Manuel Kiessling

Beginning Mobile App Development with React Native

Beginning Mobile App Development with React Native

A comprehensive tutorial-style eBook that gets you from zero to native iOS app development with JavaScript in no time.

Manuel Kiessling

This book is for sale at http://leanpub.com/beginning-mobile-app-development-with-react-native

This version was published on 2015-08-13

Also By Manuel Kiessling

The Node Beginner Book

Node入门

The Node Craftsman Book

El Libro Principiante de Node

Livro do Iniciante em Node

For Aaron, Nora and Melinda.

Contents

Trademark notes

Preface

Status

This book is currently work in progress and is about 40% done. It was last updated on August 13, 2015. The code in this book has been tested to work with React Native v0.8.0.

> As of now React Native has only been published for the iOS platform. Therefore, developing apps for Android devices is not yet covered, but will be included in a free book update once React Native for Android has been released.

Intended audience

The book will introduce readers to the React Native JavaScript framework and its mobile app development environment. In the course of the book, the reader will build a full-fledged native mobile app, learning about each React Native framework detail on the way to the final product. Furthermore, the reader will be introduced to every tool and all JavaScript language constructs needed to fully master software development with React Native: JSX, ECMAScript 6, the CSS Flexbox system, Xcode®, io.js and NPM, utilities like watchman, and more.

If you did some JavaScript programming before and want to become a mobile app developer, then this book is for you. It introduces everything that is needed to work with the React Native JavaScript framework in an easy-to-follow and comprehensive manner.

Prerequisites

In order to create React Native based iOS applications and work through the examples of this book, you need all of the following:

- A computer running Mac OS X
- The most recent stable version of Xcode® (v6.3 as of this writing)

> Xcode® is available for download at https://developer.apple.com/xcode/downloads/ or on the Mac OS X App Store℠.

Note that unless you want to run your applications on real iOS hardware like your iPhone®, you do **not** need to be enrolled to the Apple® iOS Developer Program. In other words, you can run your applications using the iOS simulator without the need to be enrolled.

In case you do not have access to an Apple® computer, you can try to set up a Virtual Machine running Mac OS X following the guide at https://blog.udemy.com/xcode-on-windows/.

> From this point on, the book presumes that you have a running installation of Mac OS X with the most recent version of Xcode® installed.

Setting up your development environment

React Native is a collection of JavaScript and Objective-C code plus supporting tools that allow to create, run, and debug native iOS applications.

In order to reach the point where we can actually start working on our first React Native application, some preparation is necessary.

On your development machine, the following components need to be made available:

- Homebrew
- io.js
- Watchman
- Google Chrome
- React Native CLI

Installing Homebrew

Homebrew is a package manager for Mac OS X. We will use it to subsequently install most of the software tools we need.

In order to install and set up Homebrew, open a Terminal shell window and run the following command:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

For further information, visit the Homebrew homepage at http://brew.sh/.

Installing io.js

io.js is a modern fork of Node.js, the popular server-side JavaScript runtime environment. React Native ships with some helper tools that are written for io.js. Also, we will regularly use the Node.js/io.js package manager, *NPM*, to install the React Native command line tool and other dependencies.

We are not going to install io.js directly. Instead, we will use Homebrew to install *nvm*, the Node Version Manager.

nvm is a nifty little tool that takes the hassle out of managing parallel installs of different versions of Node.js and/or io.js. With it, you can install as many versions of Node.js and io.js as you wish, and quickly switch to the version you want to actually use for a given project.

In order to install and set up nvm, open a Terminal window and run the following command:

```
brew install nvm
```

Afterwards, create or open file `$HOME/.bashrc`, and add the following lines at the beginning:

```
export NVM_DIR=~/.nvm
source $(brew --prefix nvm)/nvm.sh
```

Close the Terminal session, and start Terminal anew - you now have nvm enabled in your Terminal session.

Using nvm, we can now install io.js, like this:

```
nvm install iojs
nvm alias default iojs
```

This will install and setup the most recent stable version of io.js, plus the matching version of NPM, the Node Package Manager, and make this setup the default.

