible for loading the correct scene based on the route. Using the `configureScene` property, we can set the default animation for all transitions. For this example, we will animate the new view from the bottom to the top of the screen, but we can also define an animation from left to right, or any other direction. I recommend you go to the documentation to see all the available options that we have:

```
class MainApp extends Component {

  renderScene(route, navigator) {
    // Defined on step 4
  }

  render() {

    return (
      <Navigator
        ref="navigator"
        style={styles.container}
        configureScene={(route) =>
        Navigator.SceneConfigs.FloatFromBottom}
        initialRoute={{}}
        renderScene={this.renderScene}
      />
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
});
export default MainApp;
```

4. The `renderScene` method will get executed every time we push or pop a scene. Here, we need to decide which view we will render by returning the component based on the route. We could have as many views as needed. In this case, we only have the `Detail` and the `Home`:

```
 renderScene(route, navigator) {
  if (route.song) {
    return (
      <Detail song={route.song} navigator={navigator} />
    );
  }

  return <Home navigator={navigator} />;
}
```

5. We are done with the main class. Now we can work on the `Home` component. We will start by importing all the dependencies for this class:

```
import React, { Component } from 'react';
import {
  StyleSheet,
  View,
  Image,
  Text,
  ScrollView,
  TouchableHighlight,
} from 'react-native';
```

6. We are going to define the data for the songs in the `state` of the component. We will have several sections with a list of songs, for example, `'Just for you'`, `'Recently played'`, and `'Popular music'`. For now, I will define the structure of one section only, because I don't want to clutter this book, but you can duplicate the same structure of data to create more sections. Here's what the class looks like with the data:

```
class Home extends Component {
 state = {
   forYou: { // Please duplicate this data
     title: 'Just for you',
     root:
     'https://s3.amazonaws.com/crysfel/public/book/01/07',
     songs: [
       {title:'Some nice song', image: '1.jpg'},
       {title:'One more nice song', image: '2.jpg'},
       {title:'Here is one more song', image: '3.jpg'},
       {title:'Really nice song', image: '4.jpg'},
       {title:'I love this song', image: '5.jpg'},
       {title:'This is a song', image: '6.jpg'},
     ],
   },
 };

 onSelectSong(song) {
   // Defined on step 10
 }

 renderSong(section, song, index){
   // Defined on step 9
 }

 renderSection(options) {
  // Defined on step 8
```

```
  }

  render() {
    // Defined on step 7
  }
}
export default Home;
```

7. In the `render` method, we are going to define a title bar and three sections. Remember to duplicate the data on the state in order for this to work correctly:

```
  render() {
   const {
     forYou,
     played,  // Name of the duplicated data
     popular, // Name of the duplicated data
   } = this.state;

   return (
     <View style={styles.container}>
       <Text style={styles.title}>Home</Text>
       {this.renderSection(forYou)}
       {this.renderSection(played)}
       {this.renderSection(popular)}
     </View>
   );
 }
```

8. For each section, we are calling the `renderSection` method, which will render the title of the section and the list of songs in this group. We are going to use the `ScrollView` component to allow the user to scroll through the list horizontally. To render the songs, we need to loop the `songs` array of each section. We can do this by using the `map` method and an arrow function:

```
  renderSection(options) {
   return (
     <View style={styles.section}>
       <Text
         style={styles.sectionTitle}
       >
         {options.title.toUpperCase()}
       </Text>
       <ScrollView
         horizontal
         showsHorizontalScrollIndicator={false}>
         {
          options.songs.map(
```

```
            (song, index) =>
            this.renderSong(options, song, index)
          )
        }
      </ScrollView>
    </View>
  );
}
```

9. Now we need to render the song. We will display the image and the title for each song. The image is hosted on Amazon S3. We are going to use the `TouchableHighlight` component because we want to detect the press event to show the detail of each song. It's important to mention that this component only accepts a single child; therefore, we need to use a wrapper to group the image and the title; otherwise, we will get errors:

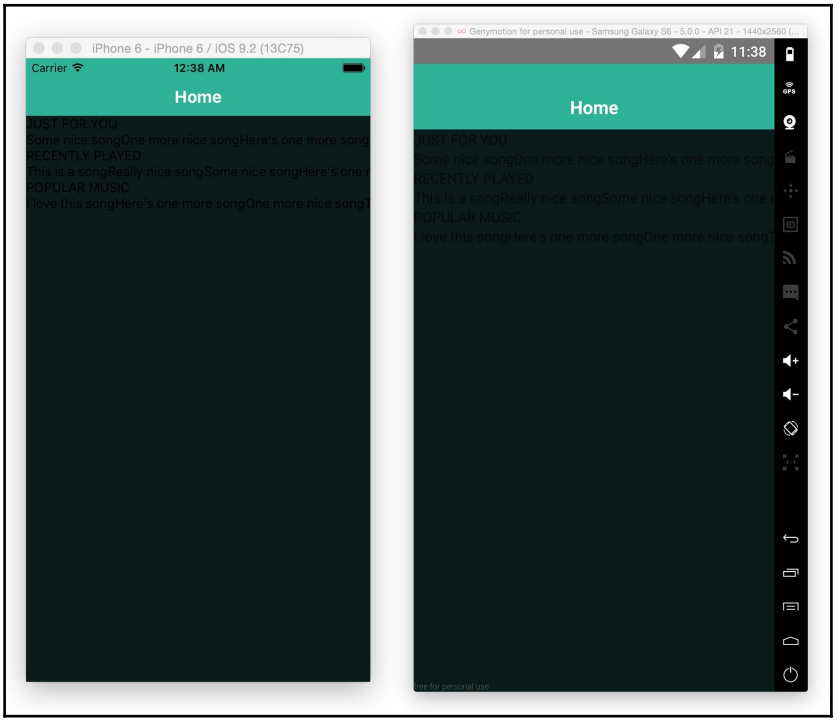
```
renderSong(section, song, index){
 return (
   <TouchableHighlight
     onPress={() => this.onSelectSong(song)}
     style={styles.song} key={index}
   >
     <View>
       <Image
         source={{uri:`${section.root}/${song.image}`}}
         style={styles.image}
       />
       <Text style={styles.songTitle}>{song.title}</Text>
     </View>
   </TouchableHighlight>
 );
}
```

10. In the previous step, we are calling the `onSelectSong` method when the user presses the button. This function is the one that will run the transition on the navigator. All we need to do is call the `push` method on the navigator and pass the details object. The `renderScene` method will check if the `song` object is there, and will show the scene that we need:

```
onSelectSong(song) {
 this.props.navigator.push({song});
}
```

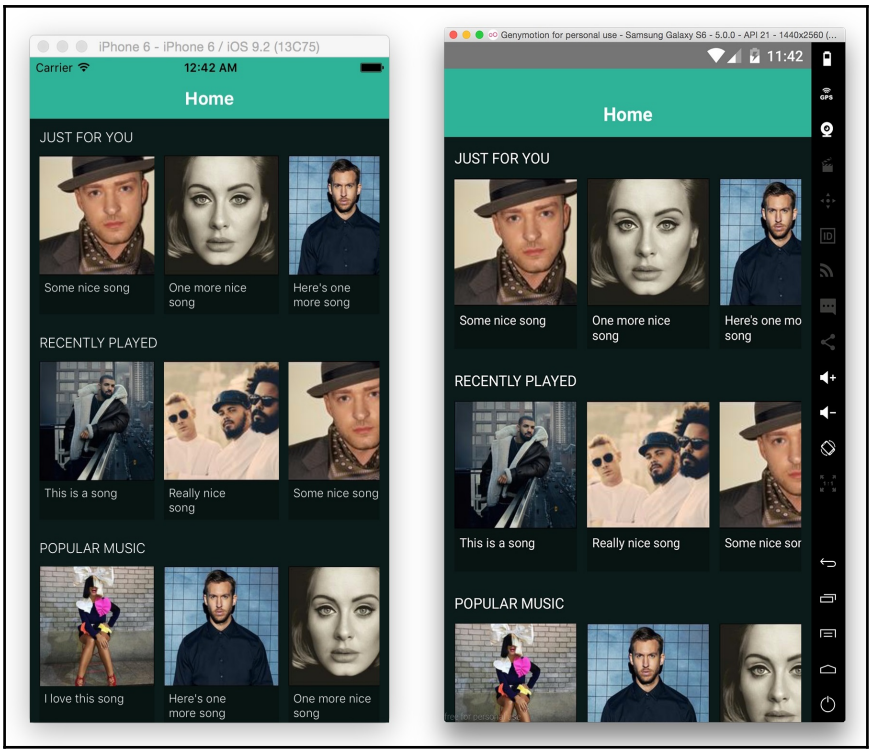
11. We are done with JSX. Now we need to add some styles to make this look pretty. Let's start by adding styles to the title bar and the main `container`:

```
const styles = StyleSheet.create({
 container: {
   flex: 1,
   backgroundColor: '#0c1b1a',
 },
 title: {
   backgroundColor: '#37b298',
   color: '#fff',
   padding: 10,
   paddingTop: 30,
   textAlign: 'center',
   fontWeight: 'bold',
   fontSize: 18,
 },
});
```



12. Next, let's style the `sectionTitle`, the `image`, and the `songTitle`, something really simple–just adding some padding, color, and setting the width and height for each image:

```
section: {
  padding: 10,
},
sectionTitle: {
  color: '#fff',
  fontWeight: '200',
  paddingBottom: 10,
},
song: {
  backgroundColor: '#081412',
  marginRight: 10,
},
image: {
   width: 120,
   height: 120,
 },
 songTitle: {
   color: '#f1f1f1',
   fontWeight: '200',
   fontSize: 12,
   flex: 1,
   padding: 5,
   width: 100,
 }
```



13. It's looking amazing! We can now scroll the list of each section horizontally. If we tap on any of the songs, we will get an error because we haven't defined the detail view. Let's start by importing the dependencies on the `Detail.js` file. We are going to import the `PropTypes` object. This object allows us to define the types of property our component will support. This step is very important because we can receive data from other components by using properties. In this example, we will receive the data from the home component, which will be an object with the song's data:

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

const { object } = PropTypes;
const root =
'https://s3.amazonaws.com/crysfel/public/book/01/07';
```

14. The class is a lot simpler. We are only defining the image and title from the song that we will receive in the properties; then we will display a list of other songs from the same artist. To make things a lot simpler, this list will be a hardcoded list of `Text` elements:

```
class Detail extends Component {
  static propTypes = {
    song: object,
    navigator: object,
  };

  render() {
    const { song } = this.props;

    return (
      <View style={styles.container}>
        <View style={styles.info}>
          <Image
            source={{uri: `${root}/${song.image}`}}
            style={styles.image}
          />
          <Text style={styles.title}>{song.title}</Text>
          <View style={styles.playContainer}>
            <Text style={styles.play}>Play song</Text>
          </View>
        </View>
        <Text style={styles.other}>01 - One more song</Text>
        <Text style={styles.other}>02 - Other song here</Text>
        <Text style={styles.other}>
        03 - This is the last song</Text>
        <Text style={styles.other}>
        04 - Maybe this is the last song?</Text>
        <Text style={styles.other}>
        05 - Why not one more song?</Text>
        <Text style={styles.other}>
        06 - Finally this is the last song</Text>
      </View>
    );
  }
}

export default Detail;
```

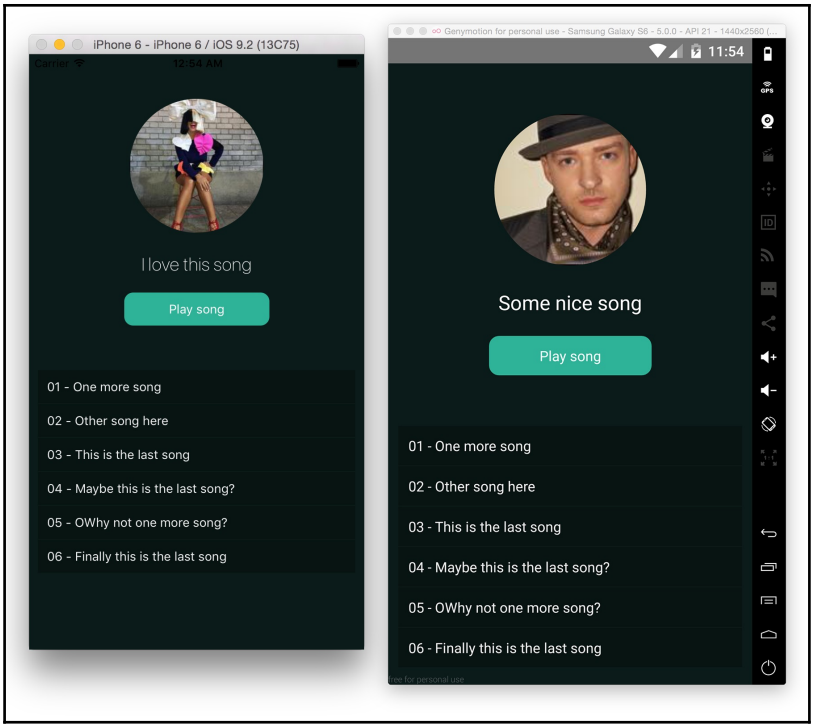
15. Now let's style this component. We are going to display the image as a circle, and then we will display the title of the song we are receiving:

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: '#0c1b1a',
    flex: 1,
  },
  info: {
    padding: 50,
    alignItems: 'center',
  },
  image: {
    width: 150,
    height: 150,
    borderRadius: 75,
  },
  title: {
    fontSize: 20,
    fontWeight: '200',
    color: '#fff',
    marginTop: 23,
  },
});
```

16. Finally, we will display the list of additional songs and the play button, which is really simple:

```
playContainer: {
  backgroundColor: '#37b298',
  padding: 10,
  paddingRight: 50,
  paddingLeft: 50,
  borderRadius: 10,
  marginTop: 20,
},
play: {
  color: '#fff',
},
other: {
  color: '#f1f1f1',
  padding: 10,
  marginRight: 10,
  marginLeft: 10,
  backgroundColor: '#081412',
  marginBottom: 1,
},
```

17. At this point, we should have something similar to the following screenshot:



There's more…

The `navigator` is a very important component. Almost every application requires some sort of navigator to add some nice transitions. In this recipe, we covered the cross-platform navigator, which works on iOS and Android. This is because everything is on the JavaScript side. If we require something native, we should take a look at `NavigatorIOS` and `ViewPagerAndroid`.

2
Implementing Complex User Interfaces

In this chapter, we will cover the following recipes:

- Creating a reusable button with theme support
- Building a complex layout for tablets using flexbox
- Including custom fonts on iOS
- Including custom fonts on Android
- Using font icons
- Dealing with universal apps
- Detecting orientation changes
- Using a WebView to open external websites
- Rendering simple HTML elements using native components
- How to create a form component

Introduction

In this chapter, we will implement complex user interfaces. We will learn more about using flexbox to create components that work on different screen sizes, how to detect orientation changes, how to work with universal apps, among many other things.

It's important to mention that we will create an empty app for each recipe in this chapter. If you are not sure how to do this using the React Native CLI, I recommend you read the first chapter of this book.

Creating a reusable button with theme support

Reusability is very important when developing software; we should avoid repeating the same thing over and over again, and instead we need to create small components that we can reuse as many times as needed.

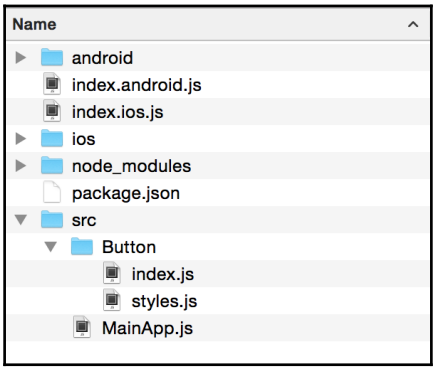
In this recipe, we will create a button component, and we are also going to define several properties to change it's look and feel. While going through this recipe we will learn how to use props and how to dynamically apply different styles to a component.

Getting ready

We need to create an empty app using the React Native CLI, let's name it `CustomComponents`. Feel free to use any other name, just make sure to use the same name in step number *9*.

How to do it…

1. In an empty app, we need to create an `src` folder at the root level of our project. Inside this folder, we need a JavaScript file called `MainApp.js`. We also need one more folder called `Button` with `index.js` and `styles.js` files:



2. Let's start by importing the dependencies for our new component. In this case, we will create a `Button` component; therefore, we'll use the `Text`, `TouchableOpacity`, and `View` components.

   If we would like to add support for icons, we should import the `Image` component as well, but to keep things simple we're not going to cover that in this recipe.

   We will define the styles in a different file later in this recipe, but for now, let's just import them all. Open the `src/Button/index.js` file and add the following code:

```
import React, { Component, PropTypes } from 'react';
  import {
  Text,
  TouchableOpacity,
  View,
} from 'react-native';
import Base, { Default, Danger, Info, Success } from './styles';
```

3. Now that we have our dependencies imported, let's define the class for this component. We are going to need some properties and two methods only. It's also required that we export this component so we can use it elsewhere:

```
const { array, string, object, bool, func, any } = PropTypes;

class Button extends Component {
  static propTypes = {
    // Defined on step 4
  };

  getTheme() {
    // Defined on step 5
  }

  render() {
    // Defined on step 6
  }
}

 export default Button; // Don't forget to export it!
```

4. Let's define the properties for this component. The `children` property will receive all the `children` components defined for the button, which means we can use images as well.

   We are going to define some Boolean configurations for the styles, for example, `danger`, `info`, and `success`. When `true`, we will assign the appropriate styles to the button. The `style` prop will receive additional styles for the main container; this will make our new component more flexible.

   The `onPress` prop will receive a callback function that will be executed when the user presses the button:

```
static propTypes = {
  children: any,
  danger: bool,
  info: bool,
  style: View.propTypes.style,
  success: bool,
  onPress: func,
};
```

5. We need to select the styles to apply to our component based on the given `props`. For this, we will define the `getTheme` method. This method will check if any of the `props` are `true` and will return the appropriate styles; otherwise, it will return the default style:

```
getTheme() {
  const { danger, info, success } = this.props;

  if (info) {
    return Info;
  }

  if (success) {
    return Success;
   }

  if (danger) {
     return Danger;
   }

   return Default;
}
```

6. It's required that all components have a `render` method. Here, we need to return the JSX elements for this component. In this case, we will get the styles for the given `props` and apply them to the `TouchableOpacity` component.

   We are also defining a label for the button; inside this label, we will render the `children` prop. If a callback function is received, then it will be executed when the user presses this component:

```
render() {
  const theme = this.getTheme();
  const {
    children,
    onPress,
    style,
    rounded,
  } = this.props;

  return (
    <TouchableOpacity
      activeOpacity={0.8}
      style={[
        Base.main,
        theme.main,
        rounded ? Base.rounded : null ,
        style,
      ]}
      onPress={onPress}
    >
      <Text style={[Base.label, theme.label]}>{children}</Text>
    </TouchableOpacity>
  );
}
```

7. We are almost done with our component; styles are still missing, but first let's work on the `MainApp.js`. We need to import the dependencies as well as the `Button` component we have created.

   We are going to display an alert message when the user clicks the button; therefore, we need to import the `Alert` component. This component works on iOS and Android:

```
import React from 'react';
import { Alert, StyleSheet, View } from 'react-native';
import Button from './Button';
```

8. Once we have all the dependencies, let's define a stateless component that renders a few buttons:

   The first button will use the default theme; the second button will use the success styles, which we will style with a nice green color as a background. The last button will display an alert when it gets pressed. For that, we need to define the callback function that will only use the `Alert` component. Here, we will just set the title and the message:

```
function onPressBtn() {
  Alert.alert('Alert', 'You clicked this button!');
}

const MainApp  = () => (
  <View style={styles.container}>
    <Button style={styles.btn}> My first button
</Button>
    <Button success style={styles.btn}> Success button
</Button>
    <Button info style={styles.btn}> Info button
</Button>
    <Button danger rounded style={styles.btn}
    onPress={onPressBtn}> Rounded button </Button>
  </View>
);
export default MainApp;
```

9. We are going to use some styles for each button; we just need to add a margin:

```
const styles = StyleSheet.create({
  container: {
      flex: 1,
      alignItems: 'center',
      justifyContent: 'center',
    },
  btn: {
    margin: 10,
  },
});
```

   Now we need to register our app in the `index.ios.js` and `index.android.js` files:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';
```

```
AppRegistry.registerComponent('CustomComponents', () => MainApp);
```

10. If we try to run the app now, we will get some errors. This is because we haven't declared the styles of our button. Let's work on that now. Inside the `styles.js` file, we need to define the base styles. These styles will be applied to every instance of the button. Here, we will define a radius, padding, font color, and all the common styles that we need for this component:

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

const BaseStyles = StyleSheet.create({
  main: {
    padding: 10,
    borderRadius: 3,
  },
  label: {
    color: '#fff',
  },
  rounded: {
    borderRadius: 20,
  },
});
```

11. Once we have the common styles for our button, we need to define the styles for the `Danger`, `Info`, `Success`, and `Default` themes. For that, we are going to define different objects for each theme, and then inside each theme we will use the same object but with specific styles for that theme.

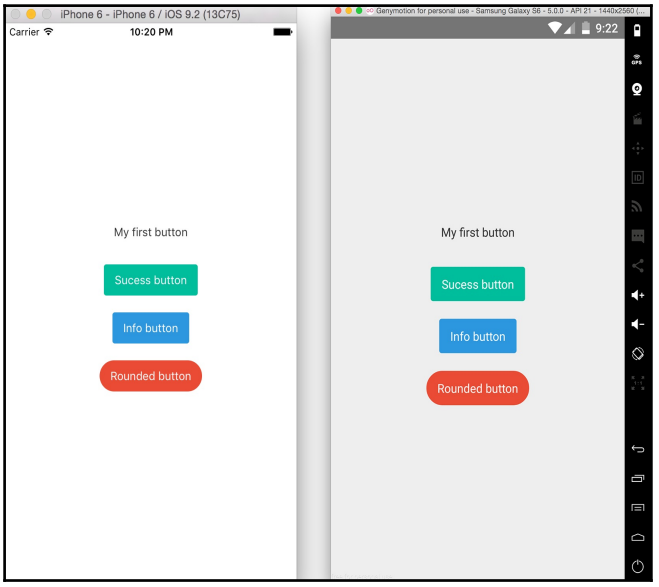
    To keep things simple, we are only going to change the `backgroundColor`, but we do have the option to use as many style properties as needed:

```
const Danger = StyleSheet.create({
  main: {
    backgroundColor: '#e74c3c',
  },
});

const Info = StyleSheet.create({
  main: {
    backgroundColor: '#3498db',
  },
});

const Success = StyleSheet.create({
  main: {
```

```
    backgroundColor: '#1abc9c',
  },
});

const Default = StyleSheet.create({
  main: {
    backgroundColor: 'rgba(0 ,0 ,0, 0)',
  },
  label: {
    color: '#333',
  },
});
```

12. Finally, let's export the styles; this step is necessary because, in the button component we are importing the common styles as well as all the available themes:

```
export default BaseStyles;
export {
  Danger,
  Info,
  Success,
  Default,
};
```

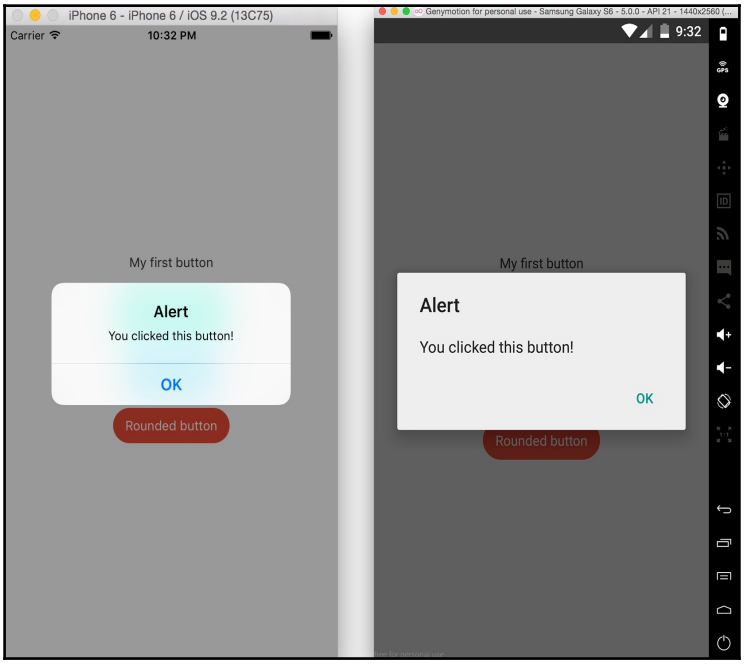


How it works…

In this example, we are using the `TouchableOpacity` component. This component allows us to define a nice animation that changes the opacity when the user presses the button.

We can use the `activeOpacity` property to set the opacity value when the button gets pressed. The value can be any number between 0 and 1, where 0 is completely transparent.

If we press the last button, we will see a native **Alert** message, as shown in the following screenshot:



Building a complex layout for tablets using flexbox

Flexbox is really convenient when it comes to creating responsive layouts. React Native uses flexbox as a layout system, and if you are already familiar with these concepts it will be really easy for you to start creating layout of any kind.

In this recipe, we will create a layout to display a list of blog posts. Each post will be a small card with an *image*, an *excerpt*, and a *button* to read more. We will use flexbox to arrange the posts on the main container based on the screen size; this will allow us to handle the screen rotation by properly aligning the cards as landscape or portrait.

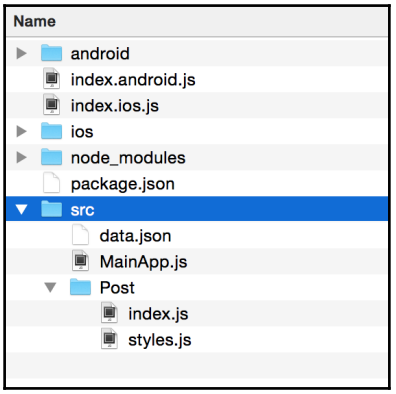
Getting ready

In order to follow the steps in this recipe, it is necessary to create an empty app using the React Native CLI. We are going to name the new app `ComplexLayout`.

How to do it…

Let's start by creating all the files and folders we will need for this recipe:

1. First, we need a `src` folder where we will write all our code; inside that folder we need a `MainApp.js` file and a `Post` folder, where we will have the code for the post component. Here, we should create the `index.js` and `styles.js` files. Finally, we need a `data.json` file where we will define a list of posts. Our project should look as shown in the following screenshot:



2. Let's open the `index.android.js` and `index.ios.js` files. We need to delete the current code and import the `MainApp` component, then use the `AppRegistry` to register and bootstrap the app:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('ComplexLayout', () => MainApp);
```

3. At the moment the `MainApp` component doesn't exist, so if we try to run or refresh the simulator, it will fail. Let's work on this component. First, we need to import the dependencies for this class.

   We are going to use a `ListView` component to render the list of posts; we also need to display `Text` and `View` as containers. We are going to create a custom `Post` component to render each post on the list, and don't forget to also load the `.json` file with the data:

```
import React, { Component } from 'react';
import { ListView, StyleSheet, Text, View } from 'react-native';
import Post from './Post';
import data from './data.json';
```

4. Let's create the class for the `MainApp` component; here, we will use the data from the `.json` file to create the `dataSource` for the list.

   In the `render` method, we are going to define something very simple, a top toolbar and the list component. We are going to use the `Post` component for every record and the `datasource` from the `state`.

   If you have any questions regarding the `ListView` component, you should take a look at the recipe in `Chapter 1`, *Getting Started*, where we created a list of orders:

```
class MainApp extends Component {
  constructor(props) {
    super(props);
    const ds = new ListView.DataSource({
       rowHasChanged: (r1, r2) => r1 !== r2,
    });
    this.state = {
      dataSource: ds.cloneWithRows(data.posts),
    };
  }
```

```
  render() {
    return (
      <View style={styles.container}>
       <View style={styles.toolbar}>
          <Text style={styles.title}>Latest posts</Text>
       </View>
        <ListView
          dataSource={this.state.dataSource}
          renderRow={post => <Post {...post} />}
          style={styles.list}
          contentContainerStyle={styles.content}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  // Defined on step 13
});

export default MainApp;
```

5. Two files are still missing: the `.json` file with the data and the `post` component. In this step, we will create the data that we are going to use for each post.

   To make things simple, I will only define one record of data, but feel free to copy and paste the same record as many times as you want. You should change the title and the image; I have uploaded six images to Amazon S3, `01.jpg` to `06.jpg`, in case you want to use them for this example:

```
{
  "posts": [
     {
        "title": "Creating custom components",
        "img":"01.jpg",
        "content": "In this post we will learn how...",
        "author": "Crysfel Villa"
   },
   // Add more records here
  ]
}
```

6. Now that we have the data, we are ready to work on the `Post` component. Here, we need to display the image, title, and button. In this case, we will use a stateless component.

   The following code uses all the components we know from `Chapter 1`, *Getting Started*. If something is unclear, please review that chapter again.

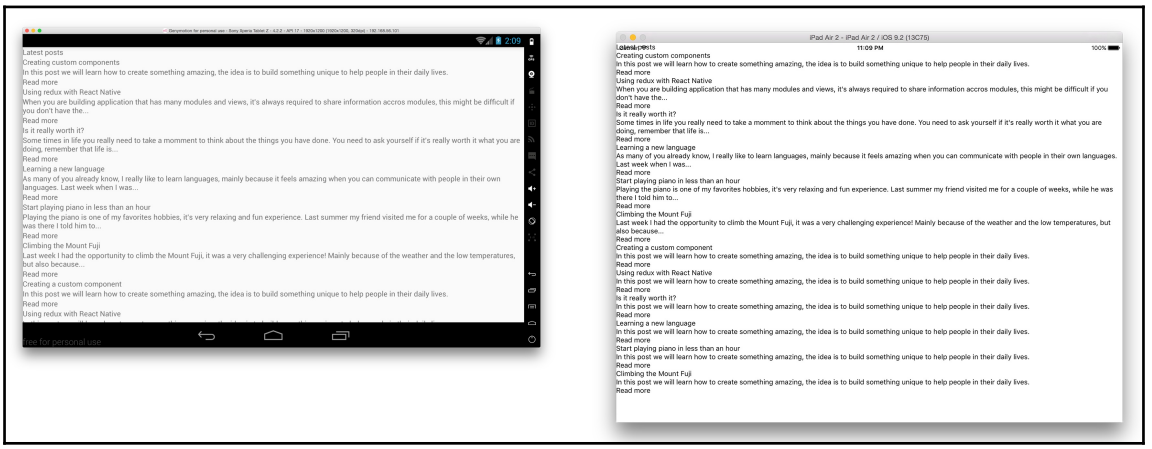
   This component receives the data as a parameter, and then we use it on the components; for example, in the `source` property of the image, we define the URL to Amazon S3 and use the parameter to set the name of the image. As mentioned before, we can use JPG images from 01 to 06:

```
import React, { PropTypes } from 'react';
import {Image, Text, TouchableOpacity, View} from 'react-native';
import styles from './styles';

const Post  = ({ content, img, title }) => (
  <View style={styles.main}>
    <Image
      source=
{{uri:`https://s3.amazonaws.com/crysfel/public/book/02/01/${img}`}}
      style={styles.image}
    />
    <View style={styles.content}>
      <Text style={styles.title}>{title}</Text>
      <Text>{content}</Text>
    </View>
    <TouchableOpacity style={styles.button} activeOpacity={0.8}>
      <Text style={styles.buttonText}>Read more</Text>
    </TouchableOpacity>
  </View>
);

const { string } = PropTypes;
Post.propTypes = {
  content: string,
  title: string,
  img: string,
};

export default Post;
```

7. Once we have defined the component, we also need to define the styles for each post. Open the `styles.js` file and write the following code:

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

const styles = StyleSheet.create({
  // Defined in later steps
});

export default styles;
```

8. If we try to run the app, we will be able to successfully see the data from the `.json` file on the screen; however, we haven't applied any styles:



9. We have everything we need on the screen; we are now ready to start working on the layout. First, let's add the styles to the `main` post container: `width`, `height`, `borderRadius`, and a few others. Open the `src/Post/styles.js` file and add the following code:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: '#fff',
    borderRadius: 3,
    height: 340,
    margin: 5,
    overflow: 'hidden',
    width: 240,
  },
});
```

10. By now, we should see small boxes vertically aligned. That's some progress, but let's add some styles to the image so we can see it onscreen.

    The `resizeMode` property will allow us to set how we want to resize the image; in this case, by selecting `cover`, the image will keep the aspect ratio:

```
image: {
  backgroundColor: '#ccc',
  height: 120,
  resizeMode: 'cover',
  },
```

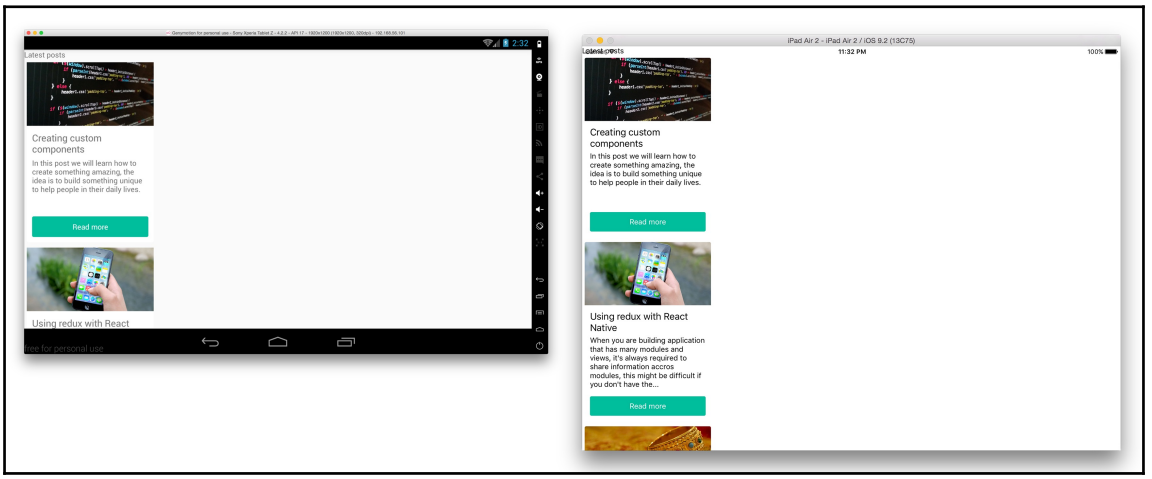
11. For the content of the post, we want to take all the available height on the card; therefore, we need to make it flexible and add some padding. For the `title`, we will only change the `font size` and add a `margin` to the bottom:

```
  content: {
    padding: 10,
    flex: 1,
  },
  title: {
    fontSize: 18,
    marginBottom: 5,
  },
```

12. Finally, for the button, we will set the `backgroundColor` to green and the text to white. We also need to add some spacing:

```
button: {
  backgroundColor: '#1abc9c',
  borderRadius: 3,
  padding: 10,
  margin: 10,
},
buttonText: {
  color: '#fff',
  textAlign: 'center',
},
```

If we refresh the simulator, we should see our posts in small cards. Currently, the cards are arranged vertically, but we need to render all of them horizontally. We are going to fix that in the following steps:



13. If we try to scroll through the content, our current implementation will not work. For now, we can only see the first three items on the list. In order to fix this problem, we need to set the height of the main container to fit the screen; for this, we will use flexbox. Let's open the `MainApp.js` file and add the following styles to the main container:

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

    Now if we try again to scroll down, we will be able to see the list of posts that we defined in the `data.json` file.

14. We need to show a toolbar at the top. For that, we just need to define some paddings and colors. This is a really simple step:

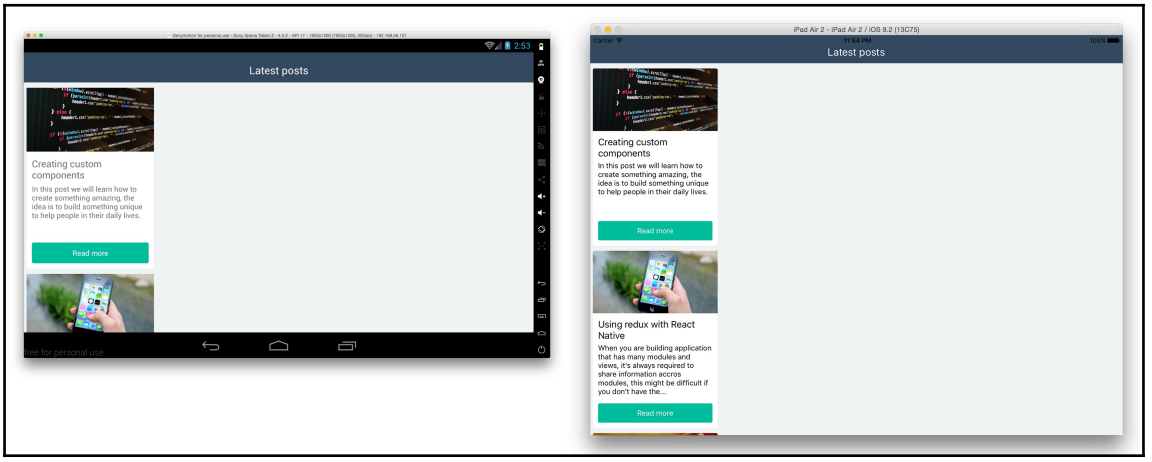
```
toolbar: {
  backgroundColor: '#34495e',
  padding: 10,
  paddingTop: 20,
},
title: {
```

```
    color: '#fff',
    fontSize: 20,
    textAlign: 'center',
  },
```

15. Let's add some simple styles to the list, just a nice background color and some padding:

```
  list: {
    backgroundColor: '#f0f3f4',
    flex: 1,
    paddingTop: 5,
    paddingBottom: 5,
  },
```

    Here, the `flex` property is making sure the list takes all the available height on the screen. We only have two components here: the toolbar and the list; the toolbar is taking about 50 px. If we make the list flexible, it will take all the available space, which is exactly what we want when rotating the device or when running the app in different screen resolutions:



16. We are almost done with this app; all we need to do is arrange the cards horizontally and make sure to render the cards in the `row`. This can be achieved with flexbox in three simple steps:

```
  content: {
    flexDirection: 'row',
    flexWrap: 'wrap',
```

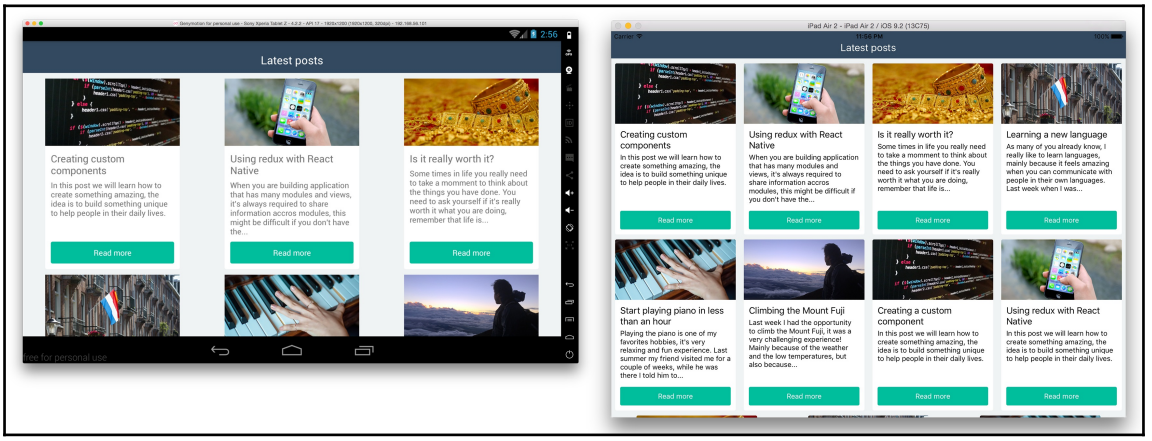
```
    justifyContent: 'space-around',
},
```

First, we need to add these styles to the `contentContainerStyle` prop in the list component. Internally, the list will apply these styles to the items container.

We need to change the `flexDirection` to `row`; this will horizontally align the cards on the list; however, there's a problem: we can only see one single row of posts, and we can't scroll horizontally.

To fix the problem, we need to wrap the items. We do this setting `flexWrap`, which will automatically move the items that don't fit in the view to the next.

We've achieved our layout; however, if we run our app on an Android device, we will see that there's a lot of space on the right side of the screen. In order to fix this problem, we will need to distribute that space between the items; to do that, we can use the `justifyContent` property and set it to `space-around`, which will automatically distribute the space around the items evenly:



We are done with our layout and it's looking really good. If you try rotating the device from landscape to portrait, the cards will automatically be arranged.

There's more…

There are some differences between iPad and the Android tablet; there are only three cards on Android and a lot of space between each card. In order to fix this problem, I would recommend calculating the width of each card based on the screen resolution; you should take a look at the *Dealing with universal apps* recipe in this chapter to learn more about handling dimensions.

Including custom fonts on iOS

At some point, we are going to need to display text with a custom font family; until now, we have been using the default font, but we can use any other that we like.

In this recipe, we will import a few fonts in Xcode and then we will display a text using different font families. We will also use different styles of a single font, such as bold and italic.

Getting ready

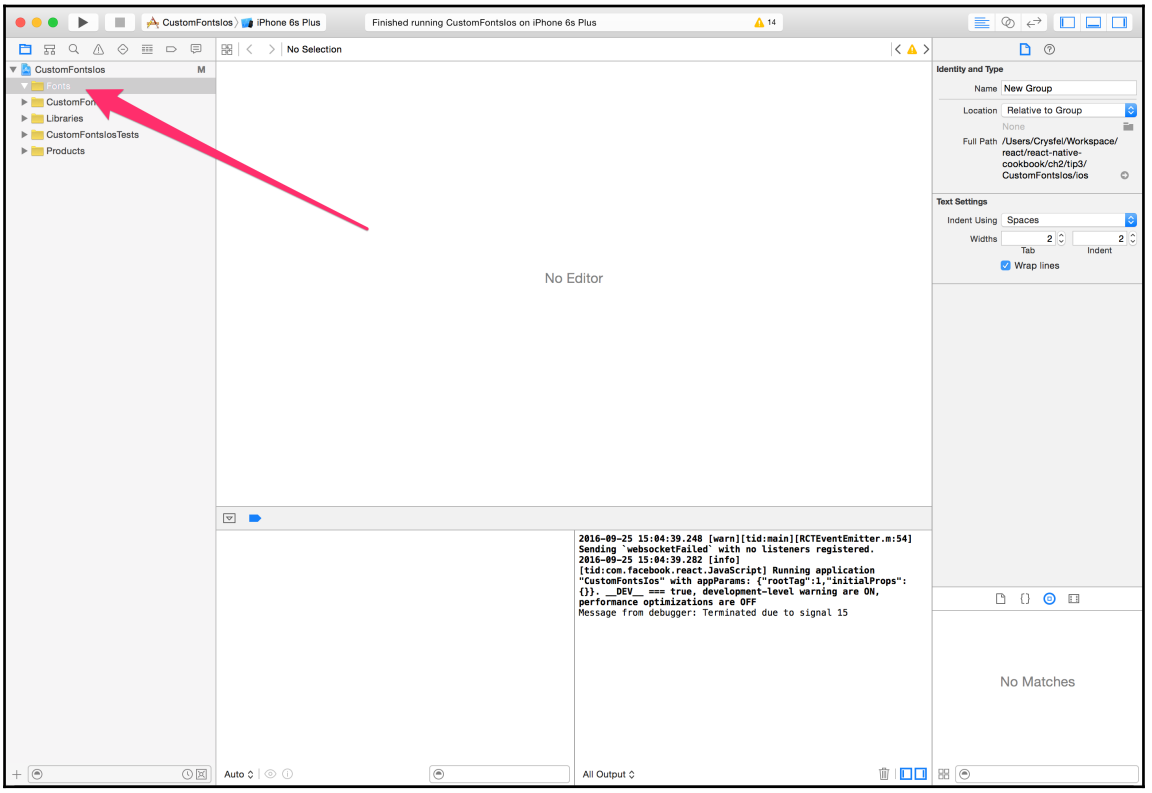
In order to work on this example, we need some fonts. You can use whatever fonts you want; I recommend going to Google fonts and downloading your favorites. For this recipe, we will use the *Roboto* font. Just make sure to use TTF. I've already prepared a few fonts that you can download from PacktPub's website.

Once you have the fonts ready, let's create an empty app using the React Native CLI; we will name it `CustomFontsIos`. If you don't know how to do this, go to `Chapter 1`, *Getting Started*, and take a look at the first recipe.

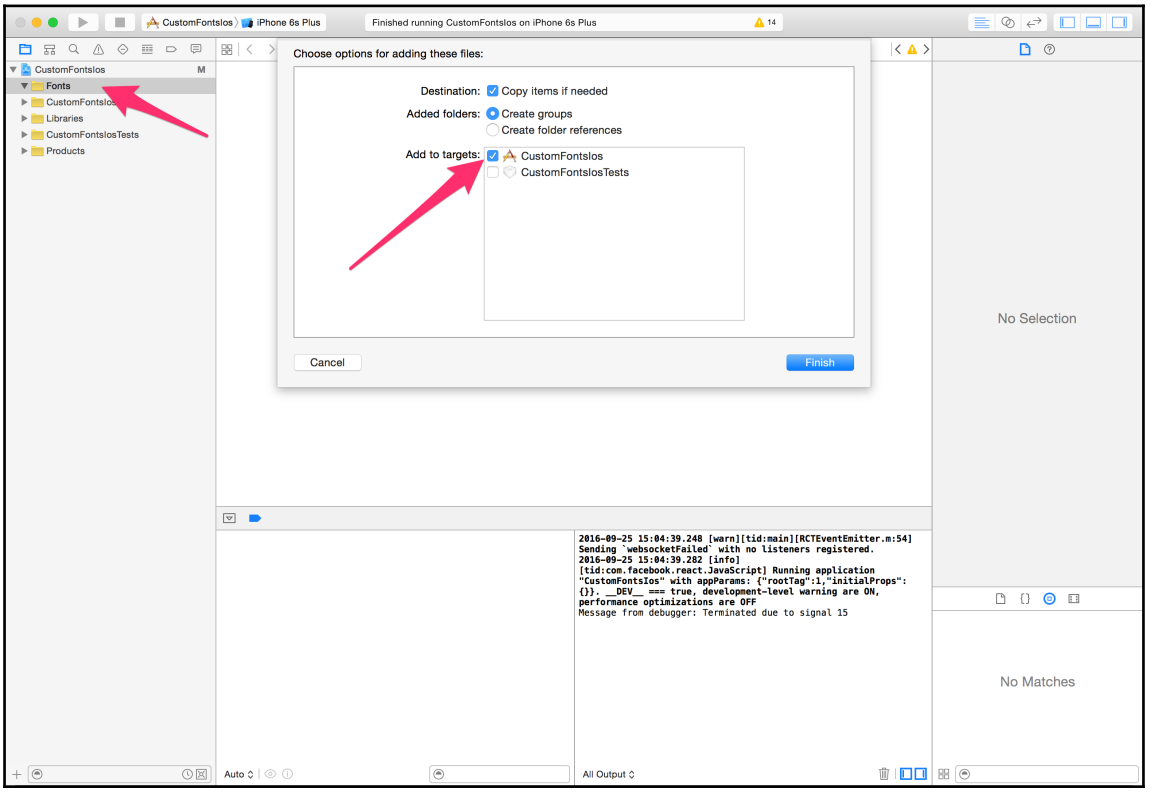
How to do it…

In order to use fonts on iOS devices, we need to import those fonts in a TTF format using Xcode. This is a simple process, but if we don't do it properly, things can go wrong.

1. Let's start by creating a new `Fonts` folder in the **Project Navigator**:

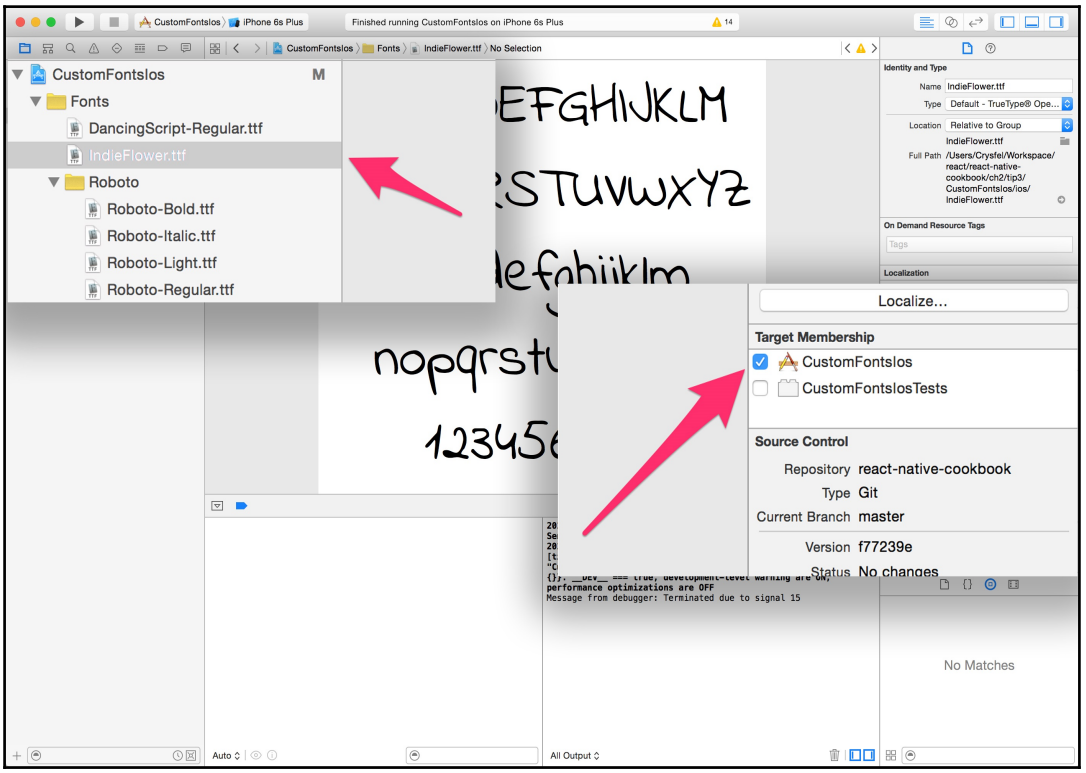


2. Now we need to import to Xcode all the fonts that we want to use, this is very simple–just drag all the font files from the folder on your system and drop them all inside the `Fonts` group on Xcode. After dropping them, you will see a confirmation message. Make sure to select **Add to targets**; this is a very important step because by selecting this option the fonts will be included in the build:



If we forget to select the target, the fonts will not be included in the app and we won't be able to use them. I've made this mistake before, and if you are new to Native Development it will take you some time to figure out the issue. Click on **Finish** to complete the import.

3. If, for some reason, we forget to select **Add to targets**, we can always select the file and check or uncheck this option directly from Xcode. Make sure all fonts that you need are included:

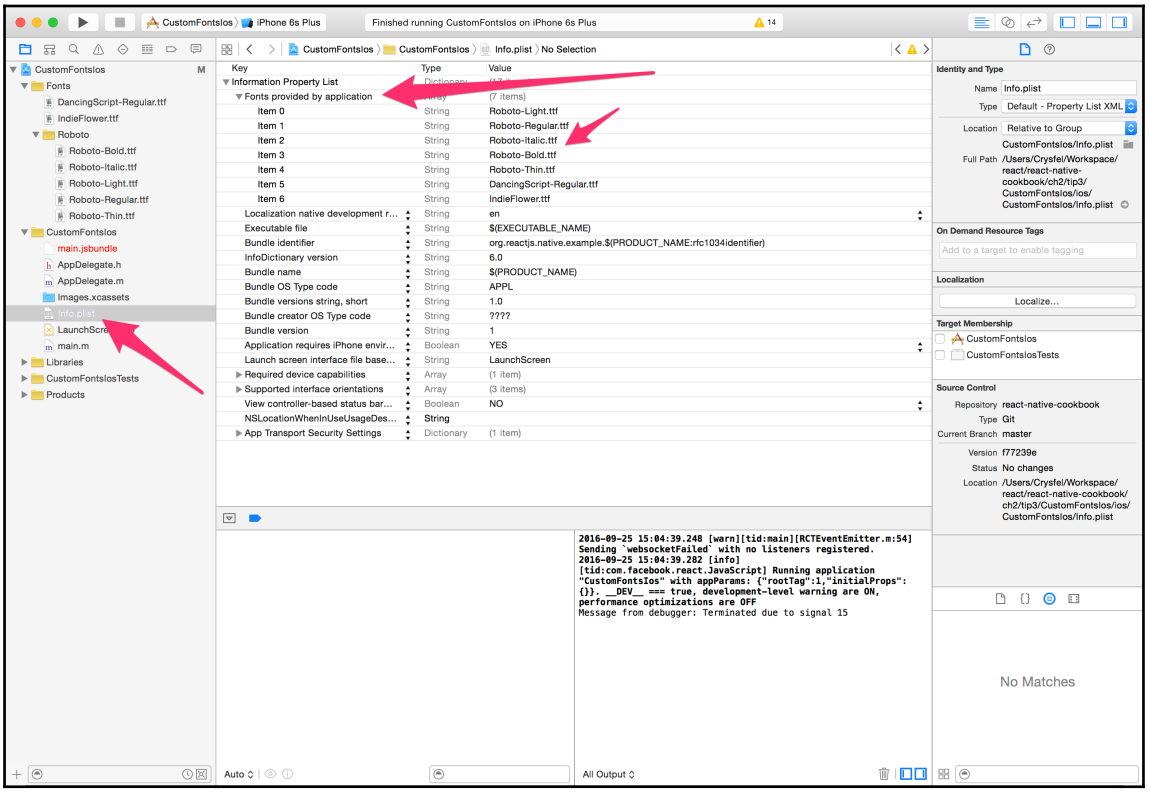


Additionally, we can group the styles of fonts. For this example, all the files for Roboto's font are grouped; this will help us to organize our project.

4. We are almost done with the setup; we just need to include the fonts in the `Info.plist` file. I personally like to edit this file using the source code editor. Right-click on the file and select **Open As** | **Source Code**. Then, write the next code inside the `<dict>` tags:

```
<key>UIAppFonts</key>
<array>
  <string>Roboto-Light.ttf</string>
  <string>Roboto-Regular.ttf</string>
  <string>Roboto-Italic.ttf</string>
```

```
    <string>Roboto-Bold.ttf</string>
    <string>Roboto-Thin.ttf</string>
    <string>DancingScript-Regular.ttf</string>
    <string>IndieFlower.ttf</string>
</array>
```



You can also use the **Property List** editor and enter the fonts one by one.

5. We are done with the setup; we should be able to use all the new fonts in our app. Let's open the `index.ios.js` file and add some text to test the new fonts:

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

```
        const CustomFontsIos = () => (
          <View style={styles.container}>
            <Text style={styles.base}>Welcome to React Native!</Text>
            <Text style={[styles.base, styles.italic]}>Welcome to React
Native!
            </Text>
            <Text style={[styles.base, styles.light]}>Welcome to React
Native!
            </Text>
            <Text style={[styles.base, styles.bold]}>Welcome to React
Native!
            </Text>
            <Text style={[styles.base, styles.dancing]}>Welcome to React
Native!              </Text>
            <Text style={[styles.base, styles.indie]}>Welcome to React
Native!              </Text>
          </View>
        );

        const styles = StyleSheet.create({
         // Defined on step 6
        });

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

6. If we run the app now, we should see the text using the default font. Let's center the text on the screen and define the `base` styles:

```
        const styles = StyleSheet.create({
          container: {
            flex: 1,
            justifyContent: 'center',
            alignItems: 'center',
            backgroundColor: '#e74c3c',
          },
          base: {
            color: '#fff',
            fontFamily: 'Roboto',
            fontSize: 25,
            textAlign: 'center',
            margin: 5,
          },
        });
```

We can see that the text is using our new font! We did this just by using the `fontFamily` property and assigning the font name.

7. Let's use different styles of the same font family; we have two weights available and the `italic` style for the `Roboto` font:

```
light: {
  fontWeight: '300',
},
bold: {
  fontWeight: 'bold',
},
italic: {
  fontStyle: 'italic',
},
```

8. It's looking good! Finally, let's use the other two font families that we have available:

```
dancing: {
  fontFamily: 'Dancing Script',
},
indie: {
  fontFamily: 'Indie Flower',
},
```

How it works…

It's worth mentioning how to get the name of the font that we are using. The simplest way, if you are on a Mac, is to right click on the file and click the **Get Info** option in the menu. Under the **General** properties, there's the **Full Name**.

We can also get the font's name through code: open the `ios/CustomFontsIos/AppDelegate.m` file and around line 20, after the `*jsCodeLocation` variable, paste the following.

```
for (NSString* family in [UIFont familyNames])
 {
   NSLog(@"%@", family);
   for (NSString* name in [UIFont fontNamesForFamilyName: family])
   {
     NSLog(@" %@", name);
   }
 }
```

After running the app again, we should be able to see the result on the Xcode console.

There's more…

In this recipe, we only added fonts to iOS. In the next recipe, we will add the same fonts on Android, and we will use the same JavaScript code and styles.

Including custom fonts on Android

In this recipe, we will include the same fonts we included on iOS, and we will use exactly the same example app.

Importing the fonts is pretty straightforward, but we need to follow some guidelines in order to make it work. There are some limitations on Android; we will learn a workaround to solve the issue regarding font weight.

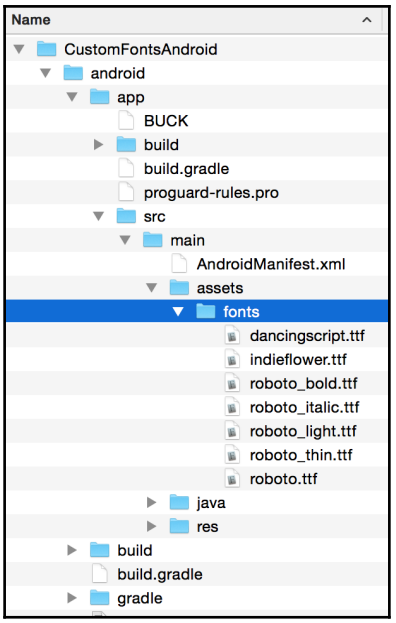
Getting ready

As with the previous recipe, we need some fonts to work with. Android supports TTF and OTF formats. For this example, we will use the same fonts we did in the previous recipe.

How to do it…

To include custom fonts on Android, let's perform the following steps:

1. We need to copy the font files to `project_name/android/app/src/main/assets/fonts`. This is the exact location where we need to add all of our `font` files. If the `assets` and `fonts` folders don't exist, go ahead and create them:



2. React Native will automatically load all the font files in this location. Internally, the `ReactFontManager` class will look for the `ttf` and `otf` files, but we need to make sure to rename the font files following these guidelines:
    - The name should be lowercase
    - There shouldn't be any spaces in the name
    - Use an underscore to define the style, for example, `roboto_bold`

3. Once we complete step 3, all we need to do is run the app on the device or simulator. The fonts should be available at this point and we should be able to use them in our app. We are going to use the same code as in the previous recipe; please follow all the steps from step number *5* until the end of the recipe. You should see something similar to the following screenshot:

4. As we can see, the last two fonts are not working as expected. There's also another issue with the font weight when we set it to `light`. To fix the problem with the last two lines of text, we need to change the value of the `fontFamily` property. We need to use the exact same name as in the `ttf` file:

```
dancing: {
  fontFamily: 'dancingscript',
},
indie: {
  fontFamily: 'indieflower',
},
```

5. If we refresh the app, we should be able to see the two custom fonts. React Native internally loads the fonts and assigns the name of the font using the filename.
6. To fix the issue regarding font weight we need to use a workaround. This is because, currently, React Native only supports `bold` as a value. If you take a look at the source code on GitHub, you will see that it's only possible to use bold. What we need to do is set the `fontFamily` to the name of the file that contains the `thin` style; let's update our code as follows:

```
light: {
  fontFamily: 'roboto_thin',
  // fontWeight: '300',
},
```

It should look as shown in the following screenshot:

Now we have the same result as in the iOS app.

Using font icons

When dealing with images, we need to think about supporting different screen densities and resolutions. For that, we will have to create different versions of the same image and will need to change the size and the name so that React Native uses the correct image for the given resolution. While this works fine, we will have to spend time maintaining all the images. In addition, the size of our app will increase because we need to bundle all those images.

We already have solved that problem on the Web by using fonts or SVGs, and it's a pretty good solution. In this recipe, we will use fonts to render icons and solve the problem of screen density.

For this recipe, we will use a third-party library to help us render the icons. The name of this library is `react-native-vector-icons` by Joel Arvidsson. We will also learn how to install other libraries using `npm`.

Getting ready

We need to create an empty app using the React Native CLI, we are going to name it `FontIcons`.

How to do it…

Font icons can be used using the following steps:

1. The first step is to install the third-party library using `npm`; this is a very simple step: just open your terminal and at the root of the project run the following command:

```
$ npm install react-native-vector-icons --save
```

   By running this command, the code will be downloaded from the `npm` repositories. A few popular fonts will be downloaded as well, for example, Font Awesome, Evil Icons, Material Icons, and a few others.

2. Once the previous command is complete, we need to install the library for iOS (Xcode) and Android. The easiest way to do this is by using the React Native CLI. Just run the following command:

```
$ react-native link
```

   This command will install the fonts for iOS and Android; this means there's no need to copy and paste the `font` files to the `android` folder or use Xcode to import the files. Remember previous recipes for using custom fonts? Everything is automatic now!

3. Let's open the `index.ios.js` and `index.android.js` files and add the following code to bootstrap our app:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';
AppRegistry.registerComponent('FontIcons', () => MainApp);
```

4. Now let's create an `src` folder and a `MainApp.js` file inside it. We are going to render several icons in this component. First, we need to import the dependencies for this class:

```
import React, { Component } from 'react';
import EvilIcon from 'react-native-vector-icons/EvilIcons';
import FAIcon from 'react-native-vector-icons/FontAwesome';
import {
  StyleSheet,
  View,
}
from 'react-native';
```

We need to import each set of icons. In this case, we are importing `Evil` and `FontAwesome` icons. We will be able to use any of the icons available in each set, and we can import all the sets that we need. At the time of writing, we can use the following sets out of the box:

- Entypo by Daniel Bruce (411 icons)
- EvilIcons by Alexander Madyankin and Roman Shamin (v1.8.0, 70 icons)
- FontAwesome by Dave Gandy (v4.7.0, 675 icons)
- Foundation by ZURB, Inc. (v3.0, 283 icons)
- Ionicons by Ben Sperry (v3.0.0, 859 icons)
- MaterialIcons by Google, Inc. (v3.0.1, 932 icons)
- Octicons by Github, Inc. (v4.4.0, 172 icons)
- Zocial by Sam Collins (v1.0, 100 icons)
- SimpleLineIcons by Sabbir and Contributors (v2.4.1, 189 icons)

5. Once we have imported the dependencies and the sets of icons, we can create the component that will only render a few icons for each set:

```
      const MainApp = () => (
        <View style={styles.content}>
          <View style={styles.row}>
            <EvilIcon style={styles.icon} name="user" size={50}
color="#e67e22" />
            <EvilIcon style={styles.icon} name="search" size={50}
color="#e67e22" />
            <EvilIcon style={styles.icon} name="trophy" size={50}
color="#e67e22" />
            <EvilIcon style={styles.icon} name="location" size={50}
color="#e67e22" />
            <EvilIcon style={styles.icon} name="gear" size={50}
color="#e67e22" />
          </View>
          <View style={styles.row}>
            <FAIcon style={styles.icon} name="briefcase" size={40}
color="#c0392b" />
            <FAIcon style={styles.icon} name="calendar" size={40}
color="#c0392b" />
            <FAIcon style={styles.icon} name="camera-retro" size={40}
color="#c0392b" />
            <FAIcon style={styles.icon} name="globe" size={40}
color="#c0392b" />
            <FAIcon style={styles.icon} name="print" size={40}
color="#c0392b" />
```

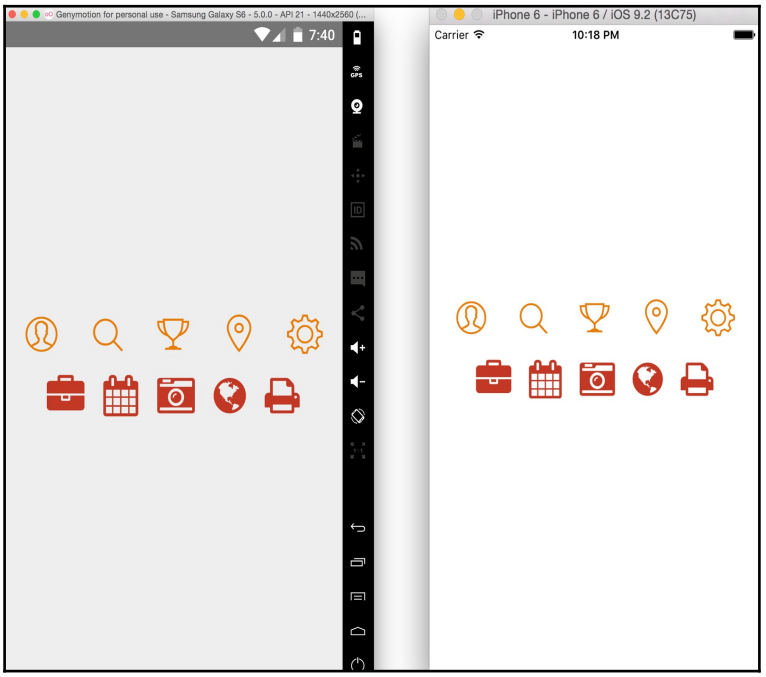
```
      </View>
    </View>
  );

  const styles = StyleSheet.create({
    // Defined on step 6
  });

  export default MainApp;
```

This is a very simple component; we are only rendering five icons for each set. I recommend opening the documentation for each set (you can just Google the name of the set) and seeing the available icons. Then, all that is required is to define the `name` property with the icon's name that we want to render, we can also define `size` and `styles`.

6. If we try to run the project as it is, we should be able to see the icons; however, let's add some styles to the containers to arrange the icons horizontally:

```
const styles = StyleSheet.create({
  content: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  row: {
    flexDirection: 'row',
  },
  icon: {
    margin: 10,
  },
});
```

All we need to do is use flexbox to `center` the content and then set the direction to `row` to horizontally render each row. Additionally, we are adding `10` pixels of margin:



There's more…

If something goes wrong, we need to make sure the fonts are installed properly on each platform. For Android, we need to make sure the `font` files are defined on `android/app/src/main/assets/fonts`. For iOS, we need to make sure the fonts are included on Xcode and added to the `Info.plist` file.

This library also includes support for the iOS tab panel; however, that component is not supported on Android, so I would recommend just using the simple `icon` component to render the icons in any place where required. There is an option to create a tab component that works on Android and iOS and then internally use these icons.

Dealing with universal apps

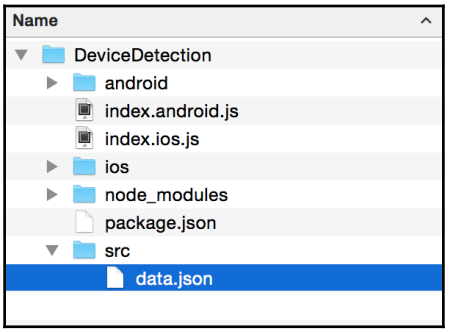
One of the benefits of using React Native is its ability to easily create universal apps. We can share a lot of code for phone and tablet apps. The layouts might be different at some point, but we can reuse the small pieces for both devices.

In this recipe, we will build an app that runs on phones and tablets. The tablet version will include a different layout, but we will reuse the same internal components.

Getting ready

For this recipe, we will show a list of contacts. For now, we will load the data from a `.json` file; later on, in the book, we will explore how to load remote data from a REST API.

Let's open the following URL and copy the generated JSON to a file called `src/data.json`. We will use this data to render the list of contacts `http://api.randomuser.me/?results=20`:



You might want to register to this service and get a key; otherwise, they will only allow you to make a few requests to the assets and API, and since we are going to load the images from there, it's a good idea to get a key.

How to do it…

1. Let's open the `index.ios.js` and `index.android.js` files. Then, add the following code, which basically imports the `MainApp` class and then registers it as the bootstrap component:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';

AppRegistry.registerComponent('DeviceDetection', () => MainApp);
```

2. Now we need to create the `src/MainApp.js` file. Here, we are going to create the main layout. This component will decide whether to render the phone or tablet UI:

```
import React, { Component } from 'react';
import { StyleSheet, View, Text } from 'react-native';
import Dimensions from './utils/Dimensions';
import data from './data.json';

class MainApp extends Component {
  renderMaster() {
    return (
      <Text>Render on phone and tablets!!</Text>
);
  }

  renderDetail() {
    if (Dimensions.isTablet()) {
      return (
        <Text>Render on tablets only!!</Text>
      );
    }
  }

  render() {
    return (
      <View style={styles.content}>
        {this.renderMaster()}
        {this.renderDetail()}
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({
  content: {
    flex: 1,
    flexDirection: 'row',
  },
});

export default MainApp;
```

This class its very straightforward. For now, we are only rendering two texts. The `renderDetail` text should be displayed on tablets only and `renderMaster` should be displayed on phones and tablets.

3. If we try to run our app as it is, it will fail with an error telling us that the `Dimensions` modules cannot be found. Let's create this class! The purpose of this utility class is to calculate whether the current device is a phone or tablet based on the screen dimensions. We are going to create the `isTablet` and `isPhone` methods, so under the `src/utils/Dimensions.js` file add the following code:

```
import { Dimensions as RNDimensions, Alert } from 'react-native';

// Tablet portrait dimensions
const tablet = {
  width: 552,
  height: 960,
};

class Dimensions {
  // Defined on step 4 and 5
}

const dimensions = new Dimensions();
export default dimensions;
```

4. Let's create two methods in this class: one to get the dimensions in portrait and the other to get the dimensions in landscape. Depending on the device rotation, the values of `width` and `height` will change, which is why we need these two methods to always get the correct values whether the device is `landscape` or `portrait`:

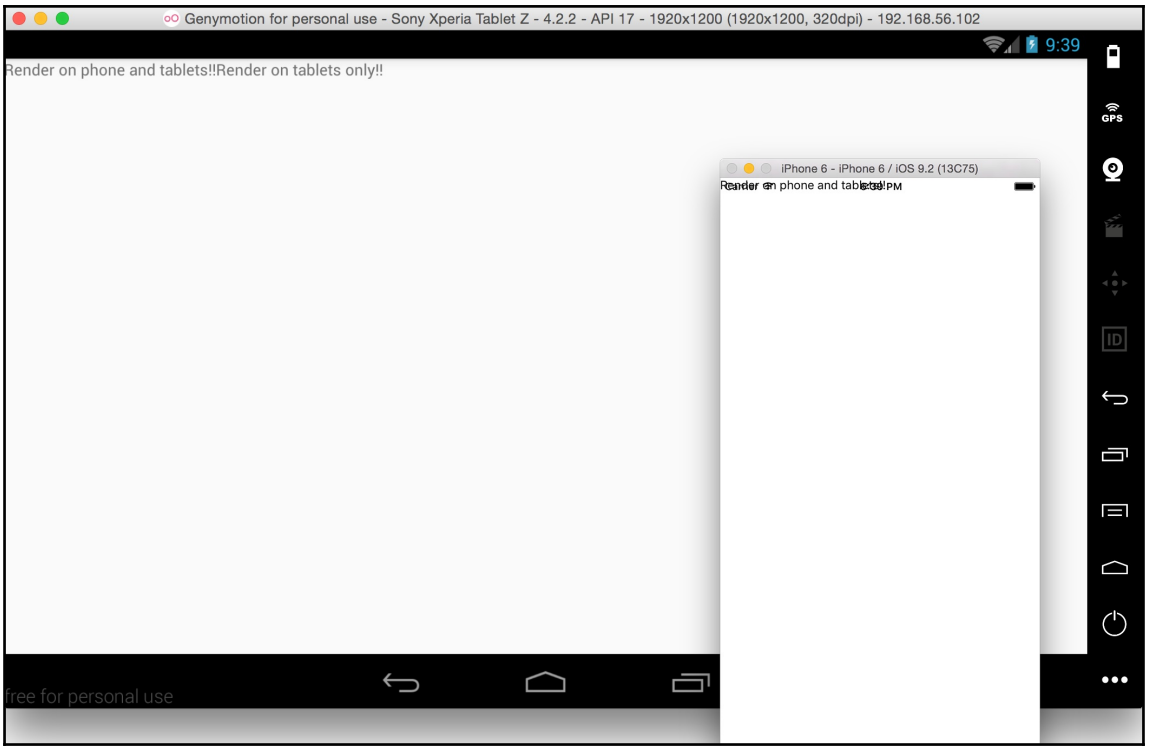
```
  getPortraitDimensions() {
    const { width, height } = RNDimensions.get("window");

    return {
      width: Math.min(width, height),
```

```
      height: Math.max(width, height),
    };
  }

  getLandscapeDimensions() {
    const { width, height } = RNDimensions.get("window");

    return {
      width: Math.max(width, height),
      height: Math.min(width, height),
    };
  }
```

5. Now let's create our utility methods to let us know if we are running the app on a tablet or a phone. To calculate this, we need to get the dimensions in portrait mode and compare them with the dimensions we have defined for a tablet:

```
isPhone() {
  const dim = this.getPortraitDimensions();
  return dim.height < tablet.height;
}

isTablet() {
  const dim = this.getPortraitDimensions();
  return dim.height >= tablet.height;
}
```

If we try to run the app, it should work! We should be able to see the two texts on the tablet and only one on the phone:



6. The utility works, but it's not so useful for rendering texts. Let's work on the master view first; here, we will render the list of contacts. In the same file (`MainApp.js`), let's import the new component and update the `renderMaster` method to return the JSX with our new component:

```
import UserList from './UserList'

class MainApp extends Component {
  renderMaster() {
    return (
      <UserList contacts={data.results} />
    );
  }

  //...

}
```

7. Let's create a new `src/UserList` folder. Inside this folder, we need to create two files: `index.js`, and `styles.js`. We are going to define the component in the `index` file. The first thing we need to do is import the dependencies, create an empty class, and export it as `default`:

```
import React, { Component } from 'react';
import {
  StyleSheet,
  View,
  Text,
  ListView,
  Image,
  TouchableOpacity,
} from 'react-native';
import styles from './styles';

class UserList extends Component {
  // Defined on next steps
}

export default UserList;
```

8. We already know how to create a list. If you are not clear on how this component works, you should read the recipe about list components in `Chapter 1`, *Getting Started*. In the constructor of the class, we will create the `datasource` and then add it to the `state`. The `render` method looks very simple:

```
render() {
  return (
    <View style={styles.main}>
      <Text style={styles.toolbar}>My contacts!</Text>
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderContact}
        style={styles.main}
      />
    </View>
  );
}
```

9. As you can see, we need to define the `renderContact` method to render the rows. We are using the `TouchableOpacity` component as the main wrapper; this will allow us to use a callback function to perform some actions when record is pressed. For now, we are not doing anything when pressing the button; we will learn more about communicating components using redux in future chapters. Also, make sure to use your key from the `randomuser.me` service. If you don't have a key, just remove the query parameter; however, in this case the images will only load a couple of times, before you eventually receive an error:

```
renderContact = (contact) => {
  return (
    <TouchableOpacity style={styles.row}>
      <Image source={{uri: `${contact.picture.large}?key=XXXX-XXXX-
XXXX-XXXX`}} style={styles.img} />
      <View style={styles.info}>
        <Text style={styles.name}>
          {this.capitalize(contact.name.first)}
          {this.capitalize(contact.name.last)}
        </Text>
        <Text style={styles.phone}>{contact.phone}</Text>
      </View>
    </TouchableOpacity>
  );
}
```

10. We don't have a way to capitalize the texts using styles; we need to use JavaScript for that. Let's create the `capitalize` function to set the first letter of the given string to uppercase:

```
capitalize(value) {
  return value[0].toUpperCase() + value.substring(1);
}
```

11. We are almost done with this component; all that's left is the `styles`. Let's open the `src/UserList/styles.js` file and start styling the main container and the `toolbar`:

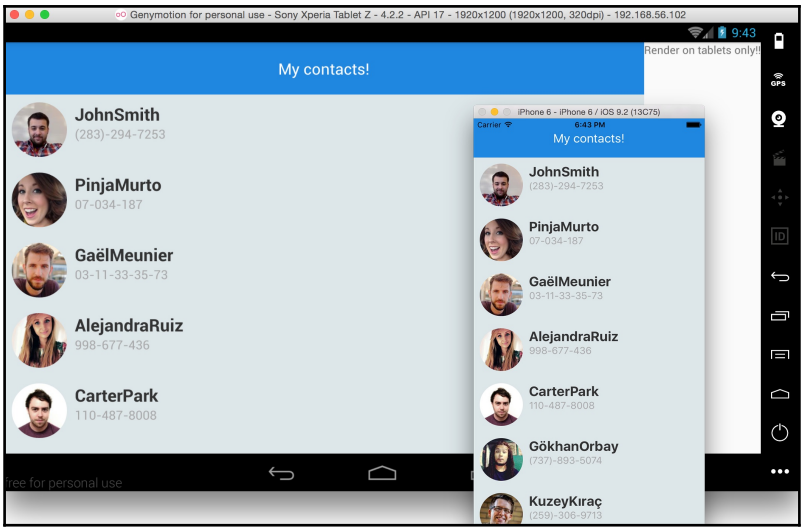
```
import { StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  main: {
    flex: 1,
    backgroundColor: '#dde6e9',
  },
  toolbar: {
```

```
    backgroundColor: '#2989dd',
    color: '#fff',
    padding: 20,
    textAlign: 'center',
    fontSize: 20,
  },
  // More styles defined on step 12
});

export default styles;
```

12. Now for each record, we are going to render the image of each contact on the left, and on the right side, the name and the phone number, which is really simple:

```
row: {
  flexDirection: 'row',
  padding: 10,
},
img: {
  width: 70,
  height: 70,
  borderRadius: 35,
},
info: {
  marginLeft: 10,
},
name: {
  color: '#333',
  fontSize: 22,
  fontWeight: 'bold',
},
phone: {
  color: '#aaa',
  fontSize: 16,
},
```

If we try to run our app, we should be able to see a really nice list on the phone as well as the table, and the same component on two different devices:



13. As of now, we are displaying two different layouts based on the current device! Now we need to work on the user details view, which will show the selected contact. Let's open the `MainApp.js` file, import the `UserDetail` views, and update the `renderDetail` method, as follows:

```
import UserDetail from './UserDetail'

class MainApp extends Component {

  renderDetail() {
    if (Dimensions.isTablet()) {
      return (
        <UserDetail contact={data.results[0]} />
      );
    }
  }

  //...
}
```

As mentioned earlier, in this recipe we are not focusing on sending data from one component to the other, but instead on rendering a different layout in tablets and phones. Therefore, we will always send the first record to the user details view for this recipe.

14. To make things simple and the recipe as short as possible, for the user details view we will only display a toolbar and some text showing the first and last name of the given record. We are going to use a stateless component:

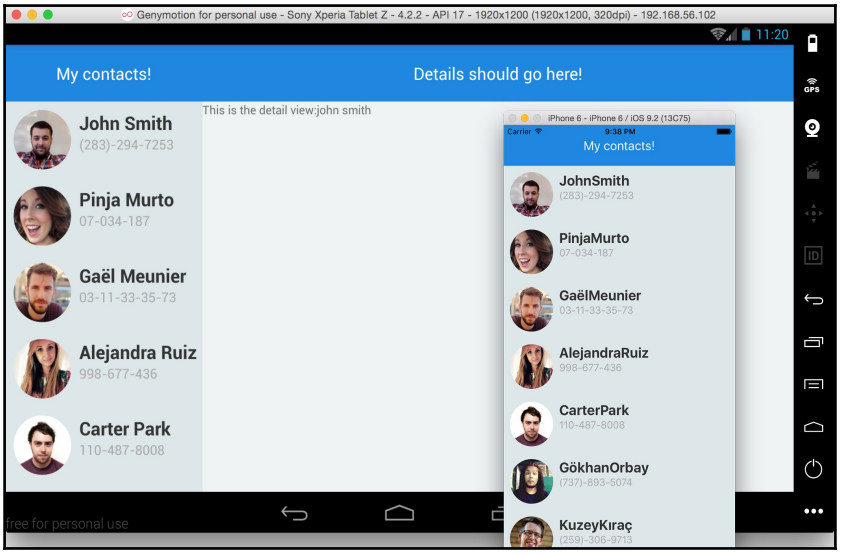
```
import React, { PropTypes } from 'react';
import {
  View,
  Text,
} from 'react-native';
import styles from './styles';

const UserList = ({ contact }) => (
  <View style={styles.main}>
    <Text style={styles.toolbar}>Details should go here!</Text>
    <Text>
      This is the detail view:{contact.name.first}
{contact.name.last}
    </Text>
  </View>
);

UserList.propTypes = {
  contact: PropTypes.object,
};

export default UserList;
```

15. Finally, we need to style this component. We want to assign three-quarters of the screen to the details page and one-quarter to the master list. This can be done with flexbox with a simple style. We need to add the following code to the `src/UserDetail/styles.js` file:

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

const styles = StyleSheet.create({
  main: {
    flex: 3,
    backgroundColor: '#f0f3f4',
  },
  toolbar: {
    backgroundColor: '#2989dd',
    color: '#fff',
    padding: 20,
    textAlign: 'center',
    fontSize: 20,
  },
});
```

```
export default styles;
```

If we try to run our app again, we will see that on the tablet, it will render a nice layout showing the list as well as the detail, while on the phone it only shows the list of contacts:



How it works…

In step number *3*, the first line imports the dependencies and React Native provides a class to get the dimensions of the current device. This is what we need in order to figure out whether our app is running on a phone or tablet.

In the same step we defined a `tablet` constant, which is an object containing the `width` and `height` that we will use to calculate if the device is a tablet or not. The values of this constant are based on the smallest Android tablet available on the market; however, if we decide to drop Android support, we might want to use the iPad's dimensions instead.

In step number *4*, we got the width and height by calling the `RNDimensions.get("window")` method, and then we got the maximum and minimum values depending on the orientation we wanted.

In step number *9*, we used an arrow function to define the `renderContact` method; this is a very important step, mainly to keep the correct binding scope. If we use a regular function, we will have a problem when using it on the list as a callback function for each record. The list will change the scope and we won't be able to access the `this.capitalize` method.

Detecting orientation changes

When building complex interfaces, it's very common to render different UI components based on the device's orientation. This is especially true when dealing with tablets.

In this recipe, we will render a menu based on the screen orientation. In landscape, we will render an expanded menu with icons and texts, and in portrait we will only render the icons.

Getting ready

To support orientation changes we are going to use a third-party library called `react-native-orientation` by Yamill Vallecillo. It works for iOS and Android by listening to the native orientation change on each platform. I've been using this library for a couple of projects, and it works pretty well.

We will also use `react-native-vector-icons` by Joel Arvidsson. The *Using font icons* recipe in this same chapter describes how to use this library.

How to do it…

1. The first thing we need to do is install the libraries using `npm`. Let's go to the root of the project and run the following commands:

```
$ npm install react-native-orientation --save
$ npm install react-native-vector-icons --save
$ react-native link
```

2. After we have installed and linked the library, we just need to add some configurations. This will allow Android to set the orientation when rotating the device. Let's open the `android/app/src/main/java/com/deviceorientation/MainActivity.java` class and add the following code:

```
import android.content.Intent; // <--
import android.content.res.Configuration; // <--

public class MainActivity extends ReactActivity {

    //...

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
      super.onConfigurationChanged(newConfig);
      Intent intent = new Intent("onConfigurationChanged");
      intent.putExtra("newConfig", newConfig);
      this.sendBroadcast(intent);
    }
}
```

3. We are done with the setup; now let's start working on the app! Open the `index.ios.js` and `index.android.js` files and add the following code:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';

AppRegistry.registerComponent('DeviceOrientation', () => MainApp);
```

4. Let's create the `src/MainApp.js` file. This is the main component of our app. Here, we are going to listen for the orientation changes and send it to any component that requires it. Let's import all the dependencies:

```
import React, { Component } from 'react';
import { Alert, StyleSheet, View, Text } from 'react-native';
import Orientation from 'react-native-orientation';

class MainApp extends Component {

  state = {
    orientation: null,
  };

  // Defined on step 5, 6, 7 and 8
```

```
}

const styles = StyleSheet.create({
  // Defined on step 9
});

export default MainApp;
```

5. When extending from the `Component` class, we automatically have several methods available that will be automatically executed as part of the component's lifecycle. I recommend you read the documentation to have a better understanding of this important concept, but in a few words the component's lifecycle allows us to execute code at a given stage of the cycle.

   In this case, we need to set the initial value of the orientation when the component is being created; therefore, we need to use the `componentWillMount` method:

```
componentWillMount() {
    const orientation = Orientation.getInitialOrientation();
    this.setState({
      orientation,
    });
  }
```

6. We have the initial orientation, but if we start rotating the device the state will always have the initial orientation. We need to listen for the orientation changes; for that, we will use two methods from the component's lifecycle: `componentDidMount` to start listening and `componentWillUnmount` to remove the listener:

```
  componentDidMount() {
    Orientation.addOrientationListener(this.onOrientationChange);
  }

  componentWillUnmount() {
    Orientation
      .removeOrientationListener(this.onOrientationChange);
  }
```

7. Every time the device is rotated, the `onOrientationChange` function will be executed. This function is very simple. All we need to do is update the state:

```
onOrientationChange = (orientation) => {
    this.setState({
      orientation,
```

```
    });
  }
```

8. In order to test our code, we need to define the `render` method; for now, we will only show the content of the orientation that we have to `state`:

```
render() {
  return (
    <Text>{this.state.orientation} </Text>
  );
}
```

   If we try to run our app, we should be able to see landscape or portrait on the screen, based on the current device's orientation.

9. Once the `orientation` library is working properly, we can focus on the UI. As mentioned before, we will create a menu that renders the options slightly differently based on the current orientation. Let's import this component and update the `render` method:

```
import Menu from './Menu';

class MainApp extends Component {
  //...

  render() {
    return (
      <View style={styles.content}>
        <Menu orientation={this.state.orientation} />
        <View style={styles.main}>
          <Text>Main Content</Text>
        </View>
      </View>
    );
  }
}
```

10. There are a few styles that we need to define as well, something that is very simple to do. Setting the `flex` direction to `row` will allow us to display the two components horizontally, and then we can are define some colors and center the main content:

```
const styles = StyleSheet.create({
  content: {
    flex: 1,
    flexDirection: 'row',
```

```
  },
  main: {
    flex: 1,
    backgroundColor: '#ecf0f1',
    justifyContent: 'center',
    alignItems: 'center',
  }
});
```

11. Let's focus on the `Menu` component; we need to create a new `src/Menu/index.js` file. Here, we will define the `Menu` class. This component will receive the `orientation` property and it will decide how to render the options based on the `orientation` value. Let's start by importing the dependencies for this class:

```
import React, { Component, PropTypes } from 'react';
import { StyleSheet, View, Text } from 'react-native';
import Icon from 'react-native-vector-icons/Foundation';
```

12. Once we have imported all we need, let's define the main class. We are going to define the orientation's `PropType` as a `string`. On the `state`, we will define an array of `options`; for simplicity, I've defined a single option, but go ahead and duplicate the same record as many times as you want. You should change the values.

> I recommend you take a look at the Foundation icons website to choose the correct value; otherwise, you might get warnings when setting the wrong value for the icon:

```
class Menu extends Component {
  static propTypes = {
    orientation: PropTypes.string,
  };

  state = {
    options: [
      {title: 'Dashboard', icon: 'graph-pie'},
      // Please duplicate the above line
      // to add more options to the menu
    ],
  };

  // Defined on step 13 and 14
}

const styles = StyleSheet.create({
```

```
  // Defined on step 15
});

export default Menu;
```

13. The `render` method for this component will loop through the array of `options` in the `state`:

```
render() {
  return (
    <View style={styles.content}>
      {this.state.options.map(this.renderOption)}
    </View>
  );
}
```

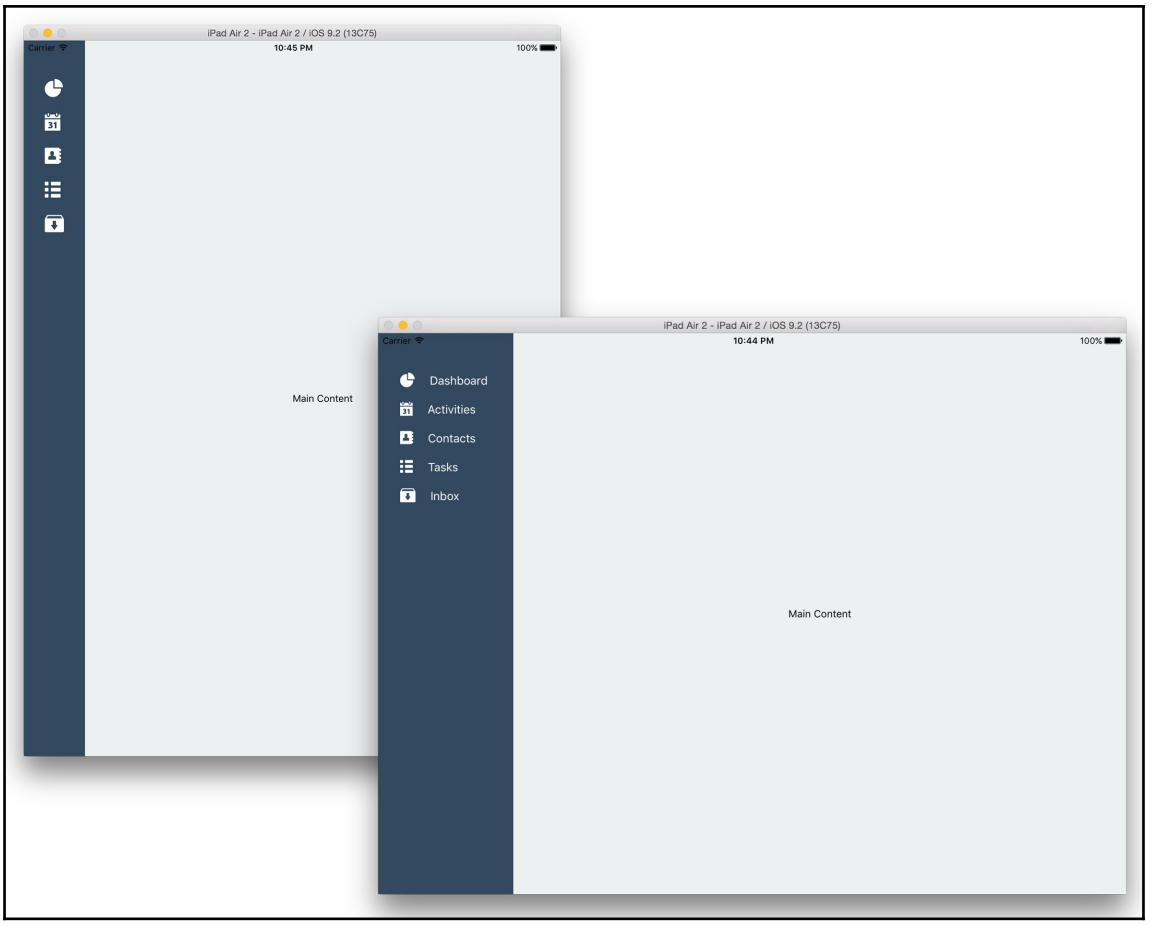
14. Inside the JSX, there's a call to `renderOption`. In this method, we are going to render the icon and the label for each option. Based on the orientation, we will show or hide the label and change the icon's size:

```
renderOption = (option, index) => {
  const isLandscape = this.props.orientation === 'LANDSCAPE';
  const title = isLandscape
    ? <Text style={styles.title}>{option.title}</Text>
    : null;
  const iconSize = isLandscape ? 27 : 35;

  return (
    <View key={index} style={[styles.option, styles.landscape]}>
      <Icon name={option.icon} size={iconSize} color="#fff" />
      {title}
    </View>
  );
}
```

15. Finally, we need to define the styles for the menu. First, we will set the `backgroundColor` to a nice dark blue, and then for each option we will change the `flexDirection` to render the icon and label horizontally. The rest of the styles are only margins and paddings:

```
const styles = StyleSheet.create({
  content: {
    backgroundColor: '#34495e',
    paddingTop: 50,
  },
  option: {
    flexDirection: 'row',
    paddingBottom: 10,
  },
  landscape: {
    paddingRight: 30,
    paddingLeft: 30,
  },
  title: {
    color: '#fff',
    fontSize: 16,
    margin: 5,
    marginLeft: 20,
  },
});
```

    If we try to run our application, it will display the menu on the initial orientation, and then when we rotate the device, the labels will be shown or hidden according to the orientation:



How it works…

The `react-native-orientation` library listens to orientation changes natively, and this improves the performance while detecting a change.

We only need to define a single listener in the app, usually in the top level component. Then, we send the orientation value to the children components. If we are using redux, we should save the value in the global store and then send it to the children using the `connect` function.

In step number *4*, we created an empty class with an `orientation` property of the state. We are going to use this property to save the current device orientation and then send it to the children components.

In step *14*, we looped though each option on the menu. When dynamically creating a new component, we always need to set a `key` property. This property should be unique as its used internally by React. In this case, we are sending the index of the `loop` iteration; that way, we can assure that every item will have a unique `key` value.

There's more…

At the time of writing, there will be some issues on Android when installing the `react-native-orientation` package. The callback function that sets the orientation will not work on Android; fortunately, this has been fixed in the GitHub repository, but the code has not been released to `npm`.

The fix should be released in a couple of weeks; I hope you don't have any problems just by following the steps in the recipe. However, if you do need to get the code from GitHub instead of npm, you only have to update your `package.json` file, like this:

```
{
  "name": "DeviceOrientation",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node_modules/react-native/packager/packager.sh"
  },
  "dependencies": {
    "react": "15.3.2",
    "react-native": "^0.34.0",
    "react-native-orientation":
"git+https://github.com/yamill/react-native-orientation",
    "react-native-vector-icons": "^2.1.0"
  }
}
```

Next, we need to remove the `node_modules` folder and install everything again. Let's open the terminal, navigate to the root of our project and run the following commands:

```
$ rm -rf node_modules
$ npm i
```

That should fix the problems.

Using a WebView to open external websites

For many apps, it's required that it can open external links, whether its to show a third-party website, your online help, or the terms and conditions of your app.

In this recipe, we will see how to open a `WebView` by clicking on a button in our app and dynamically setting the URL value.

How to do it…

1. Similar to our previous recipes, we will start by creating a new app using the React Native CLI, and we will name it `Browser`. After the new project has been created, open the `index.ios.js` and `index.android.js` files; we need to add the following code to bootstrap our app:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';

AppRegistry.registerComponent('Browser', () => MainApp);
```

2. Now let's create the `src/MainApp.js` file. Here, we will define a `Navigator` component to open the `WebView` when the user presses any of the available links. Let's start by importing the dependencies for this class:

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

3. The main class will have a list of `links` to the `state`; we are going to dynamically render buttons using this data. Please go ahead and duplicate the same object on the list to add more links; all we need to define for each item is the `title` and the `url`:

```
class MainApp extends Component {

  state = {
    links: [
```

```
        {
          title:'My Blog',
          url: 'https://medium.com/@crysfel/latest'
        },
        // Please duplicate the previous record with other links
      ],
    };

    // Defined on later steps
  }

  const styles = StyleSheet.create({
    // Defined on later steps
  });

  export default MainApp;
```

4. It's required that every new component has a `render` method. In this case, we are going to use a `Navigator` component to render the scenes of our app. The initial route will be an empty object; we are going to show the next scenes with a nice animation starting from the bottom to the top:

```
render() {
    return (
      <Navigator
        ref="navigator"
        renderScene={this.renderScene}
        initialRoute={{}}
        configureScene={(route) => (
          Navigator.SceneConfigs.FloatFromBottom
        )}
      />
    );
  }
```

5. There's a call to the `renderScene` method. Here, we need to decide which component to render based on the route.

   In this example, we will only be showing two views. By default, we will display the list of `links`, but if there's a `url` property on the `route`, we will display the `WebView` instead.

   For now, we will only display the default view. We will come back to this method to update it and show the `WebView` in later steps:

```
renderScene = (route, navigator) => {
```

```
    return (
      <View style={styles.content}>
        <Text>Home</Text>
        <View>
          {this.state.links.map(this.renderButton)}
        </View>
      </View>
    );
  }
```

6. In the `renderButton` method, we need to create a button for each link. We need to use the `title` property as the label, and when pressing the button, we need to send the `url` as a parameter for the `onPressButton` callback:

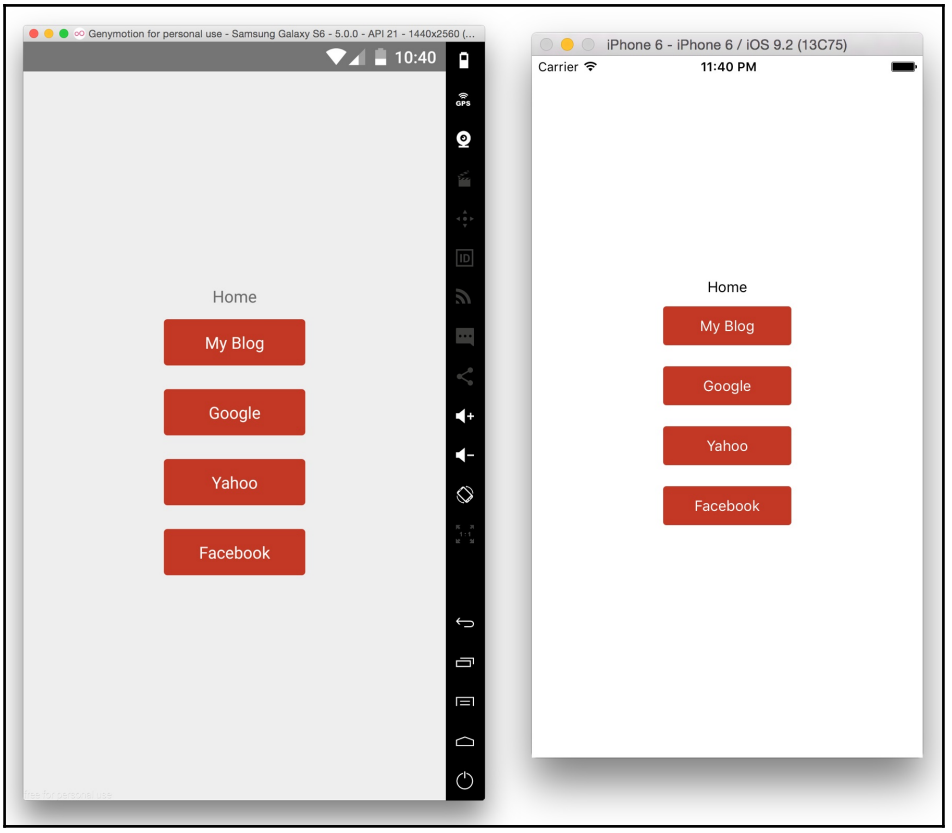
```
renderButton = (btn, index) => {
  return (
    <TouchableOpacity
      key={index}
      onPress={() => this.onPressButton(btn.url)}
      style={styles.btn}
    >
      <Text style={styles.text}>{btn.title}</Text>
    </TouchableOpacity>
  );
}
```

7. Before running our app, let's add some styles to those buttons, something really simple, such as some colors and paddings:

```
const styles = StyleSheet.create({
  content: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  btn: {
    flex: 1,
    margin: 10,
    backgroundColor: '#c0392b',
    borderRadius: 3,
    padding: 10,
    paddingRight: 30,
    paddingLeft: 30,
  },
  text: {
    color: '#fff',
    textAlign: 'center',
```

```
  },
});
```

If we try to run our app, we should be able to see the list of links. However if we try to press any of them, an error will be thrown because we haven't defined the callback function yet:



8. We are half-way through this recipe! Let's create the callback function of the press event; in this function, we will use the `navigator` component to push the new route. For this example, the route is an object that only contains a `url` property, but we can define any property on the route object:

```
onPressButton(url) {
  this.refs.navigator.push({ url });
}
```

9. We need to update the `renderScene` method to check if the `url` property is present on the `route`. If it's present, we need to return the `Browser` component instead of the default view:

```
renderScene = (route, navigator) => {
  if (route.url) {
    return (
      <Browser url={route.url} navigator={navigator} />
    );
  }

  // ...
}
```

Don't forget to import the `Browser` component at the top of the file:

```
import Browser from './BrowserView';
```

10. We are done with the `MainApp` class, and now we need to focus on the `Browser` component. Let's create a `src/BrowserView/index.js` file and import the dependencies for this class. The `WebView` component works on Android and iOS out of the box:

```
import React, { Component, PropTypes } from 'react';
import {
  Text,
  View,
  TouchableOpacity,
  WebView,
  StyleSheet,
} from 'react-native';
```

11. For this component, we only need two props: the `url`, which is a `string`, and the `navigator` as an `object`. The values of these props will be provided by the parent class, which is `MainApp`:

```
class BrowserView extends Component {
  static propTypes = {
    url: PropTypes.string,
    navigator: PropTypes.object,
  };

  // Defined on step 12 and 13
}

const styles = StyleSheet.create({
```

```
  // Defined on step 14
});

export default BrowserView;
```

12. For this view, we are going to have a top `toolbar` with a `back` button and a `WebView` using all the available space. For the `WebView`, we will set the `uri` value from the props that we receive from the parent class:

```
render() {
  return (
    <View style={styles.container}>
      <TouchableOpacity
        style={styles.toolbar}
        onPress={() => this.goBack()}
      >
        <Text style={styles.text}>Back</Text>
      </TouchableOpacity>
      <WebView
        source={{ uri: this.props.url }}
        style={styles.content}
      />
    </View>
  );
}
```

13. When pressing the back button, we'll call the `goBack` function, which will remove the current route from the `navigator` to show the previous route:

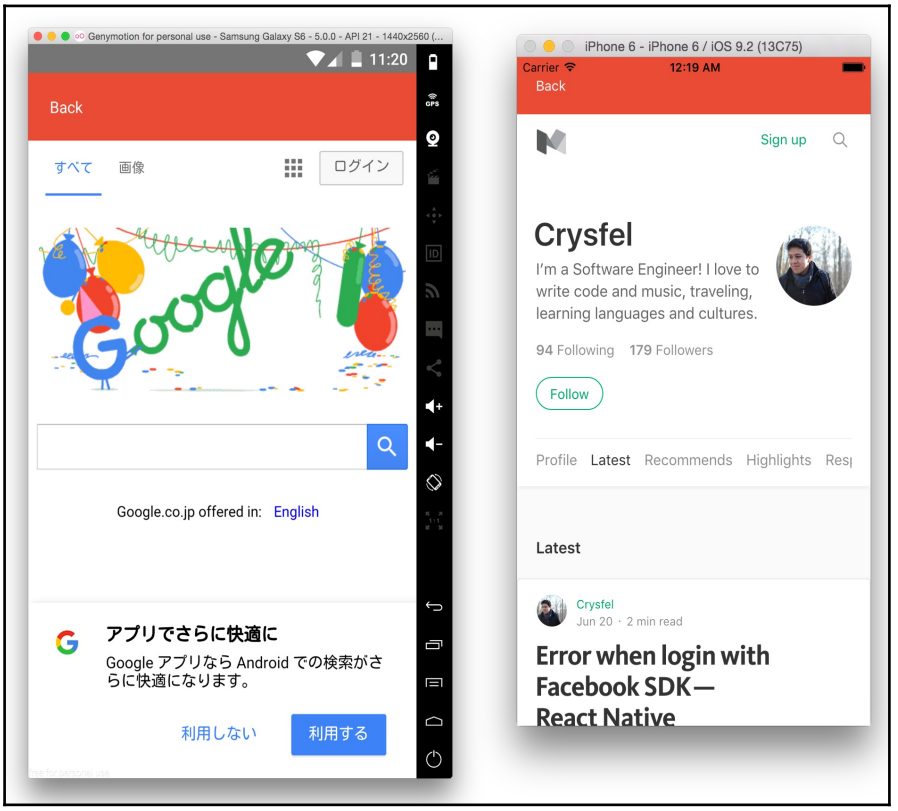
```
goBack() {
  this.props.navigator.pop();
}
```

14. To complete this component all we need to do is define a few styles, such as some colors to the toolbar, and we also need to make the height of the `WebView` flexible:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#e74c3c',
    padding: 20,
  },
  text: {
    color: '#fff',
```

```
  },
  content: {
    flex: 1,
  },
});
```

If we try to run our app, we should be able to open the links when pressing the buttons. There's one limitation on iOS, though: since iOS 9, it's required that only secured URLs are opened, meaning only HTTPS websites are allowed to successfully load on the `WebView`. There's a workaround to fix this issue. You can temporally disable certain domains in the `Info.plist` file, but for security reasons we should not do that for productions apps:



How it works…

Using a `WebView` to open external sites is a great way to keep the user into our app. Many apps out there do this, allowing the user to consume external content and then come back to the app easily.

In step number *4*, we defined the `Navigator` component. Here, it's very important to assign the `ref` property, mainly because we need to access this component in a different method on this same class.

In step number *6*, we used an arrow function to bind the function to the scope of the current class instance; this is necessary because we are using this function when looping through the array of links.

In step number *8*, we pushed the route to the `navigator`. A `route` is a plain JavaScript object; here, we can define any property that we need. Usually we would send the required data for the next view, but in this example only the `url` is needed.

It's also worth mentioning that the `refs` object in this class contains all the references defined by the components. In this case, we only defined the `navigator`, but we could define a `ref` prop on any other component.

In step number *13*, we removed the current route. The `pop` method removes the latest route from the stack, which means we can `push` as many routes as needed, and then we `pop` them all.

Rendering simple HTML elements using native components

We have learned how to use a `WebView` to render a third-party website; however, sometimes we only need to render text with HTML tags on it.

In this recipe, we will render text in the HTML format using native components. To achieve this, we will use a third-party library called `react-native-htmlview` by James Friend.

Getting ready

In order to follow the steps in this recipe, we need to create an empty app using the React Native CLI. We will name our new app `RenderHtml`. Feel free to use any other name, just make sure to use the correct name in step *2*.

How to do it…

1. The first thing we need to do is install the library using `npm`. Open the terminal and go to the root of the project, and then run the following commands:

```
$ npm i react-native-htmlview --save
```

2. Once the installation is completed, let's open the `index.ios.js` and `index.android.js` files to bootstrap our app:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';

AppRegistry.registerComponent('RenderHtml', () => MainApp);
```

3. We are going to load a JSON file from the filesystem. Let's create an `src/data.json` file. Here, we will have a record with a `content` property with a string as a value including simple HTML tags, for example, `<strong>`, `<i>`, `<a>`, `<h3>`, and `<p>`. Please go ahead and add more text to the `content` property:

```
{
  "title": "Learning a new skill",
  "content":
    "<p>Lorem ipsum dolor sit amet,
     <a href="http://crysfel.com">consectetur adipisicing</a>
     elit, sed do <strong>eiusmod tempor</strong>
     incididunt ut labore et
  <i>dolore magna aliqua</i>.
    </p>
 <h3>The problem with your current skills</h3>
<p>...</p>"
}
```

4. Now that we have some data to test with, we need to work on the `MainApp` component. Let's create a new `src/MainApp.js` file and import all the dependencies for this class:

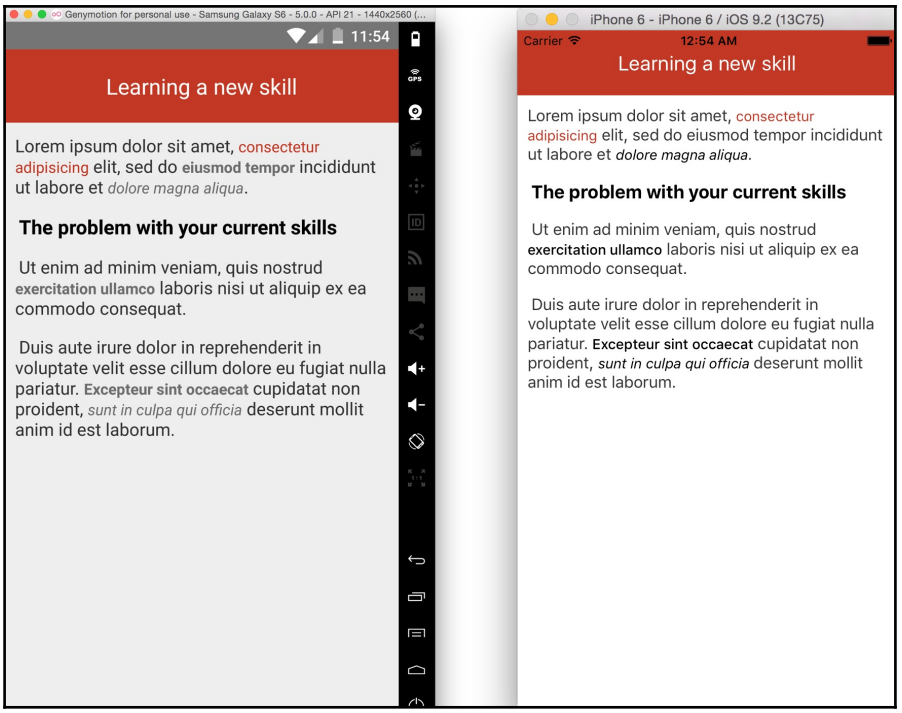
```
import React, { Component } from 'react';
import {
  Alert,
  StyleSheet,
  ScrollView,
  View,
  Text,
} from 'react-native';
import HTMLView from 'react-native-htmlview';
import data from './data.json';
```

5. The class for this component will be as simple as possible. We will only render a top toolbar with the `title` property from the JSON file and the `HTMLView` component with the content's data. If the content is too long, we will need to be able to scroll down, so we are using the `ScrollView` component:

```
class MainApp extends Component {
  onLinkPress(url) {
    Alert.alert('Link press', `URL: ${url}`);
  }

  render() {
    return (
      <View style={styles.main}>
        <Text style={styles.title}>{data.title}</Text>
        <ScrollView style={styles.content}>
          <HTMLView
            value={data.content}
            stylesheet={styles}
            onLinkPress={this.onLinkPress}
          />
        </ScrollView>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  // Defined on step 6
});

export default MainApp;
```

6. Before running our app, let's add some styles to add some colors to the toolbar, set the `height` of the `ScrollView` to fit the available space, and most importantly, set styles to the content:

```
const styles = StyleSheet.create({
  main: {
    flex: 1,
  },
  content: {
    flex: 1,
    padding: 10,
  },
  title: {
    backgroundColor: '#c0392b',
    color: '#fff',
    padding: 20,
    fontSize: 20,
    textAlign: 'center',
  },
  p: {
    color: '#333',
    fontSize: 16,
  },
  h3: {
    fontSize: 18,
    fontWeight: 'bold',
    color: '#000',
  },
  a: {
    color: '#c0392b',
    fontWeight: 'normal',
  }
});
```



How it works…

In step number *5*, we created the `HTMLView` component. This component will receive the HTML content that we want to render as a string; it will also receive a callback function to be executed when a link is pressed inside the view. In this example, we are just showing an alert message, but we could use a `WebView` to open the given URL.

Internally, the `HTMLView` component will parse the HTML string and will create a tree. Then, each node will be replaced by a `Text` element and the node will be ignored if the tag name is not supported.

At the time of writing, the supported tags on the input string are the following:

- `<pre>`
- `<li>`
- `<br>`
- `<p>`
- `<a>`

- `<img width="" height="" />`
- `<h1> <h2> <h3> <h4> <h5>`

If we want to set custom styles to the content, we can use the element name. For example, we have added styles to the `p`, `h3`, and `a` elements in step *6*.

How to create a form component

Every application requires a way to input data, whether it's a simple registration and login form or a more complex component with many input fields and controls.

In this recipe, we will create a form component to handle text inputs. We will collect data using different keyboards and show an alert message with the resulting information.

Getting ready

We need to create an empty app using the React Native CLI, and we will name it `FormComponents`.

How to do it…

1. We will start by bootstrapping our app. Let's open the `index.ios.js` and `index.android.js` files and add the following code:

```
import React from 'react';
import MainApp from './src/MainApp';
import { AppRegistry } from 'react-native';

AppRegistry.registerComponent('FormComponents', () => MainApp);
```

2. Now we need to create a `src/MainApp.js`file and import the required dependencies for this class:

```
import React from 'react';
import {
  Alert,
  StyleSheet,
  ScrollView,
  View,
  Text,
```

```
  TextInput,
} from 'react-native';
import UserForm from './UserForm';
```

3. Since this component is going to be very simple, we are going to create a stateless component. We will only render a top toolbar inside a `ScrollView` for the `UserForm` component:

```
const MainApp = () => (
  <View style={styles.main}>
    <Text style={styles.toolbar}>Fitness App</Text>
    <ScrollView style={styles.content}>
      <UserForm />
    </ScrollView>
  </View>
);
const styles = StyleSheet.create({
  // Defined on step 4
});

export default MainApp;
```

4. We should add some nice styles to these components, just some colors and paddings as well as setting all the available height space to the `scroller` component:

```
const styles = StyleSheet.create({
  main: {
    flex: 1,
    backgroundColor: '#ecf0f1',
  },
  toolbar: {
    backgroundColor: '#1abc9c',
    padding: 20,
    color: '#fff',
    fontSize: 20,
  },
  content: {
    padding: 10,
  },
});
```

5. We have defined the main component, now we need to work on the actual form. Let's create a new JavaScript file under `src/UserForm/index.js`. Then let's import all the dependencies for this class:

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

6. This is the class that will render the inputs and will keep track of the data. We are going to save the data on `state`; therefore, we need to initialize the state as an empty object:

```
class UserForm extends Component {
  state = {};

  // Defined on steps 7, 8, 9 and 10
}

const styles = StyleSheet.create({
 // Defined on step 11
});
export default UserForm;
```

7. In the `render` method, we are going to define the components that we want to display, in this case, only three text inputs and a button. We are going to define a `renderTextfield` method that accepts a configuration object as a parameter; we will define the `name` of the field, the `label`, and the `keyboard` type. In addition, we will have a `renderButton` method that will render the save button:

```
render() {
  return (
    <View style={styles.panel}>
      <Text style={styles.instructions}>
        Please enter your contact information
      </Text>
      {this.renderTextfield({ name: 'name', label: 'Your name' })}
      {this.renderTextfield({ name: 'phone', label: 'Your phone
number', keyboard: 'phone-pad' })}
      {this.renderTextfield({ name: 'email', label: 'Your email
address', keyboard: 'email-address'})}
```

```
            {this.renderButton()}
          </View>
        );
      }
```

8. To render the text fields, we are going to use the `TextInput` component, this component is provided by React Native, and it works on iOS and Android. There are some differences that we need to be aware of, but many of the configurations work on both platforms.

   In this example, we will use the `label` as a `placeholder`; when the user starts typing, the label will disappear. The `keyboardType` property allows us to set the keyboard that we want to use; we have four available keyboards on both platforms, `default`, `numeric`, `email-address`, and `phone-pad`:

```
    renderTextfield(options) {
      return (
        <TextInput
          style={styles.textfield}
          onChangeText={(value) => this.setState({ [options.name]:
value })}
          placeholder={options.label}
          value={this.state[options.name]}
          keyboardType={options.keyboard || 'default'}
        />
      );
    }
```

9. We already know how to render buttons and respond to the press action. If this is not clear to you, I recommend you read the first recipe in this chapter, *Creating a reusable button with theme support* about the buttons:

```
    renderButton() {
      return (
        <TouchableOpacity
          onPress={this.onPressButton}
          style={styles.btn}
        >
          <Text style={styles.btnText}>Save</Text>
        </TouchableOpacity>
      );
    }
```

10. We need to define the `onPressButton` callback. Here, we will show an alert with the input data that we have in the state:

```
onPressButton = () => {
  const { name, phone, email } = this.state;

  Alert.alert('User's data',`Name: ${name}, Phone: ${phone},
  Email: ${email}`);
}
```

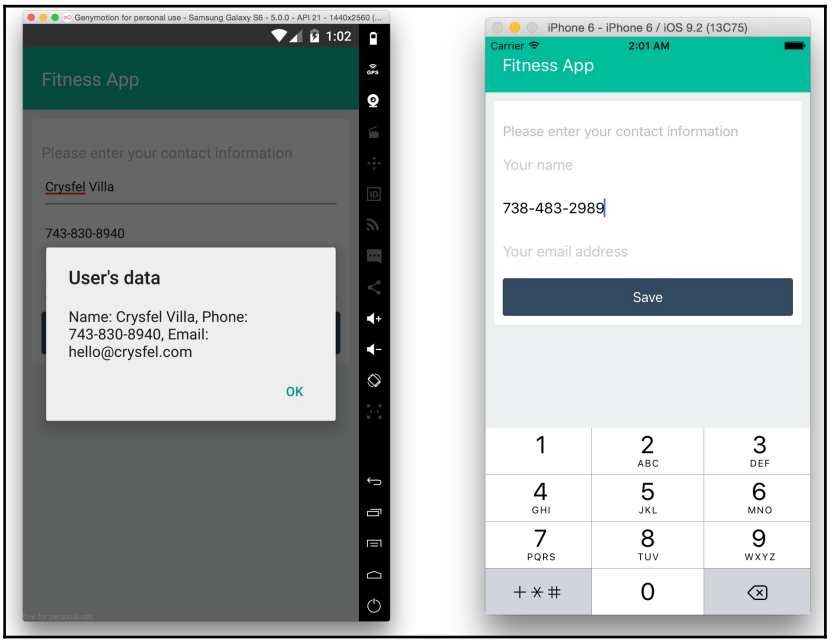
11. We are almost done with this recipe! All we need to do is apply some nice styles to the main container and the instructions, some colors, paddings and margins, nothing really fancy:

```
const styles = StyleSheet.create({
  panel: {
    backgroundColor: '#fff',
    borderRadius: 3,
    padding: 10,
    marginBottom: 20,
  },
  instructions: {
    color: '#bbb',
    fontSize: 16,
    marginTop: 15,
    marginBottom: 10,
  },
});
```

12. Finally, the text inputs and the button. Let's just add dimensions, colors, paddings, and margins:

```
  textfield: {
    height: 40,
    marginBottom: 10,
  },
  btn: {
    backgroundColor: '#34495e',
    borderRadius: 3,
    padding: 12,
    flex: 1,
  },
  btnText: {
    textAlign: 'center',
    color: '#fff',
    fontSize: 16,
  },
```

If we run our app, we should be able to see a nice form. It should work on Android and iOS as expected:



How it works…

In step number *8*, we defined the `TextInput` component. In React Native, we can use two types of input, `controlled`, and `uncontrolled` components.

A `controlled` component will have a `value` property; the component will always display the content of the `value` prop. This means that we need a way to change the value when the user starts typing on the input. If we don't update that value, then the text in the input won't change (even if the user tries to type something).

In order to update the `value`, we can use the `onChangeText` callback and set the new value. In this example, we are using the state to keep track of the data and we are setting a new key on the state with the content of the input.

An `uncontrolled` component, on the other hand, will not have a `value` prop assigned. We can assign an initial value using the `defaultValue` prop. The `uncontrolled` components have their own state, and we can get their value by using an `onChangeText` callback, just as we can do it with `controlled` components.

3
Animating the User Interface

In this chapter, we will cover the following recipes:

- Simple animations
- Running several animations at the same time
- Animating notifications
- Expanding and collapsing containers
- Loading animation
- Removing items from a list component
- Creating a Facebook reactions widget
- Displaying images in full screen

Introduction

In order to provide a good user experience, we definitely need to add some animations to call the user's attention, to go from one screen to the other, to highlight specific actions, or just to add a distinctive touch to our product.

Compared with hybrid apps using PhoneGap, animations in React Native are very fluid. We can animate many elements at the same time and we will get nice and fluid movements.

There's a limit, of course if we want to build a game and animate hundreds of elements at the same time, the frame rate will drop considerably, especially because each frame is calculated on the main JavaScript thread.

There's a work in progress feature to move all the processing from JavaScript to the native side. As of version 0.39, we can choose to use the native driver to run all these calculations on the native world; unfortunately, this is not complete, and there are some issues when running the native driver.

All the recipes in this chapter are using the JavaScript implementation. The React Native team has promised to use the same API when moving all the processing to the native side, so we don't need to worry about breaking changes on the existing API.

Simple animations

In this recipe, we will learn the basics of animations. We will use an image to create a simple linear movement from the right to the left of the screen.

Getting ready

In order to go through this recipe, we need to create an empty app. We are going to call it `SimpleAnimations`. If you are not clear on how to create an empty app, please go to `Chapter 1`, *Getting Started*, and follow the steps in the first recipe.

We are going to use a PNG image; feel free to use any image that you want or download the one provided for this recipe, which is a nice image of a cloud.

How to do it…

1. Let's create an `src` folder at the root of the project. Inside this folder, we are going to need a `MainApp.js` file, where we are going to write all the code for this recipe. As mentioned before, we are going to animate an image, so let's create an `images` folder and paste our image there.

2. Open the `index.ios.js` and `index.android.js` files and add the following code:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('SimpleAnimations', () => MainApp);
```

   Here, we are just importing the `MainApp` class (we are going to define it shortly) and use it to bootstrap the app. If you are using a different name when creating the app, please use it instead of `SimpleAnimations`.

3. Let's import the dependencies for the `MainApp` class. The `Animated` class will be responsible for creating the values for the animation. It provides a few components that are ready to be animated, and it also provides several methods and helpers to run smooth animations.

   The `Easing` class provides several helper methods for calculating movements. For example, we can create exponentials, bounces, and elastic movements among many others.

   We are going to use the `Dimensions` class to get the current device size; this way, we will know the exact position at which to initialize the animation:

```
import React, { Component } from 'react';
import {
  Animated,
  Easing,
  Dimensions,
  StyleSheet,
} from 'react-native';
```

4. We should initialize some constants that we are going to need in our program. In this case, we are going to get the device dimensions, set the size of the image, and require a nice image to move around:

```
const { width, height } = Dimensions.get('window');
const cloudImage = require('./images/cloudy.png');
const imageWidth = 80;
```

5. Now let's create the `MainApp` component. We are going to use two methods from the components lifecycle system. If you are not familiar with this concept, please go ahead and read the React documentation (`http://reactjs.cn/react/docs/component-specs.html`), there's already a really nice tutorial on how this works. You can also read more about this in `Chapter 2`, *Implementing Complex User Interfaces*, in this book, in the recipe about *Detecting orientation changes*:

```
class MainApp extends Component {
  componentWillMount() {
    // Defined on step 6
  }

  componentDidMount() {
    // Defined on step 9
  }

  startAnimation () {
    // Defined on step 7
  }

  render() {
    // Defined on step 8
  }
}

const styles = StyleSheet.create({
  // Defined on step 10

});

export default MainApp;
```

6. In order to create an animation, we need to define a standard value to drive the animation. `Animated.Value` is a class that handles the animation values for each frame over time. The first thing we need to do is to create an instance of this class when the component is created:

```
componentWillMount() {
  this.animatedValue = new Animated.Value();
}
```

> In this case, we are using the `componentWillMount` method, but we can also use the constructor or even the default values of a prop.

7. Once we have created the animated value, we can define the animation. We are going to create a loop when the image reaches the end of the animation we will start the same animation again:

```
startAnimation () {
  this.animatedValue.setValue(width);
  Animated.timing(
    this.animatedValue,
    {
      toValue: -imageWidth,
      duration: 6000,
      easing: Easing.linear,
    }
  ).start(() => this.startAnimation());
}
```

8. We have our animation in place, however as of now we are only calculating the values for each frame over time, but we are not doing anything with those values. The next step is to render the image on the screen and set the property on the styles that we want to animate. In this case, we want to move the element on the *x*-axis, therefore we should update the `left` property:

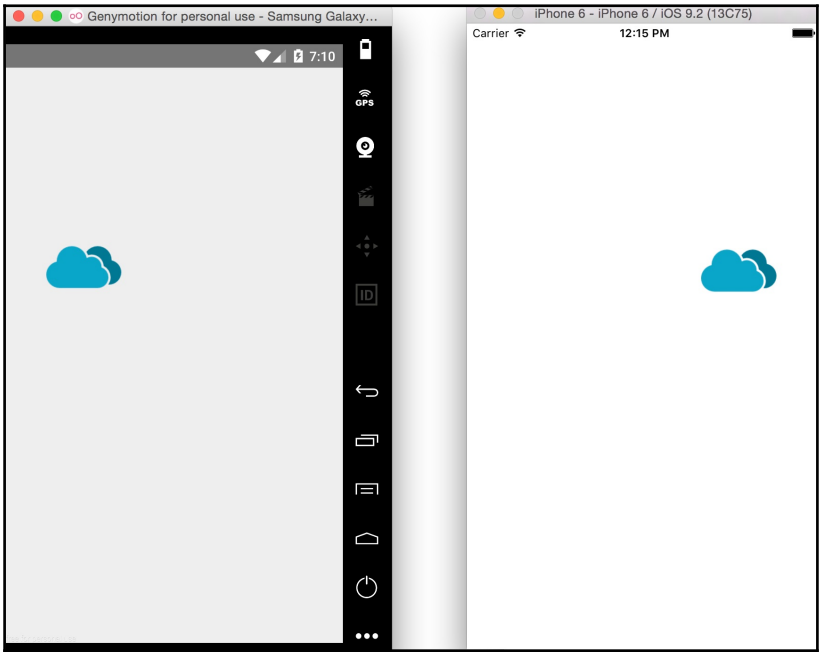
```
render() {
  return (
    <Animated.Image
      style={[
        styles.image,
        { left: this.animatedValue },
      ]}
      source={cloudImage}
    />
  );
}
```

9. If we refresh the simulator we will see the image on the screen, but it's not being animated yet. In order to fix this, we need to call the `startAnimation` method. We will start the animation once the component is fully rendered using the `componentDidMount` lifecycle method:

```
componentDidMount() {
  this.startAnimation();
}
```

10. If we run the app again we will see how the image it's moving at the top of the screen, just what we want! As a final step, let's add some styles to the image:

```
const styles = StyleSheet.create({
  image: {
    height: imageWidth,
    position: 'absolute',
    top: height/3,
    width: imageWidth,
  },
});
```



How it works…

In step number *7*, we set the animation values. The first line is resetting the initial value every time we call this method. For this example, the initial value will be the `width` of the device, this will move the image to the right side of the screen, we want to start our animation from there.

Then we use the `Animated.timing` function to create an animation based on time. For the first parameter, we need to use the `animatedValue` we already have. The second parameter is an object with configurations for the animation. In this case, we are going to set the end value to minus the width of the image; this will place the image to the left side of the screen. We will complete the animation there.

With the entire configuration in place, the `Animated` class will calculate all the frames required in 6 seconds to perform a linear animation from right to left.

We have many other helpers, in this case, we are using a `linear` movement, but we could use an elastic or bounce movement as well. Take a look at the `Easing` class documentation and try setting different values to the `easing` property.

Once the animation is configured correctly we need to run it, we do this by calling the `start` method. This method receives an optional callback function parameter; the callback function will be executed when the animation is completed. In this case, we are running again the same animation. This will create an infinite loop, which is what we want to achieve.

In step number *8*, we are rendering the image. If we want to animate an image we should always use the `Animate.Image` component. Internally, this component will handle the values of the animation and will set each value on every frame to the native component, this will avoid running the render method on the JavaScript side on every frame, allowing us to have smooth animations.

Along with the `Image`, we can also animate the `View`, `Text`, and `ScrollView` components. There's support for all these four components out of the box, but we could always create a new component and add support for animations. All these components are able to handle the styles changes; all we need to do is pass the `animatedValue` to the property that we want to animate, in this case, the `left` property, but we could use any of the available styles on each component.

There's more…

There are some limitations with the current implementation; for example, we need to do some additional calculations in order to use the `opacity` property or to rotate the image. We will cover that in the next recipe.

Running several animations at the same time

In this recipe we will learn how to use the same animation values in several elements. This way we will reuse the same values but by using an interpolation we will be able to get different values for other elements.

The animation will be similar to the previous recipe. We will have two clouds: one will be smaller with a slower movement, the other one will be a little bit bigger and will move faster. At the center of the screen we will have a static plane. We won't add any animation to the plane, but the moving clouds will create the illusion of movement.

Getting ready

Let's start this recipe by creating an empty app; we are going to call it `MultipleAnimations`. Feel free to use any other name, just make sure to use the correct name when registering the bootstrap component.

We are going to use three different images, two clouds and an airplane. You can use your own assets or download the ones provided on for this recipe.

How to do it…

1. As always, let's create the `src` folder with a `MainApp.js` file; here we will write all our code. In addition, we need to create an `images` folder and copy/paste the 3 assets here.

   Then open the `index.ios.js` and `index.android.js` file and replace with the following code. If you are using a different name for the app, make sure to update the last line accordingly:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent(
  'MultipleAnimations', () => MainApp
);
```

2. Now we can work on the `MainApp` class, so let's add the dependencies. Take a look at the first recipe in this same chapter if something is not clear in the following code:

```
import React, { Component } from 'react';
import {
  View,
  Animated,
  Image,
  Easing,
  Dimensions,
  StyleSheet,
} from 'react-native';
```

3. Additionally, we need to define some constants and require the images that we are going to use for the animations:

```
const { width, height } = Dimensions.get('window');
const cloudImage = require('./images/cloud.png');
const cloudsImage = require('./images/cloudy.png');
const planeImage = require('./images/transport.png');
const cloudWidth = 60;
```

4. In the next step we are going to create the `animatedValue` instance when the component gets created, then we will start the animation when the component is fully rendered. We are creating an animation that runs in an infinite loop, the initial value will be `1` and the final value will be `0`. If you are not clear about this code, make sure to read the first recipe in this same chapter:

```
class MainApp extends Component {
  componentWillMount() {
    this.animatedValue = new Animated.Value();
  }

  componentDidMount() {
    this.startAnimation();
  }

  startAnimation () {
    this.animatedValue.setValue(1);
    Animated.timing(
      this.animatedValue,
      {
        toValue: 0,
        duration: 6000,
        easing: Easing.linear,
```

```
        }
      ).start(() => this.startAnimation());
    }

    render() {
      // Defined on step 5 and 6
    }
}

const styles = StyleSheet.create({
  // Defined on 7
});

export default MainApp;
```

5. The `render` method is going to be quite different, for this example, we are going to animate two images using the same `animatedValue`. The animated value will return values from `1` to `0`, however, we want to move the clouds from right to left and for that, we need to set the `left` value on each element.

   In order to set the correct values, we need to interpolate the `animatedValue`. For the smaller cloud we will set the initial left to the width of the device, but for the bigger cloud we will set the initial left far away from the right edge of the device. This will make the movement distance bigger, therefore it will move faster:

```
render() {
  const left1 = this.animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [-cloudWidth, width],
  });

  const left2 = this.animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [-cloudWidth*5, width + cloudWidth*5],
  });

  // Defined on step 6
}
```

6. Once we have the correct `left` values, we need to define the elements we want to animate. Here we will set the interpolated value to the `left` styles property:

```
render() {
 // Defined on step 5
```

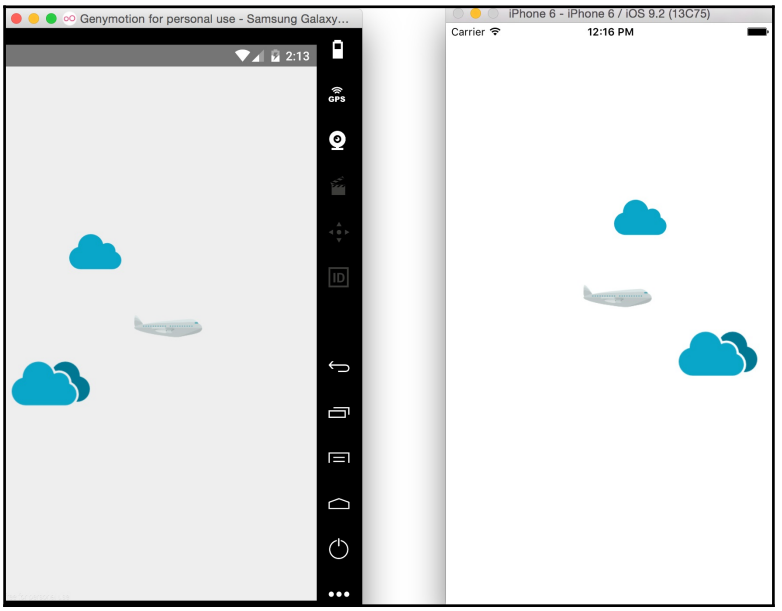
```
    return (
      <View>
        <Animated.Image
          style={[
            styles.cloud1,
            { left: left1 },
          ]}
          source={cloudImage}
        />
        <Animated.Image
          style={[
            styles.cloud1,
            styles.cloud2,
            { left: left2 },
          ]}
          source={cloudsImage}
        />
        <Image
          style={[
            styles.cloud1,
            styles.plane,
          ]}
          source={planeImage}
        />
      </View>
    );
  }
```

7. As for last step, we need to define some styles, just to set the `width` and `height` of each cloud as well as assigning the `top`:

```
const styles = StyleSheet.create({
  cloud1: {
    position: 'absolute',
    width: cloudWidth,
    height: cloudWidth,
    top: height/3 - cloudWidth/2,
  },
  cloud2: {
    width: cloudWidth*1.5,
    height: cloudWidth*1.5,
    top: height/2,
  },
  plane: {
    width: cloudWidth*1.3,
    height: cloudWidth*1.3,
    top: height/2 - cloudWidth,
    left: width/2 - cloudWidth/2,
```

```
    }
});
```

If we refresh our app, we should see a nice animation! It's running really smoothly, as shown in the following screenshot:



How it works…

In step number *5*, we defined the interpolations to get the `left` value for each cloud. The `interpolate` method receives an object with two required configurations, `inputRange` and `outputRange`.

The `inputRange` configuration receives an array of values, these values should always be ascending values, and you could use negative values as long as the values are ascending.

The `outputRange` should match the number of values defined on `inputRange`, these are the values that we need as a result of the interpolation.

For this case, `inputRange` goes from `0` to `1`, which are the values of our `animatedValue`. In `outputRange`, we defined the limits of the movement that we need.

Animating notifications

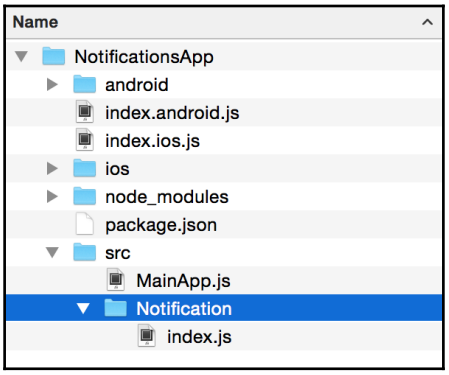
In this recipe, we will create a notification component from scratch. When showing the notification, the component will slide in from the top of the screen; after a few seconds, we will automatically hide it by sliding it out.

Getting ready

We are going to create an app called `NotificationsApp`. You can use any other name that you like, just make sure to use the correct name when registering the bootstrap component in step number 2.

How to do it…

1. Let's start by creating an `src` folder at the root of the project, inside of this we will create a `MainApp.js` file and a `Notification` folder. Inside the `Notification` folder we need to create an `index.js` file:



2. Open the `index.ios.js` and `index.android.js` files, remove the existing code and replace with the following. Make sure to use the same name that you defined when creating the app with the React Native CLI:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('NotificationsApp', () => MainApp);
```

3. Now let's work on the `MainApp` component. First, let's import all the required dependencies:

```
import React, { Component } from 'react';
import {
  Text,
  TouchableOpacity,
  StyleSheet,
  View,
} from 'react-native';
import Notification from './Notification';
```

4. Once we have all the dependencies imported, we can define the main class. In this case, we are going to initialize the `state` with a `notify` property equal to `false`. We are going to use this property to show or hide the notification; by default, the notification will not be shown on screen. To make things simple, we will define the `message` property, in the `state`, with the text we want to display.

```
class MainApp extends Component {
  state = {
    notify: false,
    message: 'Lorem ipsum dolor sit amet...',
  };

  onToggleNotification = () => {
    // Defined on step 7
  }

  render() {
    // Defined on step 5 and 6
  }
}

const styles = StyleSheet.create({
  // Defined on step 8
});
```

```
export default MainApp;
```

5. Inside the `render` method, we need to show the notification only if the `notify` property is `true`, we can achieve that by using an `if` statement:

```
render() {
  const notify = this.state.notify
    ? <Notification
        autoHide
        message={this.state.message}
        onClose={this.onToggleNotification}
      />
  : null;
  // Defined on step 6
}
```

6. In the previous step, we only defined the reference to the `Notification` component, but we are not using it yet. Let's define `return` with all the JSX for this app. To keep things simple, we are only going to define a toolbar, some text, and a button to toggle the state of the notification when pressed:

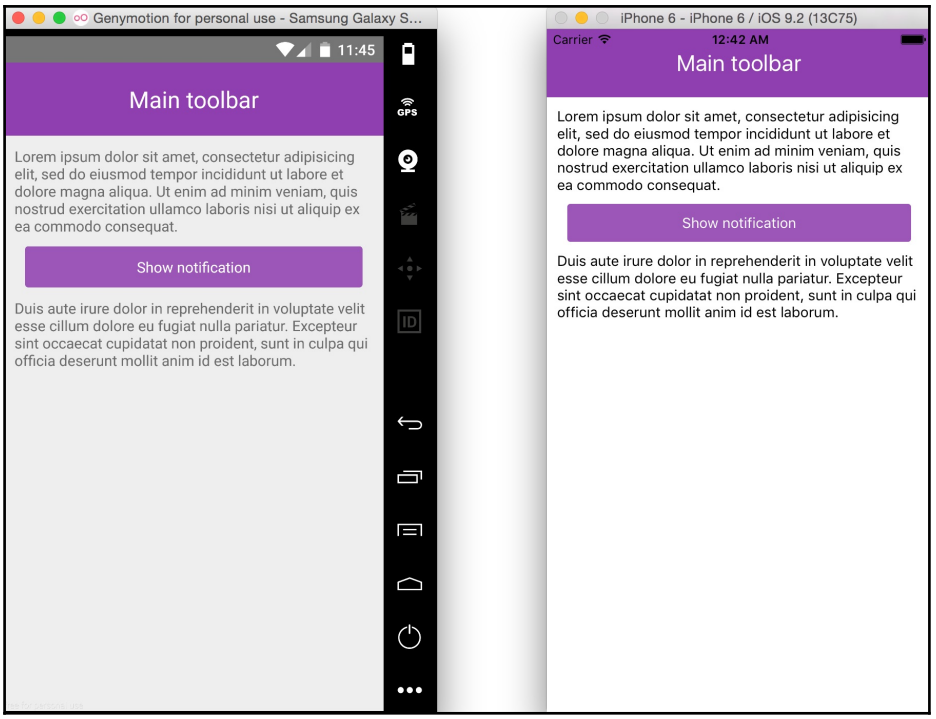
```
render() {
  // Previous code here...

  return (
    <View>
      <Text style={styles.toolbar}>Main toolbar</Text>
      <View style={styles.content}>
        <Text>Lorem ipsum...</Text>
        <TouchableOpacity
          onPress={this.onToggleNotification}
          style={styles.btn}
        >
          <Text style={styles.text}>Show notification</Text>
        </TouchableOpacity>
        <Text>Duis aute irure...</Text>
        {notify}
      </View>
    </View>
  );
}
```

7. We need to define the method that toggles the `notify` property on the `state`, this step is very simple:

```
onToggleNotification = () => {
    this.setState({
      notify: !this.state.notify,
    });
}
```

8. We are almost done with this class. The only thing left are the styles. In this case, we will only add some styles such as `color`, `padding`, and `margin`, nothing really special:

```
const styles = StyleSheet.create({
  toolbar: {
    backgroundColor: '#8e44ad',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
  content: {
    padding: 10,
    overflow: 'hidden',
  },
  btn: {
    margin: 10,
    backgroundColor: '#9b59b6',
    borderRadius: 3,
    padding: 10,
  },
  text: {
    textAlign: 'center',
    color: '#fff',
  },
});
```



9. If we try to press the button an error will appear. Let's fix that by defining the `Notification` component. Open the `Nofitication/index.js` and add the required dependencies for this component:

```
import React, { Component, PropTypes } from 'react';
import {
  Animated,
  Easing,
  StyleSheet,
  Text,
} from 'react-native';
```

10. Once we have the dependencies imported, let's define the props and the initial state of our new component. We are going to define something very simple, just a prop to receive the message to display, and two callback functions to allow running some actions when the notification appears on the screen and when it gets closed. A prop to set a number of milliseconds to display the notification before it auto hides:

```
const { bool, func, number, string } = PropTypes;

class Notification extends Component {
  static propTypes = {
    autoHide: bool,
    delay: number,
    message: string,
    onClose: func,
    onOpen: func,
  };

  static defaultProps = {
    delay: 5000,
    onClose: emptyFn,
    onOpen: emptyFn,
  };

  state = {
    height: -1000,
     };

}

function emptyFn() {}

export default Notification;
```

11. It's time to work on the animation! We need to start the animation as soon as the component gets rendered. If there's something not clear on the following code, I recommend you to take a look at the first and second recipe in this same chapter:

```
componentWillMount() {
  this.animatedValue = new Animated.Value();
}

componentDidMount() {
  this.startSlideIn();
}

getAnimation(value, autoHide) {
```

```
    const { delay } = this.props;
    return Animated.timing(
      this.animatedValue,
      {
        toValue: value,
        duration: 500,
        easing: Easing.cubic,
        delay: autoHide ? delay : 0,
      }
    );
  }
```

12. So far we defined a method to get the animation. For the slide in movement, we need to calculate the values from `0` to `1`, after the animation is completed, we need to run the `onOpen` callback and in the case, the `autoHide` property is set to `true`, we will automatically run the slide out animation to remove the component:

```
startSlideIn () {
  const { onOpen, autoHide } = this.props;

  this.animatedValue.setValue(0);
  this.getAnimation(1)
    .start(() => {
      onOpen();
      if (autoHide){
        this.startSlideOut();
      }
    });
}
```

13. Similarly, for the slide out movement, we need to calculate the values from `1` to `0`. We are sending as a parameter the `autoHide` value to the `getAnimation` method. This will automatically delay the animation for 5 seconds (or for any other value we've defined). After the animation is completed, we need to run the `onClose` callback function; this will remove the component on the `MainApp` class:

```
startSlideOut() {
  const { autoHide, onClose } = this.props;

  this.animatedValue.setValue(1);
  this.getAnimation(0, autoHide)
    .start(() => onClose());
}
```

14. Finally, the `render` method. Here we will get the `message` value from the `props`. We also need the `height` of the component to move the component to the initial position of the animation; by default it's `-1000` but we will set the correct value at runtime on the next steps.

    The `animatedValue` goes from `0` to `1` or `1` to `0`, therefore we need to interpolate it to get the actual values. The animation will go from minus the height of the component to `0`; this will result in a nice slide in/out animation:

```
render() {
 const { message } = this.props;
  const { height } = this.state;
  const top = this.animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [-height, 0],
  });

  // Defined on step 15
}
```

15. To keep things as simple as possible, we will return an `Animated.View` with a text. Here we are setting the `top` style with the interpolation result, this means we will animate the top style.

    As mentioned before, we need to calculate the height of the component at runtime. In order to achieve that, we need to use the `onLayout` property of the view, this function will be called every time the layout updates and will send as a parameter the new dimensions of this component:

```
render() {
  // Previous code here...

  return (
    <Animated.View
      onLayout={this.onLayoutChange}
      style={[
        styles.main,
        { top }
      ]}
    >
      <Text style={styles.text}>{message}</Text>
    </Animated.View>
  );
```

```
}
```

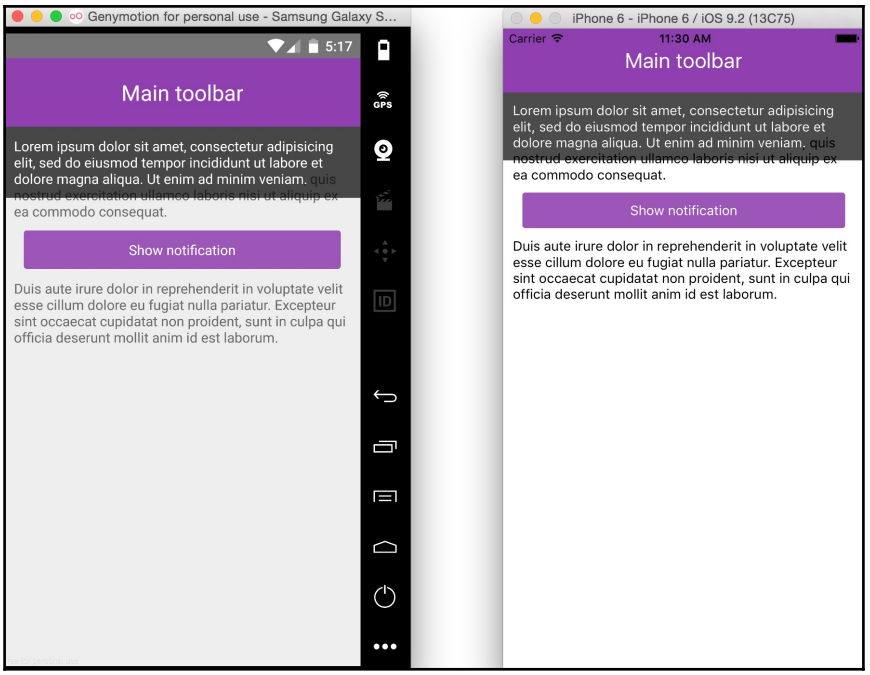
16. The `onLayoutChange` method looks very simple; we only need to get the new `height` and update the `state`.