Installing Watchman

React Native uses *Watchman* to monitor and react to changes in source code files. This is explained in more detail in a later chapter.

In order to install and set up Watchman, open a Terminal window and run the following command:

```
brew install --HEAD watchman
```

The `--HEAD` parameter ensures that the most recent version of Watchman is installed.

Installing Google Chrome

Google Chrome isn't strictly necessary to create applications with React Native, but it allows to debug them during development. The details are explained in a later chapter.

Head over to https://www.google.com/chrome/browser/desktop/index.html to download the latest version.

Installing the React Native CLI

The *React Native Command Line Interface* is a small io.js helper script that allows to set up new React Native projects.

In order to install and set up *React Native CLI*, open a Terminal window and run the following command:

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

With this, we are good to go to create our first application.

> You might also want to install a decent code editor. While Xcode® is part of our development environment, it's certainly not the most well-suited JavaScript editor out there. Check out *TextWrangler* at http://www.barebones.com/products/textwrangler/ if you prefer a simple yet sufficient tool, or *IntelliJ IDEA Community Edition* for a full-fledged IDE at https://www.jetbrains.com/idea/download/. *Atom*, available at https://atom.io/, lies somewhat in between. All three tools are usable free of charge.

Creating your first native iOS application using React Native

In this chapter, we will create a very simple application and make it run in the iOS simulator. By doing so, we are using a whole lot of different components that in total allow us to end up with a working app. Once our app runs, we will look under the hood in order to understand all those components and get a feeling for the inner workings of React Native applications. These insights form the basis that allows us to build more complex applications while truly understanding what we are doing.

Creating the app structure

Open a new Terminal window. On the command prompt, type the following and hit enter:

```
react-native init HelloWorld
```

This results in some work being done which gives you a new folder named *HelloWorld*. We will analyze its contents later - for now, we want to get our feet wet as quickly as possible.

Starting the app

Start Xcode®, choose *File* ▸ *Open...* from the menu bar, navigate to the *HelloWorld* folder, and open file *HelloWorld.xcodeproj*.

Then, choose *Product* ▸ *Destination* from the menu bar, and from the list beneath *iOS Simulator*, choose *iPhone 6*.

Now, hit **⌘-R** in order to build and run the application. This will open two new windows: a terminal window that talks about the *React packager*, and the iOS simulator which shows a visually rather unimpressive app that greets you with a *Welcome to React Native!* message.

> In case you get an ugly red error screen in the iOS Simulator that says *Could not connect to development server*, then proceed as follows: - Switch to a Terminal window - `cd` to the root folder of the *Hello World* project - Run `npm start` and keep the Terminal open

An awful lot of very interesting things just happened in order to display this simulator screen, and dissecting all the components involved and analyzing all the ways these components interact with each other will be a very exciting trip down the rabbit hole that is React Native - but, not yet. First the doing, then the explanations.

Working on the code

As said, a lot of different components are involved that make up our application, but front and center is the JavaScript code that defines and manages the user interface of our app - after all, the central idea behind React Native is the ability to write native apps with JavaScript. So let's look at the JavaScript behind *HelloWorld*.

Fire up the editor or IDE you decided to use and point it at the *HelloWorld* folder you created. In it, you will find a file named *index.ios.js* which you need to open in your editor.

> In Xcode®, you probably saw a file named *main.jsbundle*. For now, this is **not** what we are looking for.

It's immediately obvious that this is the file behind the UI we see in the simulator:

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 */
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;

var HelloWorld = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
        <Text style={styles.instructions}>
          To get started, edit index.ios.js
        </Text>
        <Text style={styles.instructions}>
          Press Cmd+R to reload,{'\n'}
          Cmd+Control+Z for dev menu
```

```
28          </Text>
29        </View>
30      );
31    }
32  });
33
34  var styles = StyleSheet.create({
35    container: {
36      flex: 1,
37      justifyContent: 'center',
38      alignItems: 'center',
39      backgroundColor: '#F5FCFF',
40    },
41    welcome: {
42      fontSize: 20,
43      textAlign: 'center',
44      margin: 10,
45    },
46    instructions: {
47      textAlign: 'center',
48      color: '#333333',
49      marginBottom: 5,
50    },
51  });
52
53  AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

Clearly, this is JavaScript code, but if you haven't followed the latest developments in the JS world or if you did not yet play around with the "normal" React framework, then some parts of this code might look a bit odd. Not that this should stop us from fiddling around!