    This method receives an `event`; inside of this object, we have some useful information. In this case, we will access the `nativeEvent` object, then in the `layout` object, we can find the `width` and `height`, *x* and *y* position on the screen of the `Animated.View` that called this function:

```
onLayoutChange = (event) => {
 const {layout: { height } } = event.nativeEvent;
  this.setState({ height });
}
```

17. As the very last step, we will add some styles to the notification component. We want this component to animate on top of anything else; therefore we need to set the `position` as `absolute`, set the `left` and `right` to `0`, and add some colors and paddings:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: 'rgba(0, 0, 0, 0.7)',
    padding: 10,
    position: 'absolute',
    left: 0,
    right: 0,
  },
  text: {
    color: '#fff',
  },
});
```



How it works…

In step *5*, we defined the `Notification` component. This component receives three parameters, a flag to automatically hide the component after a few seconds, the message that we want to display and a callback function that will be executed when the notification gets closed.

When the `onClose` callback gets executed, we will toggle the `notify` property to remove the `Notification` instance and clear the memory.

In step *6*, we defined the JSX to render the components of our app. It's important to render the `Notification` component after the others this way, the component will automatically appear on top of all other components. If we decide to render the `Notification` before the other components, we will have to define a `Index` greater than any other component.

In step *10*, we are defining the `propTypes` and the `state` of our component. The `defaultProps` object sets the default values for each property; these values will be applied in case no value is assigned to the given property.

For the callback functions, we are defining an empty function. This way, we don't have to check if those props have a value before trying to execute them.

For the initial `state`, we are defining the `height` property. The actual `height` value is going to be calculated at runtime, based on the content received in the `message` prop. Therefore, we need to initially render the component far away from the original position. The reason for this is because there's a small delay when the layout is calculated and we don't want to show the notification and then move it to the correct position.

In step *11*, we are creating the animation. The `getAnimation` method receives two parameters. One is the destination `value` and the another one is a flag to `delay` the animation. We use this method in step *12* and *13*.

In step *15*, we are returning the JSX for this component. The `onLayout` function is very useful to get the dimensions of the component when there are updates on the layout. For example, if the device orientation changes, the dimensions might change, in that case, we would like to update the initial or final coordinate for the animation.

There's more…

The current implementation works pretty well; however, there's a performance problem and we need to address that. The `onLayout` method gets executed on every frame of the animation, and we are updating the state on every frame, which leads to re-render the component on every frame! We should not do this; instead, we should only update it once to get the actual height.

To fix this we could add a simple validation just to update the state if the current value is different than the initial value. This will avoid updating the `state` on every frame and we won't force the render over and over again:

```
onLayoutChange = (event) => {
  const {layout: { height } } = event.nativeEvent;
  if (this.state.height === -1000) {
    this.setState({ height });
  }
}
```

This would be enough, but we can go further and make sure the `height` gets updated also when the orientation changes; however, we will stop here, as this recipe is getting quite big already.

Expanding and collapsing containers

In this recipe, we will create a custom container with a `title` and `content`. When pressing the title, the content will collapse or expand.

We will explore the layout animation API for this example, as it will help us to accomplish our goal in a few steps only.

Getting ready

Let's start by creating a new app, using the React Native CLI. We will call it `CollapsibleApp`, but feel free to use any other name.

Once we have created the app, let's create two files at the root of the project, `src/MainApp.js` and `src/Panel/index.js`. These are the only two files we will be working on.

How to do it…

1. Let's start by opening the `index.ios.js` file; remove the existing code and add the following to bootstrap the app:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('CollapsibleApp', () => MainApp);
```

2. Now, let's open the `index.android.js` file and replace with the following code:

```
import React from 'react';
import { AppRegistry, UIManager } from 'react-native';
import MainApp from './src/MainApp';

const EnableAnimations = () => {
  if (UIManager.setLayoutAnimationEnabledExperimental) {
    UIManager.setLayoutAnimationEnabledExperimental(true);
  }

  return <MainApp />;
}
```

```
    AppRegistry.registerComponent('CollapsibleApp', () =>
EnableAnimations);
```

3. Let's focus on the `Panel` component. First, we need to import all the dependencies that we are going to use for this class:

```
import React, { Component, PropTypes } from 'react';
import {
  View,
  LayoutAnimation,
  StyleSheet,
  Text,
  TouchableOpacity,
} from 'react-native';
```

4. Once we have the dependencies, let's declare the properties this component will receive. The `propTypes` object will contain a `title` property to set the main title of the panel, a `children` property to receive the content of the panel, a `style` property to add custom styles, an `expanded` property to initialize the container and two callbacks that will be executed when expanding or collapsing the panel.

```
const { any, bool, func, string } = PropTypes;

class Panel extends Component {
  static propTypes = {
    children: any,
    title: string,
    expanded: bool,
    onCollapse: func,
    onExpand: func,
    style: any,
  };

  static defaultProps = {
    expanded: false,
    onCollapse: emptyFn,
    onExpand: emptyFn,
  };
}

function emptyFn() {}

const styles = StyleSheet.create({
   // Defined on step
});

export default Panel;
```

5. We are going to use the `height` property on the `state` to expand or collapse the container. The first time this component gets created, we need to check the `expanded` property in order to set the correct initial `height`. We will do this check on the `constructor`:

```
 constructor(props) {
  super(props);

  this.state = {
    height: this.props.expanded ? null : 0,
  };
}
```

6. Let's render the needed JSX elements for this component. We need to get the `height` value from `state` and set it to the content's style view. When pressing the `title` element, we will execute the `onToggle` method, defined in later steps, to change the `height` value to the `state`:

```
render() {
  const { children, style, title } = this.props;
  const { height } = this.state;

  return (
    <View style={[styles.main, style]}>
      <TouchableOpacity onPress={this.onToggle}>
        <Text style={styles.title}>
          {title}
        </Text>
      </TouchableOpacity>
      <View style={{ height }}>
        {children}
      </View>
    </View>
  );
}
```

7. As mentioned before the `onToggle` method, will get executed when pressing the `title` element, here we will only toggle the `height` on the `state`. And call the animation we want to use when updating the styles on the next render cycle:

```
onToggle = () => {
  LayoutAnimation.spring();
  this.setState({
    height: this.state.height === null ? 0 : null,
  })
}
```

8. To complete this component, let's add some simple styles. We need to set the `overflow` to `hidden`, otherwise, the content will be shown when the component is collapsed:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: '#fff',
    borderRadius: 3,
    overflow: 'hidden',
  },
  title: {
    fontWeight: 'bold',
    padding: 15,
  }
});
```

9. Once we have our `Panel` component defined, let's use it on the `MainApp` class. First, we need to require all the dependencies:

```
import React from 'react';
import {
  Text,
  StyleSheet,
  View,
} from 'react-native';
import Panel from './Panel';
```

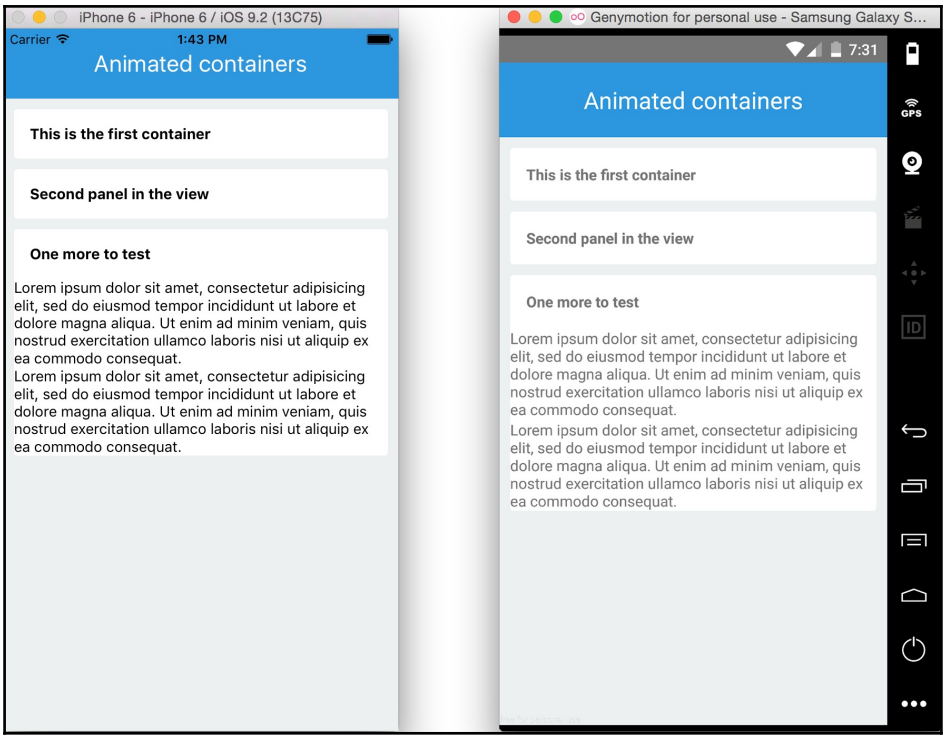
10. In the previous step, we are importing the `Panel` component, we are going to declare a few instances of this class on the JSX. Because we won't use the state or any other components lifecycle, we will use a stateless component:

```
const MainApp = () => (
  <View style={styles.main}>
    <Text style={styles.toolbar}>Animated containers</Text>
    <View style={styles.content}>
      <Panel

        style={styles.panel}
      >
        <Text>Lorem ipsum dolor sit amet...</Text>
      </Panel>
      <Panel

        style={styles.panel}
          >
        <Text>Lorem ipsum dolor...</Text>
      </Panel>
```

```
        <Panel
          expanded

          style={styles.panel}
         >
          <Text>Lorem ipsum...</Text>
          <Text>Lorem ipsum...</Text>
        </Panel>
      </View>
    </View>
  );
  export default MainClass;
```

11. Finally, let's add some styles to the toolbar and the main container. Something very simple, just `padding`, `margin`, and `color`:

```
const styles = StyleSheet.create({
  main: {
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#3498db',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
  content: {
    padding: 10,
    backgroundColor: '#ecf0f1',
    flex: 1,
  },
  panel: {
    marginBottom: 10,
  }
});
```



How it works…

In step number *2*, we are bootstrapping the Android app. The code here is quite different than the iOS version because on Android we need to enable the layout animation API using the `UIManager`. If we don't do this step, the layout animation won't work.

In step *4*, we are declaring the `props` for the `Panel` class. Here we initialize the container as collapsed by default, but if the `expanded` property is set to `true`, then we will render the container expanded.

In step number *5*, we are setting the initial `height` of the content. If the `expanded` prop is set to `true`, then we should show the content. By setting the `height` value to `null`, the layout system will calculate the `height` based on the content, otherwise, we need to set the value to zero, this will hide the content when the component is collapsed.

In step number *6*, we are defining all the JSX for the `Panel` component. There are a couple of concepts on this step. First, we get the `children` property from the `props` object, this will contain any elements defined between `<Panel>` and `</Panel>` when using this component on the `MainApp` class. This is very helpful because by using this property, we are allowing this component to receive any other components as children.

In the same step, we are getting the `height` from the `state` and setting it as the styles of the main `View`. This will update the `height` and automatically the component will expand or collapse. We are also declaring the `onPress` callback; when pressing the title element, we will toggle the `height` on the state.

In step number *7*, we are toggling the `height` value. Here, we are using the `LayoutAnimation` class. By calling the `spring` method, the layout system will animate whatever changes happened on the layout on the next render. In this case, we are only changing the `height`, but we can change anything else that we want, for example, `opacity`, `position`, or `color`.

The `LayoutAnimation` class contains a couple of animations defined already, in this example we are using the `spring`, but we can use the `linear`, `easeInEaseOut`, or create your own using the `configureNext` method.

If we remove the `LayoutAnimation` we won't see any animation, the component will expand and collapse by jumping from `0` to total height, but by adding that single line, we will be able to see a really nice and smooth animation. It's that simple, React will do everything for you. Of course, if you need more control over the animation you definitely will need to use the Animation API instead.

Loading animation

In this recipe, we will continue working with the `LayoutAnimation` class. Here we will create a button when the user presses the button we will show a loading indicator and animate the styles.

Getting ready

Before we start, we need to create an empty app. We are going to call it `ButtonLoading`, feel free to use any other name, just make sure to use the correct name when registering the bootstrap component.

In addition, we need to create the `src/MainApp.js` and `src/Button/index.js` files.

How to do it…

1. Open the `index.ios.js` file, remove the current code and add the following. Make sure to use the name of the app, in this case, `ButtonLoading`:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('ButtonLoading', () => MainApp);
```

2. Open the `index.android.js` file, remove the existing code and add the following. The code is a little bit different because for Android we need to enable the layout animations:

```
import React from 'react';
 import { AppRegistry, UIManager } from 'react-native';
 import MainApp from './src/MainApp';

 const EnableAnimations = () => {
   if (UIManager.setLayoutAnimationEnabledExperimental) {
     UIManager.setLayoutAnimationEnabledExperimental(true);
   }

   return <MainApp />;
 }

AppRegistry.registerComponent('ButtonLoading', () =>
EnableAnimations);
```

3. Let's open the `src/Button/index.js` and import all the dependencies for this component:

```
import React, { Component, PropTypes } from 'react';
import {
  ActivityIndicator,
  LayoutAnimation,
  StyleSheet,
  Text,
  TouchableOpacity,
  View,
} from 'react-native';
```

4. We are going to use only four props for this component, a `label`, a `loading` flag to either show the loading indicator or the label, a callback function to execute it when the button is pressed and custom styles:

```
const { any, bool, func, string } = PropTypes;

class Button extends Component {
  static propTypes = {
    label: string,
    onPress: func,
    loading: bool,
    style: any,
  };

  static defaultProps = {
    loading: false,
    onPress: emptyFn,
  };
  // Defined on later steps...
}

function emptyFn() {}

export default Button;
```

5. The `render` method of this component will be as simple as possible; we will render the label and the activity indicator depending on the `loading` prop:

```
render() {
  const { loading, style } = this.props;

  return (
    <TouchableOpacity
      style={[
        styles.main,
        style,
        loading ? styles.loading : null,
      ]}
      activeOpacity={0.6}
      onPress={this.onPressButton}
    >
      <View>
        {this.renderLabel()}
        {this.renderActivityIndicator()}
      </View>
    </TouchableOpacity>
  );
```

```
}
```

6. In order to render the `label`, we need to check if the `loading` prop is `false`, then we return only a `Text` element with the `label` we received from the `props`:

```
renderLabel() {
  const { label, loading } = this.props;
  if(!loading) {
    return (
      <Text style={styles.label}>{label}</Text>
    );
  }
}
```

7. To render the activity indicator, the `loading` prop should be `true`. If so, we will return the `ActivityIndicator` component. It will be small and the `color` will be white:

```
renderActivityIndicator() {
  if (this.props.loading) {
    return (
      <ActivityIndicator size="small" color="#fff" />
    );
  }
}
```

8. One more method is pending in our class. We need to inform the parent of this component that this button has been pressed; we do this by calling the `onPress` callback. We are also using the `LayoutAnimation` to queue an animation on the next render:

```
onPressButton = () => {
  const { loading, onPress } = this.props;

  LayoutAnimation.easeInEaseOut();
  onPress(!loading);
}
```

9. To complete this component, we need to add some styles. We need to define some colors, rounded corners, alignments, paddings and so on. For the `loading` styles, we will update the paddings to create a nice circle when we show the activity indicator:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: '#e67e22',
    borderRadius: 20,
    padding: 10,
    paddingLeft: 50,
    paddingRight: 50,
  },
  label: {
    color: '#fff',
    fontWeight: 'bold',
    textAlign: 'center',
    backgroundColor: 'transparent',
  },
  loading: {
    padding: 10,
    paddingLeft: 10,
    paddingRight: 10,
  },
});
```

10. We are done with the `Button` component. Now it's time to work on the `MainApp` class, let's start by importing all the dependencies:

```
import React, { Component } from 'react';
import {
  Text,
  StyleSheet,
  View,
} from 'react-native';
import Button from './Button';
```

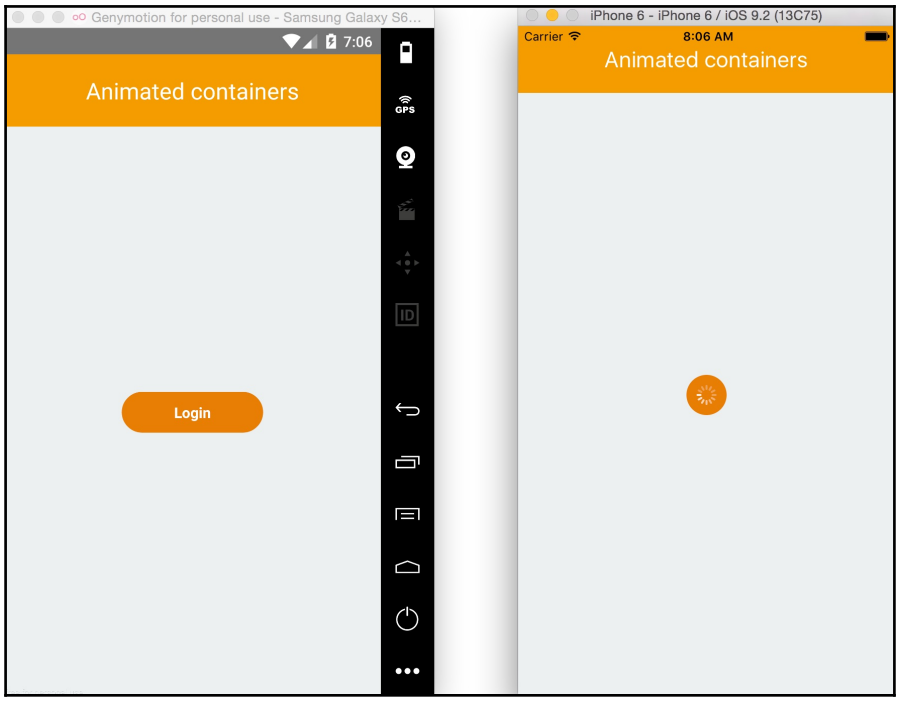
11. The class is very simple, we will only define a `loading` property on the `state` to handle the `Button` animation. We are also rendering a `toolbar` and a `Button`:

```
class MainApp extends Component {

  state = {
    loading: false,
  };

  onPressBtn = (loading) => {
    this.setState({ loading });
  }

  render() {
    const { loading } = this.state;

    return (
      <View style={styles.main}>
        <Text style={styles.toolbar}>Animated containers</Text>
        <View style={styles.content}>
          <Button
            label="Login"
            loading={loading}
            onPress={this.onPressBtn}
          />
        </View>
      </View>
    );
  }
}

export default MainApp;
```

12. Finally, we will add some `styles`. Just some colors, paddings, and alignments to center the button on screen:

```
const styles = StyleSheet.create({
  main: {
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#f39c12',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
```

```
    content: {
      padding: 10,
      backgroundColor: '#ecf0f1',
      flex: 1,
      alignItems: 'center',
      justifyContent: 'center',
    },
  });
```



How it works…

In step number *6*, we are adding the `render` method. Here we are getting the `loading` prop, and based on that value, we add the loading styles to the main container or not. We are also calling two methods, one to render the label and the other one to render the activity indicator.

In step number *8*, we are executing the `onPress` callback. By default, we declared an empty function; this way, we don't need to check if the value is present or not.

The parent of this button should be responsible for updating the loading property when running the `onPress` callback. From this component, we are only responsible for informing the parent when this button has been pressed.

The `LayoutAnimation.eadeInEasiOut` method is only queuing an animation on the next render phase, which means the animation is not executed right away. We are responsible for changing the styles that we want to animate. If we don't change any styles, then we won't see any animation.

The `Button` component doesn't know how the `loading` property gets updated. It might be because of a fetch request, a time-out, or any other action. The parent component is the responsible for updating the `loading` prop. Whenever changes happen, we apply the new styles to the button and run a smooth animation.

In step *11*, we are defining the content of the `MainApp` class. Here we define the `Button` component that we already have. When the button is pressed, we are only updating the state of the loading property. This way, we will show the animation every time the button is pressed.

We are not doing anything else when pressing the button, but if we are using redux, we should call an action here to fetch some data from an API.

Removing items from a list component

In this recipe, we will learn how to drag elements from a `ListView` to remove them. We are going to use the `PanResponder` to handle the drag events.

Getting ready

We need to create an empty app. For this recipe, we will name it `ListRemoveItems`. Feel free to use any other name, just make sure to use the correct name in step *1*.

We also need to create three files: `src/MainApp.js`, `src/ContactList/index.js`, and `src/ContactList/ContactItem.js`.

How to do it…

1. Open the `index.ios.js` and `index.android.js` files, remove the existing code, and add the following:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('ListRemoveItems', () => MainApp);
```

2. Let's start by importing the dependencies for the `MainApp` class. Open the `src/MainApp.js` file and add the following code:

```
import React from 'react';
import {
  Text,
  StyleSheet,
  View,
} from 'react-native';
import ContactList from './ContactList';
```

3. This component will be very simple. All we need to render is a `toolbar` and the `ContactList` component that we imported in the previous step:

```
const MainApp = () => (
  <View style={styles.main}>
    <Text style={styles.toolbar}>Contacts</Text>
    <ContactList style={styles.content} />
  </View>
);

const styles = StyleSheet.create({
  main: {
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#2c3e50',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
  content: {
    padding: 10,
    flex: 1,
```

```
  },
});

export default MainApp;
```

4. We have all we need to start working on the actual list. Let's open the `src/ContactList/index.js` file and import all the dependencies:

```
import React, { Component, PropTypes } from 'react';
import {
  ListView,
  ScrollView,
} from 'react-native';
import ContactItem from './ContactItem';
```

5. Then we need to define some data. Ideally, we would fetch the data from an API; however, to keep things simple and focus only on the drag functionality, let's just define the data in this same file. For this example, I will define only two records, but feel free to create more:

```
const { any } = PropTypes;
const data = [
  {id: 1, name: 'Crysfel Villa'},
  {id: 2, name: 'Stan Bershadskiy'},
  // Define more items here.
];
```

6. The `state` for this component will only contain two properties, the data for the list and a Boolean value that will be updated when the dragging starts or ends:

```
class ContactList extends Component {
  constructor(props) {
    super(props);

    this.ds = new ListView.DataSource({
      rowHasChanged: (r1, r2) => r1 !== r2
    });
    this.state = {
      dataSource: this.ds.cloneWithRows(data),
      swiping: false,
    };
  }
  // Defined at later steps
}

export default ContactList;
```

7. The `render` method only needs to display the list. In the `renderScrollComponent` property, we are enabling the scroll only when the user is not swiping an item on the list. If the user is swiping, we are going to disable the vertical scrolling:

```
render() {
  const { dataSource, swiping } = this.state;

  return (
  <ListView
     key={data}
     enableEmptySections
     dataSource={dataSource}
     renderScrollComponent={
      (props) => <ScrollView {...props} scrollEnabled={!swiping}/>
     }
     renderRow={this.renderItem}
   />
  );
}
```

8. The `renderItem` method will return each item in the list. Here we need to send the `contact` information as a prop and three callbacks:

```
renderItem = (contact) => (
  <ContactItem
    contact={contact}
    onRemove={this.onRemoveContact}
    onDragEnd={this.onToggleSwipe}
    onDragStart={this.onToggleSwipe}
  />
);
```

9. We need to toggle the value of the swiping property on `state`; this will lock and unlock the vertical scroll on the list:

```
onToggleSwipe = () => {
  this.setState({ swiping: !this.state.swiping });
}
```

10. When removing an item, we need to find the `index` of the given `contact` and then remove it from the original list. After that, we need to update the `datasource` on the state to re-render the list with the resulting data:

```
onRemoveContact = (contact) => {
    const index = data.findIndex(
```

```
        (item) => item.id === contact.id
      );
      data.splice(index, 1);

      this.setState({
          dataSource: this.ds.cloneWithRows(data),
      });
    }
```

11. We are done with the list, now let's focus on the items. Let's open the `src/ContactList/ContactItem.js` file and import the dependencies:

```
import React, { Component, PropTypes } from 'react';
import {
  Animated,
  Easing,
  PanResponder,
  StyleSheet,
  Text,
  TouchableHighlight,
  View,
} from 'react-native';
```

12. We need to define the props for this component. The `propTypes` object will have a contact object to receive the data to render, an `onPress` function to execute when the item is pressed, an `onRemove` function to execute when the contact gets removed and two functions to listen for the drag events. On the `state`, we only need to define an animated value to hold the `X` and `Y` value of the dragging:

```
const { func, object } = PropTypes;

class ContactItem extends Component {
  static propTypes = {
    contact: object,
    onPress: func,
    onRemove: func,
    onDragStart: func,
    onDragEnd: func,
  };

  static defaultProps = {
    onPress: emptyFn,
    onRemove: emptyFn,
    onDragEnd: emptyFn,
    onDragStart: emptyFn,
  };
```

```
    state = {
      pan: new Animated.ValueXY(),
    };
  }

  function emptyFn() {}

  export default ContactItem;
```

13. When the component is created, we need to configure `PanResponder`. We will do this on `componentWillMount`. The `PanResponder` is responsible for handling gestures; it provides a simple API to capture the events of the finger:

```
componentWillMount() {
  this.panResponder = PanResponder.create({
    onMoveShouldSetPanResponderCapture: this.onShouldDrag,
    onPanResponderMove: Animated.event(
      [null, { dx: this.state.pan.x }]
    ),
    onPanResponderRelease: this.onReleaseItem,
    onPanResponderTerminate: this.onReleaseItem,
  });
}
```

14. Now let's define the actual functions that will get executed for each callback defined in the previous step. We can start with the `onShouldDrag` method:

```
onShouldDrag = (e, gesture) => {
  const { dx } = gesture;
  return Math.abs(dx) > 2;
}
```

15. The `onReleaseItem` is a little bit more complicated. We are going to split this method into two steps. First, we need to figure out if the current item needs to be removed or not, in order to do that we need to set a threshold:

```
onReleaseItem = (e, gesture) => {
  const { onRemove, contact,onDragEnd } = this.props;
  const move = this.rowWidth - Math.abs(gesture.dx);
  let remove = false;
  let config = {  // Animation to origin position
    toValue: { x: 0, y: 0 },
    duration: 500,
  };

  if (move < this.threshold) {
    remove = true;
```

```
      if (gesture.dx > 0) {
        config = { // Animation to the right
          toValue: { x: this.rowWidth, y: 0 },
          duration: 100,
        };
      } else {
        config = { // Animation to the left
          toValue: { x: -this.rowWidth, y: 0 },
          duration: 100,
        };
      }
    }

    // Add the code here on the next step
  }
```

16. Once we have the configurations for the animation, we are ready to move the item! We also need to execute the `onDragEnd` callback and if the item <ie>need</ie>s to be removed, we need to run the `onRemove` function:

```
onReleaseItem = (e, gesture) => {
  // Previous step...

     onDragEnd();
  Animated.spring(
    this.state.pan,
    config,
  ).start(() => {
    if (remove) {
      onRemove(contact);
    }
  });
```

17. We have all the dragging system in place. Now we need to define the `render` method; something very simple, just a text to display the contacts name, the touch element, the animated view, and a wrapper:

```
render() {
  const { contact, onPress } = this.props;

  return (
    <View style={styles.row} onLayout={this.setThreshold}>
      <Animated.View
        style={[styles.pan, this.state.pan.getLayout()]}
        {...this.panResponder.panHandlers}
      >
        <TouchableHighlight
```

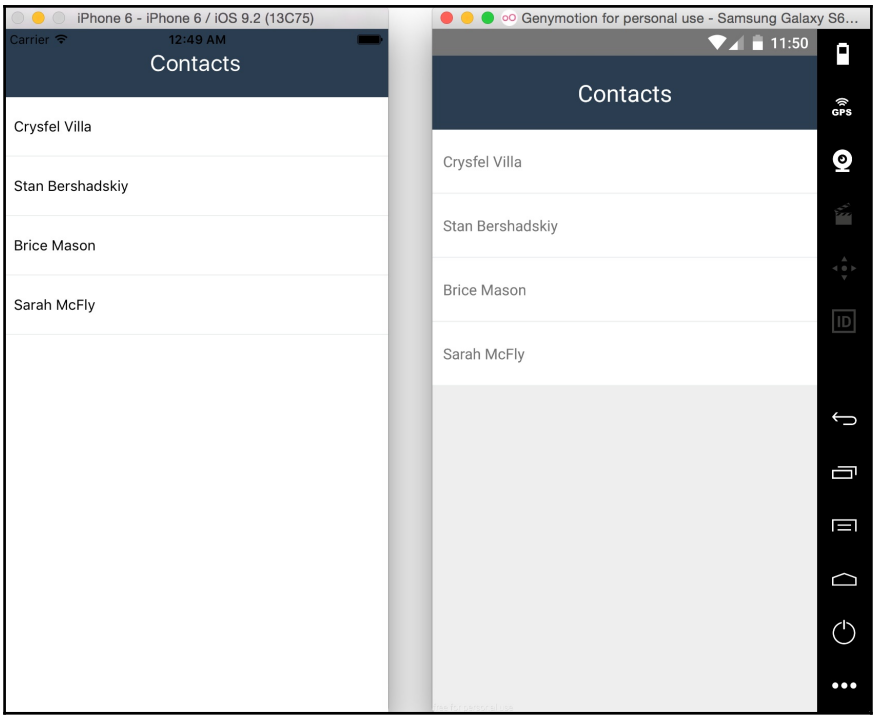
```
            style={styles.info}
            onPress={() => onPress(contact)}
            underlayColor="#ecf0f1"
          >
            <Text>{contact.name}</Text>
          </TouchableHighlight>
        </Animated.View>
      </View>
    );
  }
```

18. We need one more method on this class. The `setThreshold` will get the current `width` of the `row` and set the `threshold`. In this case, it will be a third of the width. These values are required to decide whether to remove the item or not:

```
setThreshold = (event) => {
  const { layout: { width } } = event.nativeEvent;
  this.threshold = width/3;
  this.rowWidth = width;
}
```

19. Finally, some styles for the row. By now we should be familiar with this code, we are only adding some colors and paddings.

```
const styles = StyleSheet.create({
  row: {
    backgroundColor: '#ecf0f1',
    borderBottomWidth: 1,
    borderColor: '#ecf0f1',
    flexDirection: 'row',
  },
  pan: {
    flex: 1,
  },
  info: {
    backgroundColor: '#fff',
    paddingBottom: 20,
    paddingLeft: 10,
    paddingTop: 20,
  },
});
```



How it works…

In step number *6*, we are defining the `swiping` property in the state. This property is just a flag that will be set to `true` when the dragging starts and `false` when is completed. We need this information to lock the vertical scrolling on the list while dragging around the item.

In step number *8*, we are defining the content of each row in the list. The `onDragStart` property receives the `onToggleSwipe` method that will be executed when the dragging starts. We are also going to execute the same method when the dragging is completed.

In the same step, we are sending to each item the `onRemoveContact` method. As the name suggests, we are going to remove from the list the current item when the user swipes it out.

In step *12*, we are defining the props and state of the item component. So far, we have been doing animation using a single value, but for this case, we need to handle the *x* and *y* coordinates, therefore we need an instance of `Animated.ValueXY`. Internally this class handles two `Animated.Value` instances, therefore the API is almost identical.

In step *13*, the pan responder gets created. The gesture system in React Native is very similar to the event system on the web. There are two phases when there's a touch event, the capture, and the bubble. In our case we need to use the capture phase to figure out if the current event is pressing the item or it's trying to drag it.

The `onMoveShouldSetPanResponderCapture` will capture the event; then we need to decide if we will drag the item or not by returning true or false.

The `onPanResponderMove` will get the values from the animation on each frame and it will set it to the `pan` object in the `state`. We need to use `Animated.event` to access the animation values for each frame; in this case, we only need to get the `x` value. Later we will use this value to run a different animation and return the element to its original place or remove it from the screen.

The `onPanResponderRelease` callback will be executed when the user releases the item. If for any other reason the dragging gets interrupted, `onPanResponderTerminate` will get executed.

In step *14*, we need to check if the current event is a simple press or a dragging. We can do this by checking the delta on the *x*-axis. If the touch event has been moved more than 2 pixels, then the user is trying to drag the item otherwise is trying to press the button. We evaluate the difference as an absolute number because the movement could be from left to right or right to left; we want to allow both movements.

In step *15*, we need to get the distance with respect to the width of the device. If this distance is lower than the threshold, then we need to remove these items. We are defining the `config` object for each animation, by default it will return to the original position. But, if we need to remove the item, we need to check the direction and set the configuration accordingly.

In step *17*, we are defining the JSX. We set the styles that we want to animate; in this case, it's the `left` property. Instead of manually creating an object, we can call the `getLayout` method from the animated value. It will return the top and left properties with the existing values.

In that same step, we are setting the event handlers to the animated view. This will allow binding the dragging configuration we have defined in the previous steps to this animated view.

We are also calling the `onPress` callback with the current `contact` information.

Creating a Facebook reactions widget

In this recipe, we will create a component similar to the Facebook's reaction widget. We will have a like button image, when pressed we will show four icons with a slide-in animation and opacity from `0` to `1`.

Getting ready

Let's create an empty app called `FacebookReactions`; feel free to use any other name, just make sure to use the correct one when registering the bootstrap component.

We are going to need some images to display a fake timeline, a picture of your cat will work, just make sure it's about 900×600 pixels or so. We also need five icons to display the emotions, such as like, angry, laughing, heart, and surprised. Feel free to use your own assets or download the ones provided with this book.

Let's create three JavaScript files, `src/MainApp.js`, `src/Reactions/index.js` and `src/Reactions/Icon.js`. We need to copy the pictures of our cat or the ones provided with this book to `src/images/`, the reaction icons should be on `src/Reactions/images/`.

How to do it…

1. Open the `index.ios.js` and `index.android.js` files, remove the existing code and add the following:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('FacebookReactions', () => MainApp);
```

2. We are going to create a fake timeline on the `MainApp` class, let's start by importing the dependencies:

```
import React from 'react';
import {
  Dimensions,
  Image,
  Text,
  ScrollView,
  StyleSheet,
```

```
  View,
} from 'react-native';
import Reactions from './Reactions'
```

3. First, we need to import some images to render in our timeline. The JSX is very simple, just a `toolbar`, a `ScrollView` with two `Images` and two `Reaction` components:

```
const image1 = require('./images/01.jpg');
const image2 = require('./images/02.jpg');
const { width } = Dimensions.get('window');

const MainApp = () => (
  <View style={styles.main}>
    <Text style={styles.toolbar}>Reactions</Text>
    <ScrollView style={styles.content}>
      <Image source={image1} style={styles.image}
resizeMode="cover" />
      <Reactions />

      <Image source={image2} style={styles.image}
resizeMode="cover" />
      <Reactions />
    </ScrollView>
  </View>
);

export default MainApp;
```

4. Finally, we need the `styles` for this component. Nothing fancy, just some colors, size to the image, and some paddings here and there:

```
const styles = StyleSheet.create({
  main: {
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#3498db',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
  content: {
    flex: 1,
  },
  image: {
```

```
    width,
    height: 300,
  },
});
```

We are ready to start working on the main component of this recipe. Let's import the dependencies. We will work on the `Icon` component at later steps:

```
import React, { Component, PropTypes } from 'react';
import {
  Image,
  Text,
  TouchableOpacity,
  StyleSheet,
  View,
} from 'react-native';
import Icon from './Icon';
```

5. Let's define the props and initial state; we also need to require the like `icons` to show it on screen:

```
const image = require('./images/like.png');
const { array } = PropTypes;

class Reactions extends Component {

  static propTypes = {
    icons: array,
  };

  static defaultProps = {
    icons: [
      'like', 'heart', 'angry', 'laughing', 'surprised',
    ],
  };

  state = {
    show: false,
    selected: '',
  };

  // Defined at later steps
}

export default Reactions;
```

6. Let's define the functions that toggle the show value and the function that sets the current selection in the `state`:

```
onSelectReaction = (reaction) => {
  this.setState({
    selected: reaction,
  });
  this.toggleReactions();
}

toggleReactions = () => {
  this.setState({
    show: !this.state.show,
  });
};
```

7. Now we need to define the JSX for this component. We are going to display an image when pressed; we will call the `toggleReactions` method that we already have:

```
render() {
  const { style } = this.props;
  const { selected } = this.state;

  return (
    <View style={[style, styles.container]}>
      <TouchableOpacity onPress={this.toggleReactions}>
        <Image source={image} style={styles.icon} />
      </TouchableOpacity>
      <Text>{selected}</Text>
      {this.renderReactions()}
    </View>
  );
}
```

8. If you noticed, we are calling the `renderReactions` method. Here we will render all the icons that we want to display when the user presses the main button:

```
renderReactions() {
  const { icons } = this.props;
  if (this.state.show) {
    return (
      <View style={styles.reactions}>
      { icons.map((name, index) => (
          <Icon
            key={index}
```

```
            name={name}
            delay={index * 100}
            index={index}
            onPress={this.onSelectReaction}
          />
        ))
      }
      </View>
    );
  }
}
```

9. Finally, we need to set the `styles` for this component. Just sizing the image and defining some paddings. The `reactions` container will have a `0` height; this is because the icons will be floating and we don't want to add any extra space:

```
const styles = StyleSheet.create({
  container: {
    padding: 10,
  },
  icon: {
    width: 30,
    height: 30,
  },
  reactions: {
    flexDirection: 'row',
    height: 0,
  },
});
```

10. The `icon` component is missing if we try to run our app it will fail. Let's work on this component; open the `src/Reactions/Icon.js` file and add the imports:

```
import React, { Component, PropTypes } from 'react';
import {
  Animated,
  Dimensions,
  Easing,
  Image,
  StyleSheet,
  TouchableOpacity,
  View,
} from 'react-native';
```

11. Let's get the `PropTypes` that we are going to use and require the images for each reaction. We are going to use an object for the icons. This way, we will be able to easily retrieve an image by the key name:

```
const { func, number, string } = PropTypes;
const icons = {
  angry: require('./images/angry.png'),
  heart: require('./images/heart.png'),
  laughing: require('./images/laughing.png'),
  like: require('./images/like.png'),
  surprised: require('./images/surprised.png'),
};
```

12. Now we should define the properties this component will receive. We don't need to define an initial state:

```
class Icon extends Component {
  static propTypes = {
    delay: number,
    index: number,
    name: string,
    onPress: func,
  };

  static defaultProps = {
    delay: 0,
    onPress: emptyFn,
  };

  // Defined on next steps
}

function emptyFn() {}

export default Icon;
```

13. The icons should appear on screen with a nice animation; therefore, we need to create and run the animation when the component is mounted:

```
componentWillMount() {
  this.animatedValue = new Animated.Value(0);
}

componentDidMount() {
  const { delay } = this.props;

  Animated.timing(
```

```
      this.animatedValue,
      {
        toValue: 1,
        duration: 200,
        easing: Easing.elastic(1),
        delay,
      }
    ).start();
  }
```

14. When pressing the icon, we need to execute the `onPress` callback to inform the parent a reaction was selected. We will send the name of the reaction as a parameter:

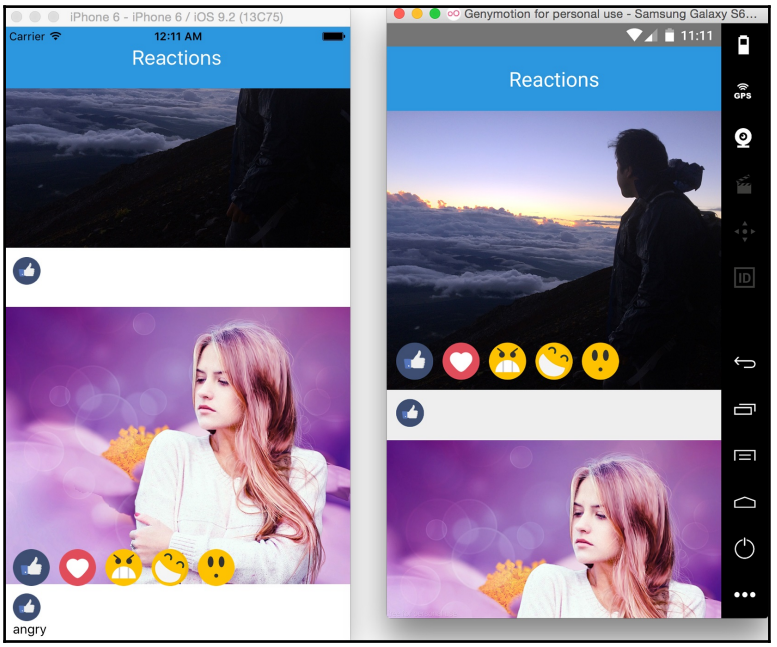
```
onPressIcon = () => {
  const { onPress, name } = this.props;
  onPress(name);
}
```

15. The last piece of the puzzle is the `render` method. We need to define the JSX for this component:

```
render() {
  const { name, index, onPress } = this.props;
  const left = index * 50;
  const top = this.animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [10, -95],
  });
  const opacity = this.animatedValue;

  return (
    <Animated.View
      style={[
        styles.icon,
        { top, left, opacity },
      ]}
    >
      <TouchableOpacity onPress={this.onPressIcon}>
        <Image source={icons[name]} style={styles.image} />
      </TouchableOpacity>
    </Animated.View>
  );
}
```

16. As the very final step, we need to add the styles for each `icon`. We need the icons to float, therefore we will set the `position` as `absolute` and the `width` and `height` as `40` pixels. After this change, we should be able to run our app:

```
const styles = StyleSheet.create({
  icon: {
    position: 'absolute',
  },
  image: {
    width: 40,
    height: 40,
  },
});
```



How it works…

In step *3*, we are defining the `Reactions` component in the timeline. For now, we are not focusing on handling data, but rather on displaying the UI. Therefore, we are not sending any callback to get the selected value.

In step *5*, we are defining the `props`, the default values, and the initial state. For the props we only have an array of icons, we could also have some callbacks to let the parent know when a reaction gets selected.

We have two properties in the state:

- The `show` property is a Boolean value. We are going to toggle it when the user presses the main button. When `false` we are not going to show the reactions; when `true`, we will run the animation to show each icon.
- The `selected` property will contain the current selection. Every time a new reaction gets selected, we are going to update this property.

In step *8*, we are rendering the icons. Here we need to send the name of the icon to every instance created, we are also sending a `delay` of 100 milliseconds for each icon, this will allow us to show the icons one after the other with a nice animation. The `onPress` prop receives the method we already defined to set the state the selected reaction.

In step *12*, we are defining the `props` this component will receive. The `name` property will receive the image to display; in the previous step we required all the images in an object, each image has a key, the property `name` should receive any of those values.

In step *13*, the animation gets created. First, we create the `animatedValue`, as mentioned before this it's the class responsible for holding the value for each frame in the animation. As soon as the component is mounted, we run the animation. We will start from `0` to `1`, with a duration of 200 milliseconds, using an elastic movement and we will delay the animation based on the received parameter.

In step *15*, we defined the JSX for the `Icon` component. Here we will animate the `top` and `opacity` properties. For the top property, we need to interpolate the values from the `animatedValue`, we want to move the icon 95 pixels to the top from its original position. The required values for the `opacity` property are from `0` to `1`, therefore we don't need to interpolate anything; we can just use the `animatedValue` as it is.

The `left` value is calculated based on the `index`; we are just moving the icon 50 pixels to the left. This will avoid rendering the icons all in the sample place.

Display images in full screen

In this recipe we will create a timeline of images, then we will allow the user to press any of the images to display it with a black background and the image in the center of the screen.

We will use an opacity animation for the background and we will slide in the image from their original position.

Getting ready

Let's create an empty app; we are going to call it `PhotoViewer`. Feel free to use any other name, just make sure to set the correct name when registering the bootstrap component.

In addition, we need to create three files for this project. `src/MainApp.js` to display the timeline, `src/PostContainer/index.js` to show each image in the timeline and `src/PhotoViewer/index.js` to show the selected image in full screen.

How to do it…

1. Let's open the `index.ios.js` and `index.android.js` files. Remove the existing code and add the following. Make sure to use the correct name for your app:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('PhotoViewer', () => MainApp);
```

2. We are going to display a `timeline` with images of the `MainApp class`. Let's import all the dependencies, including the two other components we will create in later steps:

```
import React, { Component } from 'react';
import {
  Dimensions,
  Image,
  Text,
  ScrollView,
  StyleSheet,
  View,
} from 'react-native';
import PostContainer from './PostContainer';
```

```
import PhotoViewer from './PhotoViewer';
```

3. As mentioned before, we are going to load the images from Amazon S3. In this step, we are going to define the data that we are going to render. Just a simple array of objects containing a `title` and the `image`:

```
const path = 'https://s3.amazonaws.com/crysfel/public/book/03/08';
const timeline = [
  { title: 'Enjoying the fireworks', image: `${path}/01.jpg` },
  { title: 'Climbing the Mount Fuji', image: `${path}/02.jpg` },
  { title: 'Check my last picture', image:  `${path}/03.jpg` },
  { title: 'Sakuras are beautiful!', image: `${path}/04.jpg` },
];
```

4. Now we need to declare the initial state of this component. We will update the `selected` and `position` properties when any of the images gets pressed:

```
class MainApp extends Component {
  state = {
    selected: null,
    position: null,
  };
  // Defined on next steps
}
export default MainApp;
```

5. In order to update the state, we are going to declare two methods. One to set the value of the image that has been pressed and one more to remove those values when the viewer gets closed:

```
showImage = (selected, position) => {
  this.setState({
    selected,
    position,
  });
}

closeViewer = () => {
  this.setState({
    selected: null,
    position: null,
  });
}
```

6. Now we are ready to work on the `render` method. Here, we need to render each image inside a `ScrollView`, this way we will be able to scroll up and down:

```
render() {
  return (
    <View style={styles.main}>
      <Text style={styles.toolbar}>Timeline</Text>
      <ScrollView style={styles.content}>
      {
        timeline.map((post, index) =>
          <PostContainer key={index} post={post}
         onPress={this.showImage} />
        )
      }
      </ScrollView>
      {this.renderViewer()}
    </View>
  );
}
```

7. In the previous step, we are calling the `renderViewer` method. Here we will show the viewer component only if there's a post `selected` in the state. We are also sending the initial position to start the animation and a callback to close the viewer:

```
renderViewer() {
  const { selected, position } = this.state;

  if (selected) {
    return (
      <PhotoViewer
        post={selected}
        position={position}
        onClose={this.closeViewer}
      />
    );
  }
}
```

8. The styles for this component are very simple, only some colors and paddings. Nothing much to explain here:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: '#ecf0f1',
    flex: 1,
  },
```

```
  toolbar: {
    backgroundColor: '#2c3e50',
    color: '#fff',
    fontSize: 22,
    padding: 20,
    textAlign: 'center',
  },
  content: {
    flex: 1,
  },
});
```

9. The timeline is completed, but if we try to run our app it will fail. Let's work on the `PostContainer` component by importing the dependencies:

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

10. We only need two `props` for this component. The `post` prop will receive the main data for the component, basically the `title` and the `image`. The `onPress` prop is a callback that we will need to run when the image gets pressed:

```
const { width } = Dimensions.get('window');

class PostContainer extends Component {
  static propTypes = {
    post: PropTypes.object,
    onPress: PropTypes.func,
  };

  static defaultProps = {
    onPress: emptyFn,
  };

  // Defined in the following steps
}

function emptyFn() {}

export default PostContainer;
```

11. This component will be inside of a `ScrollView`. This means its position will be changing when the user starts scrolling the content. When pressing the image, we need to get the current position on the screen and send this information to the parent component:

```
onPressImage = (event) => {
  const { onPress, post } = this.props;
  this.refs.main.measure((fx, fy, width, height, pageX, pageY) => {
    onPress(post, {
      width,
      height,
      pageX,
      pageY,
    });
  });
}
```

12. It's time to define the JSX for this component. To keep things simple, we are only going to render the `image` and the `title`:

```
render() {
  const { post: { image, title } } = this.props;

  return (
    <View style={styles.main} ref="main">
      <TouchableOpacity
         onPress={this.onPressImage}
         activeOpacity={0.9}
          >
        <Image
          source={{uri: image}}
          style={styles.image}
          resizeMode="cover"
        />
      </TouchableOpacity>
      <Text style={styles.title}>{title}</Text>
    </View>
  );
}
```

13. As always, we need to define some styles for this component. We are going to add some colors and paddings:

```
const styles = StyleSheet.create({
  main: {
    backgroundColor: '#fff',
    marginBottom: 30,
```

```
    paddingBottom: 10,
  },
  content: {
    flex: 1,
  },
  image: {
    width,
    height: 300,
  },
  title: {
    margin: 10,
    color: '#ccc',
  }
});
```

14. If we run the app now, we should be able to see the timeline. However, if we press any of the images, an error will be shown. We need to define the viewer, so let's open the `src/PhotoViewer/index.js` file and import the dependencies:

```
import React, { Component, PropTypes } from 'react';
import {
  Animated,
  Dimensions,
  Easing,
  Text,
  TouchableOpacity,
  StyleSheet,
} from 'react-native';
```

15. Let's define the `props` for this component. In order to center the image on the screen, we need to know the `height` of the current device:

```
const { width, height } = Dimensions.get('window');
const { func, object } = PropTypes;

class PhotoViewer extends Component {
  static propTypes = {
    onClose: func,
    post: object,
    position: object,
  };

  static defaultProps = {
    onClose: emptyFn,
  };

  // Defined on the following steps
```

```
}
function emptyFn() {}

export default PhotoViewer;
```

16. We want to run two animations when showing this component. Therefore, we need initialize and run the animation when the component is mounted. The animation is very simple, just going from `0` to `1` in `400` milliseconds, using a nice easing movement:

```
componentWillMount() {
  this.animatedValue = new Animated.Value(0);
}

componentDidMount() {
  Animated.timing(
    this.animatedValue,
    {
      toValue: 1,
      duration: 400,
      easing: Easing.in,
    }
  ).start();
}
```

17. When pressing the close button; we need to run the `onClose` callback to inform the parent this component needs to be removed:

```
onPressBtn = () => {
  this.props.onClose();
}
```

18. We are going to split the `render` method into two steps; first, we need to interpolate the values for the animations:

```
render() {
  const { post: { image, title }, position } = this.props;
  const top = this.animatedValue.interpolate({
    inputRange: [0, 1],
    outputRange: [position.pageY, height/2 - position.height/2],
  });
  const opacity = this.animatedValue;

  // Defined on step 19
}
```

19. We only need to define three elements. A `View` to animate the background, an `image` to display the image from Amazon S3, and a `close` button. We are setting the `opacity` style to the main view; this will run a nice animation from transparent to black. The image will slide in and at the same time the opacity will increase, creating a simple but nice effect:

```
  render() {
    // Defined on step 18

    return (
      <Animated.View
        style={[
          styles.main,
          { opacity },
        ]}
      >
        <Animated.Image
          source={{uri: image}}
          style={[
            styles.image,
            { top, opacity }
          ]}
        />
        <TouchableOpacity style={styles.closeBtn}
onPress={this.onPressBtn}>
          <Text style={styles.closeBtnText}>X</Text>
        </TouchableOpacity>
      </Animated.View>
    );
  }
```

20. We are almost done! The last step on this recipe is to define the styles. We need to set the position of the main container to absolute. This is because we want to show the image on top of everything. We are also moving the `close` button to the top right of the screen:

```
  const styles = StyleSheet.create({
    main: {
      backgroundColor: '#000',
      bottom: 0,
      left: 0,
      position: 'absolute',
      right: 0,
      top: 0,
    },
    image: {
```

```
    width,
    height: 300,
  },
  closeBtn: {
    position: 'absolute',
    top: 20,
    right: 20,
  },
  closeBtnText: {
    color: '#fff',
    fontWeight: 'bold',
  },
});
```

How it works…

In step *4*, we are defining two properties in the `state`. The `selected` property will hold the image's data that got pressed; this will be any of the objects defined in step *3*. The `position` property will contain the current *y* coordinate on the screen; we will need this information to animate the image from its original position to the center of the screen.

In step *6*, we loop the `timeline` array to render each `post`. We are using a `PostContainer` element for each post, sending the `post` information along with the `onPress` callback to set the pressed image in the state.

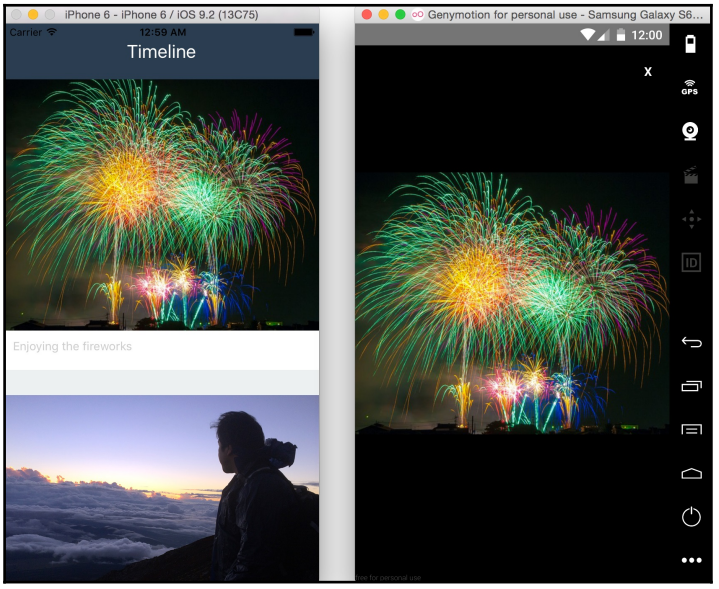
In step *11*, we need to get the current position of the image. To achieve this, we need to use the `measure` method from the component we want to get the information. This method receives a callback function and retrieves very useful information such as the `width` and `height`, but most importantly, the current position on the screen.

We are using a reference to access the component, this reference is declared when defining the JSX on the next step.

In step *12*, we are declaring the JSX of the component. In the main wrapper container, we are setting the `ref` property; we need this to get the current position of the image. Whenever we want to access a component on any of the methods of the current class, we will use a reference. We can create references by simply setting the `ref` property and assigning a name to any component.

In step *18*, we are interpolating the animations values to get the correct top values for each frame. The output of that interpolation will start from the current position of the image to the middle of the screen. This way, the values would be negative or positive, and the animation will run from bottom to top or the other way around.

We don't need to interpolate the `opacity`, as the current animated value already goes from `0` to `1`, as shown in the following screenshot:



4
Working with Application Logic and Data

In this chapter, we will cover the following recipes:

- Storing and retrieving data locally
- Retrieving data from a Remote API
- Sending data to a Remote API
- Establishing real-time communications with WebSockets
- Integrating persistent database functionality with Realm
- Masking the application upon network connection loss
- Synchronizing locally persisted data with a Remote API
- Logging in with Facebook
- Sharing content on Facebook
- Tracking application events with Facebook Analytics

Introduction

One of the most important aspects of developing an application is handling data. This data may come locally from the user, may be served by a remote server that exposes an API, or for most business applications, will be some combination of both. You may wonder what strategies are best for dealing with data, or how to even accomplish simple tasks such as making an HTTP request. Luckily, React Native makes your life that much simpler by providing mechanisms for easily dealing with data.

The open source community has taken things a step further and provided some excellent modules that can be used with React Native. In this chapter, we will discuss how to work with data in all aspects, and how it integrates into our React Native applications.

Storing and retrieving data locally

When developing a mobile app, we need to consider the network challenges. A well-designed app will allow the user to continue using the app without internet connection. This will require the app to save data locally on the device when there's no internet connection, and then sync with the server when the network is available again.

Another challenge to overcome is the fact that network connectivity might be slow or limited. In order to improve the performance of our app, we should save some data on the local device to avoid stressing our server API.

In this recipe, we will learn about a simple and basic strategy to save and retrieve data locally from the device. We will create a simple app with a text input and two buttons, one to save the content of the field and one to load the existing content. We will use the `AsyncStorage` class to achieve our goal.

Getting ready

We need to create an empty app named `SavingLocally`. You can use any other name, just make sure you use the correct name when registering the app.

How to do it…

1. Let's start by creating a `src/MainApp.js` file. We are going to work on this file to save and load the data on the local device.
2. Now, let's open the `index.ios.js` and `index.android.js` files. Remove the existing code and add the following:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('SavingLocally', () => MainApp);
```

3. If we run our app now, it won't work. This is because we need to create the `MainApp` component. Let's start by importing all the dependencies:

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

4. Now let's create the class. We are going to create a `key` constant to set the name of the key we will use to save the content. On the `state`, we will have two properties, one to keep the value from the text input component and one more to load and show the current stored value:

```
const key = '@MyApp:key';

class MainApp extends Component {
  state = {
    text: '',
    storedValue: '',
  };

  // Defined at later steps
}

const styles = StyleSheet.create({
  // Defined at later steps
});

          export default MainApp;
```

5. Let's start by loading the existing stored value. We will display the content on the UI when the app loads, therefore we need to read the local value when the component is about to be mounted:

```
componentWillMount() {
  this.onLoad();
}
```

6. The `onLoad` function loads the current content from the local storage. It's as easy as just using the key for the data we want to retrieve:

```
onLoad = async () => {
  try {
    const storedValue = await AsyncStorage.getItem(key);
    this.setState({ storedValue });
  } catch (error) {
    Alert.alert('Error', 'There was an error while loading the
data');
  }
}
```

7. Saving the data is simple as well. We only need to use a key to save anything we want:

```
onSave = async () => {
  const { text } = this.state;

  try {
    await AsyncStorage.setItem(key, text);
    Alert.alert('Saved', 'Successfully saved on device');
  } catch (error) {
    Alert.alert('Error', 'There was an error while saving the
data');
  }
}
```

8. Now we need to define a function to save the value from the input text to the `state`. When the value of the input changes, we will get the new change and set it to the `state`:

```
onChange = (text) => {
  this.setState({ text });
}
```

9. Our UI will be very simple, just a text to render the saved content, a text input component to allow the user to enter a new value and two buttons. One button will call the `onLoad` function to load the current value and the other will save the value from the input:

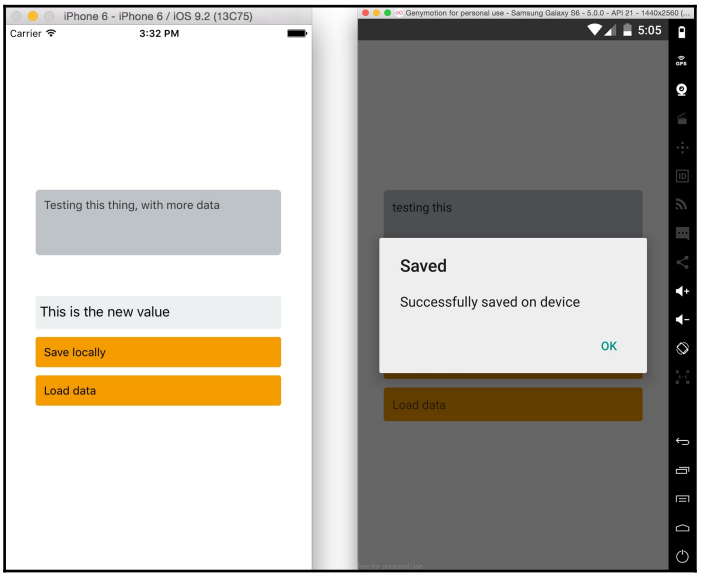
```
render() {
  const { storedValue, text } = this.state;

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

```
        <Text style={styles.preview}>{storedValue}</Text>
        <View>
          <TextInput
            style={styles.input}
            onChangeText={this.onChange}
            value={text}
            placeholder="Type something here..."
          />
          <TouchableOpacity onPress={this.onSave} style={styles.btn}>
            <Text>Save locally</Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this.onLoad} style={styles.btn}>
            <Text>Load data</Text>
          </TouchableOpacity>
        </View>
      </View>
    );
  }
```

10. Finally, let's add some styles. We are just adding some colors, paddings, margins, and things we have covered in `Chapter 1`, *Getting Started*:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  preview: {
    backgroundColor: '#bdc3c7',
    width: 300,
    height: 80,
    padding: 10,
    borderRadius: 5,
    color: '#333',
    marginBottom: 50,
  },
  input: {
    backgroundColor: '#ecf0f1',
    borderRadius: 3,
    width: 300,
    height: 40,
    padding: 5,
  },
  btn: {
    backgroundColor: '#f39c12',
    padding: 10,
```

```
    borderRadius: 3,
    marginTop: 10,
  },
});
```



How it works…

The `AsyncStorage` class allows us to save data on the local device. On iOS, this is accomplished by using dictionaries on text files. On Android, things are slightly different–it will use RocksDB or SQLite, depending on what's available.

It's not recommended to save sensitive information using this method, as the data is not encrypted.

In step *6*, we are loading the current saved data. The `AsyncStorage` API contains a `getItem` method; this method receives the key we want to retrieve as a parameter. We are using the `await/async` syntax here, as this call is asynchronous. After we get the value, we just set it to `state`; this way we will be able to render the data on the view.

In step *7*, we are saving the text from the `state`. Using the `setItem` method, we can set a new `key` with any value we want. This call is asynchronous, therefore we either need to use promises or the new `await/async` syntax.

Retrieving data from a Remote API

In previous chapters, we've been using the data from a JSON file or directly defined on the source code. While that has been working for our previous examples, it's not what we should do in real-world applications.

In this recipe, we will learn how to request data from an API. We will fetch a `GET` request from a Heroku API to get a JSON response. For now, however, we are only going to display the JSON in a text element.

This app will be really simple. We will have a text component where we are going to display the response of the API. We will also add a button to request the data when pressed.

Getting ready

We need to create an empty app, in this case, we'll name it `RemoteApi`. Feel free to use any other name; just make sure you use the correct name when registering the app.

How to do it…

1. Let's start by opening the `index.ios.js` and `index.android.js` files. Remove the previous code and add the following:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('RemoteApi', () => MainApp);
```

2. As you can see, in the previous step we are importing the `src/MainApp.js` file. We need to create this file and add all the dependencies for this class:

```
import React, { PureComponent } from 'react';
import {
  StyleSheet,
  Text,
  TextInput,
  TouchableOpacity,
  View
} from 'react-native';
```

3. We are going to define a result property on the `state`. This property will hold the response from the API. We need it to update the view when we get the response:

```
class MainApp extends PureComponent {
  state = {
    result: '',
  };

  // defined at later steps
}

const styles = StyleSheet.create({
  // defined at later steps

});

export default MainApp;
```

4. We need to send the request when the button gets pressed. Let's create a method to handle the actual request:

```
onLoad = async () => {
  this.setState({ result: 'Loading, please wait...' });

  const response = await
  fetch('https://my-bookmarks-api.herokuapp.com/api/bookmarks', {
    method: 'GET',
  });

  const result = await response.text();

  this.setState({ result });
}
```

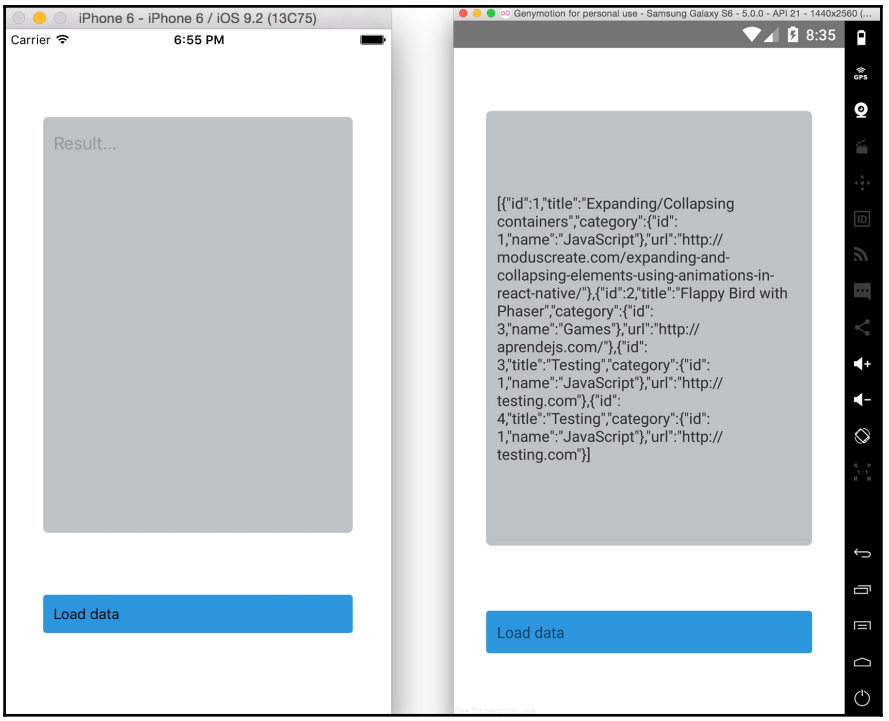
5. On the render method, we need to display the response. We can achieve this by getting the result from the `state`. We will use a text input to display the result. Editing will be disabled, and we will support multiline functionality. The button will only call the `onLoad` function we already have from the previous step:

```
render() {
  const { result } = this.state;

  return (
    <View style={styles.container}>
      <View>
        <TextInput
```

```
            style={styles.preview}
            value={result}
            placeholder="Result..."
            editable={false}
            multiline
          />
          <TouchableOpacity onPress={this.onLoad} style={styles.btn}>
            <Text>Load data</Text>
          </TouchableOpacity>
        </View>
      </View>
    );
  }
```

6. Finally, let's add some styles–alignment, colors, margins, and paddings. Nothing really fancy:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  preview: {
    backgroundColor: '#bdc3c7',
    width: 300,
    height: 400,
    padding: 10,
    borderRadius: 5,
    color: '#333',
    marginBottom: 50,
  },
  btn: {
    backgroundColor: '#3498db',
    padding: 10,
    borderRadius: 3,
    marginTop: 10,
  },
});
```



How it works…

In step *4*, we are sending the request to the Heroku API. We are using the `fetch` method to handle the requests. The first parameter is a string with the URL of the endpoint, the second parameter is a configuration object; here, we are only defining the method as `GET`, but we can also define headers, cookies, parameters, and so many other things.

We are also using `async/await` syntax, but we could also use promises to get the response and finally set it to the `state`.

We are using an `arrow` function here to properly handle the scope. This will automatically set the correct scope when this method is assigned to the `onPress` callback.

Sending data to a Remote API

In the previous recipe, we learned how to get data from an API using `fetch`. In this recipe, we will learn how to POST data to the same endpoint to add new bookmarks.

Getting ready

Before going through this recipe, we need to create an empty app named `SendingData`. You can use any other name; just make sure you set the correct name when registering the app.

How to do it…

1. Let's open the `index.ios.js` and `index.android.js` files, remove the previous code, and add the following:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('SendingData', () => MainApp);
```

2. Now we need to create the `src/MainApp.js` file, import the dependencies, and define three properties on the `state`:

```
import React, { PureComponent } from 'react';
import {
  Alert,
  ScrollView,
  StyleSheet,
  Text,
  TextInput,
  TouchableOpacity,
  View
} from 'react-native';

const endpoint =
'https://my-bookmarks-api.herokuapp.com/api/bookmarks';

class MainApp extends PureComponent {
  state = {
    result: '',
    title: '',
    url: '',
  };
  // Defined later
}

const styles = StyleSheet.create({
```

```
    // Defined later
});

export default MainApp;
```

3. After saving a new bookmark, we will request all the bookmarks from the API, just to make sure the new data is saved correctly. We are going to define an `onLoad` method to fetch the new data. This code is explained in the previous recipe:

```
onLoad = async () => {
  this.setState({ result: 'Loading, please wait...' });

  const response = await fetch(endpoint, {
    method: 'GET',
  });

  const result = await response.text();

  this.setState({ result });
}
```

4. Let's work on saving the new data. First, we need to get the values from the `state`. Additionally, we could run some validations to make sure the `title` and the `url` are not empty. On the `fetch` call, we need to define the content type of the request; in this case, we will send the data in a JSON format within the body containing all the information. For now, we will hard code the category ID, which is `1`, but ideally, we should get this value from the category API. After the response is completed, we get the JSON response, check if it was successful or not, and show an alert message:

```
onSave = async () => {
  const { title, url } = this.state;

  const response = await fetch(endpoint, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json;charset=UTF-8',
    },
    body: JSON.stringify({
      category_id: 1,
      title,
      url,
    }),
  });
```

```
          const result = await response.json();

          if (result.success === false) {
            Alert.alert('Error', 'There was an error while saving the
bookmark');
          } else {
            Alert.alert('Sucess', 'Bookmark successfully saved');
            this.onLoad();
          }
        }
```

5. The saving functionality is completed. Now we need the methods to save the `title` and `url` to the `state`. These methods will be executed as the user types in the input text to keep track of the values:

```
onTitleChange = (title) => this.setState({ title });
onUrlChange = (url) => this.setState({ url });
```

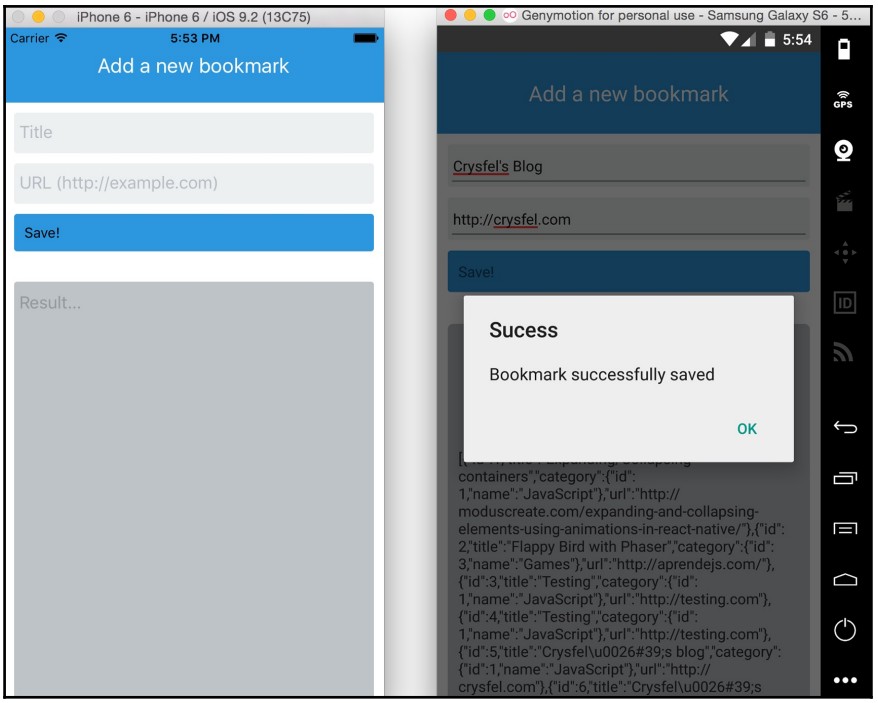
6. We have everything we need for the functionality, but the UI is missing. The `render` method will display a toolbar, three input texts, and a button:

```
render() {
  const { result, title, url } = this.state;

  return (
    <View style={styles.container}>
      <Text style={styles.toolbar}>Add a new bookmark</Text>
      <ScrollView style={styles.content}>
        <TextInput
          style={styles.input}
          onChangeText={this.onTitleChange}
          value={title}
          placeholder="Title"
        />
        <TextInput
          style={styles.input}
          onChangeText={this.onUrlChange}
          value={url}
          placeholder="URL (http://example.com)"
        />
        <TouchableOpacity onPress={this.onSave} style={styles.btn}>
          <Text>Save!</Text>
        </TouchableOpacity>
        <TextInput
          style={styles.preview}
          value={result}
          placeholder="Result..."
```

```
            editable={false}
            multiline
          />
        </ScrollView>
      </View>
    );
  }
```

7. Finally, let's add the styles. Nothing really complicated here–just some colors, margins, paddings, sizing, just the usual, actually:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
  },
  toolbar: {
    backgroundColor: '#3498db',
    color: '#fff',
    textAlign: 'center',
    padding: 25,
    fontSize: 20,
  },
  content: {
    flex: 1,
    padding: 10,
  },
  preview: {
    backgroundColor: '#bdc3c7',
    flex: 1,
    height: 500,
  },
  input: {
    backgroundColor: '#ecf0f1',
    borderRadius: 3,
    height: 40,
    padding: 5,
    marginBottom: 10,
    flex: 1,
  },
  btn: {
    backgroundColor: '#3498db',
    padding: 10,
    borderRadius: 3,
    marginBottom: 30,
  },
});
```



How it works…

In step 2, we are importing all the dependencies and defining three properties on the `state`.

The `result` will contain the response from the server API; we will use it to display the value on the UI.

We will use the `title` and `url` properties to hold the values from the input text components; this will allow the user to define a new bookmark. We will send these values to the API when pressing the **Save** button.

In step *6*, we are declaring the elements on the UI. We are going to use two input text to get the data for the new bookmark, which the user will be able to type. We will also use a button to save the data by calling the `onSave` method when pressed. Finally, we will use an input text to display the result, but the user won't be allowed to type anything in this field.

Establishing real-time communications with WebSockets

In this recipe, we will integrate WebSockets in a React Native application. We are going to create a simple chat app, where we will allow clients to send and receive messages to each other.

Getting ready

In order to support WebSockets on React Native, we need to have a server to handle all the clients that will be connected. The server should be able to broadcast a message when it receives a message from any of the clients.

You can download the code for this recipe; it will contain a `server` folder with the code needed to listen and send messages to clients. We need to have the server running before writing any React Native code.

To keep things as simple as possible, we will run our server with Node.js 6.x. We will only use the `ws` package from npm. If you have downloaded the code for this recipe, you will only need to run the following commands from inside of the `server` folder:

```
$ npm install --save ws
$ npm start
```

That's it. You will have a simple WebSocket server up and running on port `3001`. The code for the server is actually really simple. In case you want to do it by yourself, here's the minimal code to create the server. Don't forget to install the `ws` package:

```
const port = 3001;
const WebSocketServer = require('ws').Server;
const wss = new WebSocketServer({ port });
wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    console.log('received: %s', message);

    wss.clients.forEach(client => {
      if (client !== ws) {
        client.send(message);
      }
    });
  });
});
```

```
console.log(`Web Socket Server running on port ${port}`);
```

We also need to create an empty React Native app. For this recipe, we will name it `WebSockets`, but feel free to use any other name.

How to do it…

1. Once we have the WebSocket server running, we need to open the `index.ios.js` and `index.android.js` files, remove the existing code, and add the following:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('WebSockets', () => MainApp);
```

2. Now let's create the `src/MainApp.js` file, where we will have all the shared code for iOS and Android. Let's start by importing all the dependencies and defining the `state`:

```
import React, { Component } from 'react';
import {
  Dimensions,
  ScrollView,
  StyleSheet,
  Text,
  TextInput,
  View,
} from 'react-native';

const { width } = Dimensions.get('window');

class MainApp extends Component {
  state = {
    history: [],
  };
  // Defined at later steps
}

const styles = StyleSheet.create({
 // Defined at later steps
});

export default MainApp;
```

3. The first thing we need to do in order to integrate WebSockets into our app is to connect to the socket and set up the callback functions for receiving messages, errors, and when the connection is open or closed. We will do this when the component has been created:

```
componentWillMount() {
  this.ws = new WebSocket('ws://localhost:3001');

  this.ws.onopen = this.onOpenConnection;
  this.ws.onmessage = this.onMessageReceived;
  this.ws.onerror = this.onError;
  this.ws.onclose = this.onClose;
}
```

4. Let's define the callbacks for open/closed connections and when receiving errors. We are just going to log the actions, but in here we could show an alert message when the connection is closed and the user is still using the app, or display an alert error when an error is received:

```
onOpenConnection = () => {
  console.log('Open!');
}

onError = (event) => {
  console.log('onerror', event.message);
}

onClose = (event) => {
  console.log('onclose', event.code, event.reason);
}
```

5. When receiving a new message from the server, we need to add it to the `history` on the `state`. This way we will be able to render the new content as soon as it arrives:

```
onMessageReceived = (event) => {
  this.setState({
    history: [
      ...this.state.history,
      { owner: false, msg: event.data },
    ],
  });
}
```

6. Now let's send the message! We need to define a method that will get executed when the user presses the *Return* key on the keyboard. We need to do two things here. First add the new message to the history, and then send the message through the socket:

```
onSendMessage = () => {
  const { text } = this.state;

  this.setState({
    text: '',
    history: [
      ...this.state.history,
      { owner: true, msg: text },
    ],
  });
  this.ws.send(text);
}
```

7. In the previous step, we got the `text` property from the `state`. We need to keep track of the value whenever the user types something in the input. Therefore, we have to create a function to listen to the keystrokes and save the value to the `state`:

```
onChangeText = (text) => {
  this.setState({ text });
}
```

8. We have all the functionality in place. Let's work on the UI. On the `render` method, we will have a toolbar, a scroll view to render all the messages on the history, and an input text to allow the user to send a new message:

```
render() {
    const { history, text } = this.state;

    return (
      <View style={styles.container}>
        <Text style={styles.toolbar}>Simple Chat</Text>
        <ScrollView style={styles.content}>
          { history.map(this.renderMessage) }
        </ScrollView>
        <View style={styles.inputContainer}>
          <TextInput
            style={styles.input}
            value={text}
            onChangeText={this.onChangeText}
            onSubmitEditing={this.onSendMessage}
```

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

9. To render the history messages, we loop through the history array and render each message. In here, we need to check if the current message belongs to the user on this device. Based on this, we will apply different styles:

```
renderMessage(item, index){
  const kind = item.owner ? styles.me : styles.friend;

  return (
    <View style={[styles.msg, kind]} key={index}>
      <Text>{item.msg}</Text>
    </View>
  );
}
```

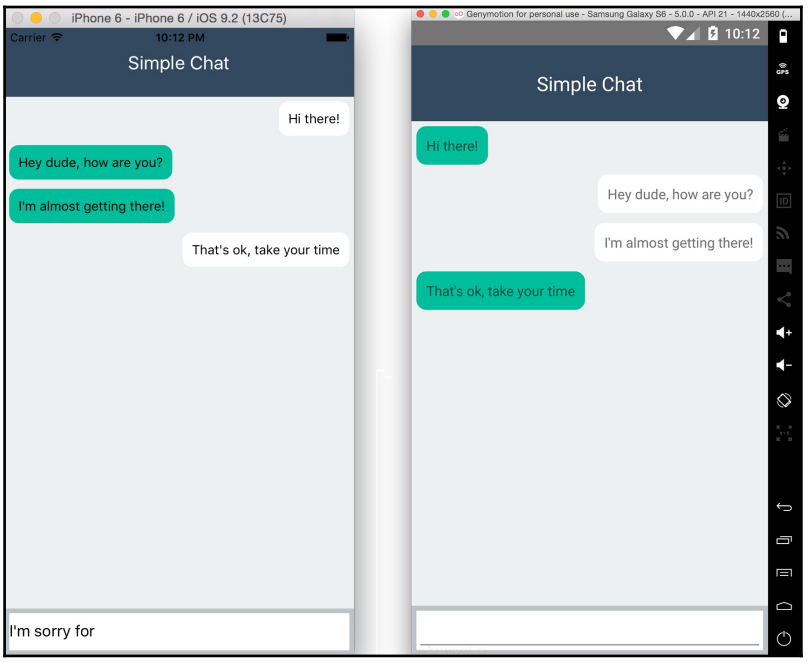
10. Finally, let's work on the styles! Let's add styles to the toolbar, the history component, and the text input. We need to set the history container as flexible because we want to take all the available vertical space.

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: '#ecf0f1',
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#34495e',
    color: '#fff',
    fontSize: 20,
    padding: 25,
    textAlign: 'center',
  },
  content: {
    flex: 1,
  },
  inputContainer: {
    backgroundColor: '#bdc3c7',
    padding: 5,
  },
  input: {
    height: 40,
    backgroundColor: '#fff',
  },
```

```
});
```

11. Now the styles for each message. We are going to create a common style's object called `msg`. Then we will create styles for my messages and styles for other people's messages. Only the color and alignment change:

```
msg: {
  margin: 5,
  padding: 10,
  borderRadius: 10,
},
me: {
  alignSelf: 'flex-start',
  backgroundColor: '#1abc9c',
  marginRight: 100,
},
friend: {
  alignSelf: 'flex-end',
  backgroundColor: '#fff',
  marginLeft: 100,
},
```



How it works…

In step *2*, we are importing the dependencies and declaring the `state`. We are also getting the width of the device because we are going to use this value for the styles. The history property will have objects representing each message on the history. Each object will have two properties only–a string with the actual message and a Boolean flag to figure out whether this message was sent by the current user.

We could have more data here, such as the name of the user, a URL of the avatar image, or anything else we need.

In step *3*, we are connecting to the socket and setting up the callbacks. We need to specify the server address as well as the port; in this case, the address is `localhost` and the port is `3001`.

In step *5*, we are defining the callback to execute when receiving a new message from the server. We are using an `arrow` function because we need to set the scope of the function to the current component. We are also adding a new object to the history array on the `state`. The new object represents the new message. It just has two properties–`owner`, and `msg`.

In step *6*, we are sending the message to the server. We need to add the message to the history because the server will broadcast the message to all other clients, but not the author of the message. In order to keep track of this message, we need to manually add it to the history.

There's more…

In order to run the app on Genymotion to test the Android version, we need to forward the port to successfully connect to the socket. Just run the following command on the Terminal–make sure the `$ANDROID_HOME` has the correct reference to the Android SDK:

```
$ANDROID_HOME/platform-tools/adb reverse tcp:3001 tcp:3001
```

Integrate persistent database functionality with Realm

As your application becomes more and more complex, you will come to the point where you need to store data on the device. This could be business data, such as user lists, to avoid having to make expensive network connections to a Remote API. Maybe you don't have an

API at all and your application works as a self-sufficient entity. Regardless of the situation, you may benefit from leveraging a database to store your data. There are several options on the React Native front:

- You can use `AsyncStorage` (we cover this in the *Storing and retrieving data locally* recipe)
- SQLite, or you can write an adapter to an OS-specific data provider, such as CoreData, or use a mobile database such as Realm

Realm is an extremely fast, thread-safe, transactional, object-based database. It is primarily designed for use by mobile devices, with a straightforward JavaScript API. It supports other features, such as encryption, complex querying, UI bindings, and more. You can read all about it at `https://realm.io/products/realm-mobile-database/`.

In this recipe, we will walk through using Realm in React Native. We will create a simple database, and perform basic operations, such as inserting records, updating, and deleting. Most importantly, we will display all the records of our schema on the UI.

Getting ready

For this recipe, we will use an empty React Native application named `RealmApp`. For our sample UI, we used `react-native-button`, for buttons to trigger database operations. To install `react-native-button` execute the following command in the Terminal:

```
$ npm install --save react-native-button
```

How to do it…

1. Open your Terminal and go into your React Native project directory. Execute the following commands:

```
$ npm install --save realm
$ react-native link realm
```

2. Open `index.ios.js` and add the following imports:

```
import Realm from 'realm';
import Button from 'react-native-button';
```

3. Next, we need to instantiate our Realm database and keep a reference to it in our class. Add the following code to the class:

```
class RealmApp extends Component {
  realm : undefined

  componentWillMount() {
    const realm = this.realm = new Realm({
      schema : [
        {
          name : 'User',
          properties : {
            firstName : 'string',
            lastName : 'string',
            email : 'string'
          }
        }
      ]
    });
    this.setState({users:realm.objects('User')});
  }
```

4. To create the `User` entries, we will use the random user generator API (`randomuser.me`). Let's create a method that will `fetch` the data:

```
getRandomUser() {
  return
fetch('https://randomuser.me/api/').then((response)=>response.json());
}
```

5. We need some functionality to create `User` objects. Add the following function:

```
createUser = () => {
  const realm = this.realm;

  this.getRandomUser().then((response) => {
    const user = response.results[0];
    const userName = user.name;
    realm.write(() => {
      realm.create('User', {
        firstName : userName.first,
        lastName  : userName.last,
        email     : user.email
      });
      this.setState({users:realm.objects('User')});
    });
  });
```

```
}
```

6. Next, we should add a way to update an object in the database. Let's put in some code to take the first record in the collection and change its information:

```
updateUser = () => {
  const realm = this.realm;
  const users = realm.objects('User');

  realm.write(() => {
    if(users.length) {
      let firstUser = users.slice(0,1)[0];
      firstUser.firstName = 'Bob';
      firstUser.lastName = 'Cookbook';
      firstUser.email = 'react.native@cookbook.com';
      this.setState(users);
    }
  });
}
```

7. Finally, let's say we need to reset the entire dataset. We need a way to delete everything in the database:

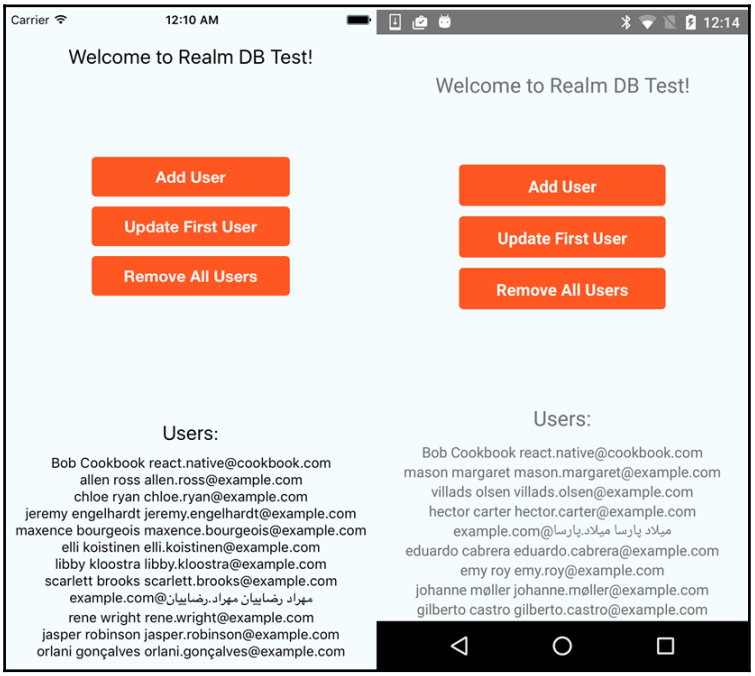
```
deleteUsers = () => {
  const realm = this.realm;
  realm.write(() => {
    realm.deleteAll();
    this.setState({users:realm.objects('User')});
  });
}
```

8. Let's build our UI, which will render a list of `User` objects, and buttons to create, update, and delete. Implement your `render` function with this:

```
const realm = this.realm;
return (
  <View style={styles.container}>
    <Text style={styles.welcome}>
      Welcome to Realm DB Test!
    </Text>
    <View style={styles.container}>
       <Button
       onPress={this.createUser}>
        Add User
      </Button>
       <Button
        onPress={this.updateUser}>
```

```
          Update First User
        </Button>
         <Button
          onPress={this.deleteUsers}>
          Remove All Users
        </Button>
      </View>
      <View style={styles.container}>
      <Text style={styles.welcome}>Users:</Text>
      {this.state.users.map((user, idx) => {
        return <Text key={idx}>{user.firstName} {user.lastName}
       {user.email}</Text>;
      })}
      </View>
    </View>
  );
```

9. For Android, open `index.android.js` and repeat steps *2*–*8*.
10. Running the application on either platform should look and function something like these screenshots of the sample app:



How it works…

The Realm database is built in C++; its core is known as the **Realm Object Store**. There are products that encapsulate this object store for each major platform (Java, Objective-C, Swift, Xamarin, and as we know from our case, React Native). The React Native product is a JavaScript adapter for Realm. From the React Native side, we do not need to worry about the implementation; rather we get a very clean API to persist and retrieve data. Steps *5*, *6*, and *7* show some basic Realm calls. If you would like to see what you can do with the API, please refer to the documentation found at `https://realm.io/docs/react-native/latest/api/`.

Mask the application upon network connection loss

Internet connection is not always available, especially when people are moving around the city, on the train, or hiking in the mountains. A good user experience will inform the user when the connection is lost.

In this recipe, we will create an app that shows a message when a network connection is lost.

Getting ready

We need to create an empty app named `MaskNetwork`. If you use a different name, make sure you use the same when registering the app.

For Android, add the following permission to your `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

How to do it…

1. Let's open the `index.ios.js` and `index.android.js` files, remove the existing code, and add the following:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('MaskNetwork', () => MainApp);
```

2. We will have the common code for iOS and Android on `src/MainApp.js`. Let's start by importing the dependencies and adding two properties to the `state`:

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

class MainApp extends PureComponent {
  state = {
    online: null,
    offline: null,
  };
  // Defined at later steps
}

const styles = StyleSheet.create({
  // Defined at later steps
});

export default MainApp;
```

3. When the component gets created, we need to find out the initial network status. We are going to use the `NetInfo` class to get the current status, and we also need to set up a callback when the status changes:

```
componentWillMount() {
  NetInfo.fetch().done(reach => {
    this.onConnectivityChange(reach);
  });

  NetInfo.addEventListener('change', this.onConnectivityChange);
```

```
}
```

4. When the component is about to get destroyed, we need to remove the listener:

```
componentWillUnmount() {
  NetInfo.removeEventListener('change', this.onConnectivityChange);
}
```

5. The callback that will get executed when the network status changes, is only going to check whether the current network type is `none`, and it will set the `state` to `true` or `false`:

```
onConnectivityChange = reach => {
  const type = reach.toLowerCase();
  this.setState({
    online: type !== 'none',
    offline: type === 'none',
  });
}
```

6. Now we know when the network is on or off, but the UI is missing. Let's just render a toolbar with some dummy text as the content:

```
render() {
  return (
    <View style={styles.container}>
      <Text style={styles.toolbar}>My Awesome App</Text>
      <Text style={styles.text}>Lorem...</Text>
      <Text style={styles.text}>Lorem ipsum...</Text>
      {this.renderMask()}
    </View>
  );
}
```

7. As you can see from the previous step, there's a `renderMask` function. This function will return a message when the network is offline or nothing if it's online:

```
renderMask() {
  if (this.state.offline) {
    return (
      <View style={styles.mask}>
        <View style={styles.msg}>
          <Text style={styles.alert}>Seems like you do not have
           network connection anymore.</Text>
          <Text style={styles.alert}>You can still continue
          using the app, with limited content.</Text>
```

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

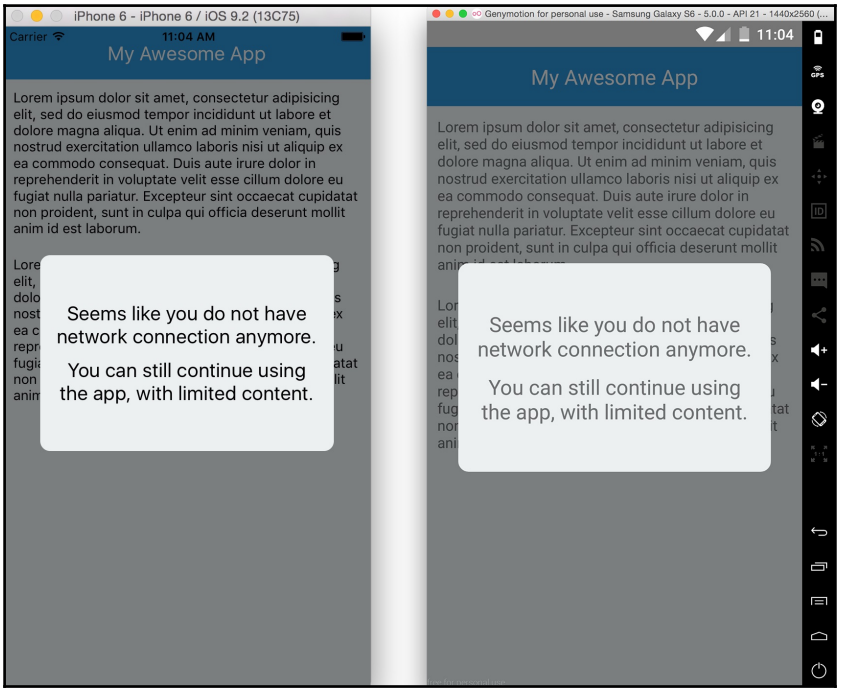
8. Finally, let's add the styles for our app. Something really simple–just styles for the toolbar and the content:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  toolbar: {
    backgroundColor: '#3498db',
    padding: 15,
    fontSize: 20,
    color: '#fff',
    textAlign: 'center',
  },
  text: {
    padding: 10,
  },
});
```

9. For the message, we will render a dark mask before all content and a container with the text at the center of the screen. For the mask, we need to set the position to absolute, set the top, bottom, right, and left to zero, add opacity to the background color, and justify and align the content to the center:

```
mask: {
  alignItems: 'center',
  backgroundColor: 'rgba(0, 0, 0, 0.5)',
  bottom: 0,
  justifyContent: 'center',
  left: 0,
  position: 'absolute',
  top: 0,
  right: 0,
},
msg: {
  backgroundColor: '#ecf0f1',
  borderRadius: 10,
  height: 200,
  justifyContent: 'center',
  padding: 10,
```

```
    width: 300,
  },
  alert: {
    fontSize: 20,
    textAlign: 'center',
    margin: 5,
  },
```



How it works…

In step 2, we are importing the dependencies and creating the initial state properties. We only declared two properties–`online` will be `true` when a network connection is available and `offline` will be `true` when it's not available.

In step 3, we are getting the initial network status and setting up a listener to check when the status changes. The network type that we will receive should be `Wi-Fi`, `cell`, `unknown`, and `none` for iOS, and `WIFI`, `MOBILE`, `NONE`, and a few others on Android. Make sure to read the documentation to see all the available values.

In step 5, when listening to network status changes, we need to lower case the network type. This is because on Android, the value is upper case, and in iOS, it is lowercase. After making sure that we will have the same value on both platforms, we need only check if there's no connectivity to set the value of `online` and `offline`.

There's more…

In `Chapter 5`, *Implementing Redux*, we take this approach to detect the network connection using Redux.

Synchronizing locally persisted data with a Remote API

When working with a mobile app, network connectivity is something that is often taken for granted. Sure, year after year great strides in availability and bandwidth are made, but everyone still hits a point where they only have one bar and the picture they just sent over iMessage is timing out. What if your application needs to make an API call for something in your application, but you just lost connectivity? Fortunately for us, React Native has a module that reacts to the network connectivity status. We can architect our application in a way that supports the loss of connectivity, and then synchronizes our data by completing an API call when the network connection restores.

This recipe will show a simple implementation of using the `NetInfo` module to control whether or not our application will make an API call. If the connectivity is lost, we will keep a reference of the pending request and complete it when the network access is restored. We will be using a sample POST API provided by `posttestserver.com`. This API accepts `POST` requests and writes the contents out to a file on the web server that you can later access.

Getting ready

For this recipe, we will use an empty React Native application named `SyncData`.

How to do it…

1. Open `index.ios.js` in your IDE.
2. First, we're going to need to import the `NetInfo` module, as well as `TouchableOpacity` for our **Submit** button.

```
import {
  //...
  NetInfo,
  TouchableOpacity
} from 'react-native';
```

3. Add an instance variable `pendingSync` to the class that will store the pending request.

```
class SyncData extends Component {
  pendingSync : undefined
```

4. Next, let's set up our state and, more importantly, get the current connection status and attach a listener to monitor the changes in connectivity:

```
componentWillMount() {
  this.setState({
    isConnected : undefined,
    syncStatus : undefined,
    serverResponse : undefined
  });

  NetInfo.isConnected.fetch().then(isConnected => {
    this.setState({isConnected});
  });
  NetInfo.isConnected.addEventListener('change',
this.onConnectedChange);
}
```

5. Now we need to implement our event listener:

```
onConnectedChange = (isConnected) => {
  var pendingSync = this.pendingSync;
  this.setState({isConnected});
  if(pendingSync) {
    this.setState({syncStatus : 'Syncing'});
    this.submitData(pendingSync).then(() => {
      this.setState({syncStatus : 'Sync Complete'});
    });
  }
```

```
}
```

6. Next, we need to implement the API call that will be performed when there is an active network connection:

```
submitData(bodyData) {
  return fetch('https://posttestserver.com/post.php', {
    method : 'POST',
    body : JSON.stringify(bodyData)
  }).then((response) => {
    return response.text();
  }).then((responseText) => {
    this.setState({
      serverResponse : responseText
    });
  });
}
```

7. The last thing we need to do before we work on our UI is create a handler for the **Submit Data** button we will create. This will either perform the call immediately or mark it as pending if there is no network connection:

```
onSubmitPress = () => {
  var isConnected = this.state.isConnected,
      submitBody = {
        name : 'React Native Cookbook',
        timestamp : Date.now()
      };
  if(isConnected) {
    this.submitData(submitBody);
  } else {
    this.pendingSync = submitBody;
    this.setState({syncStatus : 'Pending'});
  }
}
```

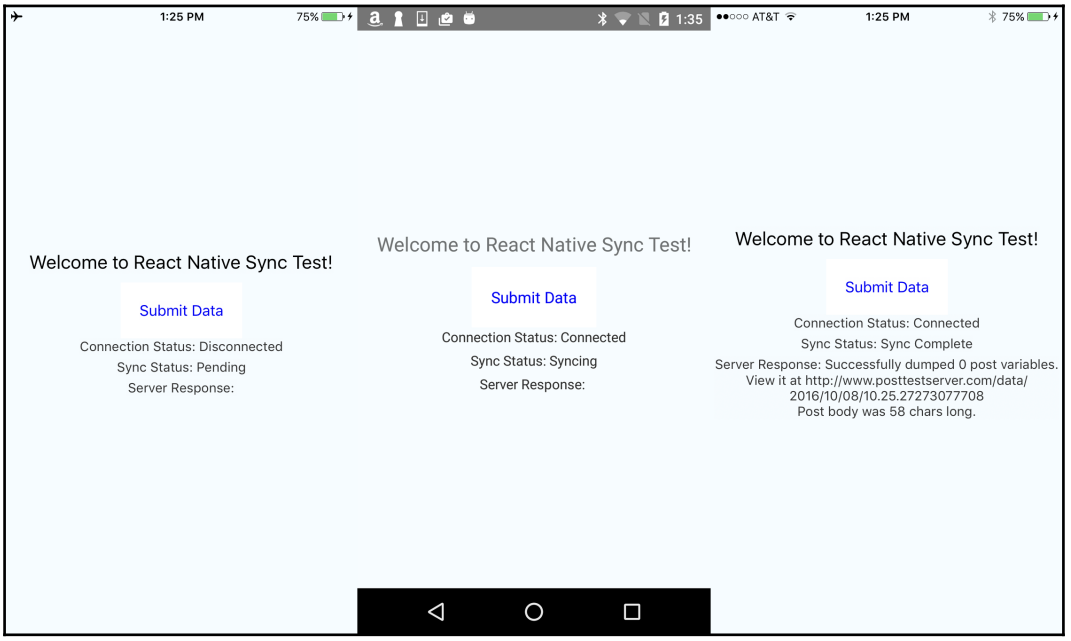
8. Now we can stub out our simple UI that will show us the connection status, sync status, and the response from the API. Add the following to your `render` method:

```
        const {
          isConnected,
          syncStatus,
          serverResponse
        } = this.state;
        return (
          <View style={styles.container}>
            <TouchableOpacity onPress={this.onSubmitPress}>
              <View style={styles.button}>
                <Text style={styles.buttonText}>Submit Data</Text>
              </View>
            </TouchableOpacity>
            <Text style={styles.instructions}>
              Connection Status: {isConnected ? 'Connected' :
'Disconnected'}
            </Text>
            <Text style={styles.instructions}>
              Sync Status: {syncStatus}
            </Text>
            <Text style={styles.instructions}>
              Server Response: {serverResponse}
            </Text>
          </View>
        );
```

9. For Android, open `index.android.js` and repeat steps *2–8*.
10. Open `AndroidManifest.xml` in `./android/app/src/main/` and add the following line:

```
        <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```

11. When you run the app (we recommend running on a device for accuracy), you should see something like this for the sync process:



How it works…

This recipe leverages the `NetInfo` module to control whether or not we can make an HTTP request. If there is no connectivity, we place the arguments that we would send to the API in a class variable. This logic is carried out in step *7*. The two `NetInfo` method calls in step *4* retrieve the current network connection status and attach an event listener to the change event. In step *5*, once the connectivity changes to a positive state, we check to see if there is a pending API call and complete the request. This recipe can be expanded on to support a queue system of pending calls, which we would see in a more realistic scenario.

Logging in with Facebook

Facebook is the largest social media platform, with well over 1 billion users worldwide. There is a pretty good chance that your user will have a Facebook account, and would want to maybe even log in using their Facebook credentials. Your app can register and link with their account. This will allow you to access data, such as their user information, pictures, and even the ability to share content. Not only has Facebook given us this wonderful framework of React Native, they also gave us a library to help facilitate connecting to the Facebook SDK.

In this recipe, we will cover the first two bits of functionality–logging in and getting basic user information from a Facebook account. We will use the Facebook iOS and Android SDK wrapper for React Native, and present the login status, user ID, and username on our sample UI.

Getting ready

For this recipe, we created a React Native application titled `FbLogin`. You will need to have an active Facebook account to test the functionality.

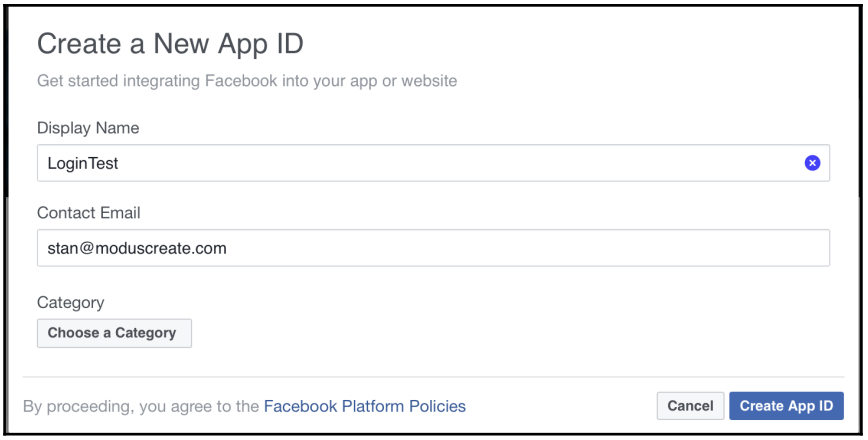
How to do it…

1. Open your Terminal and go to your React Native project. Execute the following command:

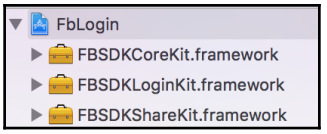
```
$ react-native install react-native-fbsdk
```

2. Now we need to set up our iOS project. Please follow the **Quick Start Guide**. The next several steps will be used as a companion to the guide at `https://developers.facebook.com/quickstarts/?platform=ios`.

3. Create a new app ID, and fill out the required information. Please note that you cannot use any Facebook-related naming in your name. In our example we use `LoginTest`:

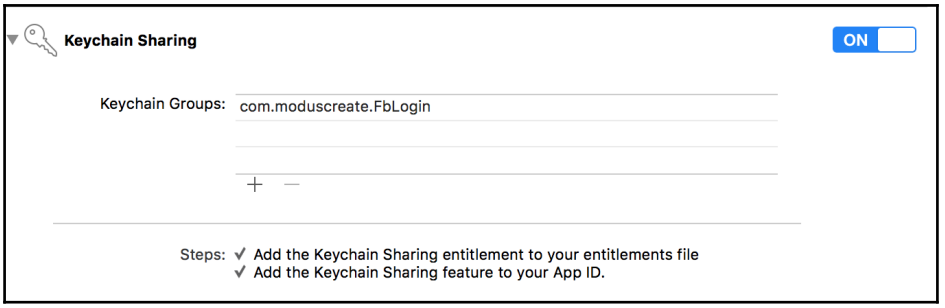


4. Make sure you save your SDK files into `~/Documents/FacebookSDK`.
5. Open Xcode and drag the `.framework` files to your project:

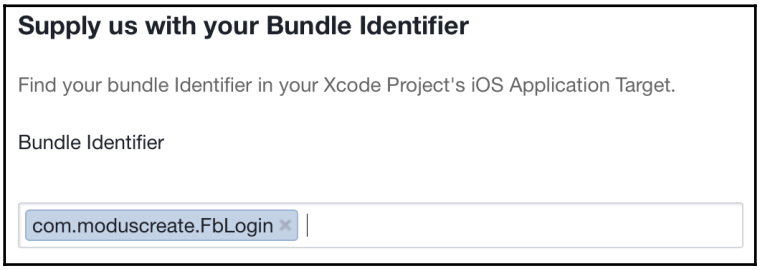


6. Open your project's **Build Settings** and add `~/Documents/FacebookSDK/` to the **Framework Search Paths**.

7. In the **Capabilities** section of your project, enable **Keychain Sharing:**



8. Complete all the changes required for the `Info.plist`, including the `LSApplicationQueriesSchemes`.
9. Back in your browser, the **Quick Start Guide** will ask for your **Bundle Identifier**. It is found in your project's **General Settings**. These identifiers must match.



10. Switch back to Xcode and open `AppDelegate.m`. Add the following code:

```
        #import <FBSDKCoreKit/FBSDKCoreKit.h>
        - (BOOL)application:(UIApplication *)app
                    openURL:(NSURL *)url

options:(NSDictionary<UIApplicationOpenURLOptionsKey,id> *)options {

          BOOL handled = [[FBSDKApplicationDelegate sharedInstance]
          application:application
            openURL:url

sourceApplication:options[UIApplicationOpenURLOptionsSourceApplicationKey]
            annotation:options[UIApplicationOpenURLOptionsAnnotationKey]
          ];
          return handled;
        }
```

11. In your `didFinishingLaunchingWithOptions` method, add the following line:

```
[[FBSDKApplicationDelegate sharedInstance] application:application
didFinishLaunchingWithOptions:launchOptions];
```

12. We're all set for the iOS setup. Now let's add a Facebook login button to our UI. Open `index.ios.js` and add the following `import` and variable declaration:

```
import FBSDK from 'react-native-fbsdk';

const {
  LoginButton,
  AccessToken,
  GraphRequest,
  GraphRequestManager
} = FBSDK;
```

13. Let's start with our `render` method. Add the following JSX:

```
<LoginButton
  publishPermissions={["publish_actions"]}
  onLoginFinished={this.onLoginFinished}
  onLogoutFinished={this.onLogoutFinished}/>
<View>
  <Text>Login Status:</Text>
  <Text>{this.state.status}</Text>
  <Text>User Name:</Text>
  <Text>{this.state.userName}</Text>
</View>
```

14. Next, we need to implement the callbacks for the `LoginButton`, as well as define the initial `state` for our view:

```
componentWillMount() {
    this.setState({status : undefined, userName : undefined});
}

onLoginFinished = (error, result) => {
  var status;
  if (error) {
    status = `Error: ${error.message}`;
  } else if (result.isCancelled) {
    status = 'Login Cancelled';
  } else {
    AccessToken.getCurrentAccessToken().then(
      (data) => {
        status = `Success! UserID: ${data.userID}`;
```

```
            this.setState({status});
            this.getUserInfo(data);
          }
        )
      }
      this.setState({status});
    }

    onLogoutFinished = () => {
      this.setState({status : 'Logged Out'});
    }
```

15. We need to make our call to get the user profile using the Graph API:

```
    getUserInfo(accessToken) {
      const infoRequest = new
GraphRequest('/me',null,this.onGraphResponse);
      new GraphRequestManager().addRequest(infoRequest).start();
    }

    onGraphResponse = (error, result) => {
      this.setState({userName : result.name});
    }
```

16. For Android functionality, we're going to work a bit backward. Repeat steps *12–15* for `index.android.js`.
17. Open your Android project in Android Studio. Open `MainApplication.java` and add the following `private` field:

```
private static CallbackManager mCallbackManager =
 CallbackManager.Factory.create();
```

18. Next, add the following methods:

```
public void onCreate() {
     super.onCreate();
     FacebookSdk.sdkInitialize(getApplicationContext());
 }
 protected static CallbackManager getCallbackManager() {
     return mCallbackManager;
 }
```

19. Finally, we need to register the `FBSDKPackage`:

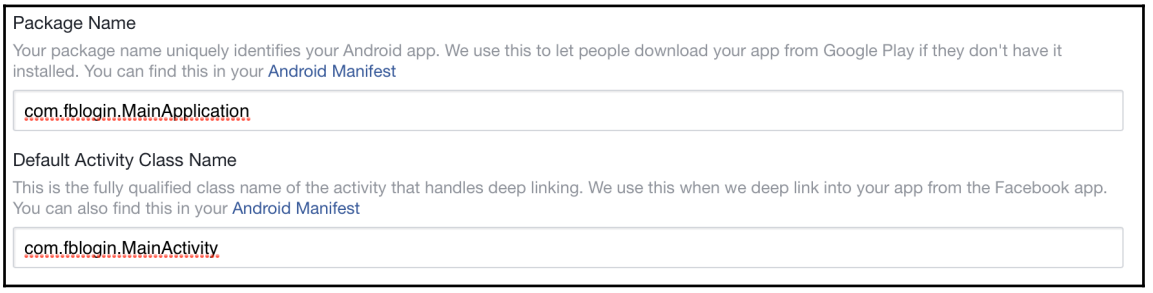
```
protected List<ReactPackage> getPackages() {
     return Arrays.<ReactPackage>asList(
             new MainReactPackage(),
```

```
                    new FBSDKPackage(mCallbackManager)
            );
        }
```

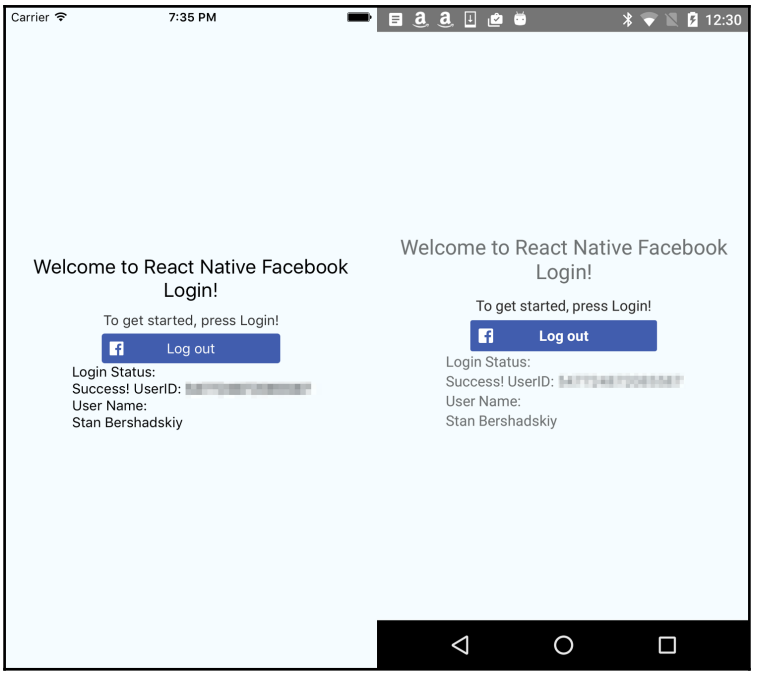
20. Open `MainActivity.java` and add the following method:

```
       @Override public void onActivityResult(int requestCode, int
resultCode,
         Intent data) {
            super.onActivityResult(requestCode, resultCode, data);

MainApplication.getCallbackManager().onActivityResult(requestCode,
            resultCode, data);
        }
```

21. Follow the **Quick Start Guide** for Android to register the app ID and key hashes for our application at `https://developers.facebook.com/quickstarts/?platform=android`.
22. You will be asked for the **Package Name** and **Default Activity Class Name**. Enter the following:



23. If you are prompted that your application is not in the Google Play store, press **Use this package name**.
24. We're all set. You can now run the application on your device or simulator. See our sample UI:



How it works…

The React Native FBSDK module is a wrapper around the native iOS and Android Facebook SDKs. It provides a common JavaScript API for interfacing with the native Facebook SDK components, such as `LoginButton`, in our case. Since it is a collection of native modules (see *Exposing custom iOS modules* and *Exposing custom Android modules* recipes in `Chapter 6`, *Adding Native Functionality* for details on how these work), we still have to go through the legwork of bootstrapping the individual SDKs on each platform. Once the setup is complete, the JavaScript calls are piped through the native layer identically on both platforms.

Sharing content on Facebook

Social media is an incredible platform to spread a message, image, or any sort of content out to the world. Facebook is the de-facto leader in social media, with the furthest reach. We live in a day and age where a celebrity's post on Facebook will generate worldwide news coverage. How awesome would it be if you had a user who has a large social media following share something on Facebook using your application?

In this recipe, we will use the Facebook SDK for React Native to share some content using our application. We will have a button that will show the Facebook share dialog and link to the Packt website.

Getting ready

For this recipe, we created a React Native application titled `FbLogin`. Your application should have the Facebook SDK integrated. If you need assistance in doing so, please follow the *Logging in with Facebook* recipe.

How to do it…

1. Open `index.ios.js` in your favorite IDE and add the following import and variable declaration:

```
import FBSDK from 'react-native-fbsdk';

const {
  ShareDialog
} = FBSDK;
```

2. Add the following to your `render` method to render the button and the status of our share process:

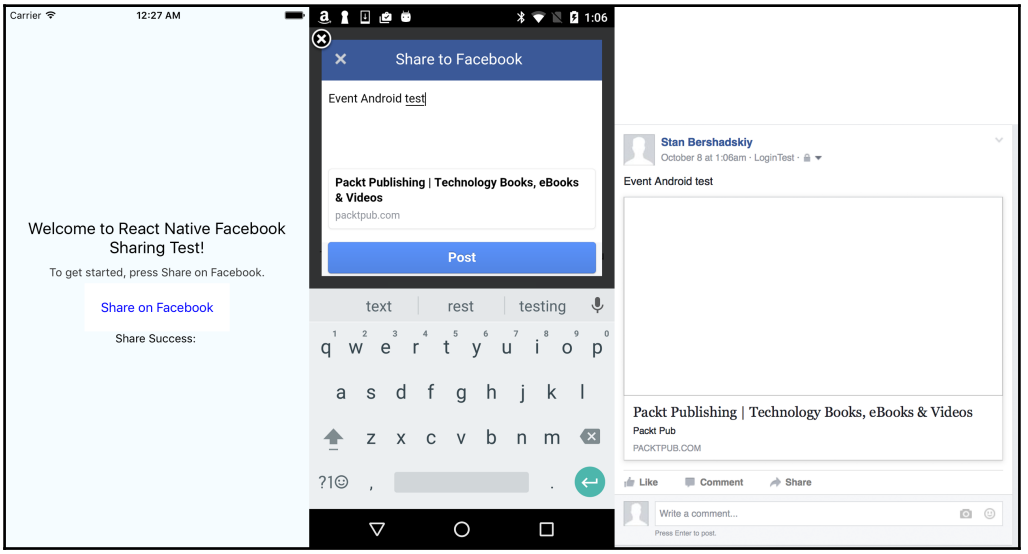
```
<TouchableOpacity onPress={this.onShareBtnPressed}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Share on Facebook</Text>
  </View>
</TouchableOpacity>
<Text>Share Success: {this.state.shareSuccess}</Text>
```

3. Next up, we need to define our initial state and implement our button callback:

```
componentWillMount() {
    this.setState({shareSuccess : undefined});
}

 onShareBtnPressed = () => {
  var shareSuccess;
  const shareLinkContent = {
    contentType: 'link',
    contentUrl: "https://packtpub.com",
```

```
      contentDescription: 'Packt Pub',
    };

    ShareDialog.canShow(shareLinkContent).then((canShow) => {
      if(canShow) {
        return ShareDialog.show(shareLinkContent);
      }
    }).then((result) => {
      if(result.isCancelled) {
        shareSuccess = 'Cancelled';
      } else {
        shareSuccess = 'Shared Successfully';
      }
      this.setState({shareSuccess});
    }, (error) => {
      shareSuccess = 'Error';
      this.setState({shareSuccess});
    });
  }
```

4. For Android, please replicate steps *1*–*3* for `index.android.js`.
5. We can now run our sample app and try sharing some content:



How it works…

This recipe uses the native Facebook share dialog. We pass in the link of Packt's homepage as the content to share. An interesting bit to note is that you do not need an explicit login call. If your app is already authenticated and linked with Facebook, it will handle the login for you. If you are logged out, it will prompt you to log in through a *Web View*.

Tracking application events with Facebook Analytics

Keeping track of what users do while active in your application is an important bit of functionality to have. Metrics and analytics will help you understand which areas of your application may need more focus to develop and enhance. Facebook provides an entire platform for this to use in app development named **Facebook Analytics**. We can register our application using the Analytics dashboard to track engagements and analyze important metrics, such as sharing, purchases, and any custom events you may define.

This recipe will hook our sample application into Facebook Analytics and begin firing launch events, as well as a few custom events when we share information.

Getting ready

For this recipe, we created a React Native application titled `FbLogin`. Your application should have the Facebook SDK integrated. If you need assistance in doing so, please follow the *Logging in with Facebook* recipe. The JavaScript code we will write in this recipe will be very similar to the work we did in the *Sharing content on Facebook* recipe.

How to do it…

1. Open your React Native iOS project in Xcode. Open up `AppDelegate.m` and add the following meeting:

```
-(void)applicationDidBecomeActive:(UIApplication *)application {
  [FBSDKAppEvents activateApp];
}
```

2. Now switch to your JavaScript IDE and open `index.ios.js`. Let's add our imports and module references:

```
import FBSDK from 'react-native-fbsdk';

const {
  AppEventsLogger,
  ShareDialog
} = FBSDK;
```

3. Add the following to your `render` method to render the button and the status of our share process:

```
<TouchableOpacity onPress={this.onShareBtnPressed}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Share on Facebook</Text>
  </View>
</TouchableOpacity>
<Text>Share Success: {this.state.shareSuccess}</Text>
```

4. Next up, we need to define our initial state and implement our button callback that will show the `ShareDialog` and track its progress with custom app events:

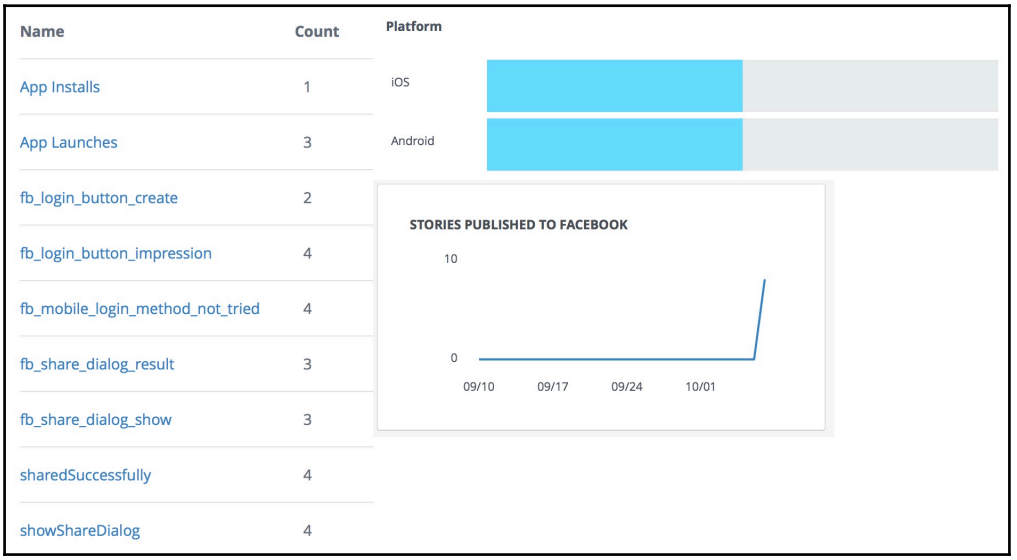
```
componentWillMount() {
    this.setState({shareSuccess : undefined});
}

onShareBtnPressed = () => {
  var shareSuccess;
  AppEventsLogger.logEvent('showShareDialog');
  const shareLinkContent = {
    contentType: 'link',
    contentUrl: "https://packtpub.com",
    contentDescription: 'Packt Pub',
  };
  ShareDialog.canShow(shareLinkContent).then((canShow) => {
    if(canShow) {
      return ShareDialog.show(shareLinkContent);
    }
  }).then((result) => {
    if(result.isCancelled) {
      shareSuccess = 'Cancelled';
      AppEventsLogger.logEvent('shareCancelled');
    } else {
      shareSuccess = 'Shared Successfully';
      AppEventsLogger.logEvent('sharedSuccessfully');
    }
```

```
      this.setState({shareSuccess});
    }, (error) => {
      shareSuccess = 'Error';
      AppEventsLogger.logEvent('shareError');
      this.setState({shareSuccess});
    });
  }
```

5. Now let's set up our Analytics for apps on Android. Open the project in Android Studio and open `MainApplication.java`.
6. Add the following in the `onCreate` method:

```
AppEventsLogger.activateApp(this);
```

7. Next, open `index.android.js` in your JavaScript IDE and repeat steps *2–4*.
8. You can go ahead and run the application, and share some content.
9. The next day (this process takes time), visit the **Facebook Analytics for Apps** dashboard at `https://www.facebook.com/analytics/`.
10. You will be presented with a list of applications. Click on your application name (in our case it would be `LoginTest`).
11. Here you will be presented with various metrics that have been tracked:



How it works…

This recipe builds on the *Sharing content on Facebook* recipe, but interjects custom app events that are tracked in the **Facebook Analytics for Apps** dashboard. When we begin tracking activations, as done in step *1* and *6*, we are provided with a lot more information about the users. These events are transmitted through the native Facebook SDK with metadata about the session and user. If the user is logged in through the Facebook SDK and has lax privacy settings, we would be able to see information such as their location, gender, and other demographic information. There would have to be over 1,000 users at least 18 years of age for results to be shown.

5
Implementing Redux

In this chapter, we will go step by step through the process of adding Redux to our app. We will cover the following recipes:

- Installing Redux and preparing our project
- Defining actions
- Defining reducers
- Setting up the store
- Communicating with a Remote API
- Connecting the store with the views
- Storing offline content using Redux
- Showing network connectivity status

Introduction

At some point in our application, we will need a better way to handle the state of our application and sending data across several components, and, overall, a robust architecture to scale our application.

In simple words, Redux is a predictable state container. We can use Redux to handle the global state of our app, which will allow us to send data to our components easily. All our data will be in a single place, and from there, we can send it to any component or view that we need to.

In order to have a better understanding of Redux, we will create an app through all the recipes. All the recipes in this chapter will depend on each other.

We will be building an app to display bookmarks, something really simple. We are going to use a `ListView` component to display categories from an API deployed on Heroku.

Installing Redux and preparing our project

In this recipe, we will install Redux in an empty application, and we will define the main folder structure of our bookmarks application.

How to do it…

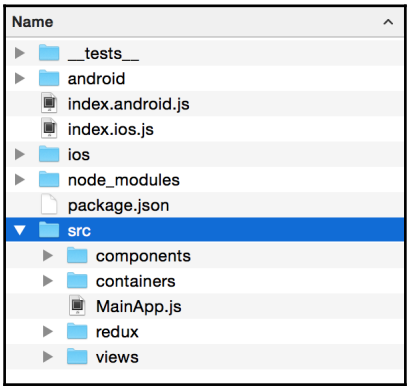
1. We need to create our application using the React Native CLI. By running the following command, we will have our initial app:

```
$ react-native init Bookmarks
$ cd Bookmarks
```

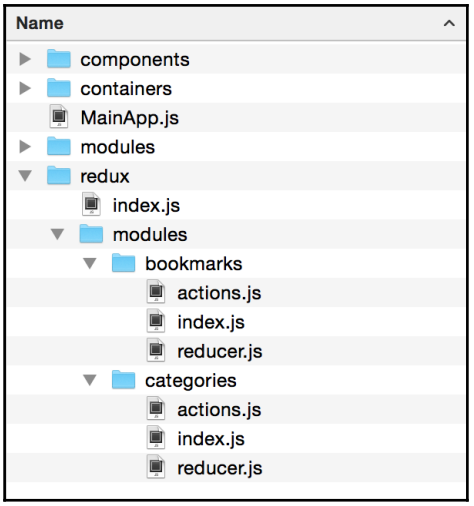
2. Once we have the initial app, we need to install Redux; we can do that by using `npm` or `yarn`. Feel free to use your favorite package manager:

```
$ npm i --save redux
$ npm i --save react-redux
```

3. Now let's create the folder structure for our application. First, we need to create an `src` folder where we will have all our source code. Inside `src`, we will have `components`, `containers`, `views`, `redux`, and `MainApp.js`:



4. Inside the `redux` folder, we need to create a `modules` folder with `bookmarks` and `categories` folders inside, as well as an `index.js` file where we will initialize all the modules. In each of the `modules` folders, we will have three files: `actions.js`, `reducer.js`, and `index.js`:



5. Open the `index.ios.js` and `index.android.js` files, remove the existing code, and add the following:

```
import React from 'react';
import { AppRegistry } from 'react-native';
import MainApp from './src/MainApp';

AppRegistry.registerComponent('Bookmarks', () => MainApp);
```

6. For now, the `MainApp.js` file will only contain an empty `View`; we will work on this component in later recipes:

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

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

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

export default MainApp;
```

How it works…

In step *2*, we installed the `redux` and `react-redux` libraries. The `react-redux` library contains the necessary bindings to integrate Redux with React.

Redux is not exclusively designed to work with React; in fact, we can use Redux with any other library out there. But, by using `react-redux`, we will be able to seamlessly integrate Redux in our React Native application.

In step *3*, we created the main folders we will use for our app:

- The `components` folder will contain reusable components. The React community calls them presentational components. In simple terms, these are the kind of components that are not aware of any business logic or Redux actions. These components only receive data via props, and should be reusable on any other project. A button or panel would be a perfect example of a presentational component.
- The `containers` folder will contain components that directly receive data from Redux and are able to call actions. In here we will define components such as a header that displays the logged in user. Usually, these components internally use presentational components.
- The `views` folder will contain the main modules in our app, these classes use containers and presentational components. In our application, we will have categories and bookmarks as two different modules internally.
- The `redux` folder will contain all the business logic of our app; in here we shouldn't have any component or view.

In step *4*, we created the two modules that we will have in our app. In this folder, we will define all the actions, the reducer, and logic that we need to display our bookmarks and categories:

- The `actions.js` file will contain all the actions needed in the module; we will talk more about the actions on the next recipe.

- The `reducer.js` file will contain all the code to handle the data in the state. We will dig deeper into this subject in later recipes.
- The `index.js` on each module will have the public actions for each one of them. We can have private and public actions; sometimes we only need an action internally, within the module.

Defining actions

An action is a payload of information that sends data to the store; this is the only way views can request or send data to the global state.

In this recipe, we will create the actions to load the initial bookmarks. For now, we will define hardcoded data, but later on, we will request this data from an API on Heroku.

Getting ready

We will continue working on the code from the previous recipe. Make sure to follow those steps in order to have Redux installed, as well as the folder structure that we will use for this project.

How to do it…

1. First, we need to define the name of the action; let's open the `src/redux/modules/bookmarks/actions.js` file. In this case, it's going to be a constant that we are going to use later, in the action creator and in the reducer:

```
export const LOAD_BOOKMARKS = 'bookmarks/load';
```

2. Now let's create an action creator as follows. Please duplicate the first bookmark and change the IDs. For now, we are hardcoding these values, but later on this same chapter we will request an API to load this data:

```
export function loadBookmarks() {
  return {
    type: LOAD_BOOKMARKS,
    payload: [
      {
        id: 1,
        title: 'Testing 1',
```

```
        url: 'http://test.com',
        category: {
          id: 1,
          name: 'Games'
        },
      },
      // Define more bookmarks here
    ],
  };
}
```

3. We can create as many action creations as needed. For example, if we want to add a new `bookmark`, we need to define an action creator, as follows:

```
export const ADD_BOOKMARK = 'bookmarks/add';

export function addBookmark(bookmark) {
  return {
    type: ADD_BOOKMARK,
    payload: bookmark,
  };
}
```

4. If we want to remove a `bookmark`; we need to define an action creator as follows.

```
export const REMOVE_BOOKMARK = 'bookmarks/remove';

export function removeBookmark(bookmark) {
  return {
    type: REMOVE_BOOKMARK,
    payload: bookmark,
  };
}
```

5. Finally, let's define an action creator to update a `bookmark`, the code is very similar to the previous steps:

```
export const UPDATE_BOOKMARK = 'bookmarks/update';

export function updateBookmark(bookmark) {
  return {
    type: UPDATE_BOOKMARK,
    payload: bookmark,
  };
}
```

How it works…

In step *1*, we are defining the action's type to load the existing bookmarks. We need to define a constant because we are going to use it in a couple places, in the action creator, in the reducer, and in our tests.

When defining an action name, it's recommended to set a namespace to avoid conflicts with other action types.

In step *2*, we are declaring an action creator. Actions are simple JavaScript objects that define that something happens in our application. We use actions to send data to the store, either from the views, from a fetch response, or a task that runs every X seconds.

There's only one single requirement: each action must have a `type` property. In addition, it's recommended to define a `payload` property to send all the data that we want. In this case, we are using an array of bookmarks, but in later steps we use a simple object.

An action is correct as long as we define the `type` property. If we want to send anything else, we should use the `payload` property as defined by the flux pattern. However, this is not required; we can even use a different name, such as `params` or `data`.

Action creators are just functions that return actions; it's convenient to define the functions to call the actions from the views by executing the action creator.

There's more…

Currently, we have defined the action creators that are simple functions. In order to use them, we need to use the `dispatch` method from the `store`. We will learn more about the store in later recipes.

Defining reducers

At this point, we have created some actions for our application–basically, a CRUD of bookmarks. As mentioned previously, actions define that something happened; however, we still need to figure out what to do with the data on the action. By using a reducer, we can define how the data will be stored on the global state.

In this recipe, we will introduce a reducer to set the bookmarks, from the actions defined on the previous recipe, to the global state.

Getting ready

This recipe depends on the previous recipe; make sure to start from the beginning of this chapter to avoid any problems or confusion.

How to do it…

1. Let's start by opening the `src/redux/modules/bookmarks/reducer.js` file. Then we need to import all the action types we defined in the previous recipe.

```
import {
  LOAD_BOOKMARKS,
  ADD_BOOKMARK,
  REMOVE_BOOKMARK,
  UPDATE_BOOKMARK,
} from './actions';
```

2. Now we should define the initial state. Basically, when the app gets initialized, we will have an empty array of bookmarks.

```
const initialState = {
  bookmarks: [],
};
```

3. We can now define the `reducer` function. It will receive two parameters: the current state and the action that has been dispatched:

```
export default function reducer(state= initialState, action) {
  // Defined on steps 4,
}
```

4. Inside the reducer function, we need to return the new state for the current action:

```
export function reducer(state= initialState, action) {
  switch (action.type) {
    case LOAD_BOOKMARKS:
      return {
        ...state,
        bookmarks: [...action.payload],
```

```
        };
      default:
        return state;
    }
  }
```

5. In order to add a new bookmark to the array, all we need to do is get the payload of the action and include it in the new array:

```
    case ADD_BOOKMARK:
      return {
        ...state,
        bookmarks: [...state.bookmarks, action.payload],
      };
```

6. If we want to remove an item from the array, we can use the filter method:

```
    case REMOVE_BOOKMARK:
      return {
        ...state,
        bookmarks: state.bookmarks.filter(
          bookmark => bookmark.id !== action.payload.id
        ),
};
```

7. Updating an item, it's a little bit trickier, as we need to find the bookmark and set the new values. We can use the map method for this:

```
    case UPDATE_BOOKMARK:
      return {
        ...state,
        bookmarks: state.bookmarks.map(bookmark => {
          if(bookmark.id === action.payload.id) {
            return {
              ...action.payload,
            };
          }
          return bookmark;
        }),
      };
```

8. The final step is to combine all the reducers that we have; in this case, we only have one, but later we will add another one for the categories. Therefore, it's a good idea to prepare everything now. Let's open the `src/redux/index.js` file and add the following code:

```
import { combineReducers } from 'redux'
import bookmarks from './modules/bookmarks/reducer'

const reducers = combineReducers({  bookmarks,});
```

How it works…

In step *1*, we are importing all the action types we declared in the previous recipe. We are going to use these types to figure out how are we going to save the data from the actions of the state.

In step *2*, we defined the initial state of the `reducer` function. For now, we only have an empty array of bookmarks, but we can have anything else that we need in order to initialize our app.

In step *3*, we are defining the `reducer` function, which receives two parameters. The first parameter is the current state and the second parameter is the action that is being dispatched.

We are setting the initial state if the given state is null. This way we make sure the bookmarks array exists from the very beginning. This will avoid errors if the update or remove actions get dispatched when there's nothing on the global state.

In step *4*, we are setting the initial bookmarks from the load action to state. As you can see we are not mutating the state, but instead, we are creating a new array with the given bookmarks on the action.

The `reducer` function should be pure; this means there shouldn't be side effects on the input values. Mutating the state or the action is considered a bad practice and we should always avoid doing it. In this case, we are returning a new object with the previous state and a new array containing the new bookmarks.

In addition, to prevent side effects, we should never execute any fetch or Ajax request inside the reducer. We should never call any non-pure functions such as `Math.random`, `Date.now`, or `Guide.raw`. Reducers should always be pure.

In step *5*, we are adding a new element to the bookmarks array. But instead of using the push method, we are creating a new array and appending the new element to the last position. We do this to avoid mutating the original array on the state.

In step *6*, we need to remove the bookmark from the state. The easiest way to do this is by using the `filter` method; we only need to ignore the element with the ID that comes on the payload's action. The `filter` method returns a new array with the filtered elements, in this case, all other elements except the removed item.

In step *7*, we update a bookmark. Because we need to keep this function pure, we use the map method that returns a new array. Then, when the ID on the payload is equal to the ID on the array, we create a new element with the payload values; otherwise, we just return the existing element on the array.

In step *8*, we use the `combineReducers` function to merge all the reducers into a single global object that will be saved in the store. This function will call each reducer with the key in the state that corresponds to that reducer; this function is exactly the same as the following:

```
import bookmarksReducer from './modules/bookmarks/reducer';

const reducers = function(state, action) {
  return {
    bookmarks: bookmarksReducer(state.bookmarks, action),
  };
}
```

As you can see, the bookmark reducer has only been called on the part of the state that needs. This will avoid sending all the data to a single reducer. Later, we will include the categories reducer.

Setting up the store

The store is responsible for saving the information on the state. It's a single global object that can be accessed via `getState`.

In this recipe, we will bring together the actions and the reducer we created in the previous recipes. We will use the existing actions to start saving data on the store. We will also learn how to log changes on the state by subscribing to the store changes.

How to do it…

1. Let's open the `src/redux/index.js` file and import the `createStore` function from the `redux` package:

```
import { combineReducers, createStore } from 'redux';
```

2. Creating the store is extremely simple; all we need to do is call the function from step 1 and send the reducers as the first parameter:

```
const store = createStore(reducers);
export default store;
```

3. That's it! We are done setting up the store, so now let's run some tests. The next steps in this recipe will be removed from the final project; these steps are just for testing our setup. Let's start by importing the action creators we would like to dispatch:

```
import {
  loadBookmarks,
  addBookmark,
  removeBookmark,
  updateBookmark,
} from './modules/bookmarks/actions';
```

4. Before dispatching any action, let's subscribe to the store. By doing this, we will listen to any change in the state. For now, we will only log the state whenever there's a change:

```
const unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);
```

5. Now we should dispatch some actions and see the resulting state on the JavaScript console. Make sure to shake the device or the simulator and allow remote debugging, to debug on Chrome:

```
store.dispatch(loadBookmarks());
```

6. If we want to add a new bookmark, we should call the `addBookmark` action creator with the data as a parameter:

```
store.dispatch(addBookmark({
  id: 2,
  title: 'One more',
```

```
  url: 'http://other.com',
}));
```

7. Updating an existing bookmark is exactly the same; however, we need to make sure the `id` exists on the list, otherwise it will be ignored:

```
store.dispatch(updateBookmark({
  id: 2,
  title: 'One more edited',
  url: 'http://other-edit.com',
}));
```

8. To remove an item, we only need to send the `id` to the action creator; that's the only thing the reducer is using to find the item on the list:

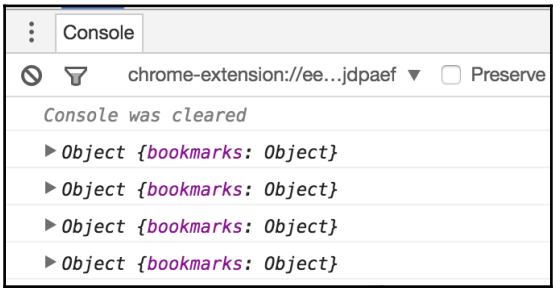
```
store.dispatch(removeBookmark({ id: 1 }));
```

9. After running all these actions, we should stop listening to changes in the state. To do this, we only need to call the unsubscribe function we have:

```
unsubscribe();
```

10. Finally, we need to import the `src/redux/index.js` file into the `MainApp.js` file. This will be enough to run our test and see the log messages on the JavaScript console:

```
import store from './redux';
```



How it works…

In step *3*, we are importing the action creators. Even if we don't have a UI, we can test and see how the state changes; we just need to call the action creators and then dispatch the action.

In step *5*, we use the `dispatch` method from the `store` instance. The `dispatch` takes an action, which is created by the `loadBookmarks` function. Internally, the reducer will be called and it will set the new bookmarks to the state.

When we have our UI in place, we will dispatch the actions in the same fashion from our components, and then the state will get updated.

Communicating with a Remote API

We are currently loading the bookmarks from hardcoded data in the action; ideally, we should make a request to an API to get the existing bookmarks.

In this recipe, we will create a middleware to fetch data from an API; we will create a common code base to handle all the requests and automatically dispatch actions when the request successfully completes or fails.

How to do it…

1. We will start by defining the action creator. Let's open the `src/redux/modules/categories/actions.js` file and add the following action types:

```
export const LOAD_CATEGORIES = 'categ/LOAD_CATEGORIES';
export const LOAD_CATEGORIES_SUCCESS = 'categ/LOAD_CATEGORIES_S';
export const LOAD_CATEGORIES_FAIL = 'categ/LOAD_CATEGORIES_F';
```

2. The action creator will define two properties only, `types` and `promise`. In the `types` property, we define the action types; the order of these actions in the array is important. In the `promise` object, we define the `url` and the `method` used on the fetch request. We can also define headers, or any other valid option the fetch method accepts:

```
export function loadCategories() {
  return {
    types: [
      LOAD_CATEGORIES,
      LOAD_CATEGORIES_SUCCESS,
      LOAD_CATEGORIES_FAIL
    ],
    promise: {
      url: 'https://my-bookmarks-api.herokuapp.com/api/categories',
```

```
      method: 'GET',
    },
  };
}
```

3. Let's create a new `src/redux/middleware` folder to keep all the middleware in a single place. Inside of this folder, we need to create the `fetchMiddleware.js` file.
4. Once we have our file structure in place, we are ready to start working on our first middleware. Let's define the main function; we need to make sure to export it as default:

```
export default function fetchMiddleware({ dispatch, getState }) {
  // Defined in next steps
}
```

5. This function needs to return another function that will get executed between the dispatch of an action and right before the action gets into the reducer:

```
export default function fetchMiddleware({ dispatch, getState }) {
  return next => action => {
    // Defined in next steps
  };
}
```

6. Let's work on the body of the inner function. We need to figure out if the current action contains a `promise` property. If that's not the case, we need to ignore the action and pass the action to the next middleware or the reducer:

```
export default function fetchMiddleware({ dispatch, getState }) {
  return next => action => {
    const { promise, types, ...rest } = action;
    if (!promise) {
      return next(action);
    }
    // defined on later steps
  }
}
```

7. If there's a `promise` property, we need to get the `types` in a separate constant. This will allow us to dispatch the actions individually, at different times. After getting each type, we will dispatch the `REQUEST` action to inform the reducer the request has started:

```
const [REQUEST, SUCCESS, FAILURE] = types;
next({ ...rest, type: REQUEST });
```

8. The last step on the middleware actually triggers the fetch request, parses the response to JSON, and dispatches the success or error actions:

```
const actionPromise = fetch(promise.url, promise);

actionPromise
  .then(response => response.json())
  .then(payload => next({ ...rest, payload, type: SUCCESS }))
  .catch(error => next({ ...rest, error, type: FAILURE }));

return actionPromise;
```

9. In steps *1* and *2* we defined a new action creator that will call our new middleware; however, we are not doing anything with the data we are receiving. On this step, we will create the reducer for the categories–something really simple just to save the categories from the API request to the state. Let's add the following code to `src/redux/modules/categories/reducer.js` file:

```
import { LOAD_CATEGORIES_SUCCESS } from './actions';

const initialState = {
  all: [],
};

export default function reducer(state = initialState, action) {
  switch (action.type) {
    case LOAD_CATEGORIES_SUCCESS:
      return {
        ...state,
        all: [...action.payload.categories],
      };
    default:
      return state;
  }
}
```

10. Now let's add the new reducer to the store. Open the `src/redux/index.js` file and import the new reducer. Then we need to combine it with the bookmark's reducer we already have from previous recipes:

```
import categories from './modules/categories/reducer';

const reducers = combineReducers({
  bookmarks,
  categories,
});
```

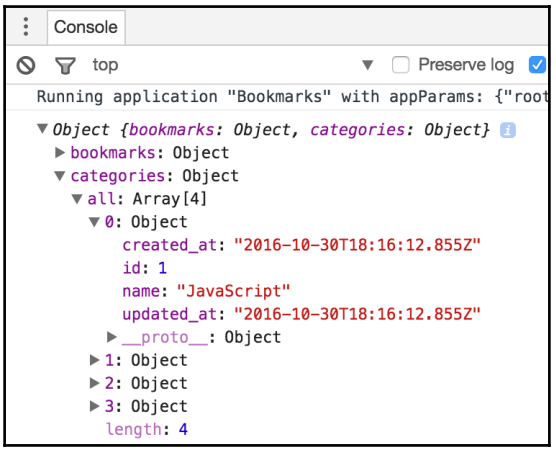
11. The final step is to plug the new middleware to the Redux store. This is really simple, we just need to import the `applyMiddleware` function from the `redux` package and import all the middleware we have, in this case only one. Finally, on the `createStore` function, we send, as a second parameter, the middleware that we want to apply:

```
import {applyMiddleware,combineReducers,createStore} from 'redux';
import fetchMiddleware from './middleware/fetchMiddleware';

const store = createStore(reducers,
applyMiddleware(fetchMiddleware));
```

12. That's all! We are done with our middleware! At this point we should be able to request data from the API using the middleware. Let's write some code to test our middleware:

```
import { loadCategories } from './modules/categories/actions';
const unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

store.dispatch(loadCategories());
```

We are just importing the `loadCategories` action, subscribing to the store changes, and then dispatching the action. If we enable remote debugging on the simulator, we will see that after a few seconds, the state gets updated with the categories from the API:



How it works…

In step *1*, we are defining three actions types. Each of these will be an action dispatched when a fetch request is executed by the middleware:

- `LOAD_CATEGORIES` will be dispatched just before the fetch request is executed.
- `LOAD_CATEGORIES_SUCCESS` will be dispatched when the request is successfully completed, meaning the server response is 200
- `LOAD_CATEGORIES_FAIL` will be executed if there's an error on the network or the server response is different than 200, for example, a 404 error

In step *2*, we are defining the action creator. The returned action will only have two properties, `types` and `promise`. The `types` property is an array that holds the three action types we defined in step 1; each of these actions will be dispatched internally on the middleware.

The order of the action types is very important. The first action will load before the fetch request is triggered, then the success and failure types.

We are also defining the `promise` property, which is an object containing the `url` and the `method` that will be used for this request. We can define in here any other option for the fetch request, such as headers or the body on a post request.

If the promise object is defined on the action, the middleware will execute a fetch request using the configurations defined here; otherwise, the middleware will ignore the action.

In step *3*, we are defining the middleware. This function is receiving an object as a parameter, and using an ES6 destructuring feature, we get the `dispatch` and `getState` functions.

These two methods are provided by the middleware API. The `dispatch` method allows us to dispatch a new action and the `getState` allows us to get the current state on the store.

We are not going to use these two functions in this middleware, but they are really helpful, and sometimes, depending on our requirements, we will need it.

In step *5*, we are returning a function; the body of this function returns another function. Basically, we have a function within a function. If you are familiar with arrow functions, this should be obvious. However, for those that are not familiar with arrow functions, the same code could be defined as follows:

```
export default function fetchMiddleware({ dispatch, getState }) {
return function (next) {
  return function (action) {
  }};
}
```

The first function provides the `next` method, which we will use to return control to the next middleware or to the reducer. The second function will provide the current action being dispatched; we will have access to all the data defined on the action creator.

In step *6*, we are destructuring the action object. We need to get the `promise` and `types` properties. We are going to use them in later steps.

In step *8*, we triggered the fetch request. The `fetch` method returns a promise; we will return this promise to allow the action creators to chain promises in case this is needed.

When the server responds, we are parsing the response to JSON. After that, we dispatch the success action with the payload from the server response. If there's an error while parsing the JSON or in the server response, the catch callback will get executed and we will dispatch the failure action.

Connecting the store with the views

So far, we have set up the state, the middleware, and some actions to load data from hardcoded data and from a remote API. But we are not showing anything on screen; our react components don't have any access to the global state.

In this recipe we will enable components to access the store that we have defined, then we will render the categories on a list.

Getting ready

This recipe depends on all the previous ones; make sure to follow each recipe from the beginning.

On the first recipe of this chapter we installed the `react-redux` library. We are finally going to use it here. Make sure you have already installed it; otherwise, use `npm` or `yarn` to install it before going through the steps.

How to do it…

1. We will start by adding the store to a parent component, and then all the children will have access to the store. On the `src/MainApp.js` file, we need to import the `Provider` component from the `redux` package, the `store` that we created in previous recipes, and the `AppNavigator` component that we will create in later steps of this same recipe:

```
import React from 'react';
import { Provider } from 'react-redux';
import store from './redux';
import AppNavigator from './views/AppNavigator';
```

2. We will use a stateless component. We don't need styles or anything else, just the `Provider` that receives the store and the `AppNavigator` to render the required view:

```
const MainApp = () => (
  <Provider store={store}>
    <AppNavigator />
  </Provider>
);
```

```
export default MainApp;
```

3. Now let's work on the `AppNavigator` class. Let's create the `src/views/AppNavigator/index.js` file; in here we will import the following dependencies. Nothing really new, just a few components from React Native and the `Categories` component, which will be a list of categories:

```
import React, { Component } from 'react';
import Categories from '../Categories';
import {
  Navigator,
  Platform,
  StyleSheet,
} from 'react-native';
```

4. We have covered the navigator component in `Chapter 1`, *Getting Started*; therefore, I won't talk much about this component. In here we are just rendering the initial view; in the next recipe, we will come back to this class to render a new component based on the route:

```
class AppNavigator extends Component {

  renderScene(route, navigator) {
            return <Categories navigator={navigator} />;
  }

  render() {
    return (
      <Navigator
        style={styles.container}
        configureScene={(route) => {
          if (Platform.OS === 'android') {
            return Navigator.SceneConfigs.FloatFromBottomAndroid;
          }

          return Navigator.SceneConfigs.FloatFromBottom;
        }}
        initialRoute={{}}
        renderScene={this.renderScene}
      />
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
```

```
    },
  });
     export default AppNavigator;
```

5. Now we need to work on the good stuff. We are going to render the list of categories that comes from the server API. Let's create a `src/views/Categories/index.js` file and import all the dependencies for this class:

```
import React, { Component, PropTypes } from 'react';
import {
  ListView,
  Platform,
  StyleSheet,
  Text,
  View,
} from 'react-native';
import { connect } from 'react-redux';
import { loadCategories } from
'../../redux/modules/categories/actions';
```

6. Let's define the class. We need a `Categories` property that will be an array of data coming from the global state. When the component is about to mount, we create the data source for the list; this might be empty at the beginning. On the `render` method we define the list view component and render the name of the category. Nothing fancy, just a simple list:

```
class Categories extends Component {

  static propTypes = {
    categories: PropTypes.array,
  };

  componentWillMount() {
    this.ds = new ListView.DataSource({rowHasChanged: (r1, r2) =>
       r1 !== r2});
    this.state = {
      dataSource: this.ds.cloneWithRows(this.props.categories),
    };
  }

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.toolbar}>Categories</Text>
        <ListView
          dataSource={this.state.dataSource}
```

```
            renderRow={(rowData) => <Text>{rowData.name}</Text>}
            enableEmptySections
          />
        </View>
      );
    }
};
```

7. This step is the most important in this recipe. In here we will map and connect the data from state to the props on this component:

```
const mapStateToProps = (state) => {
  return {
    categories: state.categories.all,
  };
}

export default connect(mapStateToProps)(Categories);
```

8. Once our component is connected to the store, we can call our actions from the component itself. Let's load the categories from the API by dispatching the action when the component is about to mount:

```
componentWillMount() {
  // ...
  this.props.dispatch(loadCategories());
}
```

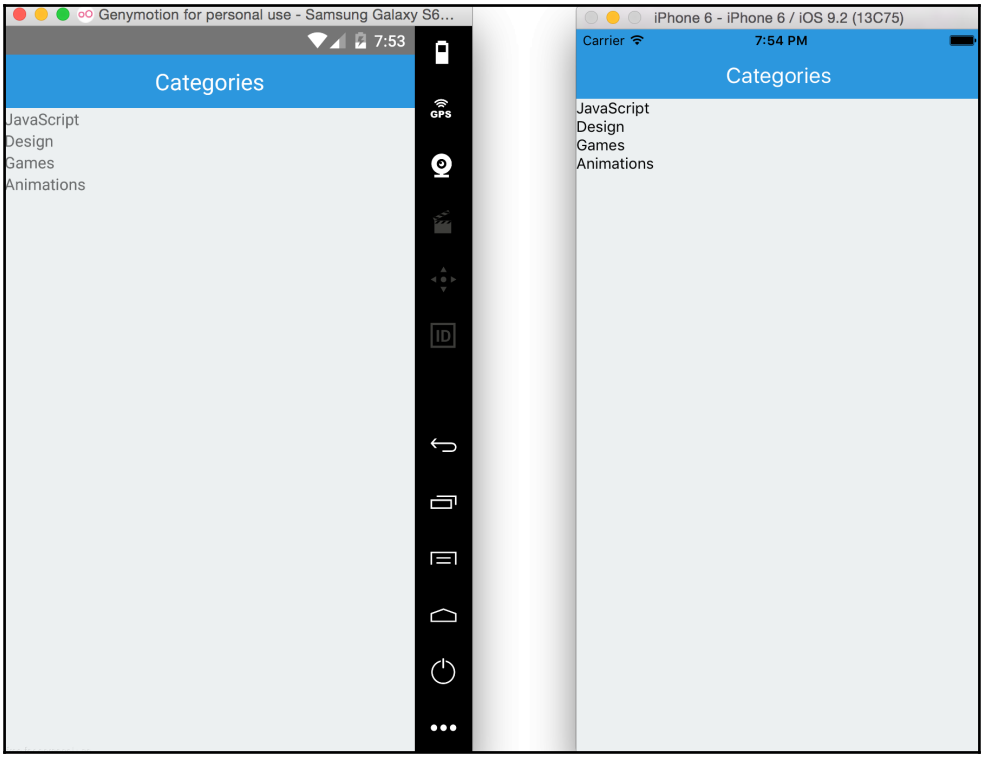
9. We are almost done! The only thing left to complete the Redux integration with this component, is to update the `dataSource` whenever there are any changes in the categories array on the global state:

```
componentWillReceiveProps(nextProps) {
  if (this.props.categories !== nextProps.categories) {
    this.setState({
      dataSource: this.ds.cloneWithRows(nextProps.categories),
    });
  }
}
```

10. We are done with the Redux integration; let's just add some styles to our app. We are going to use the `Platform` class to set a different padding to the top of the toolbar. On iOS we need to set 30px, but only 10px on Android:

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: '#ecf0f1',
```

```
    flex: 1,
  },
  toolbar: {
    backgroundColor: '#3498db',
    color: '#fff',
    fontSize: 20,
    textAlign: 'center',
    padding: 10,
    ...Platform.select({
      ios: {
        paddingTop: 30,
      },
      android: {
        paddingTop: 10,
      },
    }),
  },
});
```



How it works…

In step *2*, we are defining the main component of our application. It's a very simple component, all we need is to define the `Provider` component as the parent of all other components. This component will provide access to the store to any of the children.

The `AppNavigator` will render the module that we want to display at the time. Initially, we will display the list of categories from the API.

In step *5*, we are importing the dependencies for the class. We are going to load the categories from the API; therefore, we need to import the action that triggers the fetch request.

We also need to import the `connect` method from the `react-redux` package. This method allows us to map the state data to the props on the component; the method also provides a dispatch method to the class to allow us to dispatch actions directly from the views.

In step *7*, we are connecting the state to the props on our component. The `connect` method accepts a function as a first parameter; in here we will set the value for each property from the state. In this case, we are only getting the `categories` from the state, but usually, we will pass much more data to our component.

When we get to the point where we are mapping so many properties in here, it's usually a good sign that it's time to connect other children components to the state.

The `connect` function returns a higher order function that takes the component we want to connect, in this case, `Categories`. Internally, the `connect` method will set the store from the provider we defined on the `MainApp` component and set the props values from the state.

In step *8*, we are loading the categories from the API. The `connect` method attached a `dispatch` function as a property of the component. All we need to do is call the `dispatch` with the action creator that we need, in this case, `loadCategories`.

In step *9*, we are updating the `dataSource` to render the new data on the list. In order to update the `dataSource`, we need to check if the new `categories` on the props are different than the current ones. If that's the case, we will have to update the `dataSource`, and the list will automatically render the new information.

Storing offline content using Redux

At some point we will need to locally store the global state of our application, this will allow us to restore the state the next time the user opens the app. For example, we might want to keep the current view the user is using on the state. This way we can show the same view when the app gets loaded again.

We might also want to cache some information to avoid calling the API multiple times; this will improve the performance, as well as the traffic on your server.

On `Chapter 4`, *Working with Application Logic and Data*, we learned about different methods to store data on the device, so for this recipe, we will use `AsyncStorage`.

Getting ready

This recipe depends on the previous ones, so make sure to follow along with all the previous recipes, as we will use the same store, middleware, actions, and so on.

How to do it…

1. In order to support storing the state, we are going to use the `redux-persist` library. Let's start by installing it using `npm` or `yarn`, whichever is your favorite package manager:

```
$ npm i --save redux-persist
```

2. Open `src/redux/index.js` file and add the following dependencies to support persisting the store:

```
import { persistStore, autoRehydrate } from 'redux-persist';
import { AsyncStorage } from 'react-native';
```

3. Now we need to slightly change the way we are creating the store. First we need to apply the middleware that we need in our application; in this case it's only one:

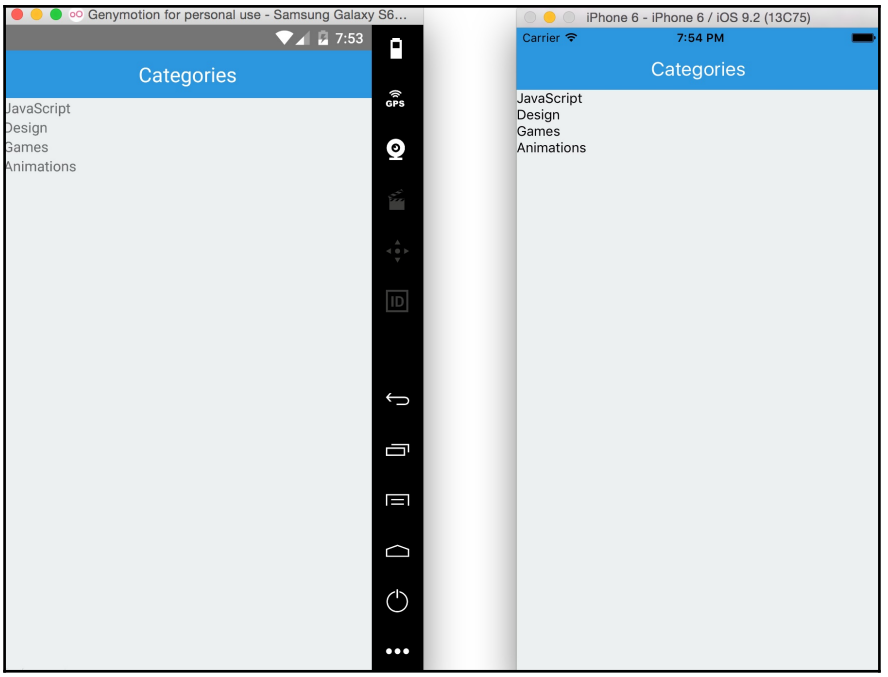
```
    const createAppStore =
applyMiddleware(fetchMiddleware)(createStore);
```

4. Then we need to call the `autoRehydrate` function, to load the store with the saved data on the device. Finally, we need to persist the store, setting the `AsyncStorage` as the engine storage:

```
    const store = autoRehydrate()(createAppStore)(reducers);

    persistStore(store, { storage: AsyncStorage });
```

5. We are done setting up the persistence. Now if we refresh the app the request will be triggered and the state will be saved automatically. The next time we refresh the app, the state will be loaded from the local device.
6. If we remove the loading action from the `src/views/Categories/index.js`, we should still see the categories in the list. By commenting this line, we can prove the data has been saved and loaded successfully:

```
componentWillMount() {
  this.ds = new ListView.DataSource({
    rowHasChanged: (r1, r2) => r1 !== r2
  });
  this.state = {
    dataSource: this.ds.cloneWithRows(
      this.props.categories
    ),
  };
  // this.props.dispatch(loadCategories());
}
```



Even though the data has been saved, we probably need to request to the server to get the latest information again. But if the device doesn't have an Internet connection, it will successfully load and display the data from the locally saved state.