Change line 20 from

```
Welcome to React Native!
```

to

```
Hello, World!
```

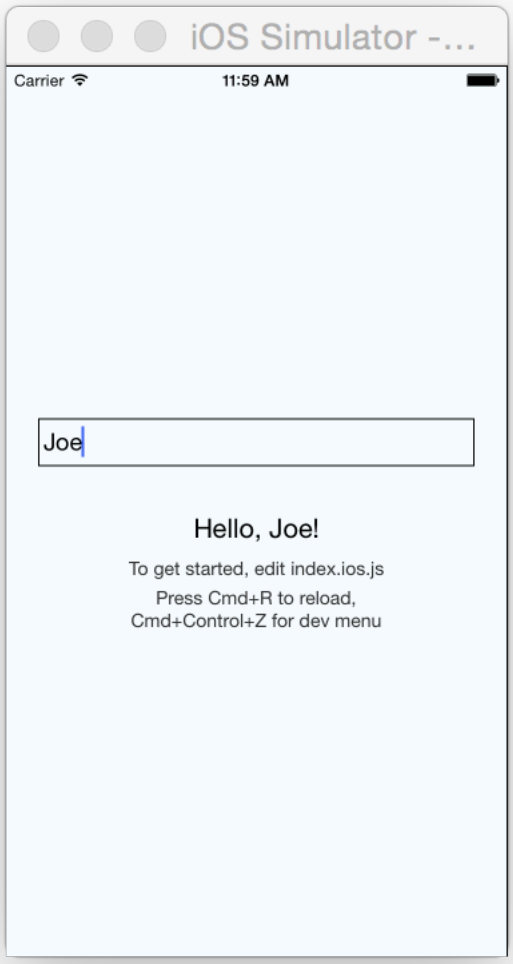
Then switch back to the iOS Simulator window and hit **⌘-R**. The UI of our app will refresh and display the new text.

Now, if you haven't done any native iOS app development before, hitting *Refresh* and seeing the changes you did to the code reflected in the Simulator probably doesn't feel like a big deal, but it actually is. The same procedure for an Objective-C or Swift based application involves a recompilation of the changed source code and a rebuild of the application that is then completely restarted in the Simulator. That doesn't only sound like it takes longer, it *does* take longer.

Once again, the explanation for why and how this "refresh" workflow actually works will follow later.

It would be great if we could make our app *do* something. Let's try to add a text input field for entering a name and change the behaviour of our app in a way that makes it greet the name we enter - if this doesn't secure us the #1 spot in the App Store charts, then I don't know what will.

What we need in order to achieve this functionality is *a)* an additional UI element plus styling that allows to input text, *b)* a function that handles the text that is input, and *c)* a way to output the input text within the greet text element. With that, the result should look like this:



The HelloWorld application with added functionality

Here is the updated code in *index.ios.js* that is needed:

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 */
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  TextInput,
  View,
} = React;

var HelloWorld = React.createClass({
  getInitialState: function() {
    return {
      name: 'World'
    };
  },
  onNameChanged: function(event) {
    this.setState({ name: event.nativeEvent.text });
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TextInput
          style={styles.nameInput}
          onChange={this.onNameChanged}
          placeholder='Who should be greeted?'/>
        <Text style={styles.welcome}>
          Hello, {this.state.name}!</Text>
        <Text style={styles.instructions}>
          To get started, edit index.ios.js
        </Text>
        <Text style={styles.instructions}>
          Press Cmd+R to reload,{'\n'}
          Cmd+Control+Z for dev menu
        </Text>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
```