A more advanced scenario would be the case where there's no Internet connection, but we still want to allow the user to save new data. We can easily save the data on the state, and the next time there's an Internet connection available, we should only synchronize the data with the server.

Saving the state. It's very convenient and easy to do; with a little configuration, we have everything in place.

Showing network connectivity status

In this recipe, we will learn how to use Redux to keep the state of the network status. When we lose network connectivity, we will display an alert message to inform the user there's no network available.

Getting ready

This recipe depends on the previous one. We are going to use the `NetInfo` class to listen to network changes, then we will keep the status of the network an the store and display an alert when the device is offline.

How to do it…

1. We will start by creating the action that we will trigger when the `network` status changes. This action will receive the current type of network the device is connected to, for example, `wifi`, `cell`, `vpn`, and so on. If the type is none, it means the device is offline. Let's create the `src/redux/modules/network/actions.js` file and add the following code:

```
export const NETWORK_CHANGE = 'network/change'


export function setConnectivity(network) {
  const status = network.toUpperCase();
  return {
    type: NETWORK_CHANGE,
    payload: {
      isOnline: status !== 'NONE',
      isOffline: status === 'NONE',
    },
  };
}
```

2. The reducer is very simple: we just need to keep track of the two properties on the action's payload. By default, the value of both properties will be null. Let's create a new `src/redux/modules/network/reducer.js` file and add the following code:

```
import { NETWORK_CHANGE } from './actions';

const initialState = {
  isOnline: null,
  inOffline: null,
};

export default function reducer(state = initialState, action) {
  switch (action.type) {
    case NETWORK_CHANGE:
```

```
        return {
          ...state,
          ...action.payload,
        };
      default:
        return state;
    }
  }
```

3. To complete the `redux` section, we need to combine the new reducer when creating the store. Let's open the `src/redux/index.js` file and add the following code to the existing one:

```
import network from './modules/network/reducer';

const reducers = combineReducers({
  bookmarks,
  categories,
  network,
});
```

4. The next step is to set up the listener to network changes. Whenever the network status changes, we will dispatch the `setConnectivity` action to set the new data on the redux state. Let's open the `src/views/AppNavigator/index.js` file and add the following dependencies:

```
import { connect } from 'react-redux';
import { setConnectivity } from
'../../redux/modules/network/actions';

import {
  Navigator,
  NetInfo,
  Platform,
  StyleSheet,
} from 'react-native';
```

5. When the component is about to mount, we need to set up the listener using the `NetInfo` class:

```
componentWillMount() {
  NetInfo.addEventListener(
    'change',
    this.onConnectivityChange,
  );
}
```

6. We should also remove the listener when the component is about to unmount:

```
componentWillUnmount() {
  NetInfo.removeEventListener(
    'change',
    this.onConnectivityChange,
  );
}
```

7. Now we need to define the connectivity change callback function. This function will dispatch the action creator to set the new status on the `redux` state:

```
onConnectivityChange = (reach) => {
  this.props.dispatch(setConnectivity(reach));
}
```

8. The final step on this component is to connect it to the Redux store. We don't need to map any prop, therefore we will not define a mapping function; we can just send nothing as the first parameter on the connect function:

```
export default connect()(AppNavigator);
```

9. We are now listening to network changes and setting the status on the `redux` state; all that is left is to actually show an alert message when there's no connection on the device. Let's open the `src/views/Categories/index.js` file and add the following mapping property:

```
const mapStateToProps = (state) => {
  return {
    categories: state.categories.all,
    isOnline: state.network.isOnline,
  };
}
```

10. Now, whenever the state changes we will have the new value on the `isOnline` property of this component. All we need to do is check if the `isOnline` prop is equal to `false` show an alert; we will do this on the `render` method as follows:

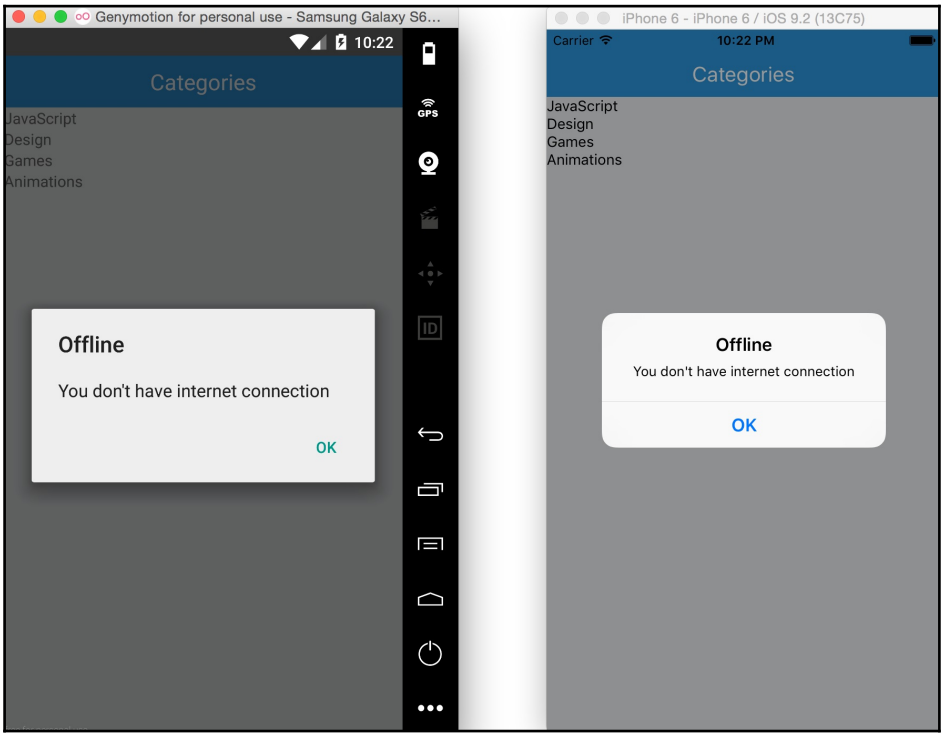
```
render() {
  if (this.props.isOnline === false) {
    Alert.alert('Offline', 'You don't have internet connection');
  }
  // rest of the code here...
}
```

11. Don't forget to import the `Alert` component to the class:

```
import {
  Alert,
  ListView,
  Platform,
  StyleSheet,
  Text,
  View,
} from 'react-native';
```

12. If we test our app, we should be able to see the `Alert` message when we turn off the Wi-Fi. On iOS this is enough to test; however, on Android, we need to test on an actual device, but before that, we need to set the permissions on the `android/app/src/main/AndroidManifest.xml` file:

```
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
```



How it works…

In step *1*, we are creating the action to set the connectivity on the `redux` state. We need to uppercase the network type because in iOS it is lowercase and in Android it is uppercase. If we want our code to run on both platforms, we need to uppercase the value of the network.

In step *2*, we created the reducer for the network status. In here we initialized the properties as null, which means we don't know if the device is online or offline in the very beginning. When checking for the connectivity status we should check for `true`/`false` values instead of falsy values.

In step *4*, we are importing the dependencies for the `AppNavigator` class. We are going to use this component because it's the parent of all the other modules. We just need a component that will be mounted at all times, mainly because we want to listen to the changes all the time.

In step *7*, we are using an arrow function to bind the method to the instance. This way, we will be able to access the correct scope when the function gets executed.

6
Adding Native Functionality

In this chapter, we will cover the following recipes:

- Exposing custom iOS modules
- Rendering custom iOS view components
- Exposing custom Android modules
- Rendering custom Android view components
- Handling the Android back button
- Reacting to changes in application state
- Copying and pasting content
- Receiving push notifications
- Authenticating via TouchID or fingerprint sensor
- Hiding application content when multitasking
- Background processing on iOS
- Background processing on Android
- Playing audio files on iOS
- Playing audio files on Android

Introduction

One of the core principles in React Native development is writing JavaScript to build a truly native mobile app. To accomplish this, many native APIs and UI components are exposed through an abstraction layer and are accessed through the React Native bridge. Through the native APIs, we can access functionality that is not available for mobile web applications, such as vibration, camera roll, contacts, native alerts, or toasts, and much more.

By exposing the native view components, we are able to leverage all the rendering performance the device has to offer, as we are not going through a WebView like in a hybrid app. This gives us a native look and feel that adapts to the platform you are using the application on. With React Native, we are able to render any native view component including maps, lists, input fields, toolbars, picker, and so on.

While React Native comes with many built-in native modules and view components, we are often in a position where we need some custom functionality leveraging the native application layer that is not provided out of the box. Fortunately, there is an extremely rich open source community supporting React Native that not only contributes to the library itself, but also publishes libraries that export some common native modules and view components. If you can't find a first or third-party library to accomplish what you need, you can always build it yourself.

In this chapter, we will present recipes that cover exposing custom native functionality, whether it is an API or view component, on both platforms. We will also show examples that leverage either built-in or community modules. The recipes cover a range of topics from rendering a basic button to creating a multithreaded process that does not block the main application threads. All the recipes in this chapter require you to have a React Native application already created.

Exposing custom iOS modules

As you begin writing exciting and engaging React Native applications, you will get to a point where you need to execute some code on the native layer. This could be anything from executing some data processing that is better suited to the native layer instead of JavaScript for performance, accessing some native functionality that is not exposed, such as file I/O, or leveraging existing native code from other applications or libraries in your React Native app.

This recipe will walk you through the process of executing some native Objective-C or Swift code and communicating with the JavaScript layer. We will build a native `HelloManager` module that will greet our user with a message. We will also show how to execute native Objective-C and Swift code, taking in arguments, and showing several ways of communicating back with the UI (JavaScript) layer.

Getting ready

For this recipe, we created a React Native application titled `NativeModule`.

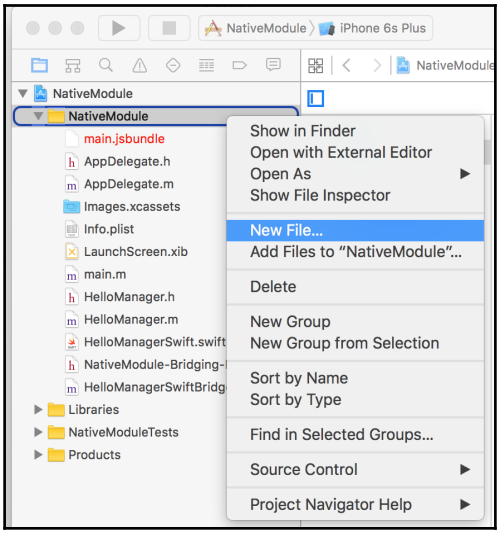
In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button --save
```

How to do it…

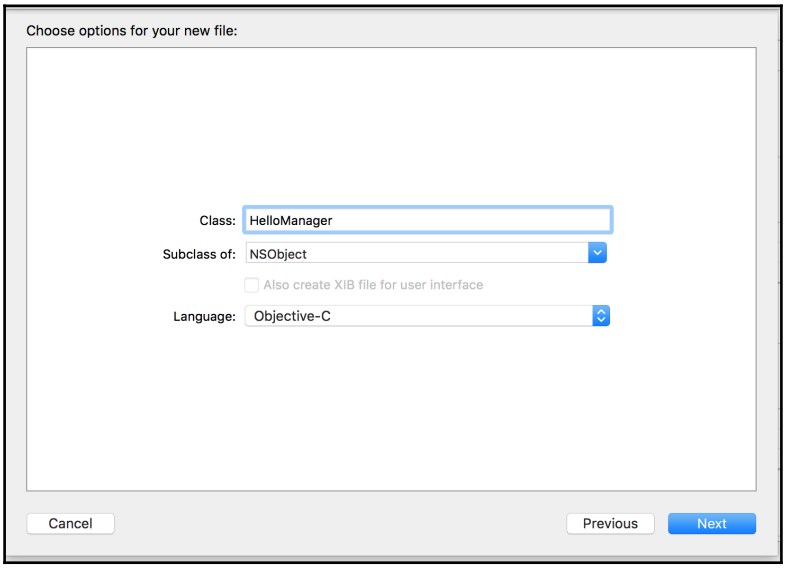
To expose custom iOS modules, we perform following steps:

1. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./NativeModule/ios/NativeModule.xcodeproj`).
2. Select and right-click on the group that matches your project name, and click on **New File…**:



3. Select **Cocoa Class** and click **Next**.

4. Enter `HelloManager` for **Class** name and leave the **Subclass of** as **NSObject** and **Language** as **Objective-C**:



5. Click **Next** and then you will have to choose into which directory you would like to place your new class. Double-click the `NativeModule` directory to create the class.
6. You will now be presented with the new **Cocoa** Class you created; this consists of a header (`.h`) and an implementation (`.m`) file.
7. Open the header file (`HelloManager.h`) and add an import for `RCTBridgeModule.h` and implement the protocol in your class:

```
#import <Foundation/Foundation.h>
#import "RCTBridgeModule.h"

@interface HelloManager : NSObject <RCTBridgeModule>

@end
```

8. Now open the implementation (`HelloManager.m`) file and let's begin adding some basic functionality.

9. In order for our application to be able to access this module from the JavaScript layer, we need to register it with the React Bridge. To do so, add `RCT_EXPORT_MODULE()` after the `@implementation` tag:

```
#import "HelloManager.h"

@implementation HelloManager
RCT_EXPORT_MODULE();
```

10. Next, we need to add our basic functionality. We will create a `greetUser` method that will take two arguments, `name` and `isAdmin`. We will take these arguments, create a greeting message using string concatenation, and then send it back to the JavaScript layer via `callback`:

```
RCT_EXPORT_METHOD(greetUser:
(NSString *)name isAdmin:(BOOL *)isAdmin callback:
(RCTResponseSenderBlock) callback) {
   NSString *greeting =
   [NSString stringWithFormat:
   @"Welcome %@, you %@ an administrator", name, isAdmin ?
   @"are" : @"are not"];
  callback(@[greeting]);
}
```

11. Now we need to build our UI that will invoke the native `HelloManager` `greetUser` method and display its output. Fortunately, the React Native bridge does all the heavy lifting for us and leaves us with a simple-to-use JavaScript object that mimics the `NativeModules` API. In this example, we will be using a `TextInput` and `Switch` to provide the `name` and the `isAdmin` value for the native modules method.

    Open `index.ios.js` in the root of our project and add `NativeModules`, `TextInput`, `Switch` to the `import { ... } from 'react-native'` block.

    We will also be using the `react-native-button` class. So add the following after the previous import block:

```
import Button from 'react-native-button';
```

    Finally, we need to make a reference to our `HelloManager` object that exists as a property of `NativeModules`. Add the following line of code before your class definition:

```
const HelloManager = NativeModules.HelloManager;
```

12. After importing all the necessary modules and submodules, it's time to put them to use. First, we're going to prepare our initial state with a `greetingMessage` that will be rendered. Inside our main class, add a `componentWillMount` function:

```
componentWillMount() {
  this.setState({
    greetingMessage: undefined
  });
}
```

13. In our components `render` function, we're going to need to add the following:

    `TextInput` to collect the user's name:

```
<TextInput

  ref="userName"
  autoCorrect={false}
  style={styles.inputField}
  placeholder="User Name"
  onChangeText={(text) => this.setState({ userName: text }) }/>
```

    `Switch` to set whether the user is an admin or not:

```
<Text style={styles.label}>Admin</Text>
<Switch style={styles.radio} onValueChange={(value) =>
  this.setState({ isAdmin: value }) }
  value={this.state.isAdmin}/>
```

    `Button` which will make the call to the `HelloManager greetUser` method:

```
<Button
  containerStyle={styles.buttonContainer}
  style={styles.buttonStyle}
  onPress={this.greetUserCallback}>
    Greet (callback)
</Button>
```

    `Text` to display the `greetingMessage` that will come back from the call:

```
<View style={styles.flexContainer}>
  <Text>Response: </Text>
  <Text>{this.state.greetingMessage}</Text>
</View>
```

14. With the UI rendering the necessary components, we now need to wire up the `Button` press handler and make the native call. Add the following methods to the class:

```
greetUserCallback = () => {
    const state = this.state;

    HelloManager.greetUser(state.userName, state.isAdmin,
    this.displayResult);
}

displayResult = (result) => {
    this.refs.userName.blur();
    this.setState({ greetingMessage: result });
}
```

How it works…

The work we did in this recipe will serve as the foundation for many of the following recipes in this chapter. It also happens to be how Facebook themselves implement many of their bundled native APIs.

There are several important concepts to keep in mind. Any native module class has to extend `RCTBridgeModule` as it contains functionality to register our class onto the React Native bridge. We register our class with the `RCT_EXPORT_MODULE` method call. The `RCT_EXPORT_METHOD` call is used to register the methods once we have registered the module. Registering the module, its respective methods and properties allows us to interface with the native layer from the JavaScript.

In summary, after the last step, the `greetUserCallback` function is executed when you press the button. The function makes a call to `HelloManager.greetUser`, then passes in the `userName` and `isAdmin` properties from the state, and the `displayResult` function as a callback. The `displayResult` sets the new `greetingMessage` on the state, causing a refresh of the UI, and now the message is displayed.

There's more…

React Native supports three ways for the native module to communicate back to the JavaScript layer: callbacks (covered in our step-by-step guide), promises, and events. To see working examples of each one of these techniques, take a look at the source code bundled with the book. If you are interested in writing native modules using the Swift language, there is a working example of the same functionality bundled in the sample application.

Rendering custom iOS view components

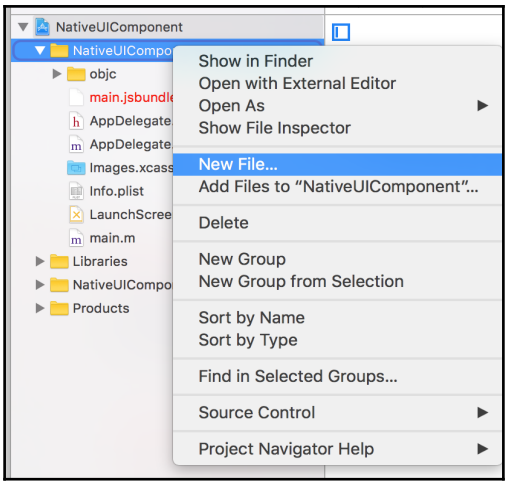
While it is very important to leverage the devices processing power in executing code on the native layer in our React Native application, it is equally important to leverage its rendering power to show native UI components. React Native can render any UI component that is an implementation of `UIView` inside an application. These components can be lists, form fields, tables, graphics, and so on.

For this recipe, we created a React Native application titled `NativeUIComponent`.

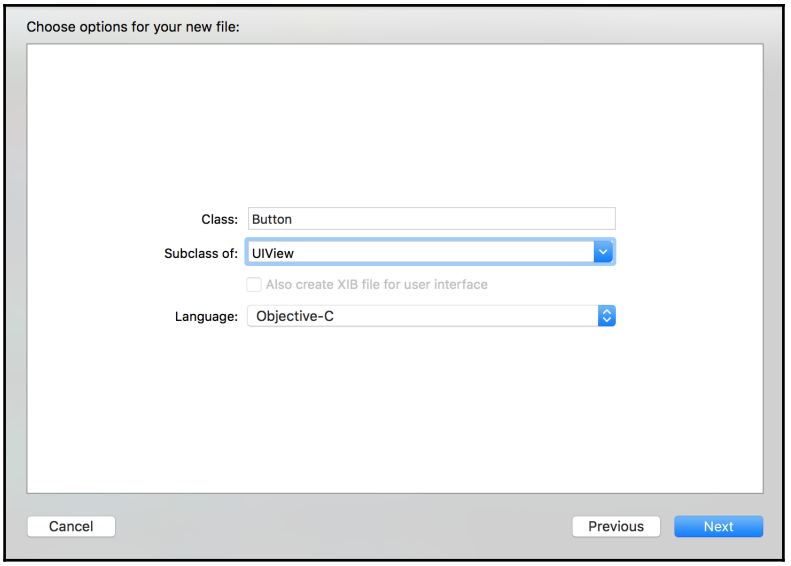
In this recipe, we will take a native `UIButton` and expose it as a React Native view component. You will be able to set the button label and attach a handler for when it is tapped.

How to do it…

1. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./NativeUIComponent/ios/NativeUIComponent.xcodeproj`).
2. Select and right-click on the group that matches your project name and click on **New File…**:



3. Select **Cocoa Class** and click **Next**.
4. Enter `Button` for **Class** name and set the **Subclass of** as **UIView** and **Language** as **Objective-C**:



5. Click **Next** and then you will have to choose into which directory you would like to place your new class. Double-click the `NativeUIComponent` directory to create the class.

6. Repeat steps 2-5 with `ButtonViewManager` as the class and `RCTViewManager` as the subclass.
7. First, we're going to implement our `Button` UI class. In the header (`Button.h`) file, we will add an `onTap` property to wire up our tap event:

```
#import <UIKit/UIKit.h>
#import "RCTComponent.h"

@interface Button : UIView

@property (nonatomic, copy) RCTBubblingEventBlock onTap;

@end
```

   Next, open up the implementation file (`Button.m`). Here we will start off by creating references for our `UIButton` instance and the string that will hold the button label:

```
#import "Button.h"
#import "UIView+React.h"

@implementation Button {
  UIButton *_button;
  NSString *_buttonText;
}
```

   The bridge will look for a setter for the `buttonText` property, in which we will set the `UIButton` instance title field. Add the following setter to your class:

```
-(void) setButtonText:(NSString *)buttonText {
  NSLog(@"Set text %@", buttonText);
  _buttonText = buttonText;
  if(_button) {
    [_button setTitle:
    buttonText forState:UIControlStateNormal];
    [_button sizeToFit];
  }
}
```

   The other part of the arguments our button will accept is the `onTap` event handler. We need to wire this to our `UIButton` instance through an action selector:

```
- (IBAction)onButtonTap:(id)sender {
  self.onTap(@{});
```

```
}
```

Finally, we need to instantiate the `UIButton` and place it inside a React `Subview`:

```
-(void) layoutSubviews {
  [super layoutSubviews];
  if( _button  == nil) {
    _button =
    [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [_button addTarget:self action:@selector(onButtonTap:)
    forControlEvents:UIControlEventTouchUpInside];
    [_button setTitle:
    _buttonText forState:UIControlStateNormal];
    [_button sizeToFit];
    [self insertSubview:_button atIndex:0];
  }
}

- (void)insertReactSubview:
(UIView *)view atIndex:(NSInteger)atIndex
{
  [self addSubview:view];
}

- (void)removeReactSubview:(UIView *)subview
{
  [subview removeFromSuperview];
}

- (void)removeFromSuperview
{
  [super removeFromSuperview];
}
```

8. Now we need to implement our `ButtonViewManager`, which will interface with our React Native application. Open up the implementation (`ButtonViewManager.m`) file and add the following:

```
#import "ButtonViewManager.h"
#import "Button.h"
#import "UIView+React.h"

@implementation ButtonViewManager
RCT_EXPORT_MODULE()

- (UIView *)view {
```

```
  Button *button = [[Button alloc] init];
  return button;
}

RCT_EXPORT_VIEW_PROPERTY(buttonText, NSString);
RCT_EXPORT_VIEW_PROPERTY(onTap, RCTBubblingEventBlock);

@end
```

9. Let's build out our UI to render this button. We're going to need our `Button` in its own React Component. Create a `Button.js` file in your project's root directory (where `index.ios.js` is located) and add the following code:

```
import React, { Component } from 'react';
import { requireNativeComponent } from 'react-native';

export default class Button extends Component {
    render() {
        return <ButtonView {...this.props} />;
    }
}

Button.propTypes = {
    buttonText : React.PropTypes.string,
    onTap : React.PropTypes.func
};

var ButtonView = requireNativeComponent('ButtonView', Button);
```

10. Open our `index.ios.js` file and add the following code after the `import { ... } from 'react-native'` block:

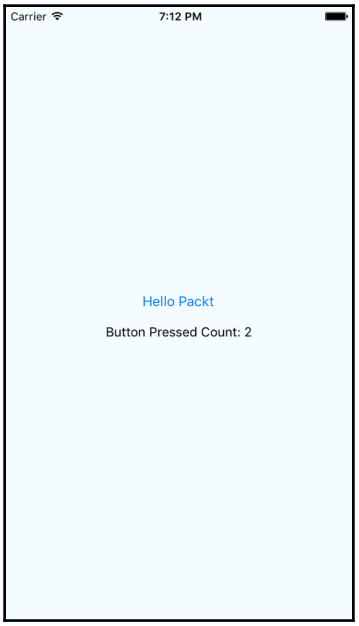
```
import Button from './Button';
```

Add the following to the class implementation:

```
componentWillMount() {
   this.setState({
     count : 0
   });
}
onButtonTap = () => {
  let count = this.state.count;
  this.setState({
    count : count+1
  });
}
```

```
render() {
  return (
    <View style={styles.container}>
      <Button buttonText="Hello Packt"
      onTap={this.onButtonTap}
      style={{height: 40, width: 80}}/>
      <Text>Button Pressed Count: {this.state.count}</Text>
    </View>
     );
}
```

Your application should look and function like this:



How it works…

For this recipe, we exposed a basic native UI component. This is the same foundation of how all the built-in UI components (for example, `Slider`, `Picker`, `ListView`, and so on) are built. The most important requirement in creating UI components is that your `ViewManager` extends `RCTViewManager` and returns an instance of `UIView`. In our case, we are wrapping a `UIButton` with a React-specific `UIView` extension. This helps us with the layout and styling of the component.

The next important factor is sending properties and reacting to component events. In the step 8 code block, we are creating and returning an instance of our `Button` class and registering the `buttonText` and `onTap` view properties that will come from the JavaScript layer.

Exposing custom Android modules

Often you will find yourself writing React Native applications for both iOS and Android. We have discussed how to go about creating native modules in iOS in the first recipe of this chapter. Now it's time to put our green Android robot hats on and write some Java!

This recipe will take us through writing our first Android native module. We will mimic the functionality of the *Exposing custom iOS modules* recipe, but for Android. We're going to create a `HelloManager` native module with a `greetUser` method that takes a `name` and `isAdmin` Boolean value arguments. It will return a greeting message that we will display.

Getting ready

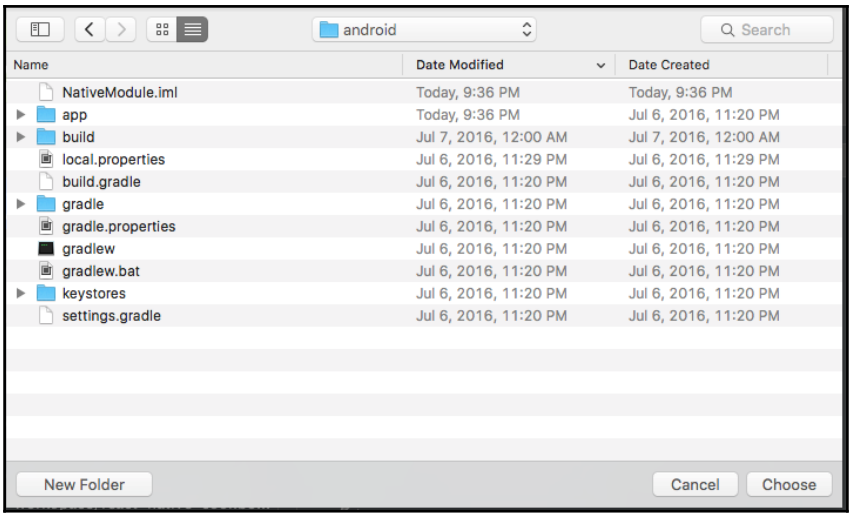
For this recipe, we created a React Native application titled `NativeModule`.

In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

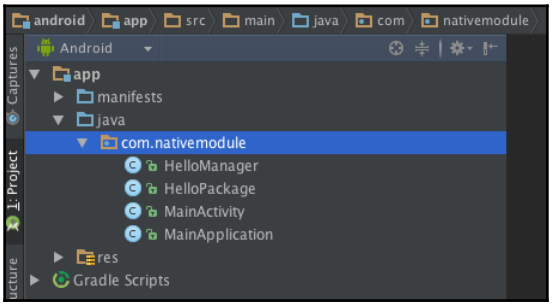
```
$ npm install react-native-button --save
```

How to do it…

1. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project:



2. Open up the left-hand Project explorer and expand the package structure until you can see the Java source files (for example, `app/java/com.nativemodule`):



3. Right-click on the package and select **New | Java Class**. Use `HelloManager` for the class name and leave the **Kind** field as **Class**.
4. Repeat step 3 for the `HelloPackage` class.
5. Now it's time to implement our `HelloManager` native module. So let's go ahead and open up the `HelloManager.java` file.

   First, we're going to have to import some classes that we will be using from React Native:

```
import com.facebook.react.bridge.Callback;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
```

```
import com.facebook.react.bridge.ReactMethod;
```

`ReactContextBaseJavaModule` is the base class for all React Native modules. Now we need to instruct our `HelloManager` class to be a subclass of it:

```
public class HelloManager extends ReactContextBaseJavaModule {
    public HelloManager(ReactApplicationContext reactContext) {
        super(reactContext);
    }
    @Override
    public String getName() {
        return "HelloManager";
    }
}
```

The `getName` method is used for the registration of the native module with the React Native bridge. This is different than iOS native modules, as those use the class name.

6. Now that we have set up our `HelloManager`, it's time to add the `greetUser` method that will expect the `name` and `isAdmin` arguments, as well as the callback to which it will post the greeting message:

```
@ReactMethod
public void greetUser
(String name, Boolean isAdmin, Callback callback) {

    String greeting =
    "Welcome " + name + ", you " +
    (isAdmin ? "are" : "are not") + " an administrator";

    callback.invoke(greeting);
}
```

7. Another step that is unique to Android is having to register the native module with the application. This is a two-step process. First, we need to add our `HelloManager` to the `HelloPackage` class we created earlier. Open up the `HelloPackage.java` and add the following:

```
public class HelloPackage implements ReactPackage {
    @Override
    public List
    <Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }
```

```
    @Override
    public List
    <ViewManager> createViewManagers
    (ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }

    @Override
    public List
    <NativeModule> createNativeModules
    (ReactApplicationContext reactContext) {
        List<NativeModule> modules = new ArrayList<>();

        modules.add(new HelloManager(reactContext));

        return modules;
    }
}
```

8. Now we need to take our `HelloPackage` that holds the instance of `HelloManager` and add it to our React Native application. Open `MainApplication.java` and instantiate `HelloPackage` in the `getPackages` method:

```
@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
          new MainReactPackage(),
          new HelloPackage()
      );
}
```

9. We're all set on the Java side of the spectrum, let's move on to the React world. We need to build our UI that will invoke the native `HelloManager greetUser` method and display its output. Fortunately, the React Native bridge does all the heavy lifting for us and leaves us with a simple-to-use JavaScript object that mimics the native modules API. In this example, we will be using a `TextInput` and `Switch` to provide the `name` and the `isAdmin` value for the native modules method.

    This is the same exact functionality as iOS. If you have completed the *Exposing custom iOS modules* recipe, the following steps are the same as steps 11-14.

Open `index.android.js` in the root of our project and add `NativeModules, TextInput, Switch` to the `import { ... } from 'react-native'` block.

We will also be using the `react-native-button` class. So add the following after the previous import block:

```
import Button from 'react-native-button';
```

Finally, we need to make a reference to our `HelloManager` object that exists as a property of `NativeModules`. Add this following line of code before your class definition:

```
const HelloManager = NativeModules.HelloManager;
```

10. After importing all the necessary modules and submodules, it's time to put them to use. First, we're going to prepare our initial state with a `greetingMessage` that will be rendered. Inside our main class, add a `componentWillMount` function:

```
componentWillMount() {
  this.setState({
    greetingMessage: undefined
  });
}
```

11. In our component's render function, we're going to need to add the following:

`TextInput` to collect the user's name:

```
<TextInput

  ref="userName"
  autoCorrect={false}
  style={styles.inputField}
  placeholder="User Name"
  onChangeText={(text) => this.setState({ userName: text }) }/>
```

`Switch` to set whether the user is an admin or not:

```
<Text style={styles.label}>Admin</Text>
      <Switch style={styles.radio} onValueChange={(value) =>
      this.setState({ isAdmin: value }) } value=
      {this.state.isAdmin}/>
```

`Button`, which will make the call to the `HelloManager greetUser` method:

```
<Button
  containerStyle={styles.buttonContainer}
     style={styles.buttonStyle}
  onPress={this.greetUserCallback}>
     Greet (callback)
</Button>
```

`Text` to display the `greetingMessage` that will come back from the call:

```
<View style={styles.flexContainer}>
  <Text>Response: </Text>
  <Text>{this.state.greetingMessage}</Text>
</View>
```

12. With the UI rendering the necessary components, we now need to wire up the Button's press handler and make the native call. Add the following methods to the class:

```
greetUserCallback = () => {
    const state = this.state;

    HelloManager.greetUser
    (state.userName, state.isAdmin,  this.displayResult);
}

displayResult = (result) => {
    this.refs.userName.blur();
    this.setState({ greetingMessage: result });
}
```

How it works…

This recipe covered the foundation for a lot of what we will be doing with React Native on Android, exposing native functionality. All native module classes need to extend `ReactContextBaseJavaModule` and implement the constructor and the `getName` method. Methods that we want to expose need to have the `@ReactMethod` annotation. Creating a React Native Android native module has more overhead in comparison to doing so on iOS, as you have to wrap your module in a class that implements `ReactPackage` and register its package with the application. This is done in steps 7 and 8.

The interaction of this recipe comes together in the last step. The `greetUserCallback` function is executed when you press the `Button`. It makes a call to `HelloManager.greetUser`, then passes in the `userName` and `isAdmin` properties from the state, and the `displayResult` function as a callback. The `displayResult` sets the new `greetingMessage` on the state, causing a refresh of the UI and now the message is displayed.

Rendering custom Android view components

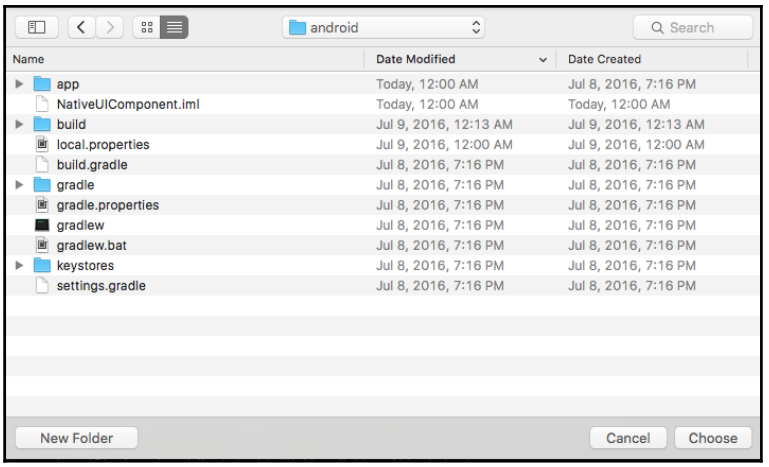
A major reason why React Native is so popular today is its ability to render truly native UI components. With native UI components on Android, we are leveraging not only the GPU rendering power, but we get the native look and feel of the components. This includes not only font and colors, but animations such as the ripple effect as well. In web or hybrid applications, there are CSS polyfills to simulate a native animation; in React Native we're getting the real deal, so to speak.

For this recipe, we created a React Native application titled `NativeUIComponent`.

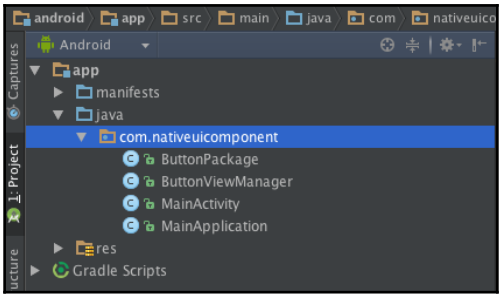
In this recipe, we will take a native `Button` and expose it as a React Native view component. You will be able to set the button label and attach a handler for when it is tapped.

How to do it…

1. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project:



2. Open up the left-hand Project explorer and expand the package structure until you can see the Java source files (for example `app/java/com.nativeuicomponent`):



3. Right-click on the package and select **New** | **Java Class**. Use `ButtonViewManager` for the class name and leave the **Kind** field as **Class**.
4. Repeat step 3 for the `ButtonPackage` class.
5. Let's begin implementing our `ButtonViewManager`. While native modules have a convention of having a suffix of module or manager, view components have a suffix of `ViewManager`. They must be subclasses of `SimpleViewManager<View>`. So let's open up our `ButtonViewManager` class and instruct it to render a `Button`.

We're going to need to import our necessary dependencies for the class:

```
import android.view.View;

import android.widget.Button;

import com.facebook.react.bridge.Arguments;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.uimanager.SimpleViewManager;
import com.facebook.react.uimanager.ThemedReactContext;
import com.facebook.react.uimanager.annotations.ReactProp;
import com.facebook.react.uimanager.events.RCTEventEmitter;
```

Now we can implement our `ButtonViewManager`; first we have to subclass `SimpleViewManager` and implement the `View.OnClickListener` interface so we can handle button presses:

```
public class ButtonViewManager extends
SimpleViewManager<Button> implements View.OnClickListener{
    @Override
    public String getName() {
        return "ButtonView";
    }
}
```

We are ready to instantiate our `Button`, set its label, and attach the tap handler. In the `ButtonViewManager` class, add the following methods:

```
@Override
protected Button createViewInstance
(ThemedReactContext reactContext) {
    Button button = new Button(reactContext);
    button.setOnClickListener(this);
    return button;
}
@ReactProp(name = "buttonText")
public void setButtonText
(Button button, String buttonText) {
    button.setText(buttonText);
}
@Override
public void onClick(View v) {
    WritableMap map = Arguments.createMap();
    ReactContext reactContext = (ReactContext) v.getContext();
    reactContext.getJSModule(RCTEventEmitter.class)
    .receiveEvent(v.getId(), "topChange", map);
}
```

6. Just like with native modules, we need to add our `ButtonViewManager` to the `ButtonPackage`, except here we're putting it in as a `ViewManager` and not a `NativeModule`:

```
public class ButtonPackage implements ReactPackage {

    @Override
    public List
<Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }

    @Override
    public List
<ViewManager>
createViewManagers
(ReactApplicationContext reactContext) {
        return Arrays.<ViewManager>asList
        (new ButtonViewManager());
    }

    @Override
    public List
<NativeModule>
createNativeModules
(ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }
}
```

7. To wrap up the Java side of things, we need to add the `ButtonPackage` to our `MainApplication` packages. Open `MainApplication.java` and add the following to `getPackages`:

```
@Override
protected List<ReactPackage> getPackages() {
  return Arrays.<ReactPackage>asList(
      new MainReactPackage(),
      new ButtonPackage()
  );
}
```

8. Let's build out our UI to render this button. We're going to need to our `Button` in its own React component. Create a file `Button.js` in your project's root directory (where `index.android.js` is located) and add the following code:

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

export default class Button extends Component {
    onChange = (event) => {
        if(this.props.onTap) {
            this.props.onTap(event.nativeEvent.message);
        }
    }

    render() {
        return <ButtonView
        {...this.props} onChange={this.onChange} />;
    }
}

Button.propTypes = {
    buttonText : React.PropTypes.string,
    onTap : React.PropTypes.func,
    ...View.propTypes
};

var ButtonView = requireNativeComponent('ButtonView', Button, {
    nativeOnly: { onChange: true }
});
```

9. Open our `index.android.js` file and add the following code after the `import { ... } from 'react-native'` block:

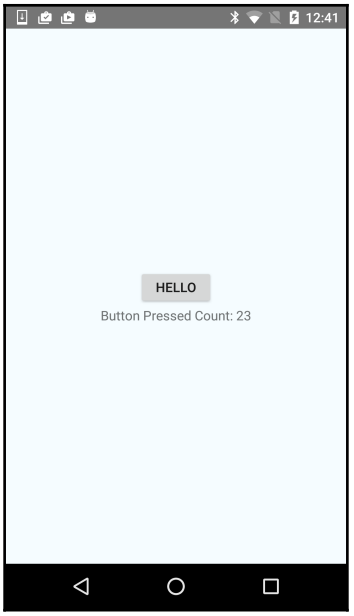
```
import Button from './Button';
```

   Add the following to the class implementation:

```
componentWillMount() {
    this.setState({
      count : 0
    });
  }
  onButtonTap = () => {
    let count = this.state.count;
    this.setState({
      count : count+1
    });
```

```
  }

  render() {
    return (
      <View style={styles.container}>
        <Button buttonText=
        "Hello Packt" onTap=
        {this.onButtonTap} style={{height: 40, width: 80}}/>
        <Text>Button Pressed Count: {this.state.count}</Text>
      </View>
    );
}
```

Your application should look and function like this:



How it works…

Here we created a React Native UI component for Android. You may find yourself having to create a good amount of native UI components if you want to have the true native Android experience from material design in your application. Currently, there are more third-party native UI components available for iOS, since it's been available to the public for longer. Therefore, you may be put in a situation where you will have to create your own. Fortunately, you can always refer back to this recipe to guide you.

The important bit in creating native UI components is that the `ViewManager` class has to extend `SimpleViewManager` and has to render a type that extends `View`. In our recipe, we render a `Button` view. For setting properties, we have to use the `@ReactProp` annotation. When we need to communicate back to the JavaScript layer, we fire an event from the native component. We implement this in step 5 of this recipe. In step 8, we create our `onChange` listener in which we execute the event handler we originally passed in.

Handling the Android back button

An important feature that all Android devices have is a dedicated back button. By default, the behavior of this back button is to go to the previous running activity. Since React Native generally only has one main React Activity, this will take you out of your application. Sometimes we may not want to have this default behavior, and use the back button inside our React Native application.

This recipe will create a simple example using React Native's latest navigation implementation `NavigationExperimental` and attach a back button listener to step back through the navigation stack.

Getting ready

For this recipe, it may be a good idea to familiarize yourself with the `NavigationExperimental` documentation. We have demonstrated this in the *Adding navigation to your application* recipe in `Chapter 1`, *Getting Started*.

For this recipe, we created a React Native application titled `BackButton`.

How to do it…

1. In order to make a meaningful example of using the Android back button, we're going to have to implement `NavigationExperimental` for our application. Let's start off by making a `Button` component that will be used to manually between navigation states. Create a file `Button.js` in your project's root directory (where `index.android.js` is located) and add the following code:

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

import {
```

```
  StyleSheet,
  Text,
  View,
  PixelRatio,
  TouchableOpacity
} from 'react-native';

export default class Button extends Component {
  render() {
    return (
      <TouchableOpacity
          style={styles.row}
          onPress={this.props.onPress}>
          <Text style={styles.buttonText}>
              {this.props.text}
          </Text>
      </TouchableOpacity>
    )
  }
}

const styles = StyleSheet.create({
  row: {
    padding: 15,
    backgroundColor: 'white',
    borderBottomWidth: 1 / PixelRatio.get(),
    borderBottomColor: '#CDCDCD',
  },
  buttonText: {
    fontSize: 17,
    fontWeight: '500',
  }
});
```

2. Now we're going to need a scene that will be rendered by the Navigator. This scene will hold the Button we created in the previous step. Create a file, `InitialScene.js` and add the following:

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

import {
  StyleSheet,
  Text,
  View,
  PixelRatio,
  ScrollView
} from 'react-native';
```

```
import Button from './Button';
export default class InitialScene extends Component {
  render () {
    return (
      <ScrollView style={styles.scrollView}>
        <Text style={styles.row}>
          Route: {this.props.route.key}
        </Text>
        <Button
          text="Next"
          onPress={this.props.onPushRoute}
        />
        <Button
          text="Back"
          onPress={this.props.onPopRoute}
        />
      </ScrollView>
    );
  }
}
```

You can add the following styles at the end of the file to give the scene some styling:

```
const styles = StyleSheet.create({
  navigator: {
    flex: 1,
  },
  scrollView: {
    marginTop: 64
  },
  row: {
    padding: 15,
    backgroundColor: 'white',
    borderBottomWidth: 1 / PixelRatio.get(),
    borderBottomColor: '#CDCDCD',
  },
  rowText: {
    fontSize: 17,
  },
  buttonText: {
    fontSize: 17,
    fontWeight: '500',
  },
});
```

3. The next step is to implement the actual Navigator itself. This will contain the `NavigationCardStack`, instructions on how to render a scene, and most importantly in this case, our back button event handler. Let's get right into it; create a `Navigator.js` file and add the following:

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

import {
  StyleSheet,
  Text,
  View,
  NavigationExperimental
} from 'react-native';

import InitialScene from './InitialScene';

const {
  CardStack : NavigationCardStack
} = NavigationExperimental;
export default class Navigator extends Component {
  constructor(props, context) {
    super(props, context);

    this._onPushRoute =
    this.props.onNavigationChange.bind(null, 'push');
    this._onPopRoute =
    this.props.onNavigationChange.bind(null, 'pop');
  }


  _renderScene = (sceneProps) => {
    return (
      <InitialScene
        route={sceneProps.scene.route}
        onPushRoute={this._onPushRoute}
        onPopRoute={this._onPopRoute}
      />
    );
  }

  render() {
    return (
      <NavigationCardStack
        onNavigationBack={this._onPopRoute}
        navigationState={this.props.navigationState}
        renderScene={this._renderScene}
        style={{flex:1}}
```

```
        />
      );
    }
}
```

Here we're rendering our `InitialScene` component that we created and putting it on the `NavigationCardStack`.

4. Now it's time to add support for the back button. We will be modifying our Navigator since it holds the prop that handles navigation state change.

   First we need to import `BackAndroid` from `'react-native'`, so add the field to the import block after `NavigationExperimental`.

   At the end of the constructor, we're going to add the event listener for the back button:

```
BackAndroid.addEventListener
('hardwareBackPress', this.onAndroidBackPress);
```

   Lastly, we need to implement the callback that we passed into the event listener:

```
onAndroidBackPress = () => {
  if(this.props.navigationState.index) {
    this._onPopRoute();
    return true;
  }
  return false;
}
```

5. To tie this all together, we need to render the Navigator from our applications entry class. We will start by importing the necessary dependencies, open `index.android.js`:

   We need to import `NavigationExperimental` from `'react-native'`, our `Navigator` component, and get a reference to `StateUtils` from `NavigationExperimental`:

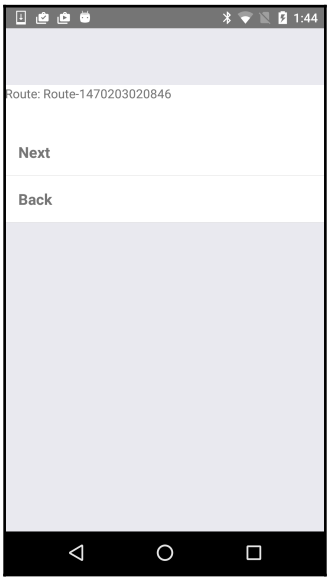
```
import {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  NavigationExperimental
} from 'react-native';
```

```
const {
  StateUtils : NavigationStateUtils
} = NavigationExperimental;

import Navigator from './Navigator';
```

Then to render the Navigator, overwrite the implementation of the application class with the following:

```
componentWillMount() {
  this.state = {
    navigationState : {
      index : 0,
      routes : [
        {key : 'Initial'}
      ]
    }
  };
}

_onNavigationChange = (type) => {
  let navigationState = this.state.navigationState,
      newNavigationState;
  switch (type) {
    case 'push':
      newNavigationState =
      NavigationStateUtils.push(navigationState, {
        key : 'Route-'+Date.now()
      });
      break;
    case 'pop':
      newNavigationState =
      NavigationStateUtils.pop(navigationState);
      break;
  }
  if(newNavigationState !== navigationState) {
    this.setState({navigationState : newNavigationState});
  }
}

render() {
  return (
    <Navigator
      navigationState={this.state.navigationState}
      onNavigationChange={this._onNavigationChange}
    />
  );
}
```

Your application should look as shown in the following screenshot, and you should be able to use the back button to navigate up the stack:



How it works…

The Android back button handler is quite trivial to implement. It should, however, be useful in a meaningful way that feels natural to the application. The implementation occurs in step 4. To prevent the default action, in this case it would exit out of the application, we have to return `true` from our event listener.

Reacting to changes in application state

The average mobile device user has several applications that they use on a consistent basis. These can be their favorite social media platform, their second favorite, a game, a media player, and hopefully your React Native app. This user may spend a short time in each application because he or she is constantly jumping across applications. What if we wanted to react to when the user leaves our app and re-enters? Maybe we need to sync data with the server or just tell the user that we're happy to see him/her use the app again and please rate us on the app store.

This recipe will show how we can react to changes in application state, that is, when the app is in the background, inactive, or active (foreground). It is applicable to both iOS and Android.

For this recipe, we created a React Native application titled `ApplicationState`.

How to do it…

1. Fortunately, React Native provides support for listening to changes through the `AppState` module. Open your `index.ios.js` or `index.android.js`. The code will be identical for both platforms.
2. Add `AppState` to the `import { ... } from 'react-native'` statement.
3. In the recipe, we're going to keep track of the previous state to see where the user came from. If it's their first time entering the app, we will welcome them, if they are returning, we will welcome them back. To do so, we need to keep a reference to previous and `currentAppStates`:

```
class ApplicationState extends Component {
  previousAppState : undefined
  currentAppState : undefined
```

4. We also need to store our status message on the state and display it on the view. Add the following functions:

```
componentWillMount() {
  this.currentAppState = 'active';
  this.setState({
    appStatus: 'Welcome!'
  });
}

render() {
  return (
    <View style={styles.container}>
      <Text style={styles.welcome}>
        {this.state.appStatus}
      </Text>
    </View>
  );
}
```

5. Currently, we're not doing anything with the application state. We are just displaying `Welcome`. Let's change that.

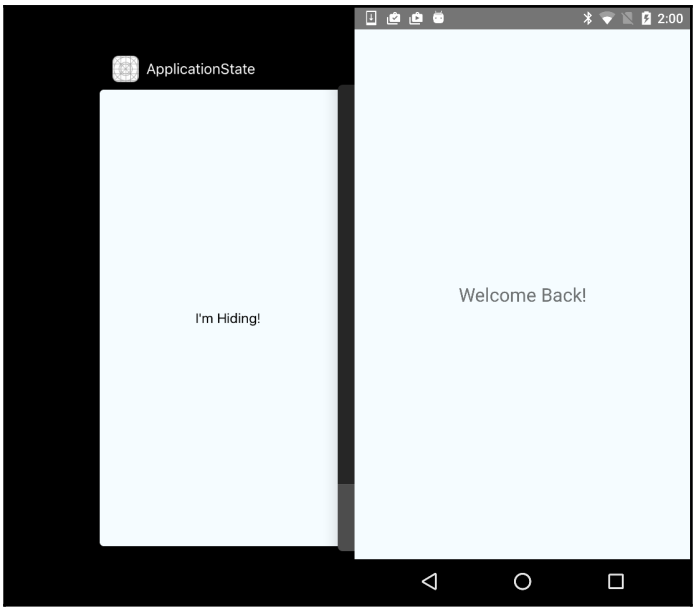
   In your `componentWillMount` function, add the following to register an event listener on `AppStateChange`:

```
AppState.addEventListener('change', this.onAppStateChange);
```

   Now we need to implement the event handler that will set the `appStatus` on the state, based on what the new application state is:

```
onAppStateChange = (appState) => {
  let appStatus;
  this.previousAppState = this.currentAppState;
  this.currentAppState = appState;
  switch(appState) {
    case 'inactive':
      appStatus = "I'm Hiding!";
      break;
    case 'background':
      appStatus = "I'm Hidden";
      break;
    case 'active':
      appStatus = 'Welcome Back!'
      break;
  }
  this.setState({appStatus});
}
```

   This is how the app will look on both iOS and Android:



How it works…

In this recipe, we use the built-in `AppState` module. The module listens to the `Activity` events on Android. On iOS it uses `NSNotificationCenter` to register a listener on various `UIApplication` events. Both platforms support the `active` and `background` states, whereas the `inactive` state is currently iOS only.

There's more…

Android does not explicitly support the `inactive` state due to its multitasking implementation. It goes between `background` and `active` states. To do the equivalent of the iOS inactive state on Android, see the recipe on *Hiding application content when multitasking*.

Copy and pasting content

One of the most used features in both desktop and mobile operating systems is the clipboard to copy and paste content. A scenario may be the user filling out a form in your awesome React Native app, where there is a field for a contact's e-mail address, a particularly lengthy and cumbersome one. Instead of typing it with a few typos, it would be easier to just open your contacts application and copy the e-mail from there and paste it into your `TextInput` field.

This recipe will show a basic example on both Android and iOS of how we can copy and paste text inside our React Native application. In our sample app, we will have both a static `Text` view and a `TextInput` field that you can copy its contents to the clipboard. Also, there will be a button that outputs the contents of the clipboard to the view.

Getting ready

For this recipe, we created a React Native application entitled `CopyPaste`.

In this recipe, we will use the `react-native-button` and `react-native-tooltip` libraries. To install them, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button react-native-tooltip --save
$ rnpm link react-native-tooltip
```

How to do it…

1. Let's start off by creating a component that takes a `Text` view and puts its content onto the `Clipboard`. We'll be building both iOS and Android versions of this component. The implementation is slightly different as we can render the native iOS tooltip easily thanks to the `react-native-tooltip` module, while Android will have to settle with using a long-press. You are encouraged though to create your own tooltip component, maybe of the native Android tooltip, to mimic the same experience on both platforms.

Create a file `ClipboardText.ios.js` in your root project directory (where your `index.ios.js` is located). First, we're going to import our dependencies; take note of the `Clipboard` module from `react-native` as well as the `ToolTip` that we installed earlier:

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

import {
  Clipboard,
  Text,
  View
} from 'react-native';

import ToolTip from 'react-native-tooltip';
```

Now we're going to implement the class. We are going to have a `useTooltip` prop flag that will decide whether we should show the native tooltip on long-press, or to just put the string onto the clipboard as we will for Android:

```
export default class ClipboardText extends Component {
  propTypes : {
    useTooltip : PropTypes.bool
  }

  onClipboardCopy = () => {
    const sourceText = this.refs.sourceText.props.children;

    Clipboard.setString(sourceText);
  }

  render() {
    const toolTipActions = [
      {
        text : 'Copy',
        onPress : this.onClipboardCopy
      }
    ];
    if(this.props.useTooltip) {
      return (
        <View>
          <ToolTip
            actions={toolTipActions}
            underlayColor='transparent'
            longPress={true}
            arrowDirection='down'>
              <Text ref="sourceText" {...this.props}/>
```

```
          </ToolTip>
        </View>
      );
    }
    return (
      <Text ref="sourceText" onLongPress={this.onClipboardCopy}
{...this.props}/>
    )
  }
}
```

Let's create the Android-specific `ClipboardText` component now. Create a file, `ClipboardText.android.js`, in the same location you did for step 1 and add the following code:

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

export default class ClipboardText extends Component {
  onClipboardCopy = () => {
    const sourceText = this.refs.sourceText.props.children;

    Clipboard.setString(sourceText);
  }

  render() {
    return (
      <Text ref="sourceText" onLongPress={this.onClipboardCopy}
{...this.props}/>
    );
  }
}
```

2. Now we can build our application view that will render our `ClipboardText` component, as well as a `TextInput`, and finally a way to output the clipboard contents. We'll start with iOS first so open up `index.ios.js` and add the following imports:

```
import {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  Clipboard,
  TextInput
} from 'react-native';

import Button from 'react-native-button';

import ClipboardText from './ClipboardText';
```

3. Before we figure out how we're going to render the page, we need to think about how we're going to store the clipboard contents, and more importantly retrieve them. Add the following functions to the `CopyPaste` class:

```
componentWillMount() {
  this.getClipboardContent =
  this.getClipboardContent.bind(this);
  this.setState({
    clipboardContent : undefined
  });
}

async getClipboardContent() {
  const clipboardContent = await Clipboard.getString();

  this.setState({
    clipboardContent
  });
}
```

4. Now that we have the plumbing done for copy and pasting, let's render the view to support it. Add the following to your `render` method:

```
<ClipboardText useTooltip=
{true} style=
{styles.header}>React Native Cookbook</ClipboardText>
```

If you want to add the `TextInput` that you can access the clipboard directly from, you can add it this way:

```
<TextInput style={styles.textInput} />
```

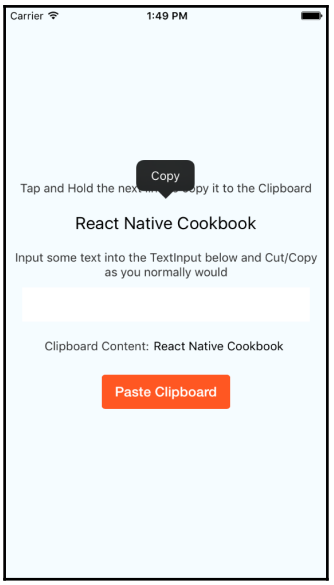
Finally, we need to display the clipboard contents and a `Button` to trigger the output of the clipboard to the state:

```
<View style={styles.row}>
  <Text style={styles.rowText}>
    Clipboard Content:
  </Text>
  <Text style={styles.content}>
    {this.state.clipboardContent}
  </Text>
</View>
<Button
  containerStyle={styles.buttonContainer}
  style={styles.buttonStyle}
  onPress={this.getClipboardContent}>
    Paste Clipboard
</Button>
```

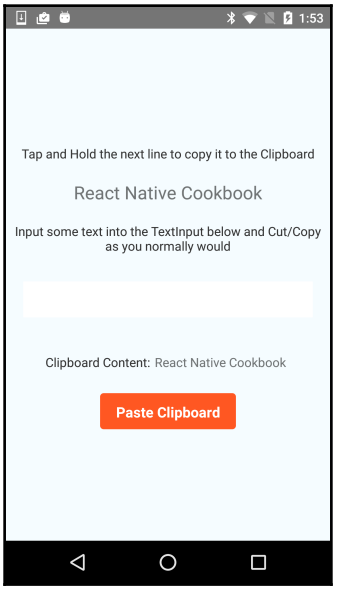
5. For Android, repeat steps 2-4. However, all the work will be done in `index.android.js`. The only other note is that since our `ClipboardText` component does not support rendering the tooltip for Android, we should remove the `useTooltip` prop arguments:

```
<ClipboardText style={styles.header}>React Native
Cookbook</ClipboardText>
```

If you build the sample application from the source code, your iOS app should look like this:



And your Android app will look like this:



How it works…

In this recipe, we built a simple component and application, wrapping the built-in `Clipboard` API. The `Clipboard` module currently only supports content of type `String`, even though the devices can copy and paste anything from images to map locations.

In step 1, we created our `TextInput Clipboard` wrapping component. For iOS, we were able to enhance it to show a native menu with our custom options to mimic the experience we see in a `WebView`. On Android, there is no existing third-party menu UI component available. In step 3, we handle retrieving the content in the `Clipboard`. It is an asynchronous process that we wrap using `async/await` to make it feel synchronous.

Receiving push notifications

In our ever-connected mobile world, it is important for our apps to encourage and entice us to open them. They should communicate that there is some piece of information waiting for us if we open the app that we must be aware of immediately. This is where the concept of push notifications comes in. Push notifications have existed since 2009 on iOS and 2010 for Android, and are frequently used by applications to let the user know of a change that they should react to. This could be a time-duration process finished, or a message has been posted directed at the user; the use cases are endless.

This recipe focuses solely on working with push notifications on both iOS and Android. We are going to register our app with Apple APNS and Google GCM and successfully receive push notifications.

Getting ready

This recipe assumes you have a working push notification server that communicates with APNS and GCM. In the recipe, we will output the device token for you to register with your server.

For this recipe, we created a React Native application titled `PushNotifications`.

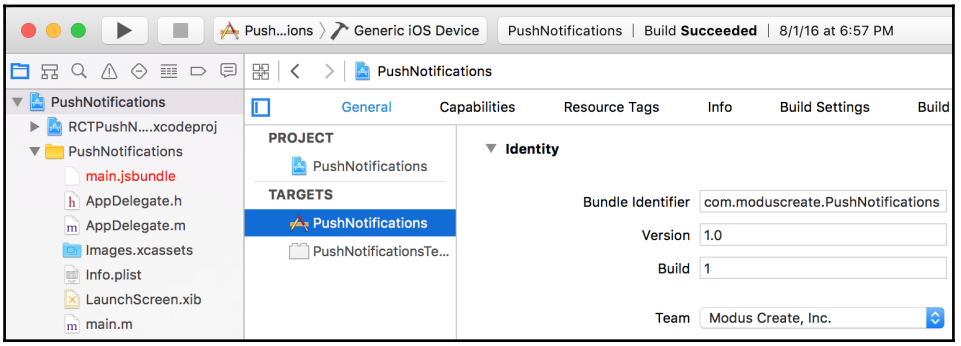
In this recipe, we will use the `react-native-push-notifications` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-push-notification --save
$ react-native link
```

Note: Push notifications for iOS require a physical device, you cannot test this on a simulator. For Android, your AVD must have Google APIs installed.

How to do it…

1. Before we begin writing React Native code to receive notifications, we have to add the push notifications entitlement to our App ID. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./PushNotifications/ios/PushNotifications.xcodeproj`):
2. Before proceeding with the entitlement configuration, make sure you change your **Bundle Identifier** to be meaningful. Select the **PROJECT** in the project navigator sidebar, then select the application target. Under **Bundle Identifier**, change it to be more meaningful. Generally the format is `com(org|net|...).companyname.appname`:



3. Please refer to the following URL for adding the push notification entitlement and generating a certificate for your server: `https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AddingCapabilities/AddingCapabilities.html#//apple_ref/doc/uid/TP40012582-CH26-SW6`

4. Now that our React Native app is ready to receive Apple push notifications, let's write some code to do so. Open up your `index.ios.js` file and add the following import:

```
import PushNotification from 'react-native-push-notification';
```

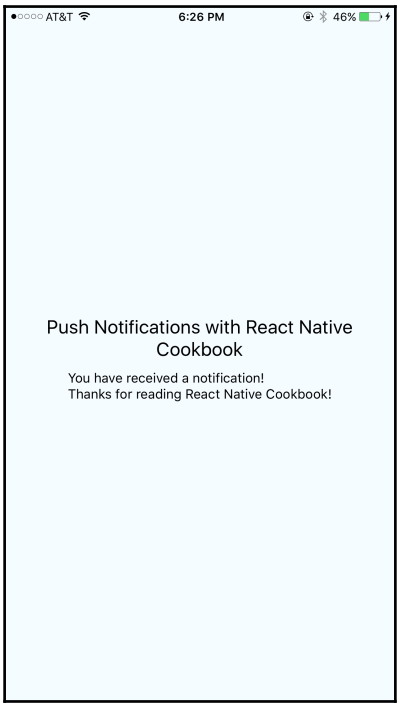
5. Next, we need register for push notifications and in our sample app we will display the message on the view:

```
componentWillMount() {
  this.setState({
    notification : undefined
  });

  PushNotification.configure({
    onRegister : function (token) {
      console.log('register', token);
    },
    onNotification : this.onNotificationReceived
  });
}

onNotificationReceived = (notification) => {
  this.setState({
    notification : notification.message
  });
}
```

6. Now if you want to display the notification message when it is received, just add the following to your `render` method:

```
<Text>{this.state.notification}</Text>
```

This is how our sample app looks running on a device:



7. It's time to move on to Android. This part is a bit more involved as there is currently no out-of-the-box support from React Native.
8. Follow the **Create an API project** to get a configuration file and add the configuration file to your project, ignoring the Gradle instructions, from `https://developers.google.com/cloud-messaging/android/client`.

   Make sure to store your server API key and sender ID somewhere for easy retrieval.

9. Go to the following URL and follow the instructions on Android Installation (`https://github.com/zo0r/react-native-push-notification/blob/master/README.md#android-installation`):

10. One feature that we may want to implement on Android that already exists on iOS is to open up our app when the user taps a notification. Open `AndroidManifest.xml` (it's located in `./app/src/main`) and add the following to our `MainActivity` definition:

```
<intent-filter>
    <action android:name="OPEN_ACTIVITY_1" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

11. When you send notifications from your server, you have to ensure that you set the `click_action` to `'OPEN_ACTIVITY_1'`:

```
message.addNotification({
    title : 'Notification',
    body : 'Thanks for reading React Native Cookbook!',
    icon : 'ic_launcher',
    click_action : 'OPEN_ACTIVITY_1'
});
```

12. Now that we're all set setting up our Android project, we can work on the UI. Repeat steps 4-6 for `index.android.js`.
13. To register with the GCM service, we must make two changes to our `PushNotification.configure` method. Add the following two properties:

```
senderID           : '<SenderIDFromGoogle>',
requestPermissions : true
```

Now you should be able to successfully receive push notifications on Android and open your application when you tap the notification.

How it works…

Push notifications are one of the more advanced recipes that we will be covering in this chapter. The level of difficulty involves more in setting up our environments, rather than reacting to them in our React Native application. Steps 3 and 8 focus on setting up our development servers.

One important distinction between iOS and Android is the action that occurs when the user taps on a notification. iOS defaults to open the application, to replicate this feature we needed to add an `intent-filter`, as seen in step 11 and 12. If you do not want to auto-launch your application, then skip step 11 and do not send the `click_action` in your notification payload.

Authenticating via TouchID or fingerprint sensor

These days, security is paramount in our applications, especially when there is any sort of authentication. We hear about security breaches and leaked passwords daily in the news, and companies are looking to implement added security measures in their apps. One such measure may be fingerprint scanning via the iOS TouchID sensor or Android fingerprint sensor. These are pretty secure mechanisms for authenticating with applications. Since React Native can easily access the native layer and allow us to create native modules with ease, there's no reason why we can't have fingerprint authentication in our apps for both iOS and Android.

This recipe covers how we can go about scanning our fingerprint using either the TouchID sensor or the Android fingerprint sensor (if it is equipped on your device, and you are deploying for API Level 23+) and have the fingerprint verified.

Getting ready

For this recipe, we created a React Native application entitled `FingerPrint`.

In this recipe, we will use the `react-native-button` and `react-native-touch-id` libraries. To install them, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button react-native-touch-id --save
```

How to do it…

1. On the iOS side of things, we are in a fortunate position in that there is a community module already out there to handle `TouchID` authentication. Open `index.ios.js` and import our modules:

```
import Button from 'react-native-button';
import TouchID from 'react-native-touch-id';
```

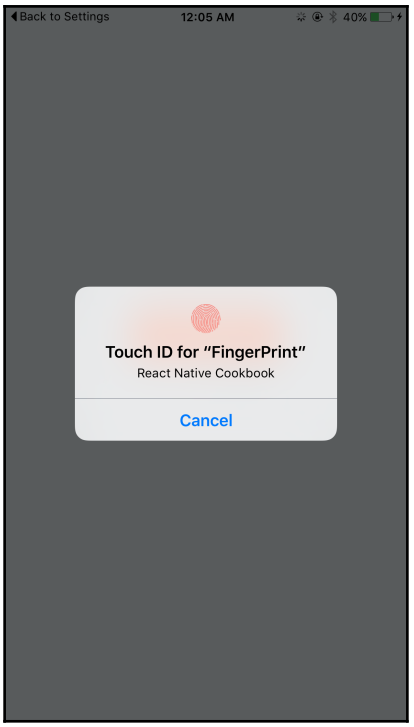
2. Now, in the class, we're going to need to set up our initial state that will hold the authentication status and an event handler for our button that will tell the OS to authenticate with TouchID. Add the following methods to your main class:

```
componentWillMount() {
  this.setState({
    authStatus : undefined
  });
}
onButtonPress = () => {
  TouchID.authenticate('React Native Cookbook')
    .then(this.onAuthSuccess)
    .catch(this.onAuthFailure);
}
onAuthSuccess = (success) => {
  this.setState({
    authStatus : 'Authenticated'
  });
}
onAuthFailure = (failure) => {
  this.setState({
    authStatus : 'Not Authenticated'
  });
}
```

3. For our sample app, let's render our button that will start the authentication and a way to display the `authStatus`. Implement the following `render` method:

```
render() {
  return (
    <View style={styles.container}>
      <Button
        containerStyle={styles.buttonContainer}
        style={styles.buttonStyle}
        onPress={this.onButtonPress}>
          Authenticate
      </Button>
      <Text style={styles.instructions}>Authentication
      Status</Text>
      <Text style={styles.welcome}>{this.state.authStatus}
      </Text>
    </View>
  ); }
```

This is what we should see when we run our app:



4. Adding fingerprint authentication will require us to create our own native module. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project.
5. Right-click on the main package (`com.fingerprint`) and select **New | Java Class**. Use `Fingerprint` for the class name and leave the **Kind** field as **Class**.
6. Repeat step 3 for the `FingerprintModule` and `FingerprintPackage` classes.
7. Open the `Fingerprint` class and add the following imports:

```
import android.hardware.fingerprint.FingerprintManager;
import android.os.CancellationSignal;

import com.facebook.react.bridge.Arguments;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.modules.core
.DeviceEventManagerModule;
```

8. Now we're going to implement the class, our `Fingerprint` class should extend `FingerprintManager.AuthenticationCallback`:

```
private FingerprintManager fingerprintManager;
private FingerprintManager.CryptoObject cryptoObject;
private CancellationSignal cancellationSignal;
private ReactContext reactContext;
public Fingerprint(FingerprintManager fingerprintManager,
ReactContext reactContext) {
    this.fingerprintManager = fingerprintManager;
    this.reactContext = reactContext;
}
public void setCryptoObject(FingerprintManager
.CryptoObject cryptoObject) {
    this.cryptoObject = cryptoObject;
}
public void authenticate() {
    cancellationSignal = new CancellationSignal();
    fingerprintManager.authenticate
    (cryptoObject, cancellationSignal, 0, this, null);
    WritableMap params = Arguments.createMap();
    params.putString("authStatus", "Waiting for fingerprint");
    sendEvent("authStatus", params);
}
@Override
public void onAuthenticationError
(int errorCode, CharSequence errString) {
    WritableMap params = Arguments.createMap();
    params.putString("authStatus", "An error has occurred");
    sendEvent("authStatus", params);
}
@Override
public void onAuthenticationFailed() {
    WritableMap params = Arguments.createMap();
    params.putString("authStatus", "Invalid fingerprint");
    sendEvent("authStatus", params);
}
@Override
public void onAuthenticationSucceeded
(FingerprintManager.AuthenticationResult result) {
    WritableMap params = Arguments.createMap();
    params.putString
    ("authStatus", "Authentication Successful!");
    sendEvent("authStatus", params);
}
@Override
public void onAuthenticationHelp
(int helpCode, CharSequence helpString) {
```

```
}
private void sendEvent(String eventName, WritableMap params) {
reactContext.getJSModule
(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
            .emit(eventName, params);
}
```

9. Now we're going to need create our React Native module wrapper around the `Fingerprint` class. Open the `FingerprintModule` class and change it to the following:

```
public class FingerprintModule extends
ReactContextBaseJavaModule{

    private FingerprintManager fingerprintManager;
    private Fingerprint fingerprint;

    public FingerprintModule
    (ReactApplicationContext reactApplicationContext) {
        super(reactApplicationContext);

        fingerprintManager = (FingerprintManager)
        reactApplicationContext.getSystemService
        (Context.FINGERPRINT_SERVICE);
        fingerprint = new Fingerprint
        (fingerprintManager, reactApplicationContext);
    }

    @Override
    public String getName() {
        return "Fingerprint";
    }

    @ReactMethod
    public void authenticate() {
        fingerprint.authenticate();
    }
}
```

10. We're going to need to add our `FingerprintModule` to our `FingerprintPackage`:

```
public class FingerprintPackage implements ReactPackage {
    @Override
    public List
    <Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
```

```
    }

    @Override
    public List
    <ViewManager> createViewManagers
    (ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }

    @Override
    public List<NativeModule> createNativeModules(
            ReactApplicationContext reactContext) {
        List<NativeModule> modules = new ArrayList<>();

        modules.add(new FingerprintModule(reactContext));

        return modules;
    }
}
```

11. Finally, we need to register the `FingerprintPackage` with our application so we can use it in our React Native JavaScript. Open `MainApplication.java` and add the following to the `getPackages` method:

```
@Override
protected List<ReactPackage> getPackages() {
  return Arrays.<ReactPackage>asList(
      new MainReactPackage(),
      new FingerprintPackage()
  );
}
```

12. Let's go ahead and implement the `FingerprintModule` and our UI. Open `index.android.js` and import the following modules:

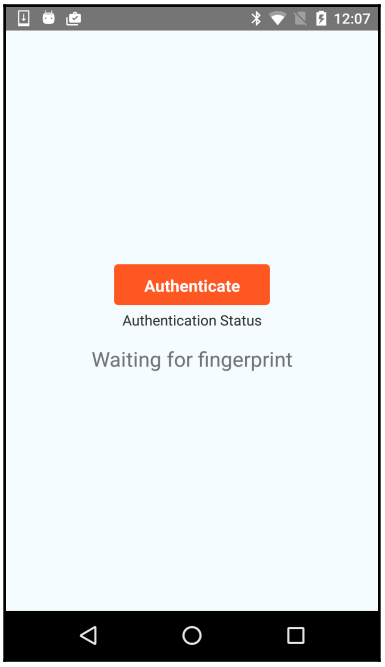
```
import {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  NativeModules,
  DeviceEventEmitter
} from 'react-native';

const {
  Fingerprint
} = NativeModules;
```

```
import Button from 'react-native-button';
```

13. Now we need to create and wire up the event handlers for authentication as we did in step 2. Since we're using our own native module, the code will be a bit different:

```
componentWillMount() {
  this.setState({
    authStatus : undefined
  });
  DeviceEventEmitter.addListener
  ('authStatus', this.onAuthStatus);
}
onAuthStatus = (eventBody) => {
  const authStatus = eventBody.authStatus;
  this.setState({authStatus});
}
onButtonPress = () => {
  Fingerprint.authenticate();
}
```

14. Fortunately, the `render` remains unchanged from the iOS version, so repeat step 3 and your application should look something like this:



How it works…

This recipe is an excellent example of when functionality is readily available and accessible on one platform, but not so much on the other. In this case, we use `react-native-touch-id` for handling the iOS Touch ID support, but since there is no available module for Android, we have no choice but to create our own. We leverage what we learned in the *Exposing custom Android modules* recipe and build a `FingerPrint` module in steps 4-11. Step 8 covers the actual implementation of the fingerprint scanning and reacting to the authentication states. In this case, all the logic occurs in the `FingerPrint` class and the `FingerPrintModule` just exposes a single `authenticate` method.

Hiding application content when multitasking

Keeping the theme of application security going, we have to be wary sometimes of unwanted eyes and hands touching our devices and potentially getting access to our applications. In order to protect the user from prying eyes while looking at sensitive information, we can mask our application when the application is hidden, but still active. Once the user returns to the application, we would simply remove the mask and the user can continue using the app as normal. A good example would be the Chase banking application; it hides your sensitive banking information when the application is not in the foreground.

This recipe will show you how to render an image to mask your application and remove it once the application returns to the foreground or active state. We will cover both iOS and Android; however, the implementation varies in its entirety. For iOS, we employ a pure Objective-C implementation for optimal performance. For Android, we're going to have to make some modifications to the `MainActivity` in order to send an event to our JavaScript layer that the application has lost focus. We will handle the rendering of the image mask there.

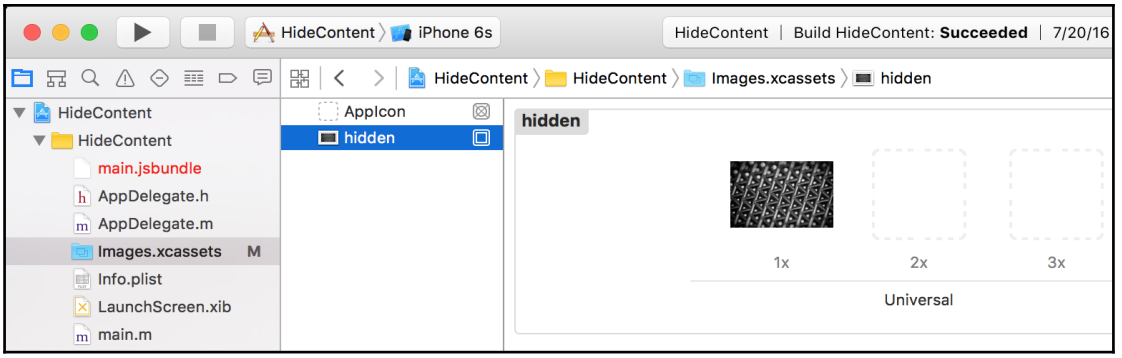
Getting ready

We're going to need to have an image handy that we want to use as the mask. In this recipe, we used an image with the filename `hidden.jpg`.

For this recipe, we created a React Native application titled `HideContent`.

How to do it…

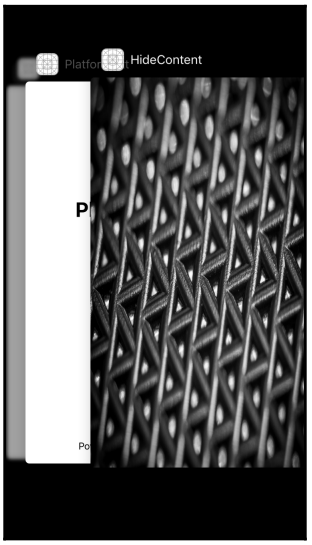
1. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./HideContent/ios/HideContent.xcodeproj`).
2. Open the `Images.xcassets` file and drag your image to the left sidebar:



3. Open `AppDelegate.m` and add the following instance variable and methods:

```
@implementation AppDelegate {
  UIImageView *imageView;
}

- (void)applicationWillResignActive:
(UIApplication *)application {
  imageView = [[UIImageView alloc]
  initWithFrame:[self.window frame]];
  [imageView setImage:[UIImage imageNamed:@"hidden.jpg"]];
  [self.window addSubview:imageView];
}

- (void)applicationDidBecomeActive:
(UIApplication *)application {
  if(imageView != nil) {
    [imageView removeFromSuperview];
    imageView = nil;
  }
}
```

Run your application and double-press the Home button or open the notifications center and you should see the following:



4. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project.
5. Open `MainActivity.java` and add the following method:

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if(getReactNativeHost()
    .getReactInstanceManager()
    .getCurrentReactContext() != null) {
        WritableMap params = Arguments.createMap();
        params.putBoolean("appHasFocus", hasFocus);
        getReactNativeHost().getReactInstanceManager()
                .getCurrentReactContext()
                .getJSModule(DeviceEventManagerModule
                .RCTDeviceEventEmitter.class)
                .emit("focusChange", params);
    }
}
```

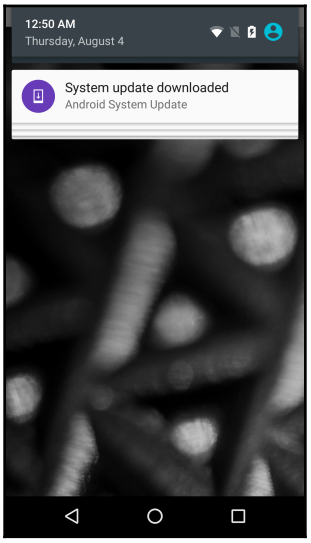
6. Copy our image (`hidden.jpg`) to the root of the React Native project (where `index.android.js` is located).
7. Open `index.android.js` and add the image and `DeviceEventEmitter` to the `import { ... } from 'react-native'` block.
8. Now we need to add an event listener for the `focusChange` event that will be fired from the `MainActivity`. Add the following code into our main class:

```
componentWillMount() {
  this.setState({
    showMask : undefined
  });
this.subscription =
DeviceEventEmitter.addListener
('focusChange', this.onFocusChange);
}
onFocusChange = (params) => {
  this.setState({showMask : !params.appHasFocus})
}
componentWillUnmount() {
  this.subscription.remove();
}
```

9. To render the image mask, we need to modify the `render` method to show the `Image` component. Add the following before the `return` statement in the `render` method:

```
if(this.state.showMask) {
  return (<Image source={require('./hidden.jpg')} />);
}
```

10. Run your application and either shake to open the developer menu or swipe down to view the control center, and you will see your image mask:



How it works…

In this recipe, we use two separate approaches for accomplishing the same task. For iOS, we handle showing the image mask in the native layer exclusively, without relying on React Native at all. The reason for this is to eliminate the round-trip over the bridge. The tradeoff is that we are writing platform-specific native code, instead of using the `inactive` state that is available to us. For Android, it is easier to rely on React Native and JavaScript to handle the image masking, therefore we write a hook in our `MainActivity` that will fire a custom event informing us that the application lost focus.