```
57    },
58    instructions: {
59      textAlign: 'center',
60      color: '#333333',
61      marginBottom: 5,
62    },
63    nameInput: {
64      height: 36,
65      padding: 4,
66      margin: 24,
67      fontSize: 18,
68      borderWidth: 1,
69    }
70  });
71
72  AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

With this, we introduce a new React Native UI element, *TextInput* (line 12), which we add to our view (line 28). An accompanying style definition is added, too (line 63). The view is set up with an initial state (line 17), and we define a function called *onNameChanged* that changes this state (line 22). This function is called whenever the value of the text input changes (line 30). Our text block is now dynamic and always reflects the value of the state variable *name* (line 33).

Re-running the application (remember, **⌘-R** while in *iOS Simulator* is all it takes) presents the new user interface which now shows a text input field. Whatever we put into this field is immediately reflected within the greet message below. Achievement unlocked.

React Native architecture explained

Ok, we played around with it, but learning to understand how React Native actually works is overdue.

A closer look at application creation

Let's reiterate the steps we did while setting up our *HelloWorld* application and take a much closer look while doing so.

The first thing we did after setting up the other prerequisites was to install *react-native-cli* using *npm*, the Node Package Manager. This gave us the `react-native` command, but although the name seems to imply it, this is not *React Native* itself, but merely a small helper tool that allows us to create new React Native projects. In case you are curious, the source code of the `react-native` program lives at https://github.com/facebook/react-native/blob/master/react-native-cli/index.js.

We then ran `react-native init HelloWorld`, which set up our first actual React Native application structure. It created a folder named *HelloWorld* - let's see what's inside:

```
HelloWorld.xcodeproj/
Podfile
iOS/
index.ios.js
node_modules/
package.json
```

To give you the full picture, here is what exactly happens when running `react-native init HelloWorld`:

- A folder named *HelloWorld* is created
- In it, the *package.json* file is created
- `npm install --save react-native` is run, which installs *react-native* and its dependencies into *HelloWorld/node_modules* and declares *react-native* as a dependency of your project in *HelloWorld/package.json*
- The globally installed `react-native` CLI tool then hands over control to the *local* CLI tool that has just been installed into *HelloWorld/node_modules/react-native/local-cli/cli.js*
- This in turn executes *HelloWorld/node_modules/react-native/init.sh*, which is yet another helper script that takes care of putting the boiler plate application code in place, like the minimal React Native code in file *index.ios.js* and the Objective-C code plus other goodies in subfolder *iOS*, and the Xcode® project definition in subfolder *HelloWorld.xcodeproj*

We already saw that the React Native based JavaScript code that makes up our actual application lives in *index.ios.js*.

The *package.json* file is no surprise for those who already worked with io.js or Node.js; it defines some metadata for our project and, most importantly, declares *react-native* (this time, this means the actual framework that makes our application possible) as a dependency of our own project.

The *node_modules* folder is simply a result of the `npm install` run that took place during project initialization. It contains the *react-native* code, which in turn consists of other NPM dependencies, helper scripts, and a lot of JavaScript and Objective-C code.

The initialization process also provided the minimum Xcode® project definitions plus some Objective-C boilerplate code, which allows us to open our new project in Xcode® and to instantly run the application without any further ado. All of this could have been done manually, but would include a lot of steps that are identical no matter what kind of application we are going to write, thus it makes sense to streamline this process via `react-native init`.

> *Podfile* is similar to *package.json*; it declares Objective-C library dependencies for the *CocoaPods* dependency manager. We will talk about this in more detail later in the book, and ignore it for now.

The takeaway here is that our *HelloWorld* project is multiple things at once. It is an Xcode® project, but it's also an NPM project. It's a React Native based JavaScript application, but it also contains some iOS glue code that is needed to make our JavaScript code run on iOS in the first place.

With this overview, we can dive one level deeper and start to look at how the different elements in our project folder interact with each other in order to end up as a working native iOS application.

How the different parts play together

The selling points of React Native are: a) it allows us to write applications for iOS using JavaScript, and b) these applications run natively and provide a fluid UI and user experience.

Now, it's no secret that iOS doesn't run JavaScript apps directly; native applications for this platform need to be written in Objective-C or, as of recently, Swift, and need to be compiled into machine code.

The component on iOS that is known to execute JavaScript is the iOS browser Safari, or, in context of apps, the *UIWebView* component. This has been used in the past to create so-called *hybrid* apps. These are actual native apps written in Objective-C, but the Objective-C code only exists in order to launch a *UIWebView* container, which is a browser engine without the browser UI surrounding it. Inside that container, a web page consisting of HTML, CSS and JavaScript is displayed.

This allows to provide users something that is very close to an actual mobile app by utilizing common web technologies. However, even though the Objective-C code that makes up the wrapper which creates the UIWebView is native, the content within the UIWebView is not - it's just a webpage, which is why these hybrid apps gained the reputation of providing a user experience that isn't as seamless as those of full native apps.

In our *HelloWorld* example, we wrote JavaScript, but we didn't create a webpage. No UIWebView is utilized in order to execute our code.

Instead, our code is run in an embedded instance of JavaScriptCore inside our app and rendered to higher-level platform-specific components. This might sound a bit cryptic, but let's follow the code. In the Xcode® Project navigator, open the file *HelloWorld/AppDelegate.m*. This is what it looks like:

```
/**
 * Copyright (c) 2015-present, Facebook, Inc.
 * All rights reserved.
 *
 * This source code is licensed under the BSD-style license found in the
 * LICENSE file in the root directory of this source tree. An additional grant
 * of patent rights can be found in the PATENTS file in the same directory.
 */

#import "AppDelegate.h"

#import "RCTRootView.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  NSURL *jsCodeLocation;

  // Loading JavaScript code - uncomment the one you want.

  // OPTION 1
  // Load from development server. Start the server from the repository root:
  //
  // $ npm start
  //
  // To run on device, change `localhost` to the IP address of your computer, and make sure your computer and
  // iOS device are on the same Wi-Fi network.
  jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle"];

  // OPTION 2
  // Load from pre-bundled file on disk. To re-generate the static bundle, run
  //
  // $ curl 'http://localhost:8081/index.ios.bundle?dev=false&minify=true' -o iOS/main.jsbundle
  //
  // and uncomment the next following line
  // jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"main" withExtension:@"jsbundle"];

  RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
                                                      moduleName:@"HelloWorld"
                                                   launchOptions:launchOptions];
```

```
42
43    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
44    UIViewController *rootViewController = [[UIViewController alloc] init];
45    rootViewController.view = rootView;
46    self.window.rootViewController = rootViewController;
47    [self.window makeKeyAndVisible];
48    return YES;
49  }
50
51  @end
```

AppDelegate.m is our apps entry point from iOS' point of view. It's the place where the native Objective-C code and our React Native JavaScript code are glued together. The key player in bringing these two worlds together is the *RCTRootView* component, whose header file is imported on line 12.

The *RCTRootView* component is a native Objective-C class provided by the React Native framework. It is the component that takes our React JavaScript code and executes it. And, in the other direction, it allows us to call into native iOS UI elements from our JavaScript code. This way, the controlling code of our applications always is actual JavaScript (it's *not* transformed into Objective-C or Swift or byte code or anything under the hood), but the UI elements that end up on the screen of our iOS device are native *UIKit* elements like those used in classical Objective-C or Swift based apps, and *not* the result of some webpage rendering.

This architecture also explains why we can simply reload our application. If we only change our JavaScript code, then no Objective-C code changes, thus no recompilation is neccessary. The RCTRootView listens to the **⌘-R** sequence and reacts by resetting the UI, retrieving the latest JavaScript code (how this is done is explained in a moment), and executing the new code. The actual native Objective-C app that wraps the RCTRootView simply keeps running.

Feeding JavaScript code into the app

How the RCTRootView receives our JavaScript code is worth a closer look. As the well-commented source code explains, there are two different approaches. The app either loads code from a URL, or from a file that is local to our Xcode® project.