Background processing on iOS

Over the last several years, the functional gap for common applications between mobile and desktop has shrunk. Users are demanding richer experiences and more robust processing than the trivial applications that originally existed. To support this, the hardware had to be upgraded to support the software demands. Most mobile devices today are powered by multi-core processors, and the operating systems now provide developers easy abstractions to execute code in the background without locking the application.

This recipe will cover the use of iOS's **Grand Central Dispatch** to execute asynchronous background processing on a new thread and communicate back to our React Native application when the processing is complete.

Getting ready

For this recipe, we created a React Native application titled `Multithreading`.

In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button --save
```

How to do it…

1. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./Multithreading/ios/Multithreading.xcodeproj`).
2. Create a new Objective-C Cocoa class called `BackgroundTaskManager`.
3. Open `BackgroundTaskManager.h` and replace it with the following:

```
#import <Foundation/Foundation.h>
#import <dispatch/dispatch.h>
#import "RCTBridgeModule.h"

@interface BackgroundTaskManager : NSObject <RCTBridgeModule> {
  dispatch_queue_t backgroundQueue;
}

@end
```

4. Open `BackgroundTaskManager.m` and replace it with the following:

```
#import "BackgroundTaskManager.h"
#import "RCTBridge.h"
#import "RCTEventDispatcher.h"

@implementation BackgroundTaskManager

@synthesize bridge = _bridge;

RCT_EXPORT_MODULE();
```

```
RCT_EXPORT_METHOD(loadInBackground) {
  backgroundQueue = dispatch_queue_create
  ("com.moduscreate.bgqueue", NULL);
  dispatch_async(backgroundQueue, ^{
    NSLog(@"processing background");
    [self.bridge.eventDispatcher sendAppEventWithName:
    @"backgroundProgress" body:@{@"status": @"Loading"}];
    [NSThread sleepForTimeInterval:5];
    NSLog(@"slept");
    dispatch_async(dispatch_get_main_queue(), ^{
      NSLog(@"Done processing; main thread");
      [self.bridge.eventDispatcher sendAppEventWithName:
      @"backgroundProgress" body:@{@"status": @"Done"}];
    });
  });
}
@end
```

5. Open `index.ios.js` and add `NativeModules` and `NativeAppEventEmitter` to the `import { ... } from 'react-native'` block.
6. We also need to import the `Button` component and make a reference to our `BackgroundTaskManager` native module:

```
import Button from 'react-native-button';

const BackgroundTaskManager =
NativeModules.BackgroundTaskManager;
```

7. In our example, we will track the status of the background process, and demonstrate that this process does not block the UI by having another button that increments a counter. Let's prepare our event handlers for these buttons:

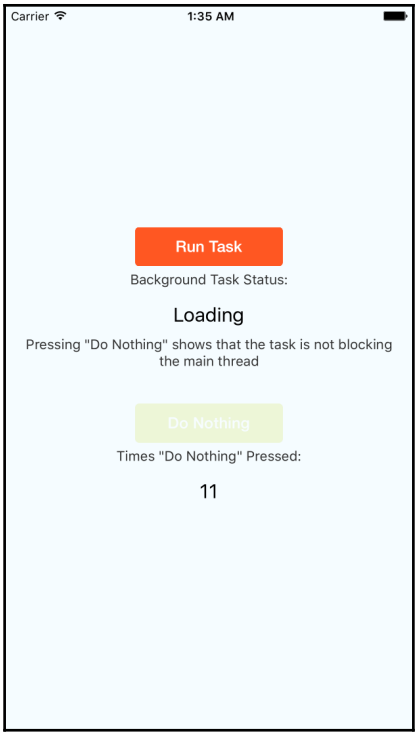
```
componentWillMount() {
  var me = this;
  me.setState({
    backgroundTaskStatus : "Not Started",
    doNothingCount : 0
  });
  this.subscription =
  NativeAppEventEmitter.addListener
  ('backgroundProgress', (e) => {
    const backgroundTaskStatus = e.status;
    me.setState({backgroundTaskStatus});
  });
}
onButtonPress() {
  BackgroundTaskManager.loadInBackground();
```

```
}
onDoNothingPress = () => {
  this.setState({
    doNothingCount : this.state.doNothingCount+1
  });
}
```

8. Now let's set up our UI to show the buttons, background process status, and counter. Change the `render` method to the following:

```
render() {
  return (
    <View style={styles.container}>
      <Button
        containerStyle={styles.buttonContainer}
        style={styles.buttonStyle}
        onPress={this.onButtonPress}>
          Run Task
      </Button>
      <Text style={styles.instructions}>
        Background Task Status:
      </Text>
      <Text style={styles.welcome}>
        {this.state.backgroundTaskStatus}
      </Text>
      <Text style={styles.instructions}>
        Pressing "Do Nothing" shows
        that the task is not blocking the main thread
      </Text>
      <Button
        containerStyle={
        [styles.buttonContainer, styles.altButtonContainer]}
        style={styles.buttonStyle}
        onPress={this.onDoNothingPress}>
          Do Nothing
      </Button>
      <Text style={styles.instructions}>
        Times "Do Nothing" Pressed:
      </Text>
      <Text style={styles.welcome}>
        {this.state.doNothingCount}
      </Text>
    </View>
  );
}
```

9. Run the application and you should see the following screen:



How it works

In this recipe, we create a native module as we did in the *Exposing custom iOS modules* recipe, but we perform some arbitrary execution in the background. In our case, we spawn a new thread, sleep it for 5 seconds, and then when it is done we let the UI know that the process has been completed. We leverage the operating systems' Grand Central Dispatch API to accomplish this.

An important part we highlight with the button we added in steps 7 and 8 is that if we were running on a single native thread, our application would lock while the background process is running. When you press a button, the bridge is invoked and messages are posted to the native layer. If the native thread is currently busy sleeping, then we cannot process the message. Fortunately, in our case we offloaded that processing using GCD to a new thread.

Background processing on Android

Multi-core processors were first introduced for Android in 2011. Generally, Android device manufacturers have boasted impressive specifications in their CPUs. It is only natural that the software should use the hardware to its full potential.

This recipe will cover using `AsyncTask` to execute short-running processes in the background without blocking the main thread.

Getting ready

For this recipe, we created a React Native application titled `Multithreading`.

In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button --save
```

How to do it…

1. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project.
2. Create two new Java classes: `BackgroundTaskManager` and `BackgroundTaskPackage`.
3. Open `BackgroundTaskManager.java` and let's begin implementing our native module that will wrap an `AsyncTask` operation. Import the following dependencies:

```
import android.os.AsyncTask;

import com.facebook.react.bridge.Arguments;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.modules
.core.DeviceEventManagerModule;
```

4. In order to execute an `AsyncTask`, we need to have a private class subclass it. Open `BackgroundTaskManager.java` and create a private inner `BackgroundLoadTask` class:

```
private class BackgroundLoadTask extends
AsyncTask<String, String, String> {
        @Override
        protected String doInBackground(String... params) {
            publishProgress("Loading");
            try {
                Thread.sleep(5000);
            } catch(Exception e){
                e.printStackTrace();
            }
            return "Done";

        }

        @Override
        protected void onPostExecute(String s) {
            WritableMap params = Arguments.createMap();
            params.putString("status", "Done");
            sendEvent("backgroundProgress", params);
        }

        @Override
        protected void onProgressUpdate(String... values) {
            WritableMap params = Arguments.createMap();
            params.putString("status", "Loading");
            sendEvent("backgroundProgress", params);
        }
    }
```

5. Now we need to have our native module provide a way for our JavaScript layer to start the `BackgroundLoadTask`. Implement the rest of the `BackgroundTaskManager` class:

```
public BackgroundTaskManager
(ReactApplicationContext reactApplicationContext) {
        super(reactApplicationContext);
    }

    @Override
    public String getName() {
        return "BackgroundTaskManager";
    }
```

```
    @ReactMethod
    public void loadInBackground() {
        BackgroundLoadTask backgroundLoadTask =
        new BackgroundLoadTask();
        backgroundLoadTask.execute();
    }

    private void sendEvent
    (String eventName, WritableMap params) {
        getReactApplicationContext().getJSModule
        (DeviceEventManagerModule.RCTDeviceEventEmitter.class)
        .emit(eventName, params);
    }
```

6. Open `index.android.js` and add `NativeModules` and `DeviceEventEmitter` to the `import { ... } from 'react-native'` block.
7. We also need to import the `Button` component and make a reference to our `BackgroundTaskManager` native module:

```
import Button from 'react-native-button';

const BackgroundTaskManager =
NativeModules.BackgroundTaskManager;
```

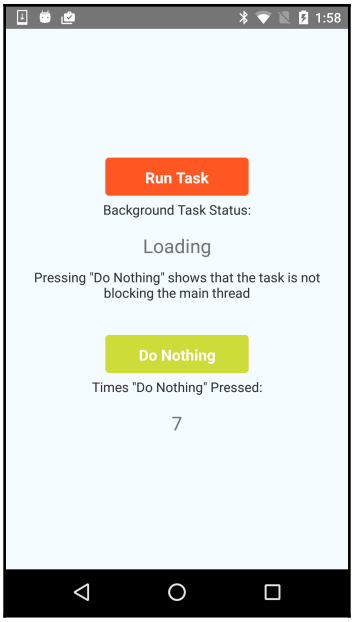
8. In our example, we will track the status of the background process, and demonstrate that this process does not block the UI by having another button that increments a counter. Let's prepare our event handlers for these buttons:

```
componentWillMount() {
  var me = this;
  me.setState({
    backgroundTaskStatus : "Not Started",
    doNothingCount : 0
  });
  DeviceEventEmitter.addListener('backgroundProgress', (e) => {
    const backgroundTaskStatus = e.status;
    me.setState({backgroundTaskStatus});
  });
}
onButtonPress() {
  BackgroundTaskManager.loadInBackground();
}
onDoNothingPress = () => {
  this.setState({
    doNothingCount : this.state.doNothingCount+1
  });
}
```

9. Now let's set up our UI to show the buttons, background process status, and counter. Change the render method to the following:

```
render() {
  return (
    <View style={styles.container}>
      <Button
        containerStyle={styles.buttonContainer}
        style={styles.buttonStyle}
        onPress={this.onButtonPress}>
          Run Task
      </Button>
      <Text style={styles.instructions}>
        Background Task Status:
      </Text>
      <Text style={styles.welcome}>
        {this.state.backgroundTaskStatus}
      </Text>
      <Text style={styles.instructions}>
        Pressing "Do Nothing" shows
        that the task is not blocking the main thread
      </Text>
      <Button
        containerStyle={
        [styles.buttonContainer, styles.altButtonContainer]}
        style={styles.buttonStyle}
        onPress={this.onDoNothingPress}>
          Do Nothing
      </Button>
      <Text style={styles.instructions}>
        Times "Do Nothing" Pressed:
      </Text>
      <Text style={styles.welcome}>
        {this.state.doNothingCount}
      </Text>
    </View>
  );
}
```

10. Run the application and you should see the following experience:



How it works…

In this recipe, we mimic the functionality we created in *Background processing on iOS* but for Android. We create an Android native module, perform some arbitrary execution in the background (sleep for 5 seconds), and update our UI upon completion. Android has several options for performing multithreaded operations natively. In our situation, we chose to use `AsyncTask` as it is geared towards short-running (several seconds) processes. It is also relatively simple to implement and the operating system manages thread creation and resource allocation.

Playing audio files on iOS

With the iPod and the digital music revolution, a digital portable media device became a must-have for even the most casual music listeners. The smartphone has superseded the media device, with many users storing large music libraries on their phones. It is only natural for other applications to want to access the music library.

In this recipe, we will create a native module to show the iOS mediaPicker and then select a music file to play. The file will play through the iOS native media player, allowing the music to play in the background and be controlled by the iOS control center.

Getting ready

For this recipe, we created a React Native application titled `AudioPlayer`.

In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button --save
```

Make sure you have music synced with your iOS device that is available in your media library.

How to do it…

1. Open the iOS Project in Xcode. The project file is located in the `ios/` directory of your React Native application (for example, `./AudioPlayer/ios/AudioPlayer.xcodeproj`).
2. Create a new Objective-C Cocoa class called `MediaManager`.
3. Open the `MediaManager` header (`.h`) file and replace it with the following imports and references for `MPMediaPickerController` and `MPMusicPlayerController`:

```
#import <Foundation/Foundation.h>
#import <MediaPlayer/MediaPlayer.h>
#import "RCTBridgeModule.h"
#import "RCTEventDispatcher.h"

@interface MediaManager : NSObject<RCTBridgeModule,
MPMediaPickerControllerDelegate>

@property (nonatomic, retain)
MPMediaPickerController *mediaPicker;
@property (nonatomic, retain)
MPMusicPlayerController *musicPlayer;
@end
```

4. First, we are going to need to tackle showing the native `MediaPicker` in our `MediaManager` implementation. Add the following methods to show and hide the `mediaPicker`:

```
#import "AppDelegate.h"

-(void)showMediaPicker {
  if(self.mediaPicker == nil) {
   self.mediaPicker =
   [[MPMediaPickerController alloc]
   initWithMediaTypes:MPMediaTypeAnyAudio];
    [self.mediaPicker setDelegate:self];
    [self.mediaPicker setAllowsPickingMultipleItems:NO];
    [self.mediaPicker setShowsCloudItems:NO];
    self.mediaPicker.prompt = @"Select song";
  }
  AppDelegate *delegate =
  (AppDelegate *)[[UIApplication sharedApplication] delegate];
  [delegate.window.rootViewController presentViewController:
  self.mediaPicker animated:YES completion:nil];
}

void hideMediaPicker() {
  AppDelegate *delegate =
  (AppDelegate *)[[UIApplication sharedApplication] delegate];
  [delegate.window.rootViewController
  dismissViewControllerAnimated:YES completion:nil];
}
```

5. Next, we're going to need to implement the two actions that the `mediaPicker` may have, picking a media item and cancelling the action. Add the following code to `MediaManager.m`:

```
@synthesize bridge = _bridge;
@synthesize musicPlayer;
-(void) mediaPicker:(MPMediaPickerController *)
mediaPicker didPickMediaItems:
(MPMediaItemCollection *)mediaItemCollection {
  MPMediaItem *mediaItem = mediaItemCollection.items[0];
  NSURL *assetURL =
  [mediaItem valueForProperty:MPMediaItemPropertyAssetURL];
  [self.bridge
  .eventDispatcher sendAppEventWithName:@"SongPlaying"
  body:[mediaItem valueForProperty:MPMediaItemPropertyTitle]];
  if(musicPlayer == nil) {
    musicPlayer = [MPMusicPlayerController systemMusicPlayer];
  }
```

```
  [musicPlayer setQueueWithItemCollection:mediaItemCollection];
  [musicPlayer play];

  hideMediaPicker();
}

-(void) mediaPickerDidCancel:
(MPMediaPickerController *)mediaPicker {
  hideMediaPicker();
}
```

6. We're going to need to expose our `MediaManager` to the React Native bridge and create a method that will be invoked to show the `mediaPicker`:

```
RCT_EXPORT_MODULE();
RCT_EXPORT_METHOD(showSongs) {
  [self showMediaPicker];
}
```

7. Open `index.ios.js` and add `NativeModules` and `NativeAppEventEmitter` to the `import { ... } from 'react-native'` block.
8. We also need to import the `Button` component and make a reference to our `MediaManager` native module:

```
import Button from 'react-native-button';

const MediaManager = NativeModules.MediaManager;
```

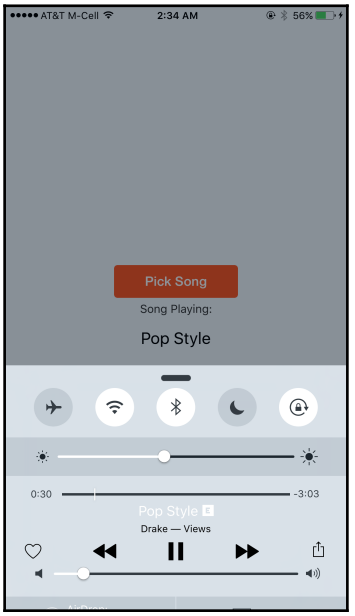
9. Our application will allow the user to pick a song and display the song name afterwards. Let's create the necessary functions that will handle the button press and setting the song title on the component's state:

```
componentWillMount() {
  this.setState({
    songPlaying : undefined
  });
  this.subscription = NativeAppEventEmitter.addListener
  ('SongPlaying', this.onSongPlaying);
}
onSongPlaying = (songPlaying) => {
  this.setState({songPlaying});
}
onShowSongsPress() {
  MediaManager.showSongs();
}
```

10. We are going to need to set up our UI with the `Button` and an area to display the song information. Implement the following `render` method:

```
render() {
  return (
    <View style={styles.container}>
      <Button
        containerStyle={styles.buttonContainer}
        style={styles.buttonStyle}
        onPress={this.onShowSongsPress}>
          Pick Song
      </Button>
      <Text style={styles.instructions}>Song Playing:</Text>
      <Text style={styles.welcome}>{this.state.songPlaying
      </Text>
    </View>
  );
}
```

11. We are ready to run our app now; pick a song and it will play!



How it works…

This recipe revolves around the Media Player iOS framework and wrapping its functionality in a native module. The Media Player framework allows us to access the native iPod library and play audio files from the library on the device using the same functionality as if you were to use `Music.app`. In step 4, we show the `mediaPicker` and handle the selection of a media item and subsequently playing the audio file using the `MediaPlayer` (`MPMusicPlayerController`) in step 5.

Playing audio files on Android

A benefit that Google likes to claim that Android has over iOS is flexibility in dealing with file storage. Android devices support external SD cards that can be filled with media files, and do not need a proprietary method of adding multimedia as iOS does.

In this recipe, we will use Android's native `mediaPicker`, which is started from an intent. We will then be able to pick a song and have it play through our application.

Getting ready

For this recipe, we created a React Native application titled `AudioPlayer`.

In this recipe, we will use the `react-native-button` library. To install it, run the following command in the terminal from your project root directory:

```
$ npm install react-native-button --save
```

Make sure you have music files available in your `Music/` directory on your Android device or emulator.

How to do it…

1. Open the Android project using Android Studio. In Android Studio, select **Open an existing Android Studio project** and open the `android` directory of the project.
2. Create two new Java classes: `MediaManager` and `MediaPackage`.

3. Our `MediaManager` will use intents to show the `mediaPicker`, `MediaPlayer` to play music, and `MediaMetadataRetriever` to parse metadata information from the audio file to send back to the JavaScript layer. Let's import all the dependencies first into our `MediaManager.java`:

```
import android.app.Activity;
import android.content.Intent;
import android.media.AudioManager;
import android.media.MediaMetadataRetriever;
import android.media.MediaPlayer;
import android.net.Uri;
import android.provider.MediaStore;

import com.facebook.react.bridge.ActivityEventListener;
import com.facebook.react.bridge.Arguments;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.modules.core.DeviceEventManagerModule;
```

4. Since we're creating a native module and starting intents, we need to add some boilerplate code as well as implement our `showSongs` method:

```
public class MediaManager extends ReactContextBaseJavaModule
implements ActivityEventListener {

    private MediaPlayer mediaPlayer = null;
    private MediaMetadataRetriever mediaMetadataRetriever = null;

    public MediaManager
    (ReactApplicationContext reactApplicationContext) {
        super(reactApplicationContext);
        reactApplicationContext.addActivityEventListener(this);
    }

    @Override
    public String getName() {
        return "MediaManager";
    }

    @Override
    public void onCatalystInstanceDestroy() {
        super.onCatalystInstanceDestroy();
        mediaPlayer.stop();
        mediaPlayer.release();
        mediaPlayer = null;
```

```
    }

    @ReactMethod
    public void showSongs() {
        Activity activity = getCurrentActivity();
        Intent intent = new Intent(Intent.ACTION_PICK,
        MediaStore.Audio.Media.EXTERNAL_CONTENT_URI);
        activity.startActivityForResult(intent, 10);
    }

    @Override
    public void onActivityResult
    (Activity activity, int requestCode,
    int resultCode, Intent intent) {
        if (intent != null) {
            playSong(intent.getData());
        }
    }

    @Override
    public void onNewIntent(Intent intent) {}
}
```

5. You may have noticed that in `onActivityResult` we call a `playSong` method. This is where we're going to start the `MediaPlayer`, parse the song's metadata, and send the event back to the JavaScript thread:

```
private void playSong(Uri uri) {
    try {
        if (mediaPlayer != null) {
            mediaPlayer.stop();
            mediaPlayer.reset();
        } else {
      mediaMetadataRetriever =
      new MediaMetadataRetriever();
            mediaPlayer = new MediaPlayer();
      mediaPlayer.setAudioStreamType
      (AudioManager.STREAM_MUSIC);
        }
        mediaPlayer.setDataSource
        (getReactApplicationContext(), uri);
        mediaPlayer.prepare();
        mediaPlayer.start();
        mediaMetadataRetriever.setDataSource
       (getReactApplicationContext(), uri);
        String artist = mediaMetadataRetriever.extractMetadata
        (MediaMetadataRetriever.METADATA_KEY_ARTIST);
String songTitle =
```

```
mediaMetadataRetriever.extractMetadata
(MediaMetadataRetriever.METADATA_KEY_TITLE);
        WritableMap params = Arguments.createMap();
        params.putString
        ("songPlaying", artist + " - " + songTitle);
        getReactApplicationContext()
          .getJSModule(DeviceEventManagerModule.
          RCTDeviceEventEmitter.class)
          .emit("SongPlaying", params);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

6. Open `index.android.js` and add `NativeModules` and `DeviceEventEmitter` to the `import { ... } from 'react-native'` block.
7. We also need to import the `Button` component and make a reference to our `MediaManager` native module:

```
import Button from 'react-native-button';

const MediaManager = NativeModules.MediaManager;
```

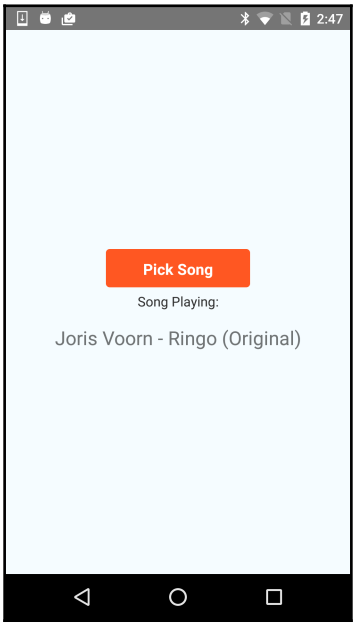
8. Our application will allow the user to pick a song and display the song name afterwards. Let's create the necessary functions that will handle the button press and setting the song title on the component's state:

```
componentWillMount() {
  this.setState({
    songPlaying : undefined
  });
  this.subscription =
  DeviceEventEmitter.addListener
  ('SongPlaying', this.onSongPlaying);
}
onSongPlaying = (params) => {
  const songPlaying = params.songPlaying;
  this.setState({songPlaying});
}
onShowSongsPress() {
  MediaManager.showSongs();
}
```

9. We are going to need to set up our UI with the `Button` and an area to display the song information. Implement the following `render` method:

```
render() {
  return (
    <View style={styles.container}>
      <Button
        containerStyle={styles.buttonContainer}
        style={styles.buttonStyle}
        onPress={this.onShowSongsPress}>
          Pick Song
      </Button>
      <Text style={styles.instructions}>Song Playing:</Text>
      <Text style={styles.welcome}>{this.state.songPlaying}
      </Text>
    </View>
  );
}
```

10. We are ready to run our app now; pick a song and it will play!



How it works…

This recipe focuses on playing audio files using the operating system's built-in audio support. In contrast to iOS, Android does not have an operating system-level UI for media playback. It requires the application to design and implement the UI. In step 4, we access the audio files stored on the device by starting a picker intent. The operating system provides us a `MediaPlayer` class that handles media playback. We cover how to play an audio file in step 5.

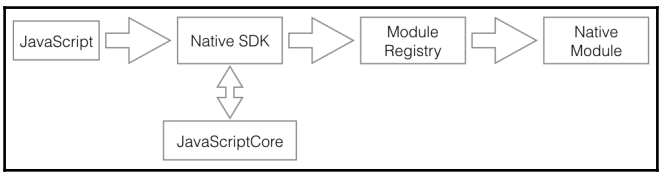
7
Architecting for Multiple Platforms

In this chapter, we will cover the following recipes:

- Building for the Universal Windows Platform
- Building for Mac OS X Desktop
- Building for Apple tvOS
- Creating platform specific UI Components
- Extending UI Components for platform-specific experiences
- Best practices for sharing code between platforms

Introduction

One of React Native's most underrated features is the architecture it is built on. The way it was designed lends itself to building native applications on a number of different platforms. When React Native was built it only supported iOS as a target platform, Android came around six months later. The way Facebook treated adding Android support architecturally was more of an add-on to what they have built for iOS instead of a critical change in the framework. The following diagram shows an overview of how calls are executed in a React Native application targeting Android:



There are several important things to note from the diagram above. First, WebKit's JavaScriptCore is used as the JavaScript interpreter and to create messages for the Bridge to execute. The Bridge then looks up the target module by passing in the iOS-based module name to the Module Registry and gets the target-specific translation. In an abstract summary to add support for a new target platform, you have to implement a Bridge that takes JavaScript messages, invokes JavaScriptCore (or a comparable JavaScript VM), a Module Registry that maps the iOS React Native components to target specific ones, and finally the components themselves need to be implemented.

In this chapter, we will show you recipes on how to build React Native apps for some of the more experimental target platforms. We will cover building for the Windows 10 platform, Mac OS Desktop, and the new Apple TV. Once we know how to build apps for all these platforms, including iOS and Android of course, we will present some recipes on how to manage building UI Components for different platforms and minimizing your headache in maintaining them.

Building for the Universal Windows Platform

At Facebook's F8 conference that happened on April 13th 2016, Microsoft announced its core support of React Native. Microsoft added React Native support to Visual Studio Code, became a core contributor to React Native itself and most importantly added support for targeting the Universal Windows Platform. The **Universal Windows Platform** (**UWP**), is a runtime that exists on all Windows 10 based devices and even the Xbox One. With React Native's third-party Windows support, we can now target a substantially larger user-base and make our apps even more accessible.

In this recipe we will cover building a React Native for Windows application. We will cover environment setup, building our first app, and running it on an emulator.

Getting ready

To get started with React Native for Windows there's a good bit of leg work that needs to be done. First and foremost, you need to be running Windows 10. If you are running Mac OS X, you can either install it via BootCamp or use a virtualization package such as VirtualBox. Once you have Windows 10 up and running you need to install Visual Studio 2015 community and the Windows 10 SDK. Once you have those installed you will need to install Microsoft Emulator for Windows 10 mobile if you do not have a mobile device to test on.

If you are running a Mac and just experimenting with Windows support you can use an evaluation VM image available from modern.ie or download an ISO from Microsoft Windows Evaluations. Both trials have a 90-day expiration, however with virtualization you can set snapshots and roll back to them. Unfortunately, if you are using VirtualBox you will not be able run the emulator as it does not support nested virtualization.

For this recipe we will create a React Native application titled **WinApp**. This app will be created as we usually create a blank React Native application using the `react-native init` command.

How to do it…

Open your command prompt and `cd` into your React Native project directory. (`cd WinApp`).

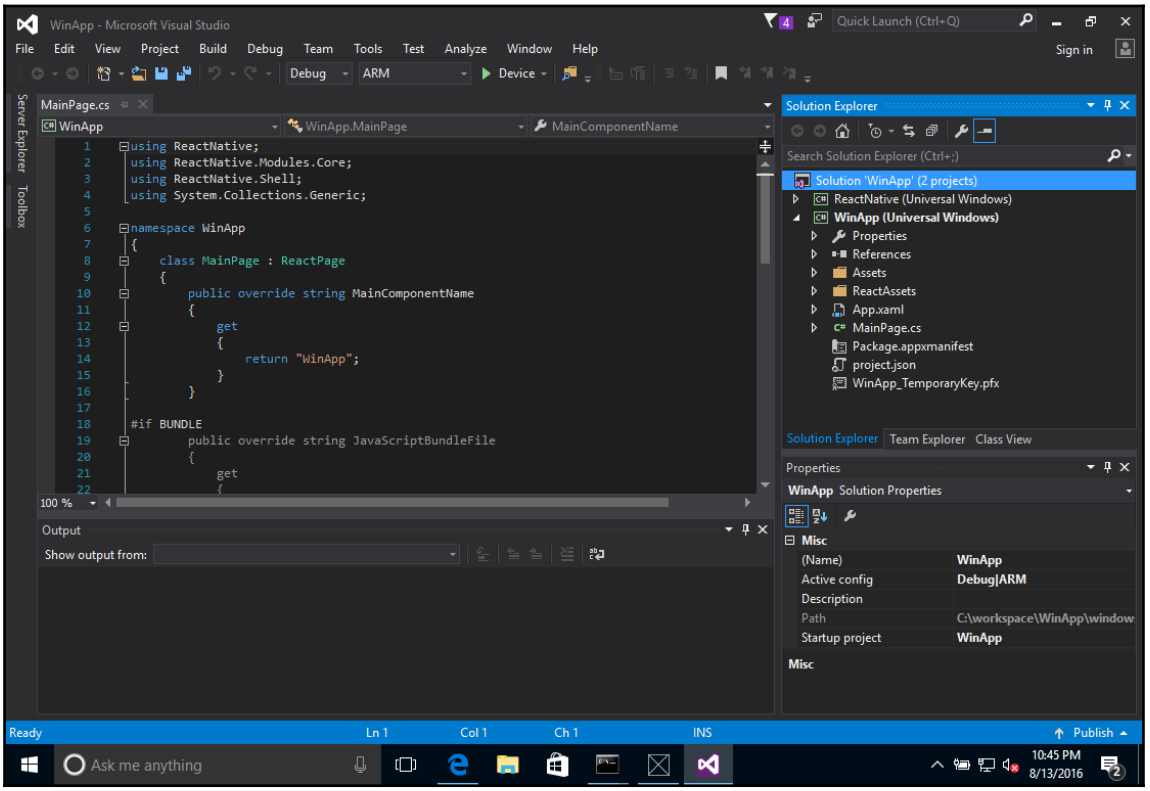
1. Now we need to install `rnpm-plugin-windows`. This plugin for the `react-native-cli` adds Windows functionality to our application as follows:

```
$ npm install rnpm-plugin-windows --save-dev
```

2. Initialize the Windows project as follows:

```
$ react-native windows
```

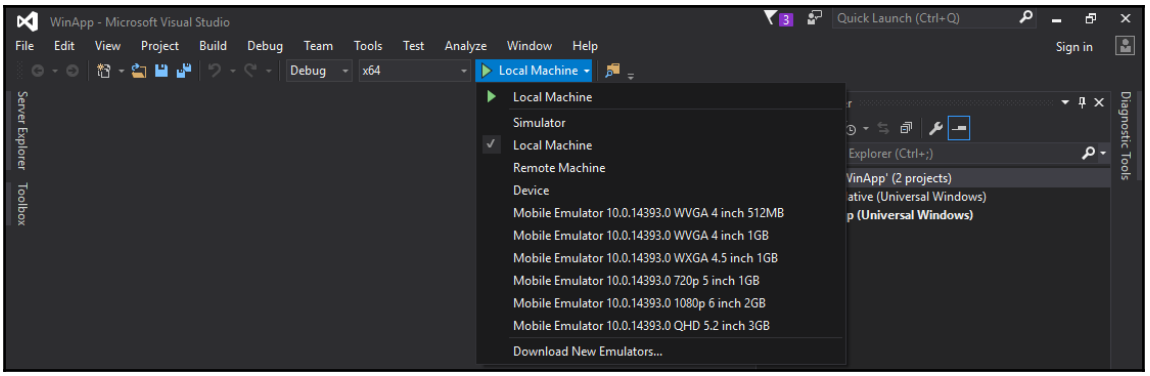
3. Open the Solution file in Visual Studio 2015. (`File > Open > Project/Solution`). The solution file is located in the `windows` directory of your project (`WinApp.sln`) as follows:



4. You may be prompted to **Install Missing Components**, please do so.
5. To run your sample app using the default settings, go back to your command prompt and this time we're going to execute the windows specific `react-native` command as follows:

```
$ react-native run-windows
```

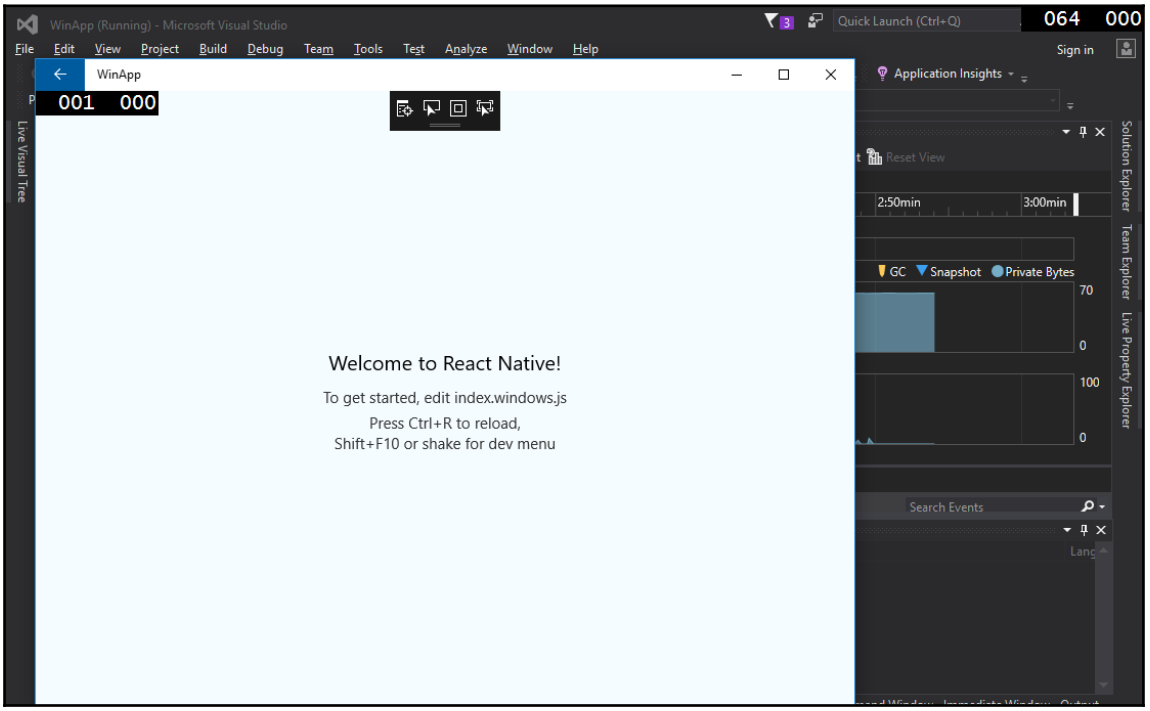
6. If you would like to run the application using a Mobile Emulator or your local device through Visual Studio, then change the deployment target to **x64** or **x86**. You should then be able to choose **Local Machine** or **Mobile Emulator** as your target platform as follows:



7. If you are using Visual Studio to start your application, then you have to open the command prompt beforehand and start the packager, if it isn't already running, as follows:

```
$ react-native start
```

8. Run your application and you should see the following:



How it works…

To add Windows support to React Native, Microsoft followed the same pattern that Facebook did when they added Android support. The major difference in Microsoft's approach is that they use **Chakra**, their own JavaScript engine instead of bundling WebKit's **JavaScriptCore** (**JSC**). Generally, this should have no impact on the developer. There may be some scenarios where Chakra's compatibility with certain ES6/ES7 features may differ from JSC's and may not be transpiled by Babel. Currently, there are no such concerns but it may happen further down the road.

When creating a React Native Windows project, we start with a base react native application and then use an `rnpm` plugin to add Windows support (step 2). This allows our application to support iOS, Android and UWP.

There's more…

As discussed in the previous chapter, often when we develop a React Native application we may need to add native functionality to our application that is not provided by the framework. If you need to expose a native API or render a native UI component, then you can follow the references below. You can also find implementations of these examples in the sample source code included with the book.

See also

- Creating Native Modules: `https://github.com/ReactWindows/react-native-windows/blob/master/docs/NativeModulesWindows.md`
- Creating Native UI Components: `https://github.com/ReactWindows/react-native-windows/blob/master/docs/NativeComponentsWindows.md`

Building for Mac OS X Desktop

So far we have an excellent understanding of two core tenants of React Native. First and foremost, with React Native you can write JavaScript and the framework will render a truly native UI for your application. Secondly, the architecture can be extended to support almost any deployment target. We mentioned in the previous recipe that you can now create a React Native application for the entire Windows 10 family, including Mobile, Desktop, and Xbox One. There should then be no reason why it isn't possible to build a React Native application for Mac OS X desktop, especially since Cocoa and Cocoa Touch have some

major similarities. Fortunately, a fork of React Native exists that supports building apps for Mac OS X. React Native for Mac OS (formerly known as React Native Desktop) serves the purpose of building desktop apps using the Cocoa API.

In this recipe, we will walk through building a desktop native application on Mac OS X using React Native. This fork is still quite experimental and does not have major support outside of the core committer and the community, so use it at your own discretion.

Getting ready

In order to create a React Native for a Mac OS X application you need the same requirements as you normally would for an iOS application. You need to be running Mac OS and have Xcode installed.

For this recipe we will create a React Native application titled `MacApp`.

How to do it…

1. React Native for Mac OS currently exists as a fork of React Native and has its own set of tooling that mimics the React Native command line tools. Before we bootstrap our application, we need to install React Native Mac OS CLI as follows:

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

2. Now we can initialize our desktop application. The process is the same as we would follow with React Native, except the command is now `react-native-macos`. See the following:

```
$ react-native-macos init MacApp
```

3. Unfortunately, there is no `run-{target}` shortcut as there is for iOS and Android so we're going to have to do it manually. Open up Xcode and open the `.xcodeproj` file located in the `macos/` directory of the application. (`./MacApp/macos/MacApp.xcodeproj`)
4. You may notice that the target platform has now changed to **My Mac**. To start the application, press the Play button.

5. Once you are running the application, if you need to access the developer menu, it is on the top OS status bar.

How it works…

React Native for Mac OS is a fork of the React Native GitHub repository. The process used to create and develop applications for Mac OS using this fork is the same as you would for iOS. This fork replaces the framework's iOS UI Components relying on `UIKit` with their Mac OS counterparts that use `AppKit` SDK. Therefore, you cannot build an application that will target iOS and Mac OS using this fork. You would have to have two separate react native applications.

There's more…

Owing to the common architecture between iOS and OS X we can easily reuse many of our custom native modules that we may have created in the past. If we are creating new custom native modules, we can easily follow the same methodologies we covered in `Chapter 6`, *Adding Native Functionality*. In the sample source code, we have included the `HelloManager` Native Module that we built in first recipe of `Chapter 6`, *Adding Native Functionality*, in a React Native Mac OS app.

Creating custom native UI Components follow a similar path except you are now exposing components using Cocoa instead of Cocoa Touch. To see some examples of accomplishing this it is recommended that you look at the source code of `react-native-macos`. There is an included Cocoa version of the `ButtonView` that we created in second recipe of `Chapter 6`, *Adding Native Functionality*. You can see the implementation of the `RCTButton` and `RCTButtonManager` here:

`https://github.com/ptmt/react-native-macos/blob/master/React/Views/.`

Building for Apple tvOS

The next great frontier in app development after desktop and mobile has become TVs. With the release of the 4th generation of Apple TV in 2015, Apple opened up the platform for developers to develop and publish apps for the Apple TV. Apple tvOS inherits many of the core frameworks we use for iOS development, with some minor changes to handle the new form factor and user control. This is a major benefit as React Native was built primarily for iOS. So what do we need to do so that we can get React Native running on our Apple tvOS? Just like in our previous recipe on Mac OS we're going to use a community fork of React Native that has support for tvOS. The fork can be found on GitHub here: `https://github.com/douglowder/react-native-appletv`. It is still a bit immature and is missing CLI tooling. However, all the automated tests built into React Native and the `UIExplorer` are passing.

This recipe will get us up and running using React Native for Apple tvOS. We will show you how to install the fork on your machine and create a React Native for Apple TV app. If you have read the prior recipes, please note that this fork is even more experimental and has less support than `react-native-windows` and `react-native-macos`. It is highly recommended to have experience with React Native, Objective-C, and iOS development.

Getting ready

As with React Native for Mac OS and iOS development, we will need to be running a Mac with the latest version of Xcode installed. This will include simulators for the Apple TV that we will be using to test our application. You can also use a physical 4th generation Apple TV if you are in possession of one.

Our sample app that we will clone will be named `TVApp`.

How to do it…

1. Since we're going to be working with a fork, we need to set up our local `npm` environment. Execute the following command in your terminal:

```
$ npm install -g sinopia && sinopia
```

2. Kill the `sinopia` node process after it runs.
3. Pull the `react-native-appletv` GitHub repository as follows:

```
$ git clone https://github.com/douglowder/react-native-appletv
```

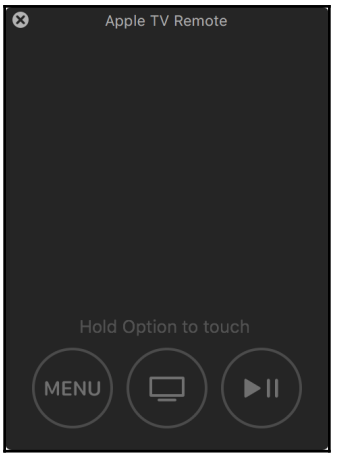
4. Now that we've pulled the repo, we should initialize the project as follows:

```
$ cd react-native-appletv && npm install
```

5. Follow the instructions starting with modifying the config file and publishing to `sinopia` found here: `https://www.npmjs.com/package/react-native-cli`.
6. With `sinopia` configured, we can initialize our app as follows:

```
$ npm uninstall -g react-native-cli
$ npm install -g react-native-cli
$ react-native init TVApp
```

7. Open up Xcode and open the `TVAppTV.xcodeproj` file located in the `ios/` directory of the application. (`./TVApp/ios/TVAppTV.xcodeproj`,note the TV Suffix**)**
8. You should now be able to run the application on your Apple TV device or simulator.
9. If you want to navigate through the application, you should show the Apple TV Remote. You can open it by selecting `Hardware > Show Apple TV Remote` in the toolbar or the keyboard shortcut *Shift* + *Command* + *R*. To scroll, you have to hold the *Alt*/option key when you touch the trackpad.



How it works…

The tvOS fork of React Native adds Apple TV support to a React Native application. This is slightly different than the Mac OS fork covered in the *Building for OS X Desktop* recipe, as the functionality is replaced entirely from iOS to Mac OS. Here we can still deploy our applications to iOS as well since both platforms use UIKit for rendering components.

With this fork we had to spoof `npm` into using a local repository for the `react-native-cli` since this fork is not deployed to the public `npm` registry. We accomplished this using `sinopia` in steps 5-6. After initializing the application using the local CLI, we are able to continue with development as per our usual react native practices.

Creating platform specific UI Components

In this chapter we have covered some of the platforms that you can target using React Native. We can build applications targeting anything from watches to desktops to game consoles. While we may be inclined to try to keep our codebase identical across all the platforms we develop for, this may take away from the user experience. In order to build a better application, it may be worth taking the time to build components that feel like they belong on their particular target.

In this recipe, we're going to take a look at a couple of strategies for making platform specific UI Components. We will cover the built-in targets iOS and Android. These patterns can be extended to other targets such as Windows or Mac OS. We will be creating a simple Toolbar component that will have a different visual representation based on its platform.

Getting ready

For this recipe we will be working off a newly created React Native app named **Platforms**.

How to do it…

1. In our React Native project directory we are going to create a file called `Toolbar.ios.js` and write the following code:

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

const Toolbar = ({items}) => (
    <View style={{flex:1, flexDirection:'row', marginTop : 24}}>
        {items.map(function (item, idx) {
            return (
                <TouchableOpacity key={idx}>
                    <View style={{padding: 20}}>
                        <Text>{item.title}</Text>
                    </View>
                </TouchableOpacity>
            )
        })}
    </View>
);
export default Toolbar;
```

2. Now we need to modify our index.ios.js to use our new `Toolbar` component. First add the following `import` before the `class` definition:

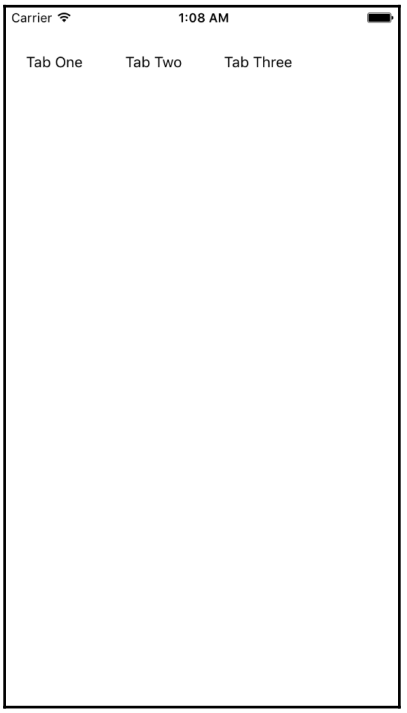
```
import Toolbar from './Toolbar';
```

3. Overwrite the render function to now render the Toolbar as follows:

```
render() {
  var toolbarItems = [
    {
      id : 1,
      title : 'Tab One',
    },
    {
```

```
        id : 2,
        title : 'Tab Two',
      },
      {
        id : 3,
        title : 'Tab Three',
      }
    ];

    return (
      <Toolbar items={toolbarItems}/>
    );
  }
```

4. If you run the app you should see the following:

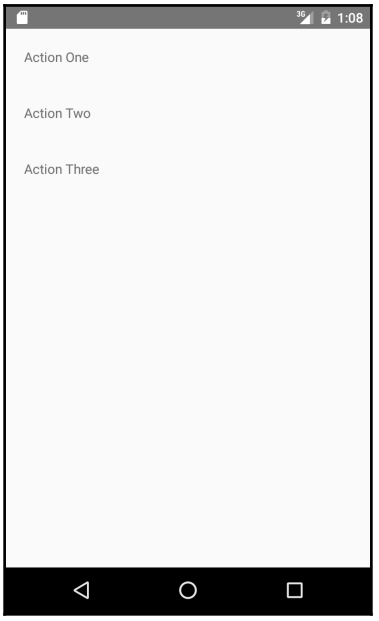


5. With the Toolbar iOS version being done, we have to build the Android counterpart. Create a file `Toolbar.android.js` and add the following code:

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

```
  View,
  TouchableNativeFeedback
} from 'react-native';

const Toolbar = ({items}) => (
    <View style={{flex:1, flexDirection:'column'}}>
        {items.map(function (item, idx) {
            return (
                <TouchableNativeFeedback key={idx}>
                    <View style={{padding: 20}}>
                        <Text>{item.title}</Text>
                    </View>
                </TouchableNativeFeedback>
            )
        })}
    </View>
);
export default Toolbar;
```

6. Repeat steps 2 and 3 for `index.android.js`.
7. If you run the app on your Android device, you should see the following:



How it works…

This recipe showed how to create iOS and Android platform-specific modules. We distinguish the classes by the suffix in their file name (`.android` or `.ios`). The react native packager sends in the platform it is requesting code for and looks for the existence of a file that matches the name with the platform in the suffix. If there is no platform-specific file, then it falls back to see if there is a `.js` file. This file would be used for all platforms.

In this recipe the extent of differentiating work we did around the platforms was mostly styling and touch interactions. This was a trivial change for the sake of the example but the `flexDirection` defined in the Toolbar class as well as the Touchable module differs from iOS and Android. The `TouchableNativeFeedback` module used on the Android version is specific only to that platform.

There's more…

There is another option for executing code based on the platform inside your JavaScript. Instead of creating `.android` and `.ios` files, you can have a single `.js` file and import the `Platform` module. In code it would look something like the following:

```
import { Platform, Text } from 'react-native';

const PlatformText = () => {
    if(Platform.OS === 'ios') {
        return (<Text>iOS</Text>);
    } else if(Platform.OS === 'android') {
        return (<Text>Android</Text>);
    }
}
```

In the included source code there is a more involved example using a `Toolbar` component that wraps `TabBarIOS` and `ToolbarAndroid`.

Extending UI Components for platform-specific experiences

When developing a React Native application you will spend a significant portion of time creating UI Components. These may be compositions of prebuilt UI Components or native components that you are exposing. Remember that Facebook's goal with React Native is to enable you to build incredible native experiences using JavaScript. iOS and Android have two different design principles that is present in their native UI Components. While we have great flexibility in regards to styling and interaction you may want your application to look and behave like it fits on its target platform.

This recipe will show a technique of how to create a base component and extend it to look and perform in a manner fitting for its platform. We will create a simple dummy button and extend it for both iOS and Android. The platform-specific versions will have a look and feel that matches their environment.

Getting ready

For this recipe we will be working off a newly created React Native app named `PlatformExt`.

How to do it…

1. In our React Native project directory we are going to create a file called `BaseButton.js` and write the following code:

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

export default class BaseButton extends Component {
    _onButtonPress = () => {
        const onTap = this.props.onTap;
        if(onTap) {
            onTap();
        }
    }
```

```
    renderButtonContent(styles) {
        return (
            <View style={styles.button}>
                <Text style={styles.label}>{this.props.label}</Text>
            </View>
        );
    }
}
```

2. Next, we will create our iOS specific `Button` class. Create a file `Button.ios.js` and add the following code:

```
import React from 'react';
import {
    StyleSheet,
    TouchableOpacity
} from 'react-native';
import BaseButton from './BaseButton';

export default class Button extends BaseButton {
    render() {
        return (
            <TouchableOpacity onPress={this._onButtonPress}>
                {this.renderButtonContent(styles)}
            </TouchableOpacity>
        );
    }
}

const styles = StyleSheet.create({
    button: {
        padding : 8,
        backgroundColor : '#007AFF'
    },
    label: {
        color : 'white',
        fontSize : 17,
        textAlign : 'center'
    }
});
```

3. Now we need to modify our `index.ios.js` to use this new `Button` component we created. First, we're going to need to import our `Button` module, add the following after the import but before the class definition:

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

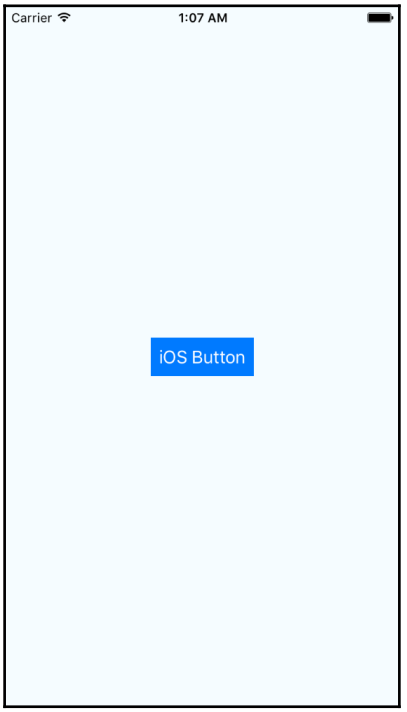
4. Next we're going to render the button, overwrite the `render` method with the following:

```
render() {
  return (
    <View style={styles.container}>
      <Button label="iOS Button" onTap={this.onTap}/>
    </View>
  );
}
```

5. We're also going to need to implement our `onTap` callback.

```
onTap() {
    console.log('iOS button pressed');
}
```

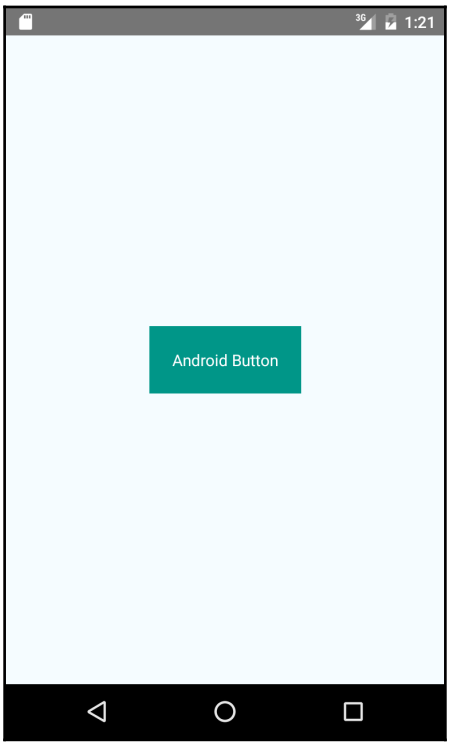
6. Running the application will show you a button that looks like this:



7. Now let's build the Android counterpart. Create a file `Button.android.js` and write the following code:

```
import React from 'react';
import {
    StyleSheet,
    TouchableNativeFeedback
} from 'react-native';
import BaseButton from './BaseButton';

export default class Button extends BaseButton {
    render() {
        return (
            <TouchableNativeFeedback onPress={this._onButtonPress}>
                {this.renderButtonContent(styles)}
            </TouchableNativeFeedback>
        );
    }
}

const styles = StyleSheet.create({
    button: {
        padding : 20,
        backgroundColor : '#009688'
    },
    label: {
        color : 'white',
        fontSize : 14,
        textAlign : 'center'
    }
});
```

8. Finally repeat steps 3-5 for `index.android.js`. You should change the log output to say Android instead of iOS for better clarity, unless you want to be confused of course. Once you run the application on your Android device you should see the following:



How it works…

In this recipe we relied on ES6 classes and inheritance to create platform specific `Button` components while sharing core functionality. The `Button` class implementations for each platform render the same JSX for the button, but surround it with their specific `Touchable` interaction. In step 1 we defined the `BaseButton` class, the platform specific `Button` classes subclass the `BaseButton` to use its `renderButtonContent` and `_onButtonPress` functions. In this situation, we are using the `BaseButton` class as if it were a mixin. However, with ES6 classes we have the flexibility to overwrite or encapsulate these function calls with subclass specific logic.

Best practices for sharing code between platforms

In this chapter, we discussed developing React Native applications for different platforms. While building our app we often develop for our favorite platform and then port to the remaining target(s). Generally, just like with the development platform, the iOS functionality of React Native is a bit more mature and stable than the Android counterpart. It should be noted that React Native was built for iOS first and the Android functionality hooks onto the existing architecture.

Whether you chose to port your application at the end or develop for multiple platforms at the same time you should attempt to keep as much of the code the same as possible and then branch off for platform specific code. This recipe will serve as a list of suggestions and examples that you should keep in mind when developing for more than a single platform.

How to do it…

- **Native Modules**: While you may not be able to share code for the module implantation, your React JavaScript code should not need to be concerned. Use the same module name and exported method signatures. See *Exposing custom iOS modules* and *Exposing custom Android modules* recipes in `Chapter 6`, *Adding Native Functionality*. You will see both modules are named `HelloManager` and the modules are defined as follows:

```
// iOS
RCT_EXPORT_METHOD(greetUser:
(NSString *)name isAdmin:
(BOOL *)isAdmin callback:(RCTResponseSenderBlock) callback) {
}
// Android
@ReactMethod
public void greetUser(String name, Boolean isAdmin, Callback
callback) {
}
```

- **Native UI Components**: Just like with Native Modules, UI Components share similar challenges that their implementation will be vastly different. Not only is the language and environment different, but in terms of UI development Android and iOS have very few similarities in their approaches. Regardless, the JavaScript layer should never be concerned about this.

  For UI Components, there is a convention of appending the platform to the class name. For example, a native `Button` component will have a `ButtonIOS` and `ButtonAndroid` class respectively. It is recommended to create a `Button` class that will render the necessary class based on the platform. The render method would look something like this (a full example is included with the source code):

```
// Button.js
render() {
   if(Platform.OS === 'ios') {
       return <ButtonIOS {...this.props}/>
   } else if(Platform.OS === 'android') {
       return <ButtonAndroid {...this.props}/>
   } else {
       return <UnimplementedView/>
   }
}

// index.ios.js || index.android.js
render() {
 return (
   <View style={styles.container}>
     <Button label="iOS Button" onPress={()=>{}}/>
   </View>
 );
}
```

- **React UI Components**: When your components are created using JavaScript only then your life is a bit simpler as you do not have to worry about porting native code and making sure the API is the same or creating a wrapper around it. An excellent practice for sharing code across platform-specific React components is to leverage ES6's inheritance as done in the *Extending UI Components for platform-specific experiences* recipe in this chapter.

8
Integration with Applications

In this chapter, we will cover the following recipes:

- Embedding a React Native application inside an iOS application
- Communicating from an iOS application to React Native
- Communicating from React Native to an iOS application container
- Handling being invoked by external iOS application
- Embedding a React Native application inside an Android application
- Communicating from an Android application to React Native
- Communicating from React Native to an Android application container
- Handling being invoked by external Android application
- Invoking an external iOS and Android application

Introduction

React Native was introduced as a solution to build native applications using JavaScript. The ultimate goal is that more developers can build truly native applications for multiple platforms. An interesting consequence of building a React Native application in an existing team is a situation where native developers and JavaScript developers have to work closely together. This generally occurs when a team is looking to adopt React Native and has an existing native application that they, want to leverage components from or integrate with.

The beauty of React Native rendering native UI views is that they can be easily embedded inside existing applications. Often, companies have already built sophisticated native applications that are critical to their line of business. There may be no immediate need to rewrite their entire application(s) in React Native if the app is not broken. However, they can begin leveraging both their JavaScript and native developers to build React Native applications that can integrate into their existing application(s).

Facebook is currently using React Native sub-applications inside of their main Facebook iOS app.

This chapter will focus exclusively on using React Native inside existing native iOS and Android applications. We will cover rendering the application and facilitating communication between the React Native application and its parent. Finally, we will also cover how our React Native app can work with other installed applications.

When working on the Android recipes, it is recommended you enable auto-import settings in Android Studio or use A*lt+Enter* to perform a quick fix code completion for the class import.

Embedding a React Native application inside an iOS application

There is a possibility that you may be in a company or have a client that has an active iOS application out in the world. This app may be well built and used frequently by its users with praise. You may be in a situation where you want to build some new functionality using React Native. You can package this React Native application as a standalone app, or you can render it inside your existing iOS application. This is the opposite of the material covered in R*endering Custom iOS View Components* recipe in `Chapter 6`, *Adding Native Functionality,* as now we're rendering our React Native view inside a vanilla iOS application.

This recipe will walk you through creating a blank iOS application and adding a sample React Native app inside it. We will cover two ways of rendering the React Native application: embedded inside the application as a nested view and a full-screen implementation. The steps discussed in this recipe serve as a baseline for rendering React Native applications inside vanilla applications.

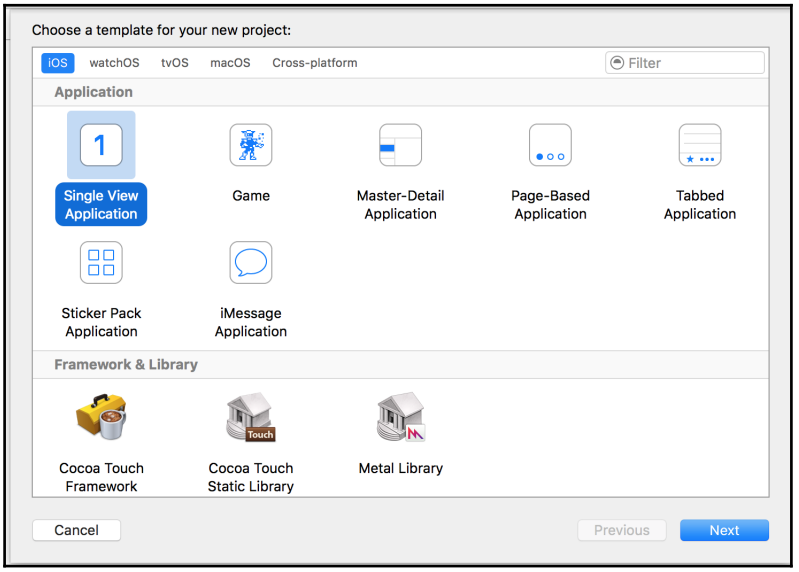
Getting ready

This recipe will be referencing an iOS application named **EmbedRN**. We will walk through creating the sample iOS application in this section. If you have an application already, you can skip to the actual recipe instructions. Please ensure, however, that you have `cocoapods` installed.

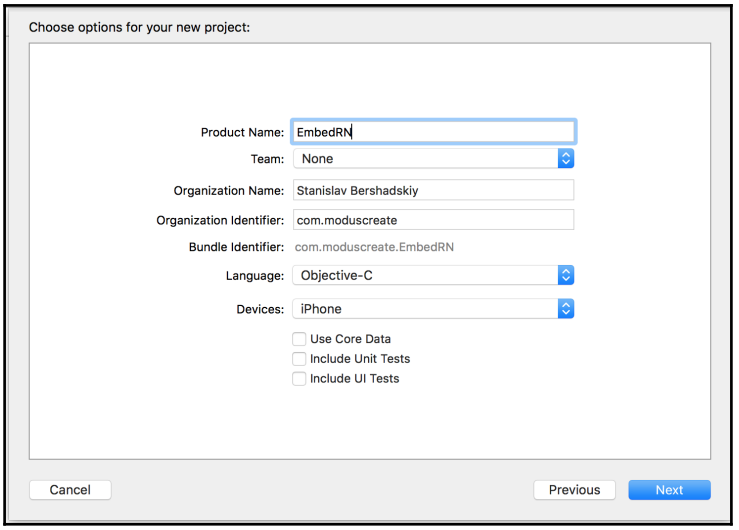
1. We need to install `cocoapods` if you do not have it already installed. Open your terminal and execute the following command:

```
sudo gem install cocoapods
```

2. Open Xcode and create a new Project using **File | New | Project**.
3. Select **Single View Application** and press **Next**:



4. Fill in the product name as **EmbedRN** and press **Next**:



5. Place the project as a sub-directory where you will hold both your React Native project and this iOS project. (for example, `EmbedApp/EmbedRN`)

How to do it…

1. In the root project directory (`./EmbedApp`), create a `package.json` file and type the following code:

```
{
  "name": "EmbedApp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start":
    "node node_modules/react-native/local-cli/cli.js start"
  },
  "dependencies": {
    "react": "15.3.1",
    "react-native": "0.32.0"
  }
}
```

2. In your terminal, run the following command:

```
$ npm install
```

3. Now we need to create a `cocoapods` Podfile. In the directory where the iOS project is (`/EmbedApp/EmbedRN`), run the following command:

```
$ pod init
```

4. Open the `Podfile` and replace its contents with the following:

```
target 'EmbedRN' do
  # Uncomment this line if you're using Swift or
  would like to use dynamic frameworks
  # use_frameworks!

  # Pods for EmbedRN
  pod 'React', :path =>
  '../node_modules/react-native', :subspecs => [
    'Core',
    'RCTText',
    'RCTNetwork',
    'RCTWebSocket', # needed for debugging
    # Add any other subspecs you want to use in your project
  ]
end
```

5. Back in the terminal in the iOS project directory, run the following command:

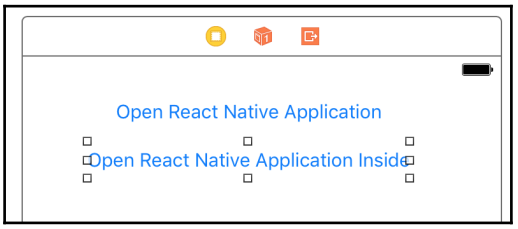
```
$ pod install
```

6. Go back to the root project directory, where we created `package.json`, and create a file `index.ios.js` with the following code:

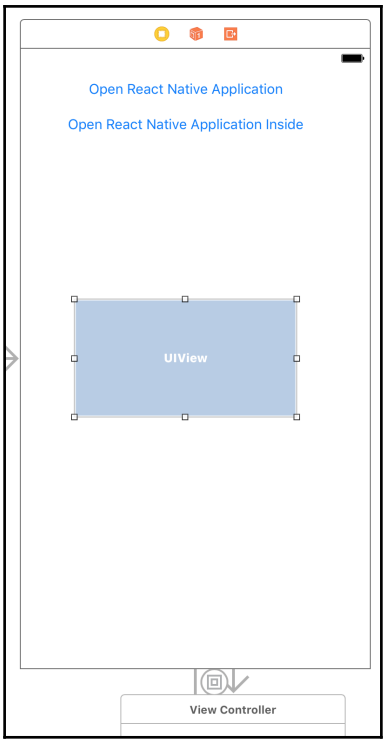
```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  View,
  Text
} from 'react-native';

class EmbedApp extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello in React Native</Text>
```

```
        </View>
      );
    }
}

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

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

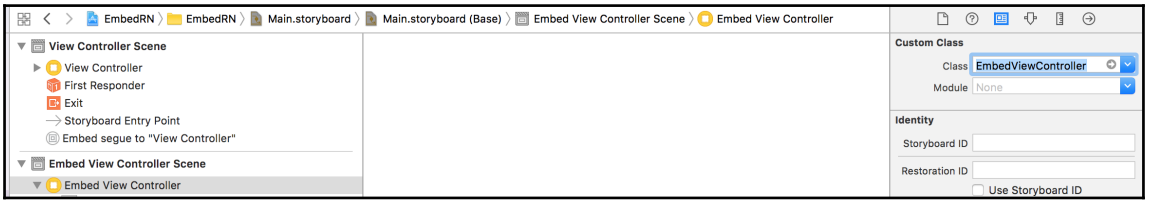
7. In Xcode, close the project and re-open it through the `.xcworkspace` file that was generated in step *5*.
8. Open `Main.storyboard` and add two buttons with labels, **Open React Native Application** and **Open React Native Application Inside:**



9. Add a Container View to your story board:



10. Create a new Cocoa Touch Class (**File|New|File**) named `EmbedViewController` with a subclass called `UIViewController`.
11. Open `Main.storyboard`, select the child **Embed View Controller Scene**, and set the class to be `EmbedViewController`:



12. Select the **Embed** segue object in the **Embed View Controller Scene** and give it an identifier of `embed`.

13. Open `ViewController.m` and add the following imports:

```
#import "RCTRootView.h"
#import "EmbedViewController.h"
```

14. Update the interface definition to add the following property:

```
@interface ViewController () {
    EmbedViewController *embedViewController;
}
```

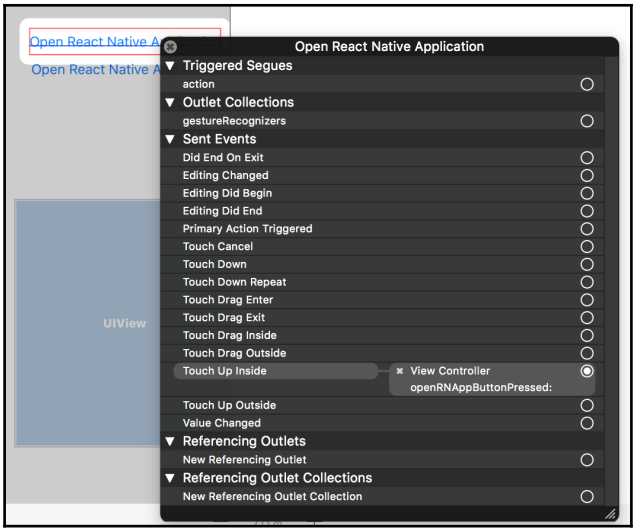
15. In `ViewController.m`, add the following methods to the implementation:

```
- (IBAction)openRNAppButtonPressed:(id)sender {
    NSURL *jsCodeLocation = [NSURL
URLWithString:
@"http://localhost:8081/index.ios.bundle?platform=ios"];
    RCTRootView *rootView =
    [[RCTRootView alloc] initWithBundleURL : jsCodeLocation
                         moduleName        : @"EmbedApp"
                         initialProperties : nil
                          launchOptions    : nil];
    UIViewController *vc = [[UIViewController alloc] init];
    vc.view = rootView;
    [self presentViewController:
    vc animated:YES completion:nil];
}
- (IBAction)openRNAppEmbeddedButtonPressed:(id)sender {
    NSURL *jsCodeLocation = [NSURL
URLWithString:
@"http://localhost:8081/index.ios.bundle?platform=ios"];
    RCTRootView *rootView =
    [[RCTRootView alloc] initWithBundleURL : jsCodeLocation
                         moduleName        : @"EmbedApp"
                         initialProperties : nil
                          launchOptions    : nil];
    [embedViewController setView:rootView];
}


- (void) prepareForSegue:
(UIStoryboardSegue *)segue sender:(id)sender {
    if([segue.identifier isEqualToString:@"embed"]) {
        embedViewController = segue.destinationViewController;
    }
}
```

16. Now we need to wire up our button actions to the buttons. Open `Main.storyboard` and *Ctrl + click* each one of the buttons, and add the following **Touch Up Inside** actions:

    Open React Native Application maps to `openRNAppButtonPressed`.

    Open React Native App Inside maps to `openRNAppEmbeddedButtonPressed`:



17. Add the following to `Info.plist`:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>localhost</key>
        <dict>
            <key>NSTemporaryExceptionAllowsInsecureHTTPLoads
            </key>
            <true/>
        </dict>
    </dict>
</dict>
```

18. In your root project directory, run the following command:

```
$ react-native start
```

19. Once we run our application, we should get something that looks like the following screenshot:

How it works…

In this recipe, we tackled rendering the React Native app in two different ways. The first and most trivial way was to replace the application's main `UIViewController` instance with our newly created one that will hold the `RCTRootView`. This is accomplished in the `openRNAppButtonPressed` method. The second and slightly more involved method, involves rendering the React Native app inline with the rest of the root application. To accomplish this, we created a Container View that links to another `UIViewController` instance. In this case, we replaced the `embedViewController` contents with our `RCTRootView` instance. This was accomplished in the `openRNAppEmbeddedButtonPressed` method. A final point to note: to set the `embedViewController` reference, we needed to listen for the segue to the container. This occurred in the `prepareForSegue` delegate method.

Communicating from an iOS application to React Native

In the previous recipe, *Embedding a React Native application inside an iOS application*, we learned how to render our React Native sub-application inside a potentially larger native iOS app. Unless you're building a glorified application container/portal, what good is just rendering the React Native application? What if we wanted to connect the React Native application with our container so that it can consume information from the parent application?

This recipe will cover that very thing, sending data from the parent iOS application to our embedded React Native application. The React Native application can accept data when it is first instantiated, and then at runtime. We will be covering how to accomplish both methods. This recipe will use a `UITextField` in the iOS app and set up one-way binding to the React Native application.

Getting ready

For this recipe, please ensure you have an iOS application with a React Native app embedded inside a Container View. If you need guidance to accomplish this, please complete the *Embedding a React Native application inside an iOS application* recipe and you will then be able to move on.

How to do it…

1. In Xcode, open `ViewController.m` and add the following imports:

```
#import "ViewController.h"
#import "RCTRootView.h"
#import "RCTBridge.h"
#import "RCTEventDispatcher.h"
#import "EmbedViewController.h"
```

2. Now we need to add a few instance variables to the `ViewController` class to hold a reference to the React Native bridge :

```
@interface ViewController () <RCTBridgeDelegate> {
    EmbedViewController *embedViewController;
    RCTBridge *_bridge;
    BOOL isRNRunning;
```

```
}
```

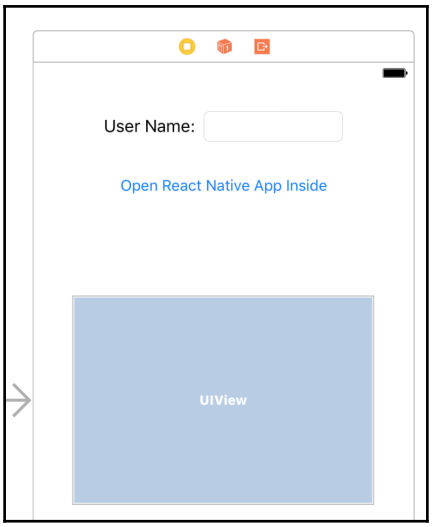
3. Next, we need to create a reference that we will wire `UITextField` to the following code:

```
@property (weak, nonatomic)
IBOutlet UITextField *userNameField;
```

4. Add the following methods to the `ViewController` implementation:

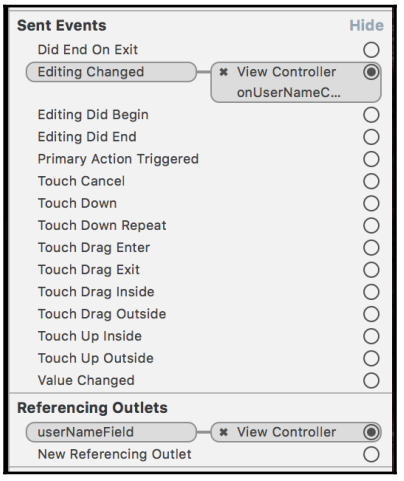
```
- (NSURL *)sourceURLForBridge:(RCTBridge *)bridge {
    NSURL *jsCodeLocation = [NSURL
 URLWithString:
 @"http://localhost:8081/index.ios.bundle?platform=ios"];
    return jsCodeLocation;
}
- (IBAction)openRNAppEmbeddedButtonPressed:(id)sender {
    NSString *userName = _userNameField.text;
    NSDictionary *props = @{@"userName" : userName};
    if(_bridge == nil) {
        _bridge = [[RCTBridge alloc]
        initWithDelegate:self launchOptions:nil];
    }
    RCTRootView *rootView =
    [[RCTRootView alloc] initWithBridge :_bridge
                      moduleName        : @"EmbedApp"
                      initialProperties : props];
    isRNRunning = true;
    [embedViewController setView:rootView];
}

- (IBAction)onUserNameChanged:(id)sender {
    if(isRNRunning == YES && _userNameField.text.length > 3) {
        [_bridge.eventDispatchersendAppEventWithName:
        @"UserNameChanged" body:
        @{@"userName" : _userNameField.text}];
    }
}
```

5. Open `Main.storyboard` and add a **Text Field** and **Label** widget called **UserName**:



6. Select the **User Name** field, show **Connections Inspector** and create the following wirings by dragging the item to the `ViewController`.

A simple way to create connections from the storyboard is to use the assistant editor with the `ViewController` implementation opened on the right side.



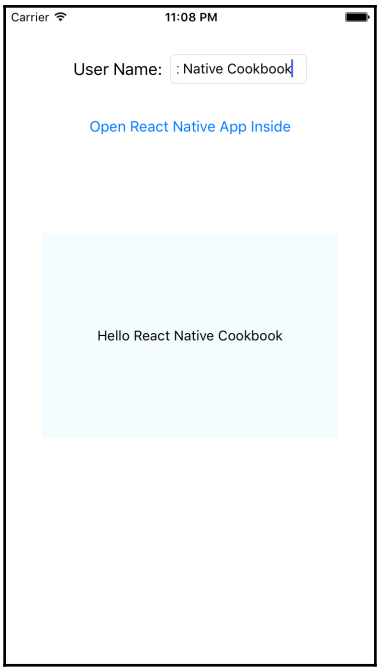
7. Now we need to update our React Native application to complete the binding. Open `index.ios.js` and import `NativeAppEventEmitter`.
8. Put the following functions inside the class definition:

```
componentWillMount() {
  this.setState({
    userName : this.props.userName
  });
NativeAppEventEmitter.addListener
('UserNameChanged', (body) => {
      this.setState({userName : body.userName});
  });
}

render() {
  return (
    <View style={styles.container}>
      <Text>Hello {this.state.userName}</Text>
    </View>
  );
}
```

9. In your root project directory, run the following command:

```
$ react-native start
```

10. Once we run our application, we should get something that looks like this:



How it works…

The binding functionality was accomplished by using the React Native bridge `eventDispatcher`. This was originally designed to be used with native modules, but as long as you have access to the `RCTBridge` instance you can use it for any communication to our React Native JavaScript layer. In step *4*, we created a `RCTBridge` instance and used it to launch our React Native application. This is the recommended way of launching React Native applications, if you need to use them in multiple applications since it will use a single JS runtime.

Communicating from React Native to an iOS application container

Most React Native applications rely on some user input, be it a form field or actions against some piece of data. It will probably be the case that our embedded React Native application will be interacted with. Just like we learned how to update our React Native application in the *Communicating from an iOS application to React Native* recipe, we should be able to update our parent application.

This recipe will cover that very concept. We will render a user input element inside our React Native application and set up a one-way binding from React Native to a UI component rendered in the parent.

Getting ready

For this recipe, please ensure you have an iOS application with a React Native app embedded inside a Container View. If you need guidance to accomplish this, please complete the *Embedding a React Native application inside an iOS application* recipe and you will then be able to move on.

How to do it…

1. In Xcode, open `ViewController.m` and add the following imports:

```
#import "ViewController.h"
#import "RCTRootView.h"
#import "RCTBridge.h"
#import "RCTEventDispatcher.h"
#import "EmbedViewController.h"
```

2. Now we need to add a few instance variables to the `ViewController` class to hold a reference to the React Native bridge:

```
@interface ViewController () <RCTBridgeDelegate> {
    EmbedViewController *embedViewController;
    RCTBridge *_bridge;
    BOOL isRNRunning;
}
```

3. Next, we need to create a reference that we will wire `UITextField` to:

```
@property (weak, nonatomic)
IBOutlet UITextField *userNameField;
```

4. Open the `ViewController` header file and add the following method declaration:

```
@interface ViewController : UIViewController
- (void) updateUserNameField:(NSString *)userName;
@end
```

5. Add the following methods to the `ViewController` implementation:

```
- (NSURL *)sourceURLForBridge:(RCTBridge *)bridge {
    NSURL *jsCodeLocation = [NSURL
URLWithString:
@"http://localhost:8081/index.ios.bundle?platform=ios"];
    return jsCodeLocation;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    [self openRNAppEmbeddedButtonPressed:nil];
}

- (IBAction)openRNAppEmbeddedButtonPressed:(id)sender {
    if(_bridge == nil) {
        _bridge = [[RCTBridge alloc] initWithDelegate:self
          launchOptions:nil];
    }
    RCTRootView *rootView =
    [[RCTRootView alloc] initWithBridge :_bridge
                      moduleName        : @"EmbedApp"
                      initialProperties : nil];
    isRNRunning = true;
    [embedViewController setView:rootView];
}

- (void) prepareForSegue:
(UIStoryboardSegue *)segue sender:(id)sender {
    if([segue.identifier isEqualToString:@"embed"]) {
embedViewController =
segue.destinationViewController;
    }
}
```

```
-(void) updateUserNameField:(NSString *)userName {
    [_userNameField setText:userName];
}
```

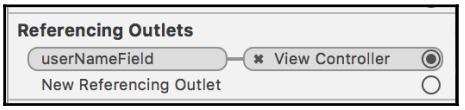
6. Next, we're going to create the `UserNameManager` native module. Create a Cocoa Touch Class called `UserNameManager` and add the following code to the implementation:

If you are not familiar with creating a native module for React Native, please refer to first recipe, *Exposing custom iOS modules*, in `Chapter 6`, *Adding Native Functionality*.

```
#import "UserNameManager.h"
#import "AppDelegate.h"
#import "ViewController.h"

@implementation UserNameManager
RCT_EXPORT_MODULE();

- (dispatch_queue_t)methodQueue
{
    return dispatch_get_main_queue();
}

RCT_EXPORT_METHOD(setUserName: (NSString *)userName) {
    AppDelegate *delegate = (AppDelegate *)[[UIApplication
    sharedApplication] delegate];
    ViewController *controller =
(ViewController*)delegate.window.rootViewController;
    [controller updateUserNameField:userName];
}
@end
```

7. Open `Main.storyboard` and add a Text Field and Label widget called **User Name**

8. Create a **Referencing Outlets** from the `UITextField` to our `userNameField` property:



9. Now let's switch to `index.ios.js` and set up our React Native application to communicate with the parent. Add the following code to the `import` block:

```
import {
  AppRegistry,
  StyleSheet,
  View,
  Text,
  TextInput,
  NativeModules
} from 'react-native';
const UserNameManager = NativeModules.UserNameManager;
```

10. Add the following code to the class implementation:

```
componentWillMount() {
    this.setState({
      userName : ''
    });
}
onUserNameChange = (userName) => {
```

```
    this.setState({userName});
    UserNameManager.setUserName(userName);
  }
  render() {
    return (
      <View style={styles.container}>
        <Text>Embedded RN App</Text>
        <Text>Enter User Name</Text>
        <TextInput
          style={styles.userNameField}
          onChangeText={this.onUserNameChange}
          value={this.state.userName}
        />
      </View>
    );
  }
```

11. In your root project directory, run the following command:

```
$ react-native start
```

12. Once we run our application, we should get something that looks like this:

How it works…

To get our React Native application updating the native app containers, we created a native module. This is the recommended way of communicating from JavaScript to the native layer. However, since we had to update a native UI component, the operation had to be performed on the main thread. This was achieved by implementing the `methodQueue` method and instructing the module to execute on the main thread, this is done in step *5*.

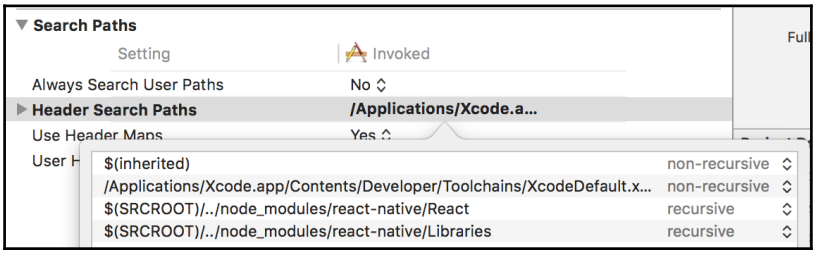
Handling being invoked by external iOS application

What good is an application if it isn't being used? We want our app to be used by our targeted users as much as possible. There are many ways of going about this, but one of the important factors is our app needs to be accessible. It needs to be so accessible, that other applications should feel inclined to link to it.

This recipe will describe how to set our application up to be invoked by other iOS applications.

How to do it…

1. Open your React Native project in Xcode and open up the project's **Build Settings**.
2. Add `$(SRCROOT)/../node_modules/react-native/Libraries` to your **Header Search Paths** field as a recursive entry:



3. Next, we need to register the URL scheme that will be used by other applications. Open `Info.plist` using the **Source Code Editor** (**right click | Open As | Source Code**). For our example, we will register our application under the `invoked://` scheme. Add the following XML code as a node of `<dict>`:

```
<key>CFBundleURLTypes</key>
<array>
   <dict>
         <key>CFBundleTypeRole</key>
         <string>Editor</string>
         <key>CFBundleURLSchemes</key>
         <array>
               <string>invoked</string>
         </array>
   </dict>
</array>
```

4. Add the following method to `AppDelegate.m`:

```
#import "RCTLinkingManager.h"

- (BOOL)application:(UIApplication *)
application openURL:(NSURL *)url
  sourceApplication:(NSString *)
  sourceApplication annotation:(id)annotation
{
  return [RCTLinkingManager application:application
  openURL:url
              sourceApplication:sourceApplication annotation:
              annotation];
}
```

5. Now let's get our UI to react to being invoked by another app. Open `index.ios.js` and import the `Linking` module from `'react-native'`.
6. Add the following code to the class implementation:

```
componentWillMount() {
  this.setState({
    status: 'App Running'
  });
  Linking.addEventListener('url', this.onAppInvoked);
}
componentWillUnmount() {
 Linking.removeEventListener('url', this.onAppInvoked);
}
onAppInvoked = (event) => {
```

```
    this.setState({
      status: `App Invoked by ${event.url}`
    });
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.instructions}>
          App Status:
        </Text>
        <Text style={styles.welcome}>
          {this.state.status}
        </Text>
      </View>
    );
  }
```

7. Running the app and invoking it from another application will result in the following UI:

How it works…

In this recipe, we register our URL schema for linking by editing the `Info.plist` file in step *3*. In step *4*, we pass on the responsibility of reacting to our application being opened to React Native by implementing the `openURL` method of `AppDelegate` and invoking `RCTLinkingManager`.

Embedding a React Native application inside an Android application

It is no secret that the Android platform holds the majority stake in the smartphone market space. For this very reason, it is almost certain that if you were to build an application and were looking to reach as many users as you could that you would build for Android as well as iOS. For many developers and teams, this may not be what they look forward to. Thankfully, Facebook made React Native to make our lives easier. However, there most probably are Android applications that you or your company may have in production that work just fine. But, you want to write the next feature using React Native, or you have a React Native application that you want to embed inside this Android app. Fortunately, Facebook has made this possible for you as well.

This recipe will dive into the steps needed to embed a React Native application inside an existing Android app. We will display our React Native application inside a Container View. The steps here are used as a baseline for the next recipes involving communication with the React Native application.

Getting ready

In this section, we will create a sample Android application using Android Studio called **EmbedApp**. If you have a base Android application you would like to work with, you can skip these steps and proceed to the actual implementation:

1. Open Android Studio and create a new Project (**File|New Project**).
2. Set the Application to name to `EmbedApp` and fill out your company domain. Press **Next**.
3. Leave Blank Activity selected as the default and press **Next**.
4. Leave the Activity properties as they are by default and press **Finish**.

How to do it…

1. Open your app's root folder in your terminal and run the following commands:

```
$ npm init
$ npm install --save react react-native
$ curl -o .flowconfig
https://raw.githubusercontent.com/facebook/react
native/master/.flowconfig
```

2. Open `package.json` and add the following property as a member of the `scripts` object:

```
"start":
"node node_modules/react-native/local-cli/cli.js start"
```

3. Create a file called `index.android.js` and add the following code:

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

class EmbedApp extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello in React Native</Text>
      </View>
    );
  }
}

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

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

4. Back in Android Studio, open `build.gradle` (from the `app` module) and add the following to the `dependencies`:

```
compile 'com.facebook.react:react-native:+'
```

5. Open the other `build.gradle` and add the following line to the `allprojects.repositories` object:

```
maven {
url "$rootDir/node_modules/react-native/android" }
```

6. Open `AndroidManifest.xml` and add the following permissions to the `<manifest>` node:

```
<uses-permission android:
name="android.permission.INTERNET" />
<uses-permission android:
name="android.permission.SYSTEM_ALERT_WINDOW"/>
```

7. Add the following attribute to the `<application>` node:

```
android:name=".EmbedApplication"
```

8. Create a new Java class called `EmbedApplication` and add the following code:

```
public class EmbedApplication extends Application
implements ReactApplication {
     private final ReactNativeHost mReactNativeHost =
     new ReactNativeHost(this) {
         @Override
         protected boolean getUseDeveloperSupport() {
             return BuildConfig.DEBUG;
         }
         @Override
         protected List<ReactPackage> getPackages() {
             return Arrays.<ReactPackage>asList(
                     new MainReactPackage()
             );
         }
     };
     @Override
     public ReactNativeHost getReactNativeHost() {
         return mReactNativeHost;
     }
 }
```

9. Create another new Java class with the name `ReactFragment` and write the following:

```
public abstract class ReactFragment extends Fragment {
     private ReactRootView mReactRootView;
     private ReactInstanceManager mReactInstanceManager;
     public abstract String getMainComponentName();
     @Override
     public void onAttach(Context context) {
         super.onAttach(context);
         mReactRootView = new ReactRootView(context);
         mReactInstanceManager =
                 ((EmbedApplication) getActivity()
                         .getApplication())
                         .getReactNativeHost()
                         .getReactInstanceManager();
     }
     @Override
     public ReactRootView onCreateView
     (LayoutInflater inflater,
     ViewGroup group, Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         return mReactRootView;
     }
     @Override
     public void onActivityCreated
     (Bundle savedInstanceState) {
         super.onActivityCreated(savedInstanceState);
         mReactRootView.startReactApplication(
                 mReactInstanceManager,
                 getMainComponentName(),
                 null
         );
     }
 }
```

10. Finally, create a Java class called `EmbedFragment` that will extend `ReactFragment`.

```
public class EmbedFragment extends ReactFragment {
     @Override
     public String getMainComponentName() {
         return "EmbedApp";
     }
 }
```

11. Open `MainActivity.java` and add `implements DefaultHardwareBackBtnHandler` to the class definition.
12. Add the following methods:

```
@Override
public void invokeDefaultOnBackPressed() {
     super.onBackPressed();
}
@Override
protected void onPause() {
     super.onPause();
     if (mReactInstanceManager != null) {
         mReactInstanceManager.onHostPause(this);
     }
}
@Override
protected void onResume() {
     super.onResume();
     if (mReactInstanceManager != null) {
         mReactInstanceManager.onHostResume(this, this);
     }
}
@Override
protected void onDestroy() {
     super.onDestroy();
     if (mReactInstanceManager != null) {
         mReactInstanceManager.onHostDestroy(this);
     }
}
@Override
public void onBackPressed() {
     if (mReactInstanceManager != null) {
         mReactInstanceManager.onBackPressed();
     }
 else {
         super.onBackPressed();
     }
}
@Override
public boolean onKeyUp
(int keyCode, KeyEvent event) {
     if (keyCode == KeyEvent.KEYCODE_MENU &&
     mReactInstanceManager != null) {
         mReactInstanceManager.showDevOptionsDialog();
         return true;
       }
     return super.onKeyUp(keyCode, event);
```

```
}
```

13. Add the following as a class field:

```
private ReactInstanceManager mReactInstanceManager;
```

14. Replace your `onCreate` method with the following code:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
     Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
     setSupportActionBar(toolbar);
    FloatingActionButton fab = (FloatingActionButton)
    findViewById(R.id.fab);
     fab.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View view) {
             Fragment viewFragment = new EmbedFragment();
             getFragmentManager().beginTransaction()
             .add(R.id.container, viewFragment).commit();
         }
     });
    mReactInstanceManager = ((EmbedApplication)
    getApplication()).getReactNativeHost()
    .getReactInstanceManager();
}
```

15. In your terminal, run the following command:

```
$ react-native start
```

16. Now you can run your Android application, and you should see the following after you press the Mail button:

How it works…

To accomplish rendering a React Native inside of our Android application, we had to perform a few steps. First, we had to define an `Application` class that implements the `ReactApplication` interface. Then we had to create a `Fragment` that would be responsible for instantiating and rendering the `ReactRootView`. With a fragment, we are able to render the React Native view in our `MainActivity`. In this recipe, we added the fragment to our main Container View. This essentially replaces all of the application content with the React Native application.

Communicating from an Android application to React Native

Now that we have covered how to render our React Native application inside an Android app in the *Embedding a React Native application inside an Android application* recipe, we need to take that to the next level. Our React Native application should be more than a dummy UI. It should be able to react to actions going on in its parent application.

In this recipe we will accomplish sending data from our Android application to our embedded React Native app. The React Native application can accept data when it is first instantiated, and then at runtime. We will be covering how to accomplish both methods. This recipe will use `EditText` in the Android app and set up one-way binding to the React Native application.

Getting ready

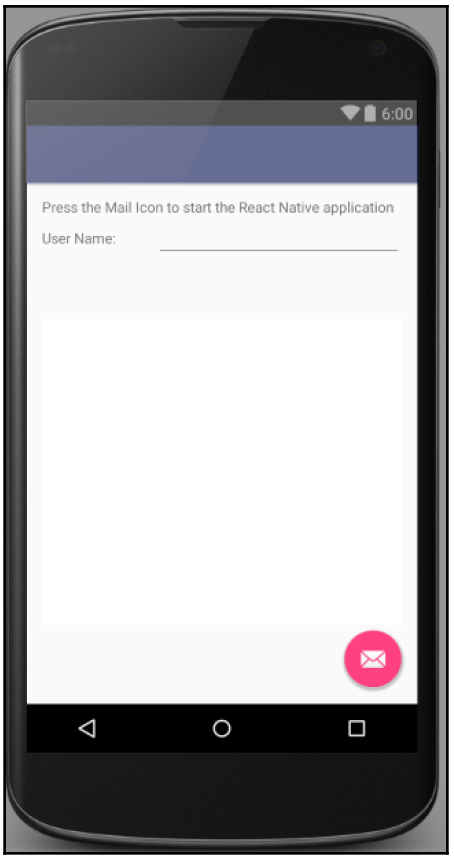
For this recipe please ensure you have an Android application with a React Native app embedded. If you need guidance to accomplish this, please complete the *Embedding a React Native application inside an Android application* recipe.

How to do it…

1. Open **Android Studio** in your **Project** and open `content_main.xml`.

2. Press the **Text** tab on the bottom to open the source editor and add/replace the following nodes:

```
<TextView
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text=
     "Press the Mail Icon to start
     the React Native application"
     android:id="@+id/textView" />
<FrameLayout
     android:layout_width="match_parent"
     android:layout_height="300dp"
     android:layout_centerVertical="true"
     android:layout_alignParentStart="true"
     android:id="@+id/reactnativeembed"
     android:background="#FFF"></FrameLayout>
 <LinearLayout
     android:orientation="horizontal"
     android:layout_width="match_parent"
     android:layout_height="75dp"
     android:layout_below="@+id/textView"
     android:layout_centerHorizontal="true">
     <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="User Name:"
         android:id="@+id/textView2"
         android:layout_weight="0.14" />
     <EditText
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:id="@+id/userName"
         android:layout_weight="0.78"
         android:inputType="text"
         android:singleLine="true"
         android:imeOptions="actionDone"/>
 </LinearLayout>
```



3. Open `MainActivity.java` and add the following class fields:

```
private ReactInstanceManager mReactInstanceManager;
private EditText userNameField;
private Boolean isRNRunning = false;
```

4. Inside the `onCreate` method, set the `userNameField` property with the following:

```
userNameField = (EditText) findViewById(R.id.userName);
```

5. Replace `FloatingActionButtononClickListener` with the following:

```
fab.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
          Fragment viewFragment = new EmbedFragment();
```

```
            if (userNameField.getText().length() > 0) {
                Bundle launchOptions = new Bundle();
                launchOptions.putString("userName",
                userNameField.getText().toString());
                viewFragment.setArguments(launchOptions);
            }
getFragmentManager().beginTransaction()
.add(R.id.reactnativeembed, viewFragment).commit();
            isRNRunning = true;
        }
});
```

6. Next, we need to add a `TextChangedListener` to our `userNameField` in the `onCreate` method:

```
userNameField.addTextChangedListener(new TextWatcher() {
     @Override
     public void beforeTextChanged
     (CharSequence s, int start, int count, int after) {}
    @Override
     public void onTextChanged
     (CharSequence s, int start, int before, int count) {}
    @Override
     public void afterTextChanged(Editable s) {
         if(isRNRunning) {
             sendUserNameChange(s.toString());
         }
     }
});
```

7. The last change we need to make for our `Activity` is to add methods that will send the event across the React Native bridge:

```
private void sendUserNameChange(String userName) {
     WritableMap params = Arguments.createMap();
     params.putString("userName", userName);
     sendReactEvent("UserNameChanged", params);
}
private void sendReactEvent
(String eventName, WritableMap params) {
     mReactInstanceManager.getCurrentReactContext()
     .getJSModule
     (DeviceEventManagerModule.RCTDeviceEventEmitter.class)
     .emit(eventName, params);
}
```

8. Open `index.android.js` and import the `NativeAppEventEmitter` module from `react-native`.
9. Add the following code to the class implementation:

```
componentWillMount() {
  this.setState({
    userName : this.props.userName
  });
  NativeAppEventEmitter.addListener
  ('UserNameChanged', (body) => {
      this.setState({userName : body.userName});
  });
}
render() {
  return (
    <View style={styles.container}>
      <Text>Hello {this.state.userName}</Text>
    </View>
  );
}
```

10. Now, if you run the application, you can enter text in the **User Name** field and start the React Native application:

How it works…

In this recipe, we render the fragment as an inline view. In step *2*, we added an empty `FrameLayout` that we target in step *5* to render the fragment. The binding functionality was accomplished by using the React Native bridge `RCTDeviceEventEmitter`. This was originally designed to be used with native modules, but as long as you have access to the `ReactContext` instance you can use it for any communication with our React Native JavaScript layer.

Communicating from React Native to an Android application container

As we discussed in the *Communicating from an Android application to React Native* recipe, it is extremely beneficial for our embedded application to be aware of what's going on around it. We should also make an effort that our Android parent application can be informed about what goes on inside the React Native application. The application should not only be able to perform business logic, but it should be able to update its UI to reflect changes in the embedded app.

This recipe shows us how to leverage native modules to update the native UI created inside the Android application. We will have a text field in our React Native app that updates a text field that is rendered in the host Android application.

Getting ready

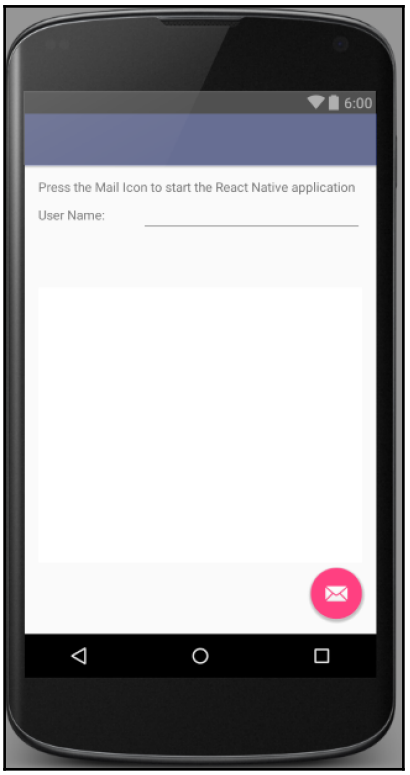
For this recipe, please ensure you have an Android application with a React Native app embedded. If you need guidance to accomplish this, please complete the *Embedding a React Native application inside an Android application* recipe.

How to do it…

1. Open **Android Studio** to your **Project** and open `content_main.xml`.

2. Press the **Text** tab on the bottom to open the source editor and add/replace the following nodes:

```
<TextView
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text=
     "Press the Mail Icon to start
     the React Native application"
     android:id="@+id/textView" />
<FrameLayout
     android:layout_width="match_parent"
     android:layout_height="300dp"
     android:layout_centerVertical="true"
     android:layout_alignParentStart="true"
     android:id="@+id/reactnativeembed"
     android:background="#FFF">
</FrameLayout>
<LinearLayout
     android:orientation="horizontal"
     android:layout_width="match_parent"
     android:layout_height="75dp"
     android:layout_below="@+id/textView"
     android:layout_centerHorizontal="true">
     <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="User Name:"
         android:id="@+id/textView2"
         android:layout_weight="0.14" />
     <EditText
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:id="@+id/userName"
         android:layout_weight="0.78"
         android:inputType="text"
         android:singleLine="true"
         android:imeOptions="actionDone"/>
 </LinearLayout>
```



3. Create a Java class named `UserNameManager`. This will be a native module that will serve the purpose of updating the `EditText` field we added to the layout.

If you are not familiar with creating a native module for React Native, please refer to the *Exposing custom Android modules* recipe in `Chapter 6`, Adding Native Functionality.

4. Add the following code to `UserNameManager.java`:

```
public class UserNameManager extends
ReactContextBaseJavaModule {
     public UserNameManager(ReactApplicationContext
     reactApplicationContext) {
         super(reactApplicationContext);
     }

     @Override
     public String getName() {
```

```
            return "UserNameManager";
        }

        @ReactMethod
        public void setUserName(final String userName) {
            Activity mainActivity =
            getReactApplicationContext().getCurrentActivity();
            final EditText userNameField =
            (EditText) mainActivity.findViewById(R.id.userName);
            mainActivity.runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    userNameField.setText(userName);
                }
            });
        }
    }
```

5. Next, create a `UserNamePackage` Java class and add the following code:

```
public class UserNamePackage implements ReactPackage {
    @Override
    public List<Class<? extends
    JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }
    @Override
    public List<ViewManager>
    createViewManagers
    (ReactApplicationContext reactContext) {
        return Collections.emptyList();
    }
    @Override
    public List<NativeModule>
    createNativeModules
    (ReactApplicationContext reactContext) {
        List<NativeModule> modules = new ArrayList<>();
        modules.add(new UserNameManager(reactContext));
        return modules;
    }
}
```

6. Add the `UserNamePackage` in the `getPackages()` in `EmbedApplication`.

7. Now we need to have our React Native UI render a `TextField` and call our `UserNameManager` native module. Open `index.android.js` and import the `TextInput` and `NativeModules` modules from `'react-native'`.
8. Create a variable reference for the `UserNameManager`.

```
const UserNameManager = NativeModules.UserNameManager;
```

9. Add the following code to the class implementation:

```
componentWillMount() {
    this.setState({
      userName : ''
    });
}
onUserNameChange = (userName) => {
  this.setState({userName});
  UserNameManager.setUserName(userName);
}
render() {
  return (
    <View style={styles.container}>
      <Text>Embedded RN App</Text>
      <Text>Enter User Name</Text>
      <TextInput
        style={styles.userNameField}
        onChangeText={this.onUserNameChange}
        value={this.state.userName}
      />
    </View>
  );
}
```

10. Running the application, starting the React Native embedded app, and adding text to the text field should resemble this screen shot:

How it works…

To get our React Native application to update the native app containers, we created a native module. This is the recommended way of communicating from JavaScript to the native layer. However, since we had to update a native UI component, the operation had to be performed on the main thread. This is achieved by getting a reference to `MainActivity` and calling the `runOnUiThread` method. This is done in the `setUserName` method of step *4*.

Handling being invoked by external Android application

Earlier in this chapter, we discussed the benefits of making our application available to other apps to increase usage in the *Handle being invoked by external iOS application* recipe. It is only fitting that we do not leave Android behind and show you how to accomplish deep linking in this recipe.

How to do it…

1. Open your React Native Android project in Android Studio and open `AndroidManifest.xml`.
2. For our example, we will register our application under the `invoked://` scheme. Change your `<activity>` node to the following:

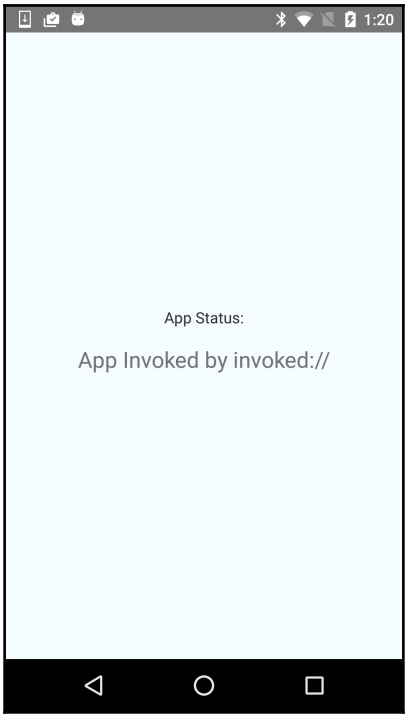
```
<activity
   android:name=".MainActivity"
   android:launchMode="singleTask"
   android:label="@string/app_name"
   android:configChanges="keyboard|keyboardHidden|
   orientation|
   screenSize">
   <intent-filter>
       <action android:name="android.intent.action.MAIN" />
       <category android:name=
       "android.intent.category.LAUNCHER" />
   </intent-filter>
     <intent-filter>
         <action android:name="android.intent.action.VIEW" />
         <category android:name=
         "android.intent.category.DEFAULT"/>
         <category android:name=
         "android.intent.category.BROWSABLE"/>
         <data android:scheme="invoked"/>
     </intent-filter>
 </activity>
```

3. Now we need to open our `index.android.js` file and make our UI react to being invoked.
4. Add the `Linking` module to the `import` block from `'react-native'`.
5. Replace the implementation with the following code:

```
componentWillMount() {
  this.setState({
    status: 'App Running'
  });
  Linking.addEventListener('url', this.onAppInvoked);
}
componentWillUnmount() {
 Linking.removeEventListener('url', this.onAppInvoked);
}
onAppInvoked = (event) => {
  this.setState({
```

```
      status: `App Invoked by ${event.url}`
    });
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.instructions}>
          App Status:
        </Text>
        <Text style={styles.welcome}>
          {this.state.status}
        </Text>
      </View>
    );
  }
```

6. Running the application and invoking it from another app will look something like this:



How it works…

In this recipe, we register our URL schema for linking by editing the `AndroidManifest.xml` file in step 2. An important bit to note is the change of the `launchMode` to `singleTask`. This prevents the operating system from creating multiple instances of our React Activity. This is important if you want to be able to properly capture the data passed along with the intent.

Invoking an external iOS and Android application

The average iOS user has dozens of apps installed on their device. A substantial percentage of them is used on a daily basis. Many of the apps today incorporate sharing and integration with other applications. An example of this would be opening the iOS Maps application for a particular address found in the host application. We can accomplish the same with React Native.

In this recipe, we are going to explore how to switch applications and pass some context to them on iOS. We will be using the built-in Linking module and opening the Maps, Phone, and Safari browser applications.

How to do it…

1. Open your `index.ios.js` file and add the Linking module to our import block:

```
import {
  //...
  TouchableOpacity,
  Linking
} from 'react-native';
```

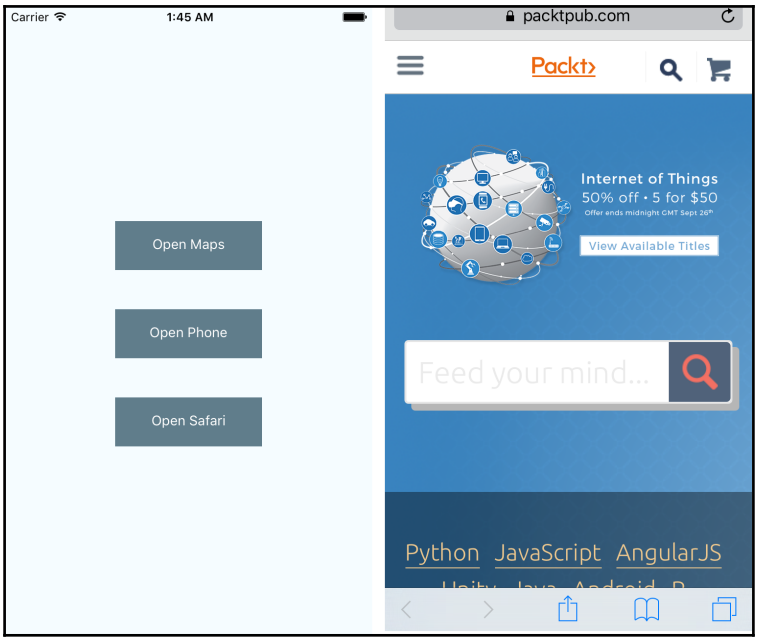
2. Next, we are going to create a generic method to link to the apps. Add the following to the class implementation:

```
openExternalApp(url) {
  Linking.openURL(url).catch(err => console.error('error
  opening external link', err));
}
```

3. Finally, we will add a few examples to our `render` function. This will be the JSX:

```
<View style={styles.container}>
  <TouchableOpacity onPress=
  {()=>this.openExternalApp('maps://')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Maps</Text>
    </View>
  </TouchableOpacity>
  <TouchableOpacity onPress=
{()=>this.openExternalApp('tel:18005551212')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Phone</Text>
    </View>
  </TouchableOpacity>
  <TouchableOpacity onPress={()=>
  this.openExternalApp('http://packtpub.com')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Safari</Text>
    </View>
  </TouchableOpacity>
</View>
```

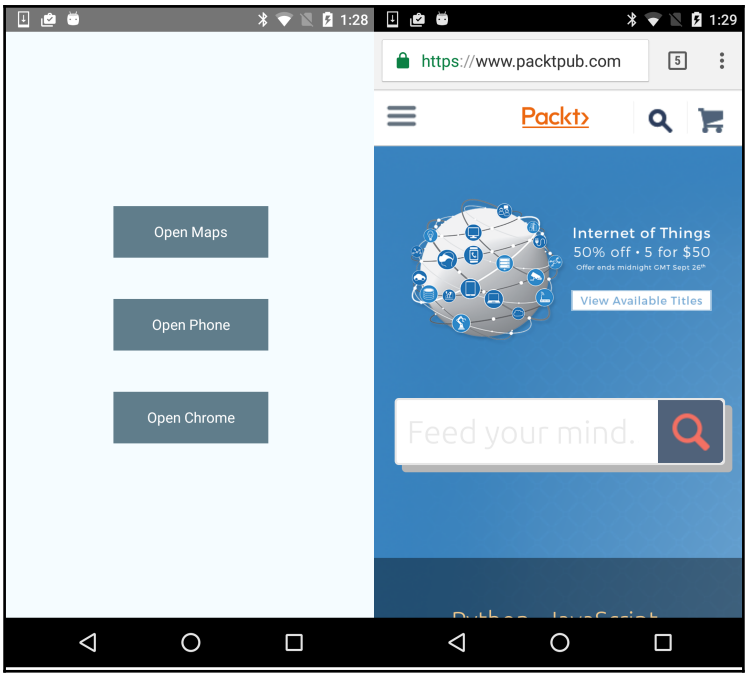
4. If we run the application, it should look something like this:



5. Open `index.android.js` and repeat steps *1* and *2*.
6. Change your `render` function to return the following JSX:

```
<View style={styles.container}>
  <TouchableOpacity onPress=
  {()=>this.openExternalApp('geo://')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Maps</Text>
    </View>
  </TouchableOpacity>
  <TouchableOpacity onPress=
  {()=>this.openExternalApp('tel:18005551212')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Phone</Text>
    </View>
  </TouchableOpacity>
  <TouchableOpacity onPress=
  {()=>this.openExternalApp('http://packtpub.com')}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Open Chrome</Text>
    </View>
  </TouchableOpacity>
</View>
```

7. Running the app on Android should look and function like this:



How it works…

Fortunately, React Native handles all the heavy lifting of opening other applications in the `Linking` API. For iOS, this involves executing the `openURL` method of the `UIApplication` instance. For Android, an `Intent` is created with the URL. The target operating system then attempts to open the URL.

9
Deploying Our App

In this chapter, we will cover the following recipes:

- Deploying development builds to an iOS device
- Deploying development builds to an Android device
- Deploying testing builds to HockeyApp
- Deploying testing iOS builds to TestFlight
- Deploying production builds to the Apple app store
- Deploying production builds to Google Play Store
- Deploying over-the-air updates
- Optimizing React Native application size

Introduction

So far throughout this book, we have discussed and showed you how to develop awesome React Native applications. As you are developing your application, you are going to hit a few stages. The first will be wanting to test your application on your personal iOS or Android device. After you are satisfied with how the app is working on your phone, you're going to want to share it with a select group of people and hopefully receive praise and adoration. Finally, you're going to reach a point where you think the world is ready to see your prized creation and you're going to release it to the designated app stores. This chapter will walk you through each one of these steps, as well as taking it one step further with pushing updates to your application and some optimization tips. For iOS, related deployment tasks, please make sure you are an **Apple Developer Program** member so you can sign your application.

Deploying development builds to an iOS device

Let's say that you are some way into building the next greatest iPhone app that will shatter app store records. You've spent the majority of your time testing your app using the iOS Simulator that comes with Xcode. While the iOS Simulator is by far the best performing and closest resembling method of running our application on an iOS device, it's still not the same as the real thing. The iOS Simulator uses the computer's CPU and GPU to render the Operating system simulation, so it may end up performing better than the actual device. When you test on a physical device you can experience the intangible factors of using the application, and see more realistic performance.

In this recipe, we will walk you through taking a React Native application and deploying it to your iPhone or iPad.

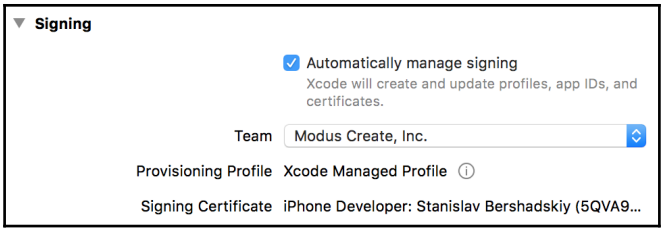
Getting ready

We created a sample React Native application named `TestDeployApp`. Make sure your device is connected via USB.

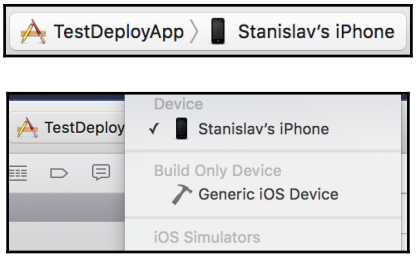
If you are not a member of the Apple Developer Program, you are able to sign your devices for development needs. You can read more about Apple's free (but limited) provisioning here: `https://goo.gl/cQAOdz`

How to do it…

1. Open your React Native iOS project in XCode.
2. Open Project Editor and select the app target.
3. Under the **Signing** section, select your **Team**:



4. Repeat step *3* for the Tests target (it will have your application name with a Tests suffix).
5. Select your device in the Destination selector:



6. To start your application on the device, press the Play button. Please make sure your device is plugged in and unlocked.

How it works…

Deploying our development build to the device simply involves running the application as we would for the Simulator but targeting the device. We use the localhost packager to create our bundle file. This file then gets saved locally on the device for the future. Please note, though, that since this is a development build, the code is not the slightest bit optimized. You will see a significant performance increase when moving to a production release.

Deploying development builds to an Android device

Naturally, when you develop an Android application, you need to test the application as you are building it. There are several options you have for testing the application locally. They involve running an Android emulator, the two most popular options being the one packaged with the Android SDK tools and Genymotion. Unfortunately, these emulators show poor performance and are cumbersome to use. The best solution is to use an actual physical device.

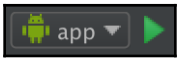
This recipe will walk you through deploying a React Native application to a physical Android device.

Getting ready

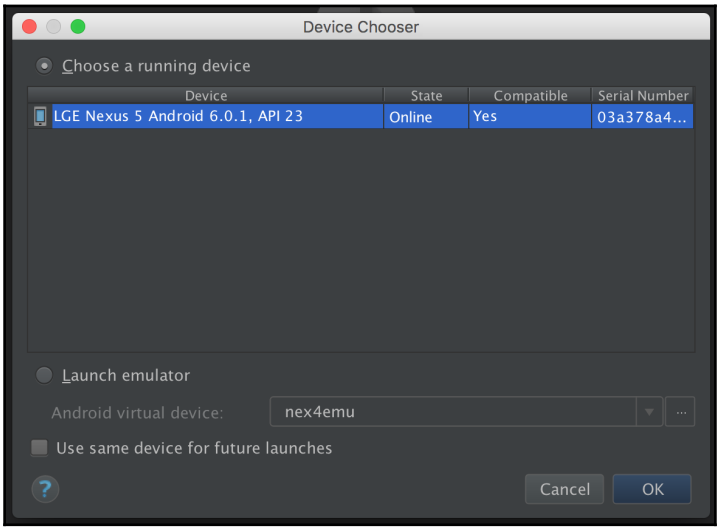
We created a sample React Native application named `TestDeployApp`. Make sure your device is connected via USB.

How to do it…

1. Open your React Native Android project in Android Studio.
2. Press the Run button:



3. Make sure the **Choose a running device** radio button is selected and your device is in the list; press **OK**:



There's more…

The React Native packager should start when you run the application. If it doesn't, please manually start the packager. If you see an error screen with the message **Could not get BatchedBridge**, **please make sure your bundle is packaged correctly**, or **Could not connect to development server**, run the following command in the terminal:

```
$ adb reverse tcp:8081 tcp:8081
```

How it works…

Deploying our development build to the device simply involves running the application as we would for an emulator, but targeting the device. We use the localhost packager to create our bundle file. This file then gets saved locally on the device for the future. The only caveat is setting up communication between the device and the development machine. Running the `adb reverse` command establishes a port forward from the device to the host computer. Please note, though, that since this is a development build, the code is not optimized the slightest bit. You will see a significant performance increase when moving to a production release.

Deploying testing builds to HockeyApp

So, you've made some excellent progress building the next best mobile app using React Native. You can see the finish line approaching, and you decide you want to open your app up for some feedback from a select group of people. To accomplish this, you need to create a signed build of your application that you will share with your group of testers. For a proper testing build, you need two things: some form of analytics or reporting and most importantly, a delivery mechanism. HockeyApp provides this and more for your testing builds on both iOS, Android, and other popular platforms.

This recipe will walk us through deploying a React Native application to HockeyApp for testing purposes. We will walk through both iOS and Android releases.

Getting ready

For this recipe, we will be using a simple React Native application called `TestDeployApp`. For iOS deployments, you will need to be enrolled in the Apple Developer Program and have `cocoapods` installed. Finally, you will need to have a HockeyApp account.

How to do it…

1. First we need to install the `react-native-hockeyapp` module in our application. Open the Terminal, go to your application's root project directory and enter the following command:

```
$ npm install react-native-hockeyapp --save
```

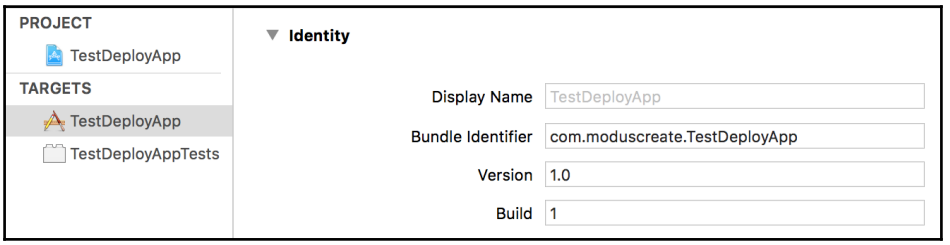
2. Go into your `ios/ directory` and initialize your Podfile:

```
$ pod init
```

3. Open your Podfile and add `pod "HockeySDK"` to your target.
4. Back in the Terminal, install the Podfile:

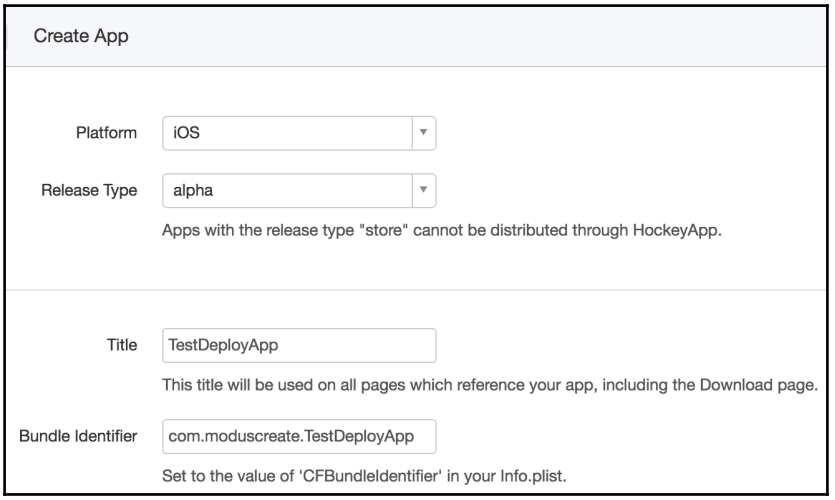
```
$ pod install
```

5. Now let's open up Xcode and open our React Native project: (`ios/TestDeployApp.xcodeproj`).
6. We recommend changing your **Bundle Identifier** to something more meaningful than the default, so please change it in your **General Settings** dialog:



7. Drag and drop `./ios/Pods/Pods.xcodeproj` into the **Libraries** group in your project navigator.
8. Drag and drop `RNHockeyApp.h/.m` files located in `./node_modules/react-native-hockeyapp/RNHockeyApp` into the same **Libraries** group.
9. Now we need to go into HockeyApp and create our application there. Log in and click **New App**.

10. We do not have our build ready yet so click **Manually**.
11. Fill out the fields to match the **Title** and **Bundle Identifier** that we defined earlier in step *6*. Press **Save**:

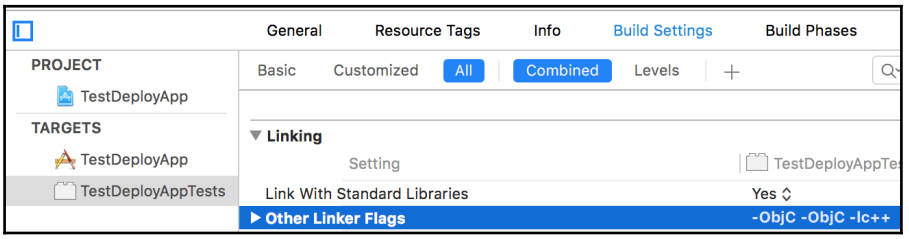


12. Make a note of the **App ID**.
13. Open `index.ios.js` and add the following code:

```
import HockeyApp from 'react-native-hockeyapp';

export default class
TestDeployApp extends Component {
  componentWillMount() {
    HockeyApp.configure(YOUR_APP_ID_HERE, true);
  }

  componentDidMount() {
    HockeyApp.start();
    HockeyApp.checkForUpdate();
  }
}
```

14. Back in XCode, set **Generic iOS Device** as your destination target and build (*cmd* + *B*) the application.

    You may see an error, **Apple Mach-O Linker (Id)**. You can fix this by adding `-lc++` to the `TestDeployAppTests` **Other Linker Flags** in the **Build Settings**:



15. Now we need to create our `.ipa` file. Click **Product > Archive**.

If you run into build errors. Please open the `Podfile` and change the `HockeySDK` import to: `pod "HockeySDK", :subspecs => ['AlllFeaturesLib']`

16. It will open the **Archives** list, press **Export**.
17. Select **Save for Development Deployment** and press **Next**.
18. Your Provisioning Team should automatically be selected, press **Choose**.
19. Leave the default Export settings and press **Next**.
20. On the summary, press **Next**.
21. Select the destination directory and press **Export**.
22. Back in the HockeyApp browser window, click **Add Version**.
23. Drag the `.ipa` file we just exported into the modal window.
24. Keep pressing **Next** and then finally click **Done** at the summary. You are now all set; you can add users to your HockeyApp application and they will be able to download your application.
25. Let's switch over to the Android side of things. Open Android Studio and open your React Native project.

26. Open `settings.gradle` and edit it to include the following:

```
include ':react-native-hockeyapp',
':app' project
(':react-native-hockeyapp').projectDir =
new File(rootProject.projectDir,
'../node_modules/react-native-hockeyapp/android')
```

27. Open `android/build.gradle` and add the following to the dependencies object:

```
classpath 'net.hockeyapp.android:HockeySDK:3.7.0'
```

28. Open `android/app/build.gradle` and add the following to the dependencies object:

```
compile project(":react-native-hockeyapp")
```

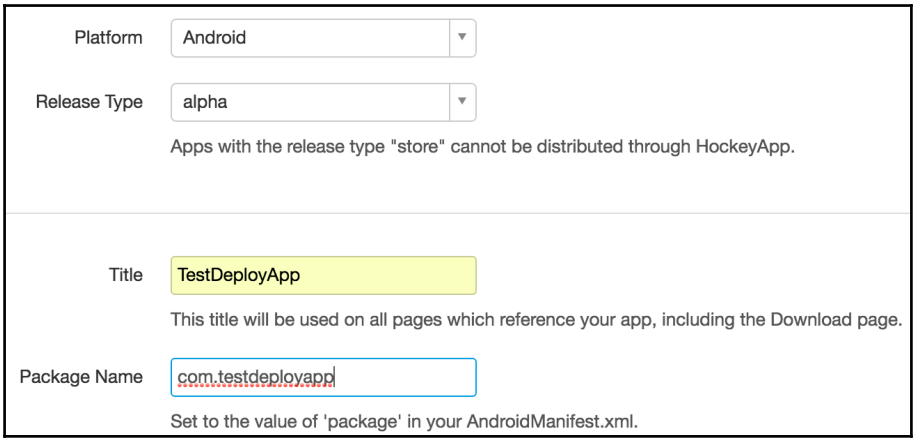
29. Now open up `AndroidManifest.xml` and add the following activities in the `<application>` node:

```
<activity android:name=
"net.hockeyapp.android.UpdateActivity" />
<activity android:name=
"net.hockeyapp.android.FeedbackActivity" />
```

30. Finally, we need to register the package in `MainApplication.java`. Your `getPackages` method should contain the following:

```
return Arrays.<ReactPackage>asList(
     new MainReactPackage(),
     new RNHockeyAppPackage(MainApplication.this)
 );
```

31. Repeat steps *9–13* but the **Platform** should be set to **Android**:



32. Now we need to build our `.apk` file. If you are not sure how to do it, follow the steps in the React Native documentation at `https://facebook.github.io/react-native/docs/signed-apk-android.html`.
33. Repeat steps *22* and *23* for our Android project. Now you're all set!

How it works…

For this recipe, we are using HockeyApp for two main features. We are using it for its beta distribution primarily, and we leverage its HockeySDK, which can support crash reporting, metrics, feedback, authentication, and in our case, notifications for updates. For iOS, beta distribution is done through the over-the-air enterprise distribution mechanism that is hosted by HockeyApp. When you sign your application, you control which devices can open the application. HockeyApp just sends the notifications and provides the URL for downloading it through its enterprise app store. Android is quite a bit simpler as there is no governance over how applications must be transferred. Therefore, HockeyApp hosts the `apk` on a web server that the tester can download and install the application from.

Deploying testing iOS builds to TestFlight

Before HockeyApp, the de facto service for beta testing distribution of mobile applications was TestFlight. In fact, it was so good that Apple purchased its parent company and integrated it straight into **iTunes Connect**. TestFlight now serves as the application testing platform for Apple. There are a few differences between TestFlight and HockeyApp to consider. First and foremost, TestFlight became iOS only when it was purchased by Apple. Secondly, there are two styles of testing in TestFlight, internal and external. **Internal testing** involves sharing the application with Developer or Admin role members of your team. There is a limit of 25 testers, with 10 devices each. **External testing** allows you to invite up to 2,000 testers, who do not have to be members of your organization. This is important, as these testers do not use up your device quota. External testing applications go through a **Beta App Review** completed by Apple. This is not as strict as Apple's review for going to the App Store, but is a good starting point.

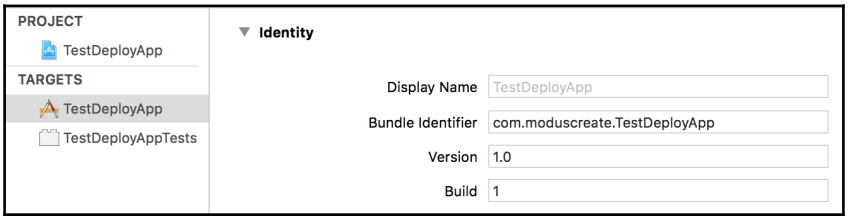
This recipe focuses on taking our React Native app and deploying a testing build to TestFlight. We will be setting up an internal test due to Apple's review process. The procedure is the same for external testing.

Getting ready

For this recipe, we will be using a simple React Native application called `TestDeployApp`. You will need to be enrolled in the Apple Developer Program and have your development and distribution certificates set up in Xcode. Your application will need to have its AppIcon set.

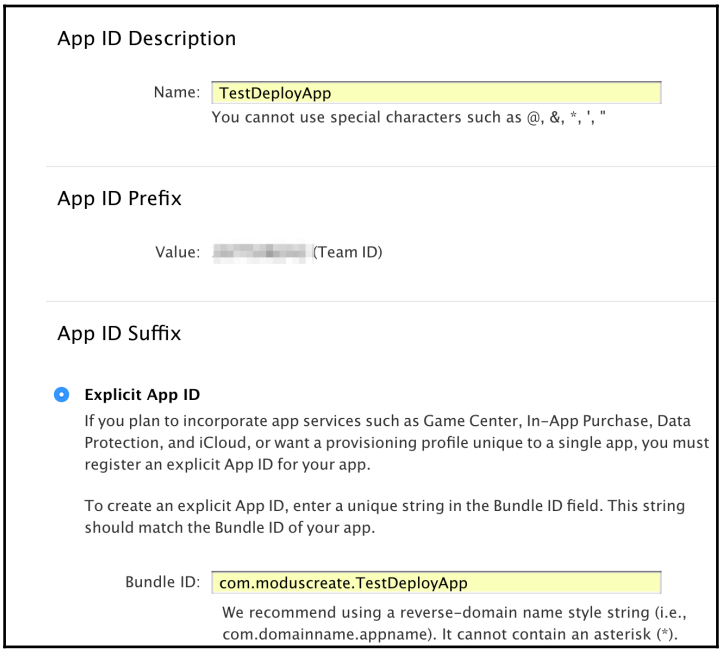
How to do it…

1. Open up Xcode and open our React Native project: (`ios/TestDeployApp.xcodeproj`).
2. We recommend changing your **Bundle Identifier** to something more meaningful than the default, so please change it in your **General Settings** dialog:

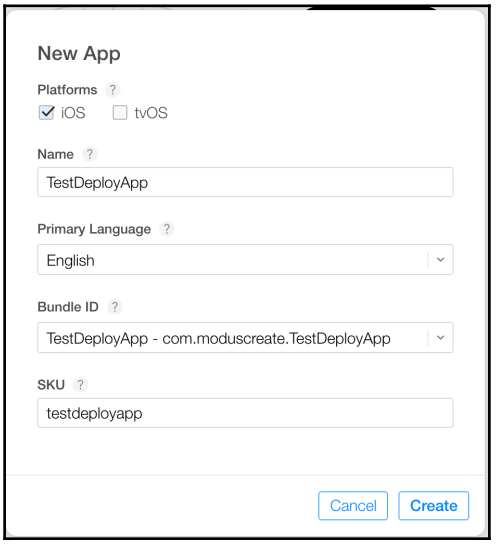


3. Log into your Apple Developer Program and go to the App IDs registration at `https//:developer.apple.com/account/ios/identifier/bundle`.

4. Fill out the **Name** and **Bundle ID** for your project, then press **Continue**, followed by **Register** and finally, **Done**:

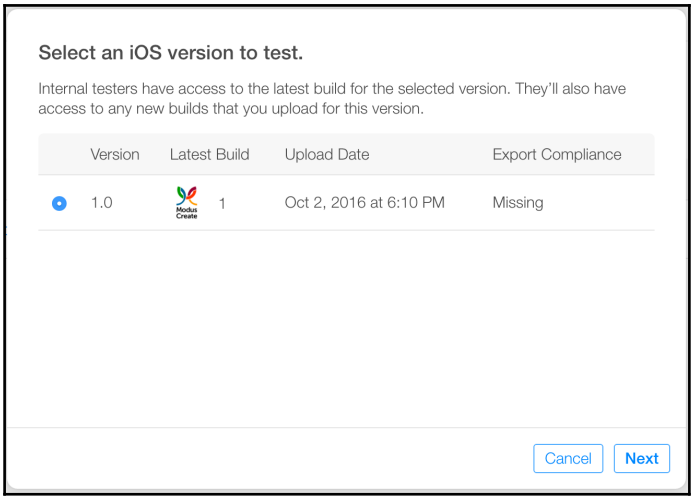


5. Log into iTunes Connect `https://itunesconnect.apple.com`.
6. Open **My Apps**.
7. Press the Plus (+) button and select **New App**.
8. Fill out the **Name** and **Language**. Select the **Bundle ID** to match the one you just created in step *4*. Write a unique application reference for the **SKU**. Press **Create**:

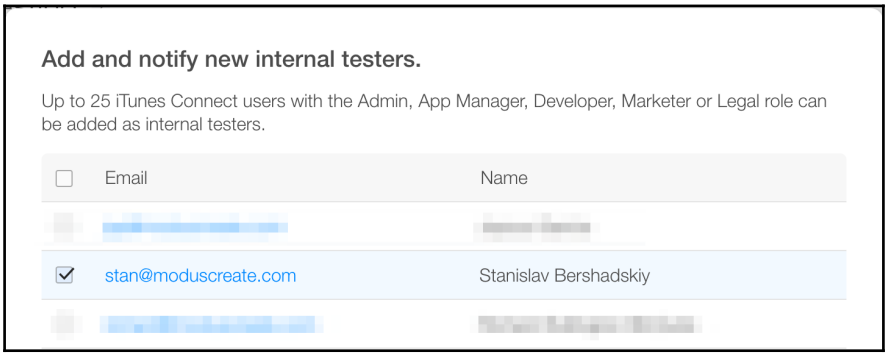


9. Open the TestFlight section for your app and fill out the **Localizable Information**.
10. Go back to XCode and lets create the `.ipa` file. Select **Generic iOS Device** for the active scheme. Click **Product | Archive**.
11. It will open the **Archives** list; press **Upload to App Store…**.
12. Your Provisioning Team should automatically be selected; press **Choose**.
13. After the archive is created, press **Upload**.
14. Wait till you receive an e-mail from iTunes Connect informing you that the build has completed processing.
15. Go back to the iTunes Connect browser window and open the **Internal Testing** view.

16. Click **Select Version to Test** and pick your build. Click **Next**:



17. At the **Export Compliance** screen press **OK**.
18. Add an internal tester and select the users you would like to test the application:



19. Finally, click **Start Testing** and confirm the modal. Your users should now get an invitation e-mail to test your awesome app!

How it works…

TestFlight serves as a first-class citizen in the App Store publishing path. Apple has integrated its support for application beta testing distribution directly into iTunes Connect. You follow the same procedure as you would when you deploy to the App Store, however, on iTunes Connect, you enable and configure the testing. For the user it is a seamless experience: they are notified to install the TestFlight app. There it will hold a link to the applications they can test. The developers benefit as they do not need to add any extra libraries or code to support TestFlight as they would with HockeyApp.

Deploying production builds to the Apple app store

You've spent the last significant portion of time, most likely more than you anticipated, building and testing your killer React Native app. Finally, the time has come to get your app in the Apple app store and release it to the masses.

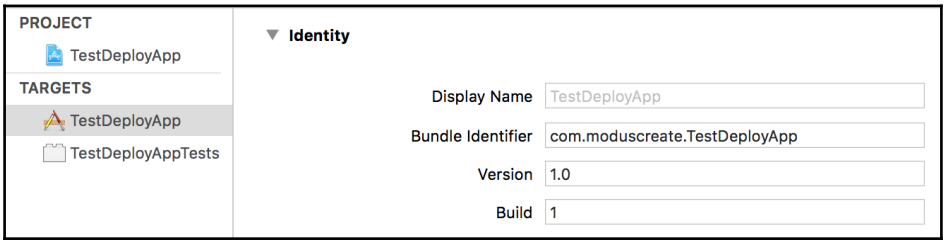
This recipe will walk us through the process of preparing our production build and submitting it to the Apple app store. This will stop right before we submit to the app store itself, as this is outside of React Native's scope and would require us to have a truly production-ready application. The rest of the process is pretty straightforward.

Getting ready

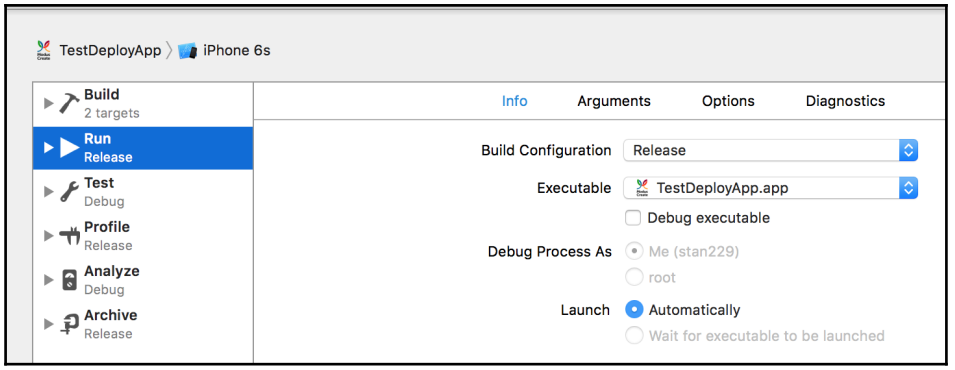
For this recipe we will be using a simple React Native application called `TestDeployApp`. You will need to be enrolled in the Apple Developer Program and have your development and distribution certificates setup in Xcode. For proper production deployments, you will need to have AppIcon set and iTunes screenshots ready.

How to do it…

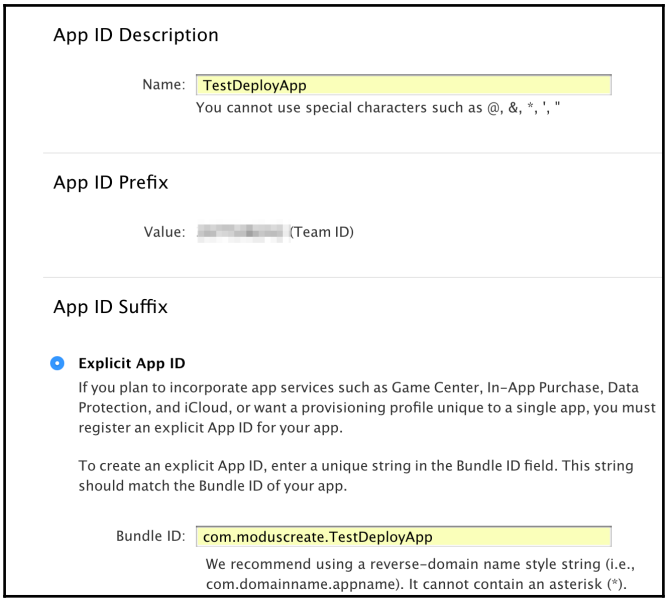
1. Open up Xcode and open our React Native project (`ios/TestDeployApp.xcodeproj`).
2. We recommend changing your **Bundle Identifier** to something more meaningful than the default, so please change it in your **General Settings** dialog:



3. If you want to test your application in Production Mode on your device, change your scheme's **Build Configuration** (**Product** | **Scheme** | **Edit Scheme…**) to **Release**:

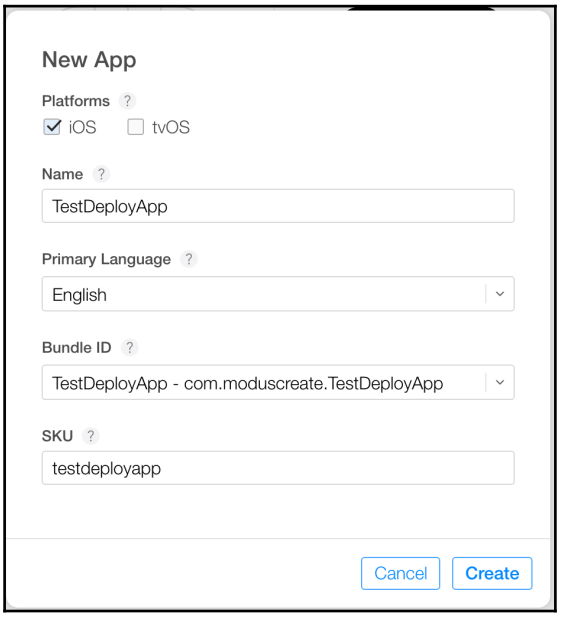


4. Log into your Apple Developer Program and go to the App ID's registration at `https://developer.apple.com/account/ios/identifier/bundle`.
5. Fill out the **Name** and **Bundle ID** for your project and press **Continue**:



6. Log into iTunes Connect at `www.itunesconnect.apple.com`.
7. Open **My Apps**.
8. Press the Plus (+) button and select **New App**.

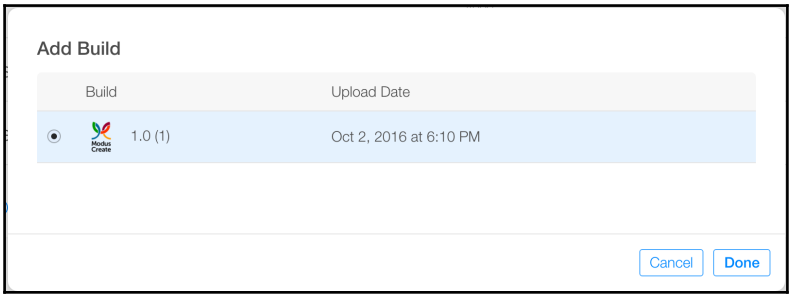
9. Fill out the **Name** and **Language**. Select the **Bundle ID** to match the one you just created in step *4*. Write a unique application reference for the **SKU**. Press **Create**:



10. Go back to XCode and let's create the `.ipa` file. Select **Generic iOS Device** for the active scheme. Click Product | Archive.
11. It will open the **Archives** list; press **Upload to App Store…**.
12. Your Provisioning Team should automatically be selected, press **Choose**.
13. After the archive is created, press **Upload**.
14. Wait till you receive an email from iTunes Connect informing you that the build has completed processing.
15. Under the **App Store** section, open **App Information** and select the category your application will reside in.
16. Open the **1.0 Prepare for Submission** section under **iOS APP**.

17. Fill out all the required fields, including **App Screenshots**, **Description**, **Keywords**, and **Support URL**.

18. Under **Build**, Select a build before you submit your app and choose your build that you submitted in step *13*:



19. Fill out **Copyright** and **App Review Information**.
20. Click **Submit for Review** and cross your fingers!

How it works…

This process is standard for publishing iOS applications to the App Store. There are no React Native specific steps we have to do in this case, since there are scripts added by the framework that hook into Xcode. These scripts will run the React Native packager to build the `main.jsbundle` file in release mode.

Deploying production builds to Google Play Store

Your Android labor of love is finally coming to an end. The next greatest thing to hit potentially almost 11,000 devices (yes, really) is ready to go live. How do you reach these users? You need to deploy your app to the Google Play Store of course.

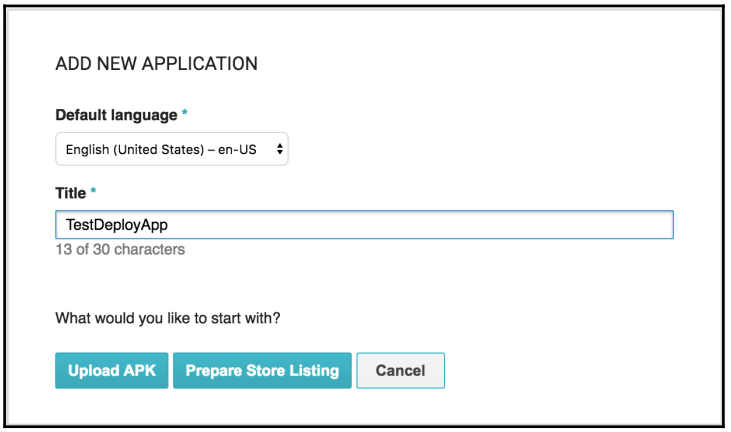
This recipe will walk us through the process of preparing our production build and submitting it to the Google Play Store. This will stop right before we submit to the app store itself, as this is outside React Native's scope and would require us to have a truly production-ready application. The rest of the process is pretty straightforward.

Getting ready

For this recipe, we will be using a simple React Native application called `TestDeployApp`. You will need to be registered for a Google Play Developer account. Finally, you will need to have all the icons and screenshots ready for the Play Store for final publishing.

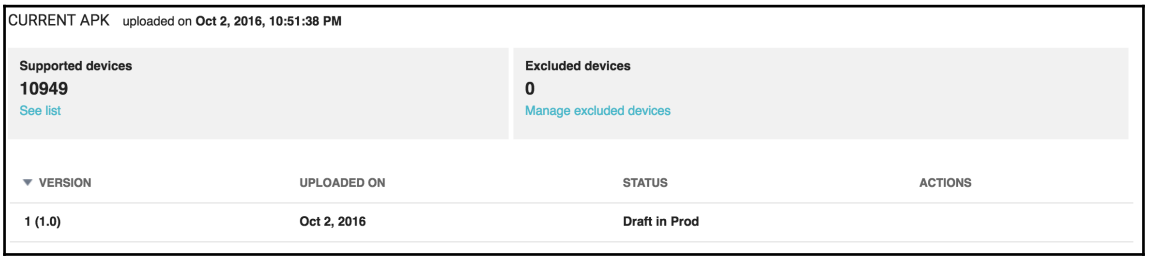
How to do it…

1. Open the React Native project in Android Studio.
2. Now we need to build our `.apk` file. If you are not sure how to do it, follow the steps in the React Native Documentation at `https://facebook.github.io/react-native/docs/signed-apk-android.html`.
3. In your browser, open the Google Play Developer Console: `https://play.google.com/apps/publish/`.
4. Click **Add new application**.
5. Fill out the **Title** and click**Upload APK**:



6. This will open the **APK** section of the Publish screen. Click **Upload your first APK to Production**.

7. Drag or select your `APK` file. After it uploads the page will be updated:



8. Next you need to go through the **S**tore Listing, Content Rating, Pricing, and Distribution sections and fill out all the necessary information.
9. Once you have satisfied all the requirements click **Publish App** and celebrate!

How it works…

This process is relatively standard for publishing Android applications to the Google Play Store. All the React Native specifics are handled in step 2 and by the gradle `assembleRelease` process. The assemble process runs the packager to create the JavaScript bundle file, compile the Java classes, package them together with the resources, and finally sign it into an `apk`.

Deploying Over-The-Air updates

An excellent side-effect of having our React Native application being written in JavaScript is that the code is loaded at runtime. This is similar to how Cordova hybrid applications work. We can leverage this functionality to push updates to our application Over-The-Air (OTA). This lets us add features and bug fixes without having to go through the App Store approval process. The only limitation to OTA updates for React Native is that we cannot push compiled (Objective-C or Java) code. The code has to be JavaScript only. There are a few popular services that provide cloud-based OTA application updates. We will be highlighting `CodePush`, a service by Microsoft. We feel this is the most mature, frequently updated and most importantly, easiest to integrate service.

This recipe will discuss setting up and pushing updates using `CodePush` for our React Native application on both iOS and Android.

Getting ready

For this recipe we will be using a simple React Native application called `TestDeployApp`. For our testing purposes we deployed the applications to physical devices running in production/release mode. This is necessary for the application to look at the `CodePush` servers for updates.

How to do it…

1. To use `CodePush`, we need to install the CLI and create a free account. In your terminal, run the following commands:

```
$ npm install -g code-push-cli
$ code-push register
```

2. Now we need to register our app with `CodePush`. Make note of the deployment keys for the applications; we will be using the **staging key** for this recipe. Run the following command for your app in the terminal:

```
$ code-push app add TestDeployApp
```

3. Still in the terminal, go into your React Native project directory and install the **React Native CodePush** `npm` module:

```
$ npm install --save react-native-code-push@latest
```

4. Now we need to link the Native Modules with our project. You will be prompted for your deployment key for Android and iOS, use the staging key from step *2*:

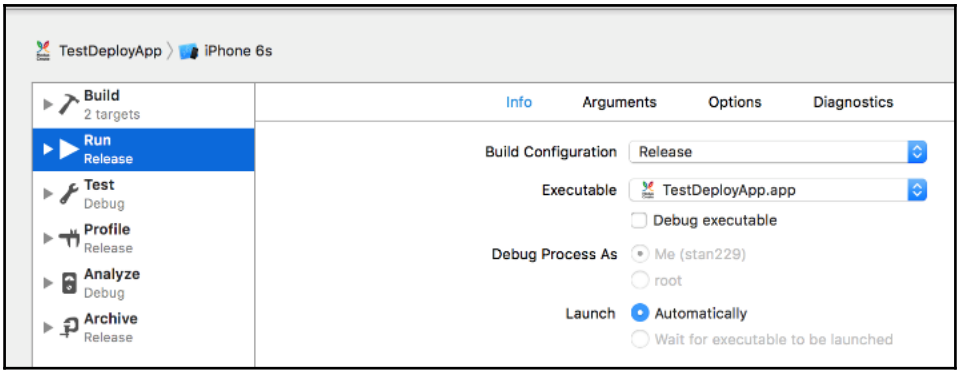
```
$ react-native link react-native-code-push
```

5. Next, we need to set our React Native application up to leverage `CodePush`. Open `index.ios.js` and add the following:

```
import codePush from 'react-native-code-push';
...
const codePushOptions = {
    updateDialog : true
}
AppRegistry.registerComponent('TestDeployApp',
() => codePush(codePushOptions)(TestDeployApp));
```

6. Repeat step *5* for `index.android.js`.

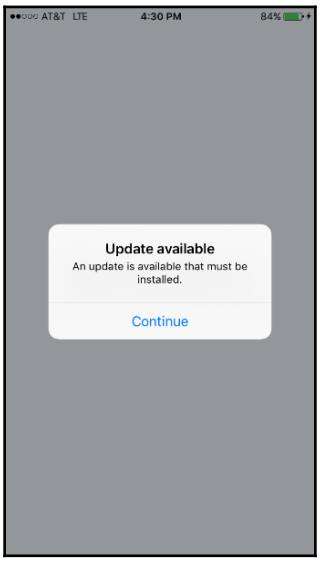
7. To test our iOS application let's deploy to our iOS device. Open the project in XCode and change your scheme's **Build Configuration** (**Product | Scheme | Edit Scheme…**) to **Release**. Press **Run**:



8. Change some of the code in your `index.ios.js`.
9. In the terminal run the following command:

```
$ code-push release-react
TestDeployApp ios -m --description
"Updating using CodePush"
```

10. Close and re-open your app on your iOS device and you should see the following prompt and change:



11. After the prompt, the application will update itself to the latest version:

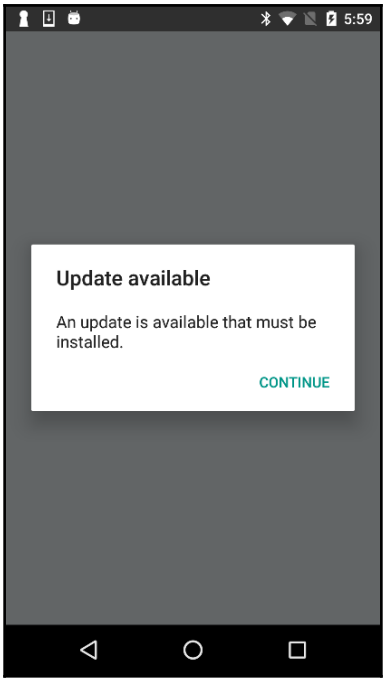
12. Now let's test on Android. First we need to create our `.apk` file. If you are not sure how to do it, follow the steps in the React Native Documentation at `https://facebook.github.io/react-native/docs/signed-apk-android.html`.
13. Make sure your Android device is plugged in and run the following command in the terminal from the `android/` directory:

```
$ adb install
app/build/outputs/apk/app-release.apk
```

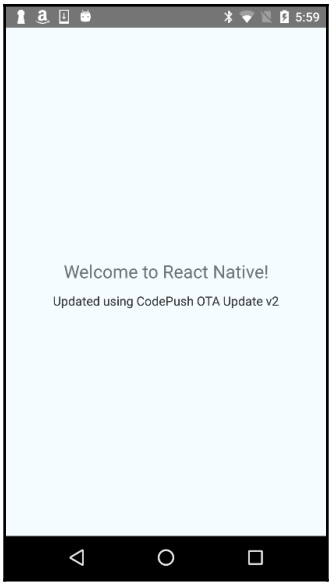
14. Change some of the code in your `index.android.js`.
15. In the terminal, run the following command:

```
$ code-push release-react TestDeployApp
android -m --description "Updating using CodePush"
```

16. Close and re-open your app on your Android device and you should see the following prompt and change:



17. After the prompt, the application will update itself to the latest version:



How it works…

`CodePush` as well as other cloud-hosted OTA update platforms, works on the same technique that has existed in React Native since its inception. React Native needs to load a JavaScript bundle when it is initialized. When we are developing the application, it looks for the React Native package that is running at `localhost:3000`. When we are deploying our application, it looks for a file named `main.jsbundle` that is attached to the application. Every time the application is opened it checks in with the `CodePush` API and sees if there is an update for the application. If there is a new update it may prompt the user about it. Download the new `jsbundle` file and restart the application.

Optimizing React Native application size

As we are ready to deploy our app to production, we may get to the point where we need to shrink our application bundle size. There are several techniques we can employ to cut some space off our packages. These may involve some sacrifices to supported devices or asset quality. Naturally, with any space-saving operations there will be consequences; otherwise they would be enabled by default.

This recipe will discuss some techniques to limit production package file size in both iOS and Android React Native applications.

Getting ready

For this recipe we will be using a simple React Native application called `TestDeployApp`. You also need to have code signing working for iOS and the ability to create `apk` files. If you need help refer to any of the previous testing and production build recipes.

How to do it…

1. First we will start off with some optimizations with our bundled assets. Often, you have to include image assets, external fonts, and other data files with your application. Let's look at some optimization techniques for these.
    - For PNG and JPEG lossy compression, you can use a service such as `www.tinypng.com`. The service claims that it can reduce file size upwards of 70%.
    - If you use the `react-native-vector-icons` library, you will notice that it bundles eight different font icon sets. Feel free to remove the icon font libraries that you are not using.
    - SVG files can also be compressed and optimized. You can use the service `compressor.io` for this.
    - Audio assets should be compressed to AAC or MP3.
2. On the JavaScript side of things, you have to keep in mind that **Babel** will take your `ES6/7/2015` code and transpile it. The transpilation will cause polyfills to be generated and used for these features. This will result in a substantially larger file size of your `jsbundle`. You can limit this by using fewer ES6 features. You should consider though the debate of **performance** versus **maintainability**. You can read more about this here at `http://moduscreate.com/optimize-es6-output-size-performance-via-babel/`.
3. For iOS there is not much we can do at the moment as all the best practices for reducing `ipa` size are enabled on the release scheme. These practices include enabling Bitcode for app thinning and setting the compiler optimization to **Fastest, Smallest [-Os]**.

4. For Android there is some room for reducing the file `apk` file size. In Android Studio, open `android/app/build.gradle` and change the following lines to the following values:

```
def enableSeparateBuildPerCPUArchitecture = true
def enableProguardInReleaseBuilds = true
```

5. If you plan to only target ARM-based Android devices, we can prevent it from building for x86 altogether. In `build.gradle` `splits abi` object, change the following line to not have x86 anymore:

```
include "armeabi-v7a"
```

How it works…

We discuss several techniques that you can employ for reducing your package bundle file size. The smaller we keep our JavaScript bundle the faster the JavaScript interpreter will parse the code and thus load the application. The smaller we keep our `ipa` and `apk` the faster users will be able to download the application, and if our application is huge it may mean the difference between marking as Wi-Fi only on the app stores. Asset optimizations in step *1* should provide significant reduction, and the result should be negligible to the user. JavaScript optimizations in step *2* will provide some marginal benefit to the user, but may have a more significant negative impact on the developers. The Android optimizations are relatively straightforward; however the Proguard enabling may impact native libraries and require some specific rules.

10
Automated Testing

In this chapter, we will cover the following recipes:

- Installing the environment
- Running the inspector to access the elements
- Integrating Appium with Mocha
- Selecting and typing into input texts
- Pressing a button and testing the result

In an ideal world, as developers, we receive the initial requirements at the beginning of the project, we take all these requirements and create a beautiful piece of software that works as it was defined. However, in the real world, this is not the case, requirements change as we build a product. In addition, once we release our amazing product to production, we need to keep adding new features and fixing issues, and things can get messy.

In order to keep a healthy code base, we need to hire a great QA Team that makes sure every change doesn't break anything of the existing functionality. Manual testing is OK but it might be expensive, therefore we should be able to automate our test. In this chapter, we will cover how to test functionality; this is also called functional testing. We will test an app from a previous recipe in this book we are going to use a real device to see if our final build doesn't introduce any bug or problem before releasing it to the public.

We are going to use Appium to run our automated tests. Appium is an open source project that, controls our device remotely. We can get elements in our app, such as buttons or inputs, then perform actions on them; after that we can validate the result and make sure everything runs as expected on every release. Appium runs an HTTP server; we can use any language to send an HTTP request, and in this case we will use the JavaScript client. The server sends instructions to the device to execute tasks remotely.

Installing the environment

In this recipe, we will install and configure our environment to run automated tests using Appium in a real device.

Getting ready

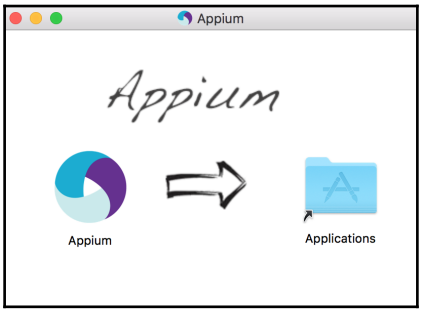
In order to go through this recipe, we need to have installed Node 6, Npm 3, Appium 1.5.3, and XCode 7.3.1. We also need a device to test; in this recipe, I will be using an iPhone 5S with iOS 8.4.

Download the `Appium` app from the home page: `http://appium.io/`

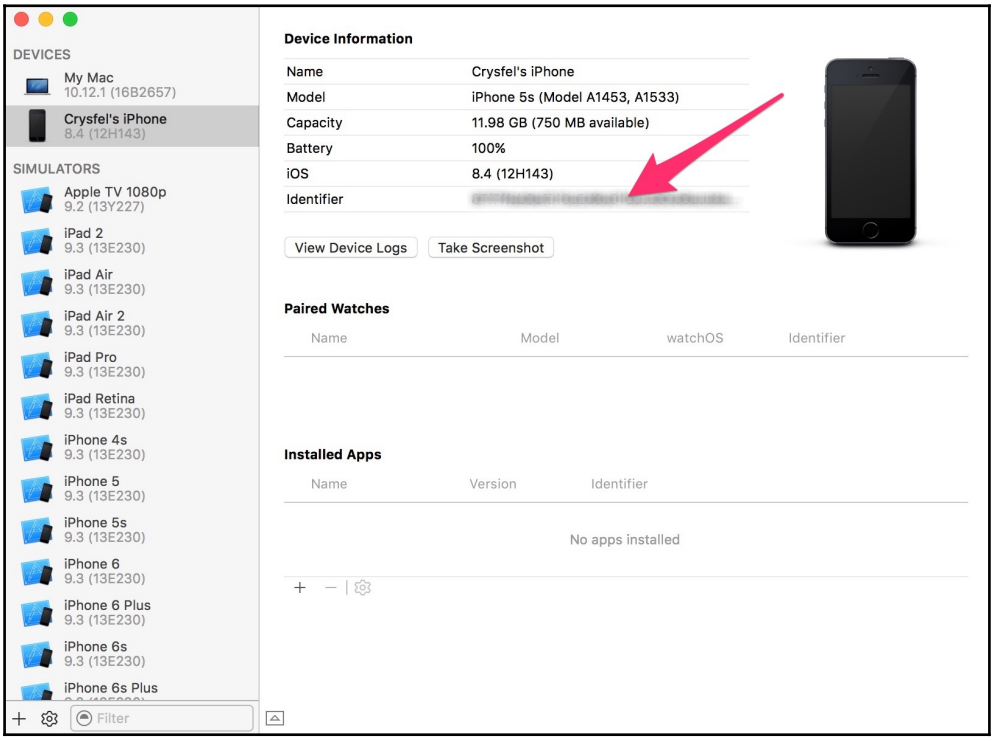
At the time of writing the `Appium` app doesn't support XCode 8.1 because Apple deprecated some internal APIs that allowed `Appium` to run. Appium 1.6 adds support for XCode 8.1 but unfortunately the app is not released yet and we need to manually build it from the source code and hack around.

How to do it…

1. Installing `Appium` is extremely easy. Once you have downloaded the app, we only need to copy the binary to the applications folder in our Mac as shown in the following screenshot:

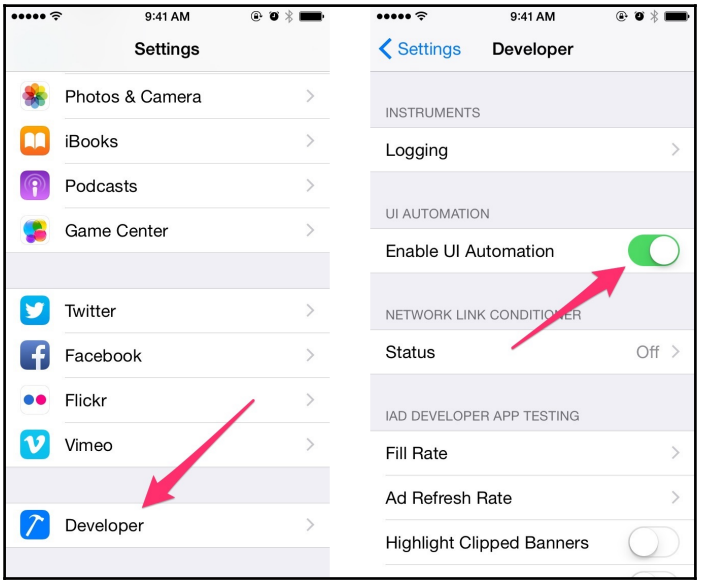


2. Before opening the `Appium` App, we need to configure our device to enable UI Automation. Connect your phone to your Mac and open XCode, then on the top menu, select **Window | Devices** (*cmd + shift + 2*). From here, we are going to need the **Identifier** number to configure Appium at later steps as shown in the following screenshot:

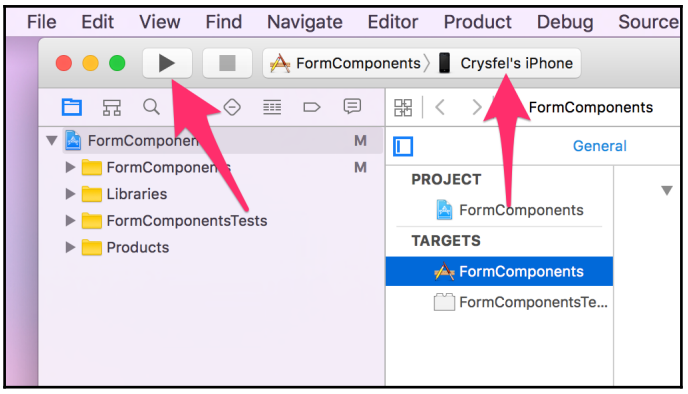


Make sure your device is already registered on the Apple developer center and the provisioning profile is installed on your phone. Otherwise you won't be able to enable automation.

3. On your phone, go to **Settings** | **Developer** and then **Enable UI Automation**. By default, it's disabled; make sure it is green in order to allow running our automated tests on this device as shown in the following screenshot:

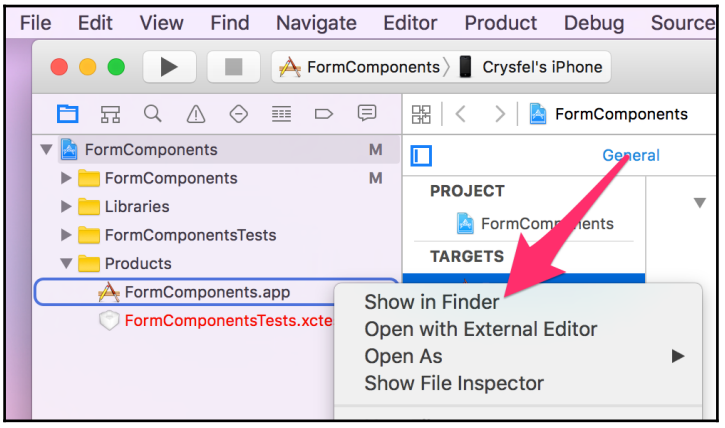


4. As mentioned before, we are going to use one of the apps we have created in previous chapters. Let's open the app created in `Chapter 2`, *Implementing Complex User Interfaces*, recipe number 10. We are going to use this app because it has a few input texts, a button, and an alert message. First, we need to make sure the build will run in our device; the easiest way to do it is to open the `ch2/tip10/FormComponents/ios/ FormComponents.xcodeproj` in XCode, then select your device from the drop-down menu and click **Run** as shown in the following screenshot:

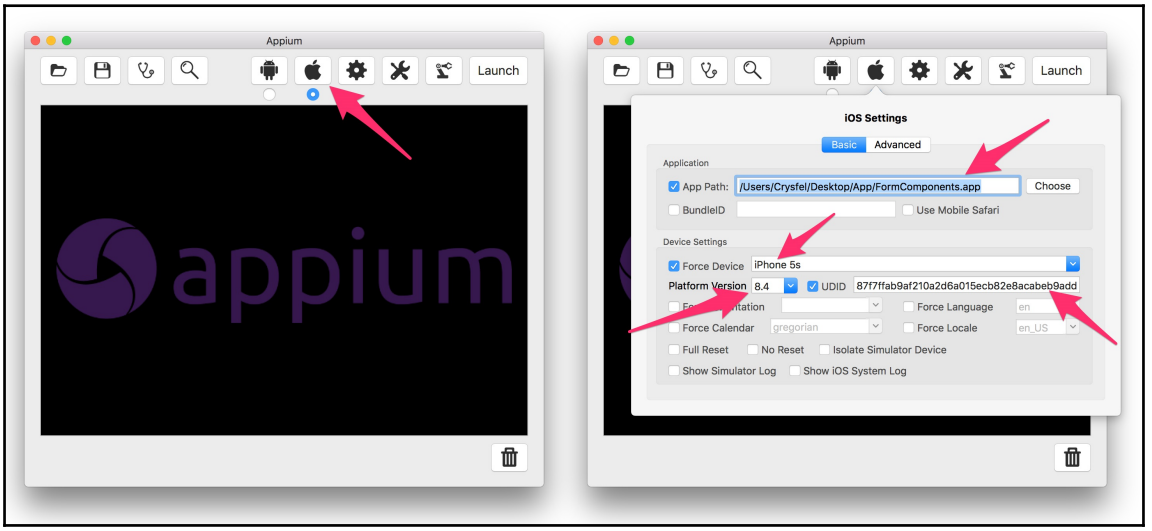


In order to successfully run the app on your device, you need to create and install the iOS Certificates on your system. You can do this by following the instructions on the Apple Developer platform: `https://developer.apple.com`.

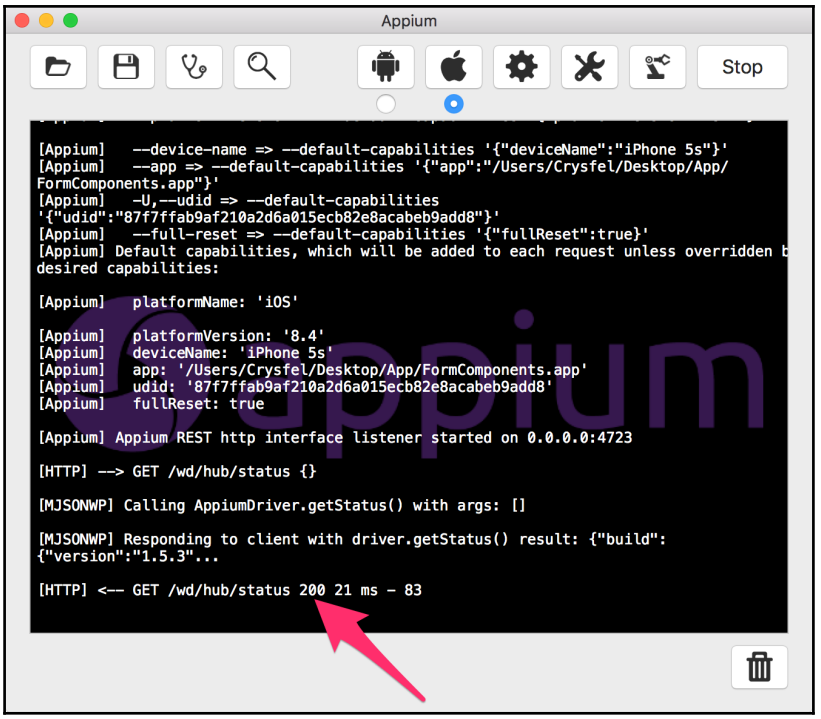
5. Once the app is successfully running on your device, we need to copy the `FormComponents.app` file to the desktop or any other destination that is convenient for you. This is the app we are going to test using `Appium`:



6. Now we are ready to configure `Appium`. Open the `Appium` app and set the following configurations:
    - **App Path**: The IPA file that we want to test. This file should be signed for development; we are going to use the app from step *5*.
    - **Force Device**: The device we want to test our app. In this case, I'm using my iPhone 5S.
    - **Platform Version**: The iOS version of the device where we are running our app.
    - **UDID**: The phone's UDID; we can get this value from step 2 of this recipe. This is required to automatically install the app in the device as shown in the following screenshot:



7. Finally, we should be able to run the server. Let's click the **Launch** button at the top-right corner. If we get 200 status responses from the server, we are ready to start writing some tests as shown in the following screenshot:



Running the Inspector to access the elements

In order to write our tests, we need a way to access the native elements on screen. After that, we should be able to input text on the fields, press the buttons, and run some validations to check that our app is working as expected from our tests.

In this recipe, we will run the `Appium` **Inspector** to get an XPath value; we are going to use this value to access the elements in our app through our JavaScript code in the tests.

Getting ready

In order to go through this recipe, it's necessary to complete the first recipe in this chapter. The Appium server should be running, we should have a `FormComponents.app` file to inspect, and don't forget to plug your phone to your Mac.

Make sure to run the React Native Packager; this is necessary because we are using a developer version of our app.

If you are having issues opening the inspector in step *1*, make sure to have `ideviceinstaller` installed, which is a command-line utility to install apps on iOS platforms from the command line.

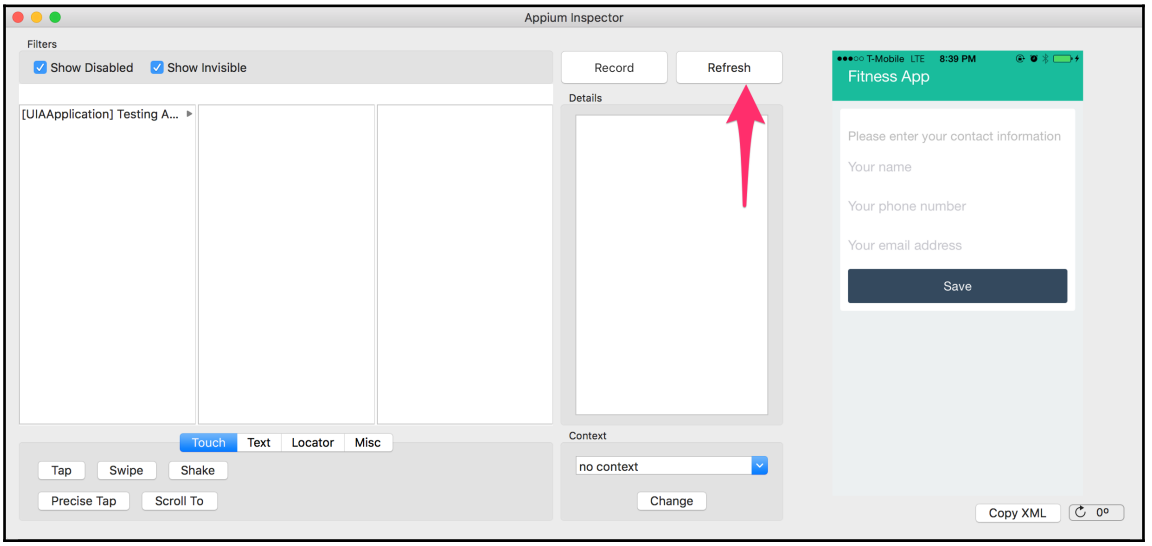
The `Appium` app comes with this utility, but sometimes the one included in the bundle doesn't work on our platform. In order to fix this problem, we need to use `brew` to manually install `ideviceinstaller` in our system:

```
$ brew install ideviceinstaller
```

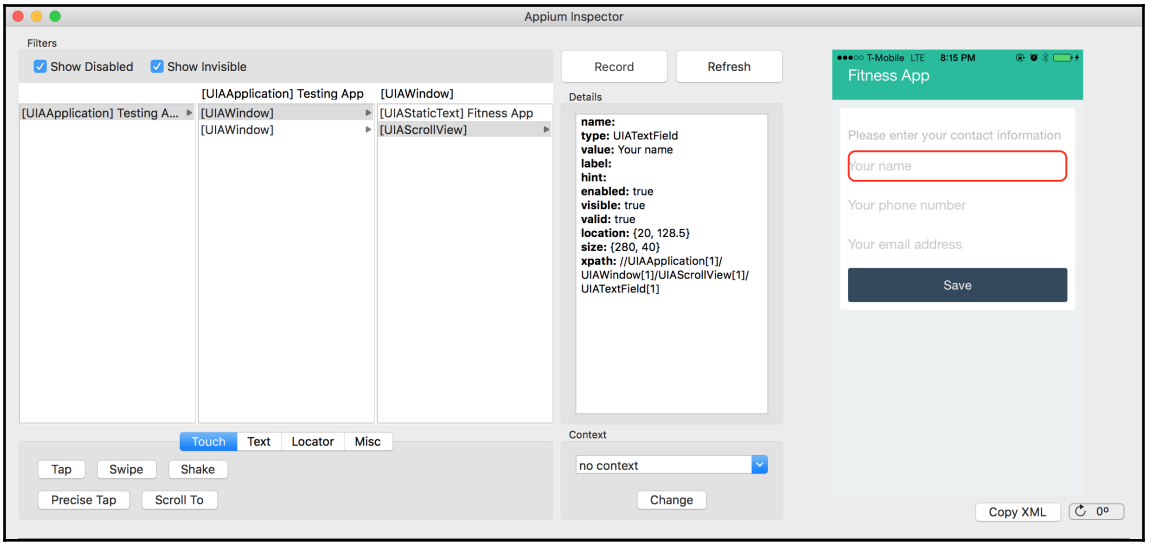
How to do it…

1. Once the `Appium` server is running, we can open the **Inspector** by clicking on the magnifier icon at the top. This will install the app on your device, then it will run the app on the device and will open the inspector app.

   The inspector will show the initial view, which is the default splash screen of the app. We need to click the **Refresh** button every time the phone shows a different view to inspect that specific view:



2. When the **Inspector** is running, we will see on the left a view of our app; in here, we can click on any of the elements and we will get the details for that element on the **Details** panel:



3. In the previous step, we inspected the details of the input text element. Once we know the XPath value, we can use it to find that element in our JavaScript tests. While the XPath displayed in the inspector will work perfectly, it might change if the app gets updated or there are several updates to the UI; therefore, it would be nice if we could use a more generic value; for example, we can use the following XPath value instead:

```
//UIATextField[@value='Your name']
```

4. In order to test our **XPath** values, we can get the XML representation of our views and then use any XPath editor. In the inspector, we need to click the **Copy XML** button; this will add to the clipboard the XML and we will be able to paste it anywhere we want.

Integrating Appium with Mocha

In this recipe, we will set up a new project to write our tests; we are going to use Mocha, chai, and the JavaScript Appium client to send commands to our device.

Getting ready

For this recipe, we will create a new repository for our automated tests. All the files and folders created here will not be on the same code base as our actual React Native app, it is going to be a completely different project.

How to do it…

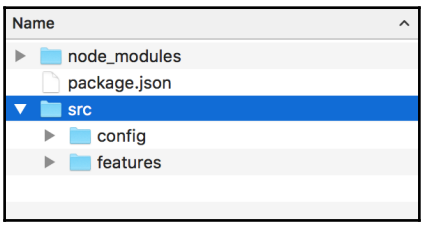
1. Let's start by creating a new folder for our project; we are going to have all our tests in here. Open the terminal, go to the new directory, and initialize the new node package with the following command:

```
$ npm init
```

2. Now we need to install all the dependencies for this project. We are going to use `npm` for this:

```
$ npm install --save chai
chai-as-promised mocha wd
```

3. Once we have all the dependencies installed, we need to create the folder structure. Let's create two folders, `src/config` and `src/features`. In the **config** folder, we will have the server Appium configurations; in the **features** folder, we will create our actual tests:



4. Open the `package.json` located on the root folder and add the following code to the scripts object. This will allow us to execute a simple command on the terminal to run all our tests using mocha:

```
"scripts": {
  "test": "./node_modules/mocha/bin/mocha
  ./src/features/"
},
```

5. Let's create the `src/config/index.js` file and define all the configurations that we are going to need for this project:

```
 // The server appium server configurations
 exports.appium = {
  host: 'localhost',
  port: 4723,
};

// Capabilities
exports.capabilities = {
  browserName: '',
  appiumVersion: '1.5.3',
  deviceName: 'iPhone 5S',
  platformVersion: '8.4',
  platformName: 'iOS',
  name: 'Sample Test',
  app: undefined
};
```

6. That's all! We are ready to start writing our tests using Mocha!

How it works…

In step *2*, we are adding all the dependencies for our automation project:

- `chai` is an assertion library. It will help us to accept or reject our tests.
- `chai-as-promised` is a library that allows chai to assert promises; we are going to use it because the Appium client returns promises for each task, therefore using this library will help us to keep a clean code base.
- `mocha` is a test framework that will allow us to run asynchronous tests easily. We can generate reports and run all our tests with a single command on the terminal.
- `wd` is the Appium JavaScript client library. All it does is to send HTTP requests to the server with the instructions that we need; for example, we can get elements on the UI, type in text inputs, or press buttons.

On step *4*, we are adding an alias to run our tests; this way, we can use npm to run all our tests as follows:

```
$ npm test
```

On step *5*, we are adding the configurations for `Appium`. We are going to define two objects, one object will contain the host and port and the other object will set the device capabilities; in here, we are defining the same configurations we did for the first recipe in this chapter.

If you noticed, we are not setting the `app` config, instead we are setting it to `undefined`, but we can set the path where our app's file is located. We are not doing it because we have already declared that on the `Appium` App and those properties will get applied if we don't set anything here.

Selecting and typing into input texts

In this recipe, we will write our first test. We are going to get the three input texts and type some values, then we will assert that the values are present in the fields.

Getting ready

For this recipe, we will continue working on the same project we created in the previous recipe.

We need to make sure the `Appium` server is running; please take a look at recipe number 1 in this chapter.

We also need to make sure the inspector is not running, otherwise the device will be busy running the app and won't be able to run our tests.

Just in case, I have to mention that the device should be connected to your Mac (sometimes I've forgotten this important step) and the React Native Packager running.

We need the packager running because our app will try to load the latest JavaScript code. If you are not using the development version of the app (meaning you didn't follow the steps in recipe number 1) but instead you are testing a build that bundles the JavaScript, the packager is not needed.

How to do it…

1. Let's start by creating a new `src/features/input.test.js` file; we are going to write our tests in here. It's important to name the files with the ending `*.test.js`. This way, when running mocha, the tests will be loaded automatically.
2. Now we need to import all the dependencies for our tests. Let's add the following code at the top of the new file:

```
const assert = require('assert');
const wd = require('wd');
const chai = require("chai");
const chaiAsPromised = require("chai-as-promised");
const config = require('../config');
```

3. We need to tell chai to use the promises and enable assertion chaining to work with the Appium client:

```
// enables chai assertion chaining
chaiAsPromised.transferPromiseness =
wd.transferPromiseness;

chai.use(chaiAsPromised);
chai.should();
```

4. Before the tests start, we need to connect to the `Appium` server and send the configurations we have defined. By using the `before` hook, we can connect to the server and initialize the driver. We are defining a driver variable at the top of the test; we need to do this because when the tests are done, we want to close and release the connection to Appium:

```
describe('Main form', function() {
  let driver;

  // Setting the maximum waiting time
  this.timeout(300000);

  before(function () {
    driver = wd.promiseChainRemote(config.appium);

    return driver.init(config.capabilities);
```

```
  });
  // Defined at later steps
});
```

5. It's very simple to release the connection, we just need to use the `after` hook to close the session:

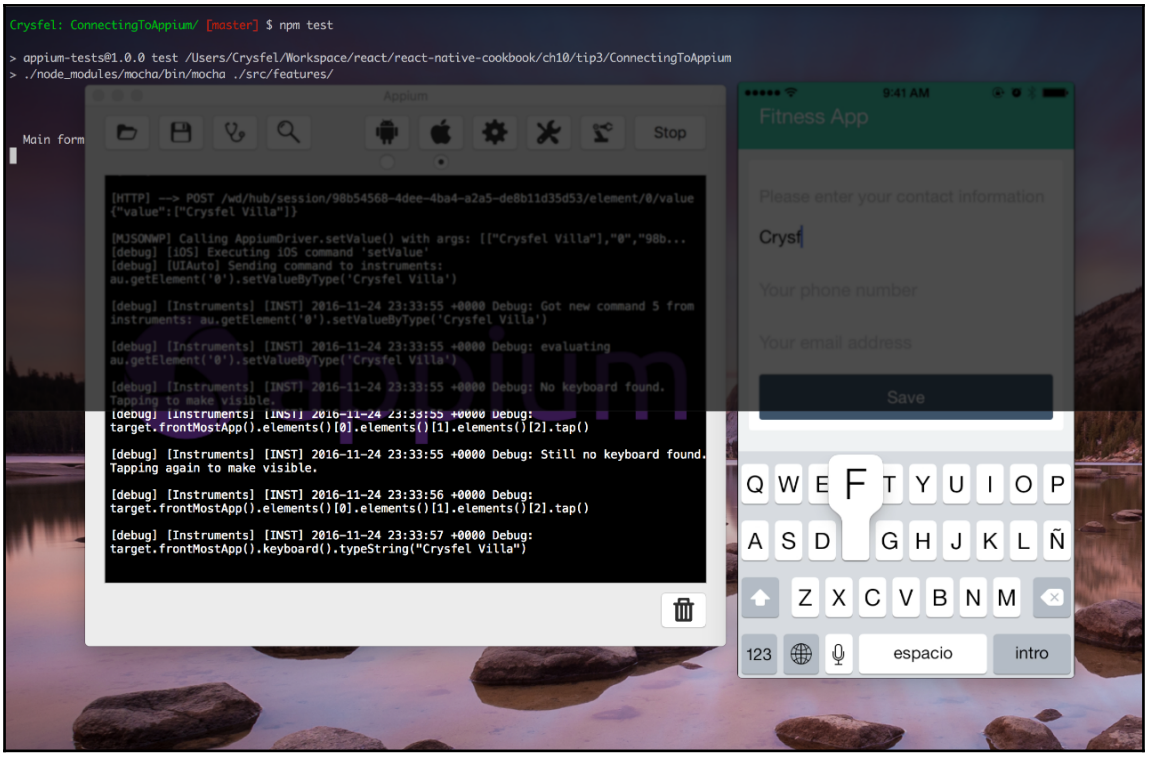
```
after(function () {
  return driver.quit();
});
```

6. In order to get an element from our app, we need to know the XPath. In recipe number 2, using the `Appium` Inspector, we were able to get the XPath for the input fields; now it's time to use it. We are going to define a new test to enter a text to the first input and then check when the value gets updated:

```
it('should enter a string to the input texts',
function() {
  return driver
    .elementByXPath
    ('//UIATextField[@value=\'Your name
    \']')
     .type('Crysfel Villa')
    .getValue()
    .should.become('Crysfel Villa');
});
```

7. Now we are ready to run our first test. Make sure your phone is plugged to your Mac and the `Appium` server is running. Then type the following command on the terminal:

```
$ npm test
```



After the tests are completed, we should see all tests green, meaning there are errors and we are good to go.

Before running your tests, make sure to close the `Appium` Inspector, otherwise the tests will fail because the device will be busy running the inspector.

8. We can continue adding new instructions to the chain. All of these are promises because everything is happening in our device and it takes some time to run. Here's an example of how to set new values to the three inputs:

```
it('should enter a string to the input texts',
function() {
  return driver
    .elementByXPath('//UIATextField[@value=
    \'Your name\']')
    .type('Crysfel Villa')
    .getValue()
    .should.become('Crysfel Villa')

    .elementByXPath('//UIATextField[@value=
    \'Your phone number\']')
    .type('3829482783')
    .getValue()
    .should.become('3829482783')

    .elementByXPath
    ('//UIAScrollView[1]/UIATextField[3]')
    .type('crysfel@gmail.com')
    .getValue()
    .should.become('crysfel@gmail.com');
});
```

How it works…

In step *4*, we are connecting the client to the `Appium` server. In here, we are using the `before` hook; this function is automatically executed for us when the tests are about to start. Mocha is the one responsible of doing this.

In step *5*, we released the connection. Again, the `after` function gets executed automatically by Mocha, after all tests are completed.

In step *6*, we are creating a simple test. Using the driver, we find the input text with the value `Your name`, then we use the `type` method to enter some text into the input; this function will set the new value using the phone's keyboard.

After the new value is typed, we get the current value from the input and execute two chai methods to verify that the new value is the one we entered; if this is true then our test will pass, otherwise it will fail.

There's more…

Sometimes the auto corrector will change the input value; make sure to either disable it on your device or use a different value to avoid running into these scenarios.

Pressing a button and testing the result

Now that we know how to get elements using XPath, we can find the button that we have and then press it to test whether there's an alert message.

Getting ready

In order to complete this recipe, we need to have the `Appium` server running, integrated Mocha with WD (take a look at recipe number *3* in this chapter), and our device should be plugged in to our Mac.

This recipe depends on the previous one; we will be updating the same files.

How to do it…

1. Let's open the `src/features/input.test.js` file; we are going to create a new test to type an e-mail address and click the button. There are several steps that we need to do in here:

```
  it('should show an alert with the email address',        function()
{
    // Find the email input

    // Type an email address

    // Hide the keyboard

    // Press the button
```

```
  // Validate that there's an alert message
});
```

2. First, we need to find the input element. We have three inputs; in previous recipes we have used the `value` property on the XPath to find the element, but in this case, we are going to use the order position; in this case, the email field is the third element on the form:

```
return driver
  .elementByXPath
  ('//UIAScrollView[1]/UIATextField[3]')
  .clear()
```

3. Once we have found the input and cleared any previous value, we need to type an email address and then hide the keyboard:

```
return driver
  .elementByXPath
  ('//UIAScrollView[1]/UIATextField[3]')
  .clear()
  .type('crysfel@gmail.com')
  .hideKeyboard()
```

4. The next step is to find the button and press it. We are going to use the `elementById` method to find the button using an ID; the ID is the label that we are using for this button:

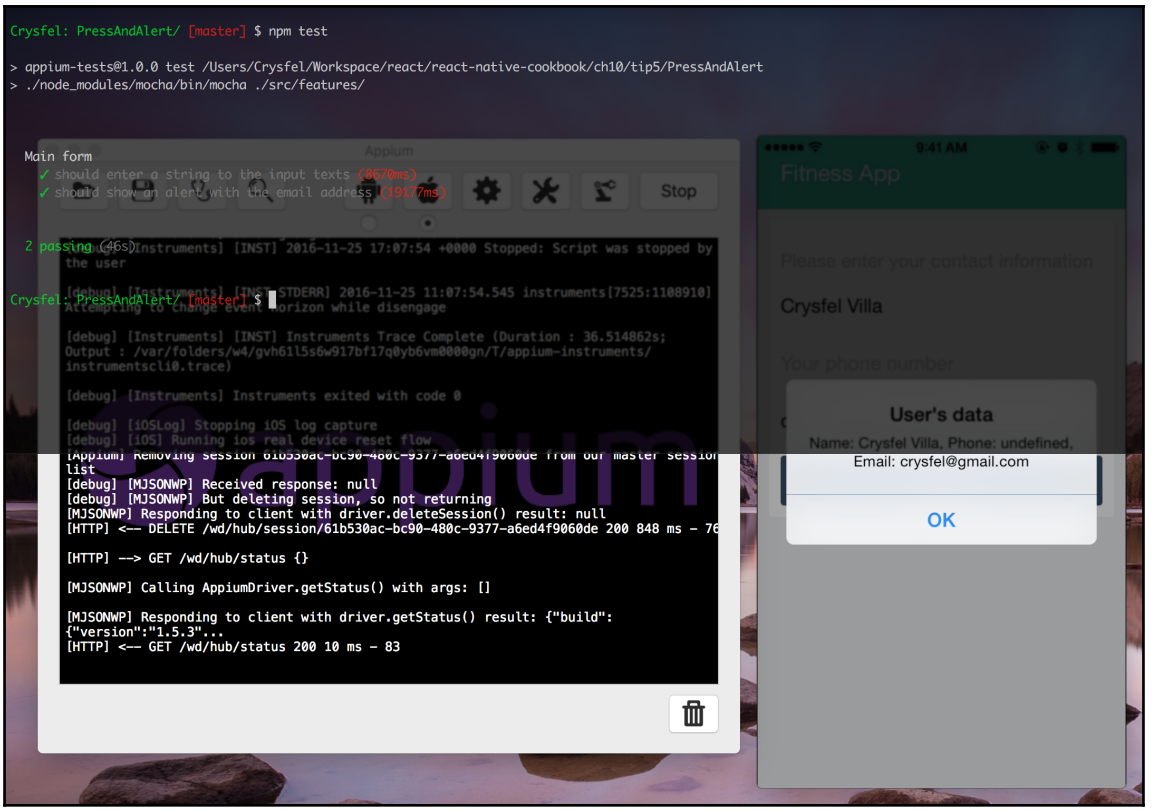
```
     return driver
.elementByXPath
('//UIAScrollView[1]/UIATextField[3]')
       .clear()
       .type('crysfel@gmail.com')
       .hideKeyboard()

       .elementById('Save') // Finding the button
       .tap()               // Pressing the button
       .sleep(1000)         // Waiting 1 second
```

5. Finally, we need to write the assertion. First we need to find the alert message, get the value of the text displayed, and compare it with the entered values. For the assertion, we are going to use chai promised:

```
return driver
  .elementByXPath
  ('//UIAScrollView[1]/UIATextField[3]')
  .clear()
  .type('crysfel@gmail.com')
  .hideKeyboard()

  .elementById('Save')
  .tap()
  .sleep(1000)

  .elementByXPath
  ('//UIAAlert/UIAScrollView/UIAStaticText[2]')
  .getValue()
  .should.eventually.equal
  ('Name: Crysfel Villa,
  Phone: undefined, Email: crysfel@gmail.com');
```

6. We are done with our test; now we can run them all and see if everything is green or red. Open the terminal, go to the project's root folder, and run the following command, (make sure to have your phone plugged-in to your Mac and the `Appium` server running):

```
$ npm test
```



How it works…

In step 2, we are finding the email input. In here we are also clearing any content that this field might contain by calling the `clear` method. This step is not required, but it's a good idea to make sure the input doesn't have anything from previous tests.

In step 3, we are typing the new value and hiding the keyboard. We need to hide the keyboard in order to press the button successfully using Appium.

In step 4, we are finding and pressing the button. We need to wait 1 second because the alert takes a few milliseconds to show up, there's a fancy animation that shows the alert and if we don't wait that second, Appium might not find the element. By waiting we make sure the alert will be there. If it's not there after 1 second, then there's an issue in our app.

In step *5*, we are defining the assertion. After finding the alert message by the XPath, we get the text value for that element, then we validate that this value should eventually be equal to the values we have entered on the previous steps.

It's important to use the `eventually` method on the assertion, because these are promises that need to be fulfilled when the `Appium` server responds to the client.

11
Optimizing the Performance of Our App

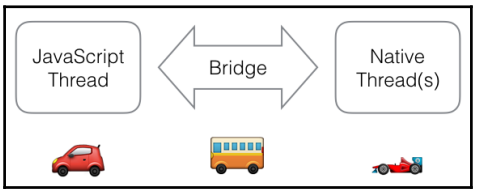
In this chapter, we will cover the following recipes:

- Optimizing our JavaScript code
- Optimizing the performance of custom UI components
- Keeping our animations running at 60 FPS
- Getting the most out of ListView
- Boosting the performance of our app
- Optimizing the performance of native iOS modules
- Optimizing the performance of native Android modules
- Optimizing the performance of native iOS UI components
- Optimizing the performance of native Android UI components

Introduction

Performance is a key criterion of almost every single piece of technology in software development. It is often debated, validated, and invalidated when discussing competing products. React Native was introduced to solve the issue of poor performance that exists in hybrid applications that wrap web applications in a native container. React Native has an architecture that lends itself to not only feature development and flexibility but excellent performance. Still, the out-of-the-box performance serves as a baseline and depends on how you use the framework; it can go in either direction.

When thinking of performance in our React Native application, it is important to think about the big picture of how React Native works. There are three major concepts of React Native and their relative performance is depicted in the following diagram:



Optimizing our JavaScript code

It's safe to say that your React Native application will be written mostly using JavaScript. You may have some native modules and custom UI components, but for the most part, all your view and business logic will be written in JSX and JavaScript. Most likely, you will be using language constructs introduced with ES6. These may be available natively by the JavaScript interpreter bundled with React Native (JavaScriptCore) or polyfilled by the Babel transpiler. Since JavaScript constitutes most of our application, this should be the first part we optimize if we want to squeeze some performance out of our application.

This recipe will provide some helpful tips in optimizing JavaScript code to be as performant as possible. There is an ongoing debate on performance versus readability. We will try to be objective in our steps, and provide you with the steps to increase performance. It is up to you to make an educated decision whether these performance optimizations are worth the potential cost in readability and maintainability.

Getting ready

This recipe is not necessarily dependent on React Native as it focuses on the JavaScript used to write a React application.

How to do it…

1. The first and foremost optimization to look for is speeding up iterations. Often, you may be using functions that take iterator functions as arguments (`forEach`, `filter`, `map`). These are generally going to be slower than doing a standard `for` loop. Depending on the size of collection you are iterating, this may make a difference. Here's an example of a faster filter function:

```
var myArr = [1,2,3,4,5,6,7],
    newArr;
// Slower:
function filterFn(element) {
    return element > 2;
}
newArr = myArr.filter(filterFn);

// Faster:
function filterArray(array) {
    var length = array.length,
        myNewArr = [],
        element,
        i;

    for(i = 0; i < length; i++) {
        element = array[i];
        if(element > 2) {
            myNewArr.push(array[i]);
        }
    }
    return myNewArr;
}

newArr = filterArray(myArr);
```

2. More importantly, when optimizing iterations, is to make sure you are storing variables you are accessing on the iteration somewhere close by:

```
function findInArray(properties, appConfig) {
    for(var i = 0; i < properties.length; i++) {
        if(properties[i].someProperty ===
        appConfig.userConfig.permissions[0]) {
            // do something
        }
    }
}
```

```
function fasterFindInArray(properties, appConfig) {
    var matchPermission = appConfig.userConfig.permissions[0],
        length  = properties.length,
        i = 0;
    for(; i < length; i++) {
        if(properties[i].someProperty === matchPermission) {
            // do something
        }
    }
}
```

3. Optimize your logical expressions. Keep your fastest and closest executing statements on the left:

```
function canViewApp(user, isSuperUser) {
    if(getUserPermissions(user).canView || isSuperUser) {
        return true;
    }
}

function canViewApp(user, isSuperUser) {
    if(isSuperUser || getUserPermissions(user).canView) {
        return true;
    }
}
```

4. While we may really enjoy ES6 constructs, please note that some of the features execute slower than their ES5 implementations. These features can include `for of`, `generators`, `Object.assign`, and some more. A good reference for performance in relation to ES5 can be found here: `https://kpdecker.github.io/six-speed/`.
5. Avoid `try-catch` as it may impact optimizations of the interpreter (it does in V8).
6. Arrays should have all members be of the same type. If you need to have a collection where the type can vary, use an object.

How it works…

JavaScript performance is a topic of constant debate, sometimes heated. It is sometimes difficult to keep up with the latest in performance metrics as Google, Apple, Mozilla, and the global open source community is hard at work on their JavaScript engines. For React Native, we focus on WebKit JavaScriptCore. The recipes in this chapter focus on using lower-level functions that take up less memory and overall have to perform fewer

operations, thus lowering the time it takes for a task to complete.

Optimizing the performance of our custom UI components

In building your React Native application, it's a safe bet that you will be creating custom UI components. These components can either be compositions of several other components or a component that builds off an existing component and adds some functionality. With the added functionality, the complexity increases as well. The increased complexity leads to more things going on and of course more things that can slow down. Fortunately, there are some ways to make sure that our custom UI components are performing the best they can. This recipe will show several techniques for getting the most out of our components.

Getting ready

This recipe requires you to have a React Native application with some custom components. As these performance suggestions may or may not provide value to your application, we will be listing some of them and you can choose whether or not to apply them to your code.

How to do it…

1. The first optimization we should look at is what we have in the `state` of our component. We should make sure that all the objects we have in the `state` are being used and can potentially change, causing a re-render. If they can be `props`, please make sure they are `props`. If you could store an object's member as a `state` instead of the entire object, please do so.
2. Let's take a look at our `render` function. Our overall goal is to keep this function the fastest performing; we should limit any long-running processes. If you can, cache computations and constant values outside the `render` function so they are not instantiated every time.
3. If you have conditional JSX that may return in the `render` function, return as early as possible. Here's a trivial example:

```
// unoptimized
render() {
    let output;
    const isAdminView = this.props.isAdminView;
```

```
    if(isAdminView) {
        output = (<AdminButton/>);
    } else {
        output = (
                <View style={styles.button}>
                      <Text>{this.props.buttonLabel}</Text>
                </View>
        );
    }
    return output;
}

// optimized
render() {
    const isAdminView = this.props.isAdminView;
    if(isAdminView) {
        return (<AdminButton/>);
    }
    return (
        <View style={styles.button}>
            <Text>{this.props.buttonLabel}</Text>
        </View>
    );
}
```

4. The most important optimization we can make, however, is to skip the render altogether if we can. This is done by implementing the `shouldComponentUpdate` function and returning `false`, thus making the component pure. Here's how we can make a component a `PureComponent`:

```
import React, {
    PureComponent
} from 'react';

export default class Button extends PureComponent {//...}
```

How it works…

To say React component performance is important is an understatement. The majority of your React Native application will consist of custom components. There will be a mix of stateful and stateless components. As highlighted by step *2*, the overall goal is to render our component in the shortest time. Better yet would be to only have to render our component once and leave it untouched; this concept is discussed in step *4*.

See also

You can find some more information about React component performance optimizations at the following URLs:

- `http://moduscreate.com/react_component_rendering_performance/`
- `http://benchling.engineering/performance-engineering-with-react/`
- `https://facebook.github.io/react/docs/advanced-performance.html`

Keeping our animations running at 60 FPS

An important aspect of our application is the fluidity of our user interface. Animations are used to give a sense of this rich user experience. We can employ animations for all sorts of interactions, from changing entire views, to reacting to a user's touch interaction on a component. The most important factor after having a visually appealing animation is to make sure it does not stop the JavaScript thread. In order to keep our animations fluid and not interrupt the UI interactions, our render loop has to render each frame in 16.67 ms.

In this recipe, we will take a look at several techniques for improving the performance of our animations. These techniques focus on preventing JavaScript execution interrupting the main thread.

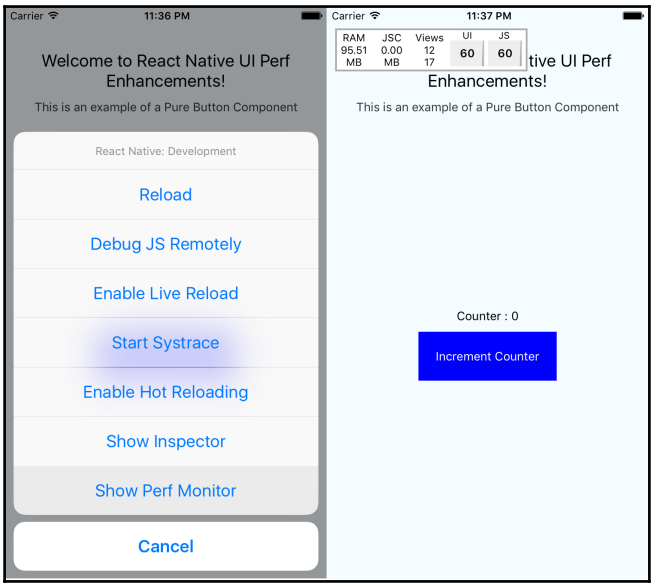
Getting ready

For this recipe, we assume you have a React Native application that has some animations defined.

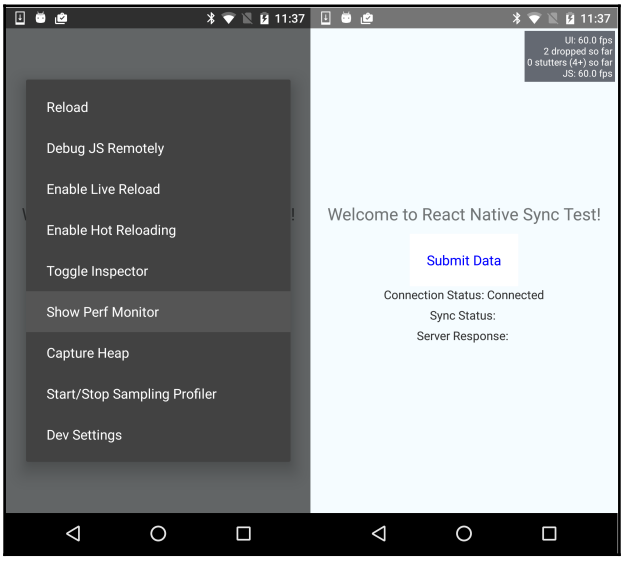
How to do it…

1. First and foremost, when debugging animation performance in React Native, we should enable the performance monitor. To do so, show the Dev Menu (shake the device or *cmd + D* from the simulator) and tap **Show Perf Monitor**.

The output in iOS will look as the following screenshot:



The output in Android will be as follows:



2. Now we have a testbed for our animations, we can begin with optimizations. If you are looking to animate a component's transition (`opacity`), or dimensions (`width`, `height`) then please use `LayoutAnimation`. You can find an example of using `LayoutAnimation` in `Chapter 3`, *Animating the User Interface*, in the *Expanding and collapsing containers* recipe.

If you want to use `LayoutAnimation` on Android, you need to add the following code when your application starts: `UIManager.setLayoutAnimationEnabledExperimental && UIManager.setLayoutAnimationEnabledExperimental(true)`

3. If you need to have a finite control over the animations, it is recommended to use the `Animated` library that comes with React Native. Fortunately, with recent React Native releases, we can offload the entire animation work onto the native UI thread. To do so, we have to add the `useNativeDriver` property to our `Animated` call. Let's take a sample `Animated` example and offload it to the native thread:

```
componentWillMount() {
  this.setState({
    fadeAnim: new Animated.Value(0)
  });
}

componentDidMount() {
  Animated.timing(
    this.state.fadeAnim,
    {
      toValue: 1,
      useNativeDriver: true
    }
  ).start();
}
```

Currently, only a subset of the functionality of the Animated library supports native offloading. Please refer to the *There's more...* section for a compatibility guide.

4. If you are unable to offload your animation work onto the native thread, there is still a solution for having a smooth experience. We can use the `InteractionManager` to execute a task after the animations have completed:

```
componentWillMount() {
  this.setState({
    isAnimDone: false
  });
}
componentWillUpdate() {
  LayoutAnimation.easeInAndOut();
}

componentDidMount() {
  InteractionManager.runAfterInteractions(() => {
    this.setState({
      isAnimDone: true
    });
  })
}

render() {
  if(!this.state.isAnimDone) {
    return this.renderPlaceholder();
  }
  return this.renderMainScene();
}
```

5. Finally, if you are still suffering from poor performance, you have no choice but to either rethink your animation or implement your view as a custom UI view component on your target platform(s). You will have to implement both your view and animation natively using the iOS and/or Android SDK. In `Chapter 6`, *Adding Native Functionality*, we covered creating custom UI components in the *Rendering custom iOS view components* and *Rendering custom Android view components* recipes.

How it works…

The tips in this recipe focus on the simple goal of preventing the JavaScript thread from locking. The moment our JavaScript thread begins to drop frames (lock), we lose the ability to interact with our application for that fraction of a second. It may seem inconsequential, but the effect is felt immediately by a user. The focus of the tips in this recipe is to offload animations onto the GPU. When the animation is running on the main thread (native layer, rendered by the GPU) you can interact with the application freely.

There's more…

For step *3*, here's a quick reference of where `useNativeDriver` is applicable:

Function	**iOS**	**Android**
`style`, `value`, `props`	X	X
`decay`		X
`timing`	X	X
`spring`		X
`add`	X	X
`multiply`	X	X
`modulo`	X	
`diffClamp`	X	X
`interpoloate`	X	X
`event`		X
`division`	X	X
`transform`	X	X

Getting the most out of ListView

React Native provides us a pretty performant list component out of the box. It is extremely flexible, supports rendering almost any component you can imagine inside of it, and renders them rather quickly. If you'd like to read some more examples of how to work with `ListView`, there are a couple of recipes in `Chapter 1`, *Getting Started* and `Chapter 3`, *Animating the User Interface* that use it. The React Native `ListView` is built on top of `ScrollView` in order to achieve the flexibility of rendering variable-height rows with any view component.

The major performance and resource drawback of the `ListView` component occurs when you have an extremely large list that you scroll through. Currently, as you scroll the list, the next *page* of rows is rendered at the bottom. The top invisible rows can be set to be removed from the render tree; we will get to this in a bit. However, the references to the rows are still in memory as long as the component is mounted. Naturally, as our component uses up the available memory, there won't be any room for quickly accessible storage for the upcoming components. This recipe will help us deal with some of these potential performance and memory resource issues.

Getting ready

For this recipe, we assume you have a React Native application that has `ListView` rendered, preferably with a large dataset.

How to do it…

1. Let's start with some optimizations we can make to our vanilla `ListView` component. If we set the `initialListSize` prop to `1`, we will speed up our initial rendering.
2. Next up, we can bump up our `pageSize` if the component we render in each row is simple.
3. Another optimization we can perform is to set the `scrollRenderAheadDistance` to a comfortable value. If your users will rarely scroll past the initial viewport or scroll slowly then you can lower the value. This would prevent the `ListView` from rendering too many rows in advance.

4. Finally, the last optimization we have to be mindful of is the `removeClippedSubviewsprop`. Currently, it is enabled by default on both iOS and Android. However, for iOS, you have to make sure the `style` for your row has `overflow` set to `true`.
5. Combining steps *1–4* would give us the following code:

```
renderRow(row) {
  return (
    <View style={{height:44, overflow:'hidden'}}>
      <Text>Item {row.index}</Text>
    </View>
  )
}

render() {
  return (
    <View style={{flex:1}}>
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderRow}
        pageSize={10}
        initialListSize={1}
        pageSize={10}
        scrollAheadDistance={200}
        />
    </View>
  )
}
```

6. Outside of `ListView` itself, it is important to consider the performance of the `View` we are rendering for the row. Please refer to the *Optimizing the performance of custom UI components* recipe and apply it to the work that is done in your `renderRow` function.

How it works…

As with many features in computer science, the more flexible and complex something is, the slower it performs. `ListView` is an excellent example of this concept. It is extremely flexible in that it can render any View in the row, but it can quickly bring your application to a halt. The result of the optimizations defined in steps *1–4* will vary across different situations based on what you are rendering and the data structure. You should experiment with the values until you find a perfect balance. If you are still unable to achieve the performance you desire, take a look at some of the community modules that provide new `ListView` implementations or alternatives.

See also

There are some third-party `ListView` implementations that promise increased performance as follows:

- **react-native-sglistview**: This takes `removeClippedSubviews` to the next level by flushing the memory when the offscreen rows are removed from the render tree (`https://github.com/sghiassy/react-native-sglistview`).
- **list-view-experiments (iOS only)**: This repository contains two separate list implementations (`https://github.com/wix/list-view-experiments`):
    - **InfiniteScrollViewChildren**: This recycles rows as you scroll. This is different from the `ListView` component as the off-screen rows are removed from the render tree. Meanwhile, in the `InfiniteScrollViewChildren` component, the rows are never destroyed but reused.
    - **TableViewChildren**: This is an implementation of the `UITableView` iOS component. This is the base list component implemented by the iOS SDK. It can only render Text nodes, but it is extremely performant.

Boosting the performance of our app

The basic premise behind React Native is using JavaScript to build a truly native app. This is different than, say, an Ionic or any other Cordova hybrid app that wraps a web application written in JavaScript and tries to act like a truly native application. Those web applications only have access to native APIs to perform processing, but cannot render native views inside their application. Thus, React Native is inherently faster than hybrid web applications. Since it's so performant out of the gate and we have discussed some excellent ways to speed up specific parts of our application in this chapter, we generally do not have to worry about overall performance as much as we would with a hybrid web app. Still, though, it may not hurt to squeeze out just a little bit more. This recipe will provide some quick wins that we can use to have a faster React Native application.

Getting ready

This recipe is applicable to any React Native application.

How to do it…

1. The simplest optimization we can employ that may provide a significant gain depends on how much you like outputting statements to the console. Performing a `console.log` statement is not a trivial task as you'd imagine for the framework. Therefore, it is recommended to remove all console statements when you are ready to bundle your application.
2. If you would like to have babel remove the console statements automatically when creating the bundle then you need to install a plugin. Open your terminal to your project directory and execute the following command:

```
$ npm install babel-plugin-transform-remove-console --save
```

3. Open your `.babelrc` file and add the following:

```
{
  "presets": ["react-native"],
  "env": {
    "production": {
        "plugins": ["transform-remove-console"]
    }
  }
}
```

4. Next, make sure when you're analyzing your performance you're running the application in production mode, preferably on a device. If you are curious about how to do so, please refer to the *Deploying testing builds to HockeyApp* recipe in `Chapter 9`, *Deploying Our App*.
5. Sometimes when you are animating a View's position or layout, you may notice performance dips in the UI thread. You can mitigate this by setting the `shouldRasterizeIOS` and `renderToHardwareTextureAndroid` props to true for iOS and Android platforms. Be mindful that this may increase memory usage significantly.
6. You may be in a position where you are changing your current view using a navigation state change and you need to perform synchronous potentially long-running processes. This can be building a `DataSource` for `ListView` or transforming data to power your view. You should experiment with processing only an initial subset of the data, enough to render the UI quickly enough. Once the animation completes between page transitions, you can use `InteractionManager` to complete loading the rest of the data. You can refer to the *Keeping our animations running at 60 FPS* recipe for instructions on how to use `InteractionManager`.
7. Finally, if you have identified a particular component or task that is slowing down your application without solution then you should move it to the native thread. Create a native module or native UI component to implement this bit of functionality.

How it works…

This recipe covers some higher-level and broader-scoped tips that exist for all React Native applications. The most significant performance gains you likely will see out of these tips are from step *7*. Ironically, it is also the most difficult and involved task of the bunch.

Optimizing the performance of native iOS module

In building our React Native application, we may have written some native modules. Hopefully, you followed some of the recipes in `Chapter 6`, *Adding Native Functionality*. You may have built these native modules to expose some bit of functionality provided by a native API or maybe you wanted to perform an intensive background task.

As we touched on earlier, working in the native layer really lets us use the device's full capacity. However, it doesn't mean that the code we write will automatically be the fastest it could be. There's always room to optimize and generate performance gains that may be significant.

In this recipe, we will provide some tips on how to make your Objective-C code run a bit faster using the iOS SDKs. We will also provide some considerations for how React Native and the bridge fits in the application flow.

Getting ready

For this recipe, you should have a React Native application that uses native modules that you have created for iOS. If you need help in writing native modules, please take a look at *Exposing custom iOS modules* in `Chapter 6`, *Adding Native Functionality*.

How to do it…

1. First and foremost, when working with native modules, we have to be mindful of the data going through the React Native Bridge. Keep the data that you place in your events and callbacks to a minimum as the data serialization between Objective-C and JavaScript types is extremely slow.
2. If you need to keep data cached in memory to be used by the native module, keep it stored in a local property or field variable. Native modules are singletons. Do this instead of returning a large object to store on the React component.
3. Let's switch over to Objective-C and iOS specific recommendations. First and foremost, please ensure you are using ARC. It's practically a given now with modern SDKs, but if you have the choice and can leverage it, please do.
4. Sometimes we have to leverage classes that are quite large because they are robust in their feature set. Instead of instantiating something like an `NSDateFormatter` each time in your method that you expose via `RCT_EXPORT_METHOD`, store the reference of this class as a property or some instance variable.

5. Staying on `NSDateFormatter`, since it is extremely heavy, we should see if we can avoid it. If your application can deal with just UNIX timestamps, then you can easily get an `NSDate` object from a timestamp with the following function:

```
- (NSDate*)dateFromUnixTimestamp:(NSTimeInterval)timestamp {
    return [NSDate dateWithTimeIntervalSince1970:timestamp];
}
```

6. Last and probably the most significant performance optimization you can make if the situation presents itself is spawning asynchronous background threads to do intensive processes. React Native fits this model well as it uses an asynchronous messaging/event system to communicate between the JavaScript and native threads. When your background process is complete, you can either invoke a callback/promise or fire an event for the JavaScript thread to pick up. To learn how to create background processes in React Native iOS native modules, please read the *Background processing on iOS* recipe in `Chapter 6`, *Adding Native Functionality*.

How it works…

Objective-C executes plenty fast, almost as fast as vanilla C. Therefore, the optimizations we perform do not have much to do with executing tasks but rather instantiations and not blocking the native threads. Most likely, you will be using ARC as described in step *3*, if you aren't, you probably have a very good reason not to be. The biggest performance boost you'll see is by properly using the Grand Central Dispatch (GCD) to spawn background processes as described in step *6*.

Optimizing the performance of native Android modules

While developing your React Native application, you may find yourself writing Android, native modules. You may be cross-developing your React Native app on both iOS and Android requiring you to replicate some bit of native functionality on both platforms. There are still some native APIs that have not been wrapped as first-party modules for Android that exist on iOS so you may have to do it yourself. Finally, you could be using some Android-specific native functionality; the reasons are endless. Hopefully, you found `Chapter 6`, *Adding Native Functionality* to be a great resource in this process.

In this recipe, we will cover several techniques for speeding up our React Native Android native modules. Many of these techniques are limited to general development Android, scope and there will be some tips for communicating with the React Native JavaScript layer.

Getting ready

For this recipe, you should have a React Native application that uses native modules that you have created for Android. If you need help in writing native modules, please take a look at the *Exposing custom Android modules* recipes, in `Chapter 6`, *Adding Native Functionality*.

How to do it…

1. First and foremost, when working with native modules, we have to be mindful of the data going through the React Native Bridge. Keep the data that you place in your events and callbacks to a minimum as the data serialization between Java and JavaScript types is extremely slow.
2. If you need to keep data cached in memory to be used by the native module, keep it stored in a private field. Native modules are singletons. Do this instead of returning a large object to store on the React component.
3. Let's focus on some Android-specific Java tips. When writing Java code for Android, you should do your best to avoid to create short-term objects. If you can, use primitives, especially for structures such as arrays.
4. It is better to reuse objects instead of relying on the garbage collector to pick up an unused reference and instantiate a new object.
5. The Android SDK provides a memory-efficient data structure for replacing the use of a `Map` which maps `Integers` to `Objects`, `SparseArray`. Here's an example:

```
SparseArray<SomeType> map = new SparseArray<SomeType>();
map.put(1, myObjectInstance);
```

There is also `SparseIntArray` that maps integers to integers and `SparseBooleanArray` that maps integers to Boolean values.

6. This may sound counterintuitive to those brought up on the OOP wave of Java development: avoid using getters and setters, access the instance field directly.

7. If working with `String` concatenation, use `StringBuilder`.
8. Last and probably the most significant performance optimization you can make if the situation presents itself is spawning asynchronous background threads to do intensive processes. React Native fits this model well as it uses an asynchronous messaging/event system to communicate between the JavaScript and Native threads. When your background process is complete, you can either invoke a callback/promise or fire an event for the JavaScript thread to pick up. To learn how to create background processes in React Native Android native modules, please read the *Background processing on Android* recipe in `Chapter 6`, *Adding Native Functionality*.

How it works…

The majority of the tips in this recipe revolve around efficient memory management. The Android OS uses a traditional-style garbage collector similar to desktop Java VM. When the garbage collector kicks in, it can take anywhere from 100-200 ms to free memory. Steps *3–7* all provide suggestions that reduce the application's memory usage.

Optimizing the performance of native iOS UI components

React Native provides us an excellent foundation to build almost any user interface using the built-in components and styling. Sometimes we may need to render iOS third-party UI components in React Native or we may want to build something from scratch and leverage the GPU even more and its multithreading capabilities. Components built in Objective-C using the iOS SDK, OpenGL, or some other drawing library will generally perform better than composing the prebuilt components using JSX. In creating these native view components, there are some situations that may have a negative impact on the application performance.

This recipe will focus on getting the most out of the iOS UIKit SDK to render our custom views. Our goal is to be rendering as quickly as we could in order for our application to run at the holy grail of 60 FPS.

Getting ready

For this recipe, you should have a React Native application that renders some custom native UI components you have written for iOS. If you need help in wrapping UI components in React Native, please take a look at the *Exposing custom iOS view components* recipe in `Chapter 6`, *Adding Native Functionality*.

How to do it…

1. First and foremost, when working with native modules, we have to be mindful of the data going through the React Native Bridge. Keep the data that you place in your events and callbacks to a minimum as the data serialization between Objective-C and JavaScript types is extremely slow.
2. If there is data that you need to store for referencing sometime in the near future, it's better to store it in the native class that you initialized. Depending on your application, you can either store it as a property on the `ViewManager`, a singleton that serves instances of the `View`, or a property on the `View` itself.
3. If your view component involves rendering multiple `UIView` instances as children of a parent `UIView` container, make sure all the instances have the `opaque` property set to `true`.
4. If you are rendering an image inside your view component (not using the React Native `Image` component) then have your image be the same dimension as the `UIImageView` component. Scaling and other image transformations are heavy operations that can impact frame-rate.
5. One of the most important benefits you can achieve with iOS view components is avoiding offscreen rendering. Avoid using the following SDK functionality if you can:
    - Classes that start with `CG`, the Core Graphics library
    - Overriding the `drawRect` implementation of `UIView`
    - Setting `shouldRasterize=YES`, using `setMasksToBounds` or `setShadow` on your `UIView` instance's `layer` property
    - Custom drawings using `CGContext`

6. If you need to add a shadow to your view, make sure you are setting the `shadowPath` to prevent offscreen rendering. Here's an example of how the initialization and shadow definition should look:

```
RCT_EXPORT_MODULE()

- (UIView *)view {
  UIView *view = [[UIView alloc] init];
  view.layer.masksToBounds = NO;
  view.layer.shadowColor = [UIColor blackColor].CGColor;
  view.layer.shadowOffset = CGSizeMake(0.0f, 5.0f);
  view.layer.shadowOpacity = 0.5f;
  view.layer.shadowPath = [[UIBezierPath
bezierPathWithRect:view.bounds] CGPath];
  return view;
}
```

How it works…

This recipe focuses on some helpful tips to let the GPU do much of the work as it can. The second part is for the GPU to do the least in the work it has to do. Enforcing the `opaque` property in step 3 tells the GPU to not have to worry about scanning for visibility of other components and calculating transparency. Steps *5* and *6* prevent offscreen rendering. Offscreen rendering generates bitmap images using the CPU (slow) and more importantly it pauses the GPU from rendering the view until it is done generating the images.

Optimizing the performance of native Android UI components

Over the last few years, Android native UI performance has grown significantly. This is primarily due to the ability to render the components and layout using GPU hardware acceleration. In your React Native application, you may be using custom view components, especially if you want to use a built-in material design component that has not been wrapped as a React component. Even though the Android platform has made a conscious effort to increase the performance of its UI, the way we render our components can quickly negate all these benefits.

In this recipe, we're going to talk through a few ways to get the most speed out of our custom Android view components.

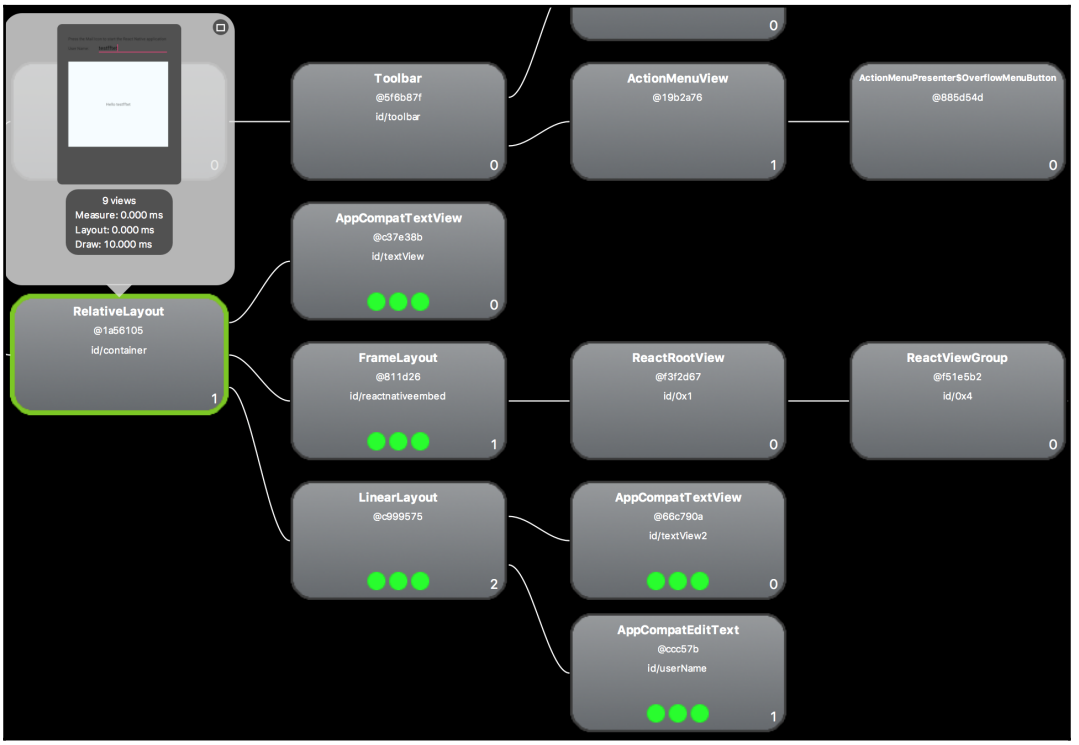
Getting ready

For this recipe, you should have a React Native application that renders some custom native UI components you have written for Android. If you need help in wrapping UI components in React Native, please take a look at the *Exposing custom Android view components* recipe in `Chapter 6`, *Adding Native Functionality*.

How to do it…

1. First and foremost, when working with native modules, we have to be mindful of the data going through the React Native Bridge. Keep the data that you place in your events and callbacks to a minimum as the data serialization between Java and JavaScript types is extremely slow.
2. If there is data that you need to store for referencing sometime in the near future, it's better to store it in the native class that you initialized. Depending on your application, you can either store it as a property on the `SimpleViewManager`, a singleton that serves instances of the `View`, or a property on the `View` itself.
3. Let's start looking at optimizing our `View` instances. You may be rendering a somewhat complicated view component that consists of other child components. These components are held in a hierarchy of layouts. Over-nesting layouts can become a very expensive operation. If you are using multi-level nested `LinearLayout` instances, try to replace them with a single `RelativeLayout`.

4. You can analyze the efficiency of your layout using the **HierarchyViewer** tool bundled inside the Android Device Monitor. In the Monitor, click **Window | Open Perspective... | Hierarchy View** and select **OK**. Here's a sample view of the layout for the app we built in the *Communicating from an Android application to React Native* recipe of `Chapter 8`, *Integration with Applications*:



5. If you are performing repeated animations on your custom view natively in Java (not using the React Native Animated API) then you can leverage hardware layers to improve performance. Just simply add a `withLayer` method call to your `animate` call:

```
myView.animate()
  .alpha(0.0f)
  .withLayer()
  .start();
```

How it works…

For better or worse, there aren't that many optimizations you can perform when it comes to rendering Android UI components. They generally revolve around not over-nesting layouts as that increases complexity by orders of magnitude. When you have layout performance issues, you most likely are suffering from the GPU overdrawing. Overdraw occurs when the GPU renders a new view over an existing view that is already rendered. You can enable **GPU Overdraw Debugging** in the Android **Developer Settings** menu. The order of severity of overdraw is No Color -> Blue -> Green -> Light Red -> Dark Red.

In Step *5*, we provided a quick tip for improving the performance of animations. Please be mindful that you will see the biggest yield for repeated animations as it creates a cache of the animation output on the GPU and replays it.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W