The default is to load from a URL. The URL is *http://localhost:8081/index.ios.bundle*. Where does this come from? The answer lies in the ominous *React packager* we encountered when launching our new app in the iOS Simulator. This packager serves several purposes, with the ultimate goal to provide to our app all the JavaScript code in the right format needed to be executed by the RCTRootView component.

In order to do so, the packager needs to do several things. First of all, it's not enough to only serve the content of our *index.ios.js* file - our very own code has to be bundled together with the React framework. If you point your browser at http://localhost:8081/index.ios.bundle while the app (and therefore, the packager) is running, you will receive a huge chunk of JavaScript that does contain our own code (search for "*Who should be greeted?*" for example), but also contains way over 30,000 lines of other code - the React framework plus some additions for React Native.

> In case the React packager currently isn't running, simply go to the project folder and run `npm start`. As defined in the *package.json*, this executes *node_modules/react-native/packager/packager.sh*.

> As we have seen, the React packager is started automatically upon building and running the app in Xcode®. Sometimes, it is not stopped automatically when closing the iOS Simulator and Xcode®. When this happens, you are not able to start another instance on the packager, because the TCP port is blocked. If this happens, look for a stray Terminal window that still runs the packager, and close it.

Modern JavaScript in React Native

Furthermore, the packager needs to convert the source JavaScript code it loads before serving it. The reason is that the JavaScript interpreter of iOS supports the *ECMAScript, 5th Edition* (or ES5) version of the language, but parts of our own code and the React code are written using JavaScript language features that go beyond the ES5 specification.

This is not to say that we couldn't write our apps in good ol' ES5 and get what we want, i.e., a working React Native app. But one of the niceties of React is that it allows to express our intentions in something far more readable and expressive than "classical" JavaScript. Let's see what this means.

In our *index.ios.js* file, there is a block of code that looks a lot like XML:

```
render: function() {
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.nameInput}
        onChange={this.onNameChanged}
        placeholder='Who should be greeted?'/>
      <Text style={styles.welcome}>
        Hello, {this.state.name}!</Text>
      <Text style={styles.instructions}>
        To get started, edit index.ios.js
      </Text>
      <Text style={styles.instructions}>
        Press Cmd+R to reload,{'\n'}
        Cmd+Control+Z for dev menu
      </Text>
    </View>
  );
}
```

This type of code is called *JSX*, the *XML-like syntax extension to ECMAScript*, to quote from the project homepage at http://facebook.github.io/jsx/. Let's quote even more from that page:

> JSX is a XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers. It's NOT a proposal to incorporate JSX into the ECMAScript spec itself. It's intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript.

The interesting bit here is the one about *transpilers*. I said that the React packager needs to convert the source code before serving it to the app. In fact, there are multiple conversions in place, and *transpiling* JSX into ES5 syntax is one of these conversions.

The transpiled version of the above code block, as we find it at http://localhost:8081/index.ios.bundle, looks like this:

```
render: function() {
  return (
    React.createElement(View, {style: styles.container},
      React.createElement(TextInput, {
        style: styles.nameInput,
        onChange: this.onNameChanged,
        placeholder: "Who should be greeted?"}),
      React.createElement(Text, {style: styles.welcome},
        "Hello, ", this.state.name, "!"),
      React.createElement(Text, {style: styles.instructions},
        "To get started, edit index.ios.js"
      ),
      React.createElement(Text, {style: styles.instructions},
        "Press Cmd+R to reload,", '\n',
        "Cmd+Control+Z for dev menu"
      )
    )
  );
}
```

Hence my claim that we would totally get away with writing straight ES5 JavaScript, but it's obvious that this would result in quite a lot more keystrokes and in much less readable code. This is not to say that the JSX syntax doesn't take some time getting used to, but in my experience it feels very natural real quick. JSX is a very central component of React and React Native, and we will constantly be using it in the course of this book.

There is another transpiler at work in the React packager. It converts from *ECMAScript 2015* to ES5. ECMAScript 2015, or simply *ES2015*, is the upcoming version of JavaScript. It brings a whole lot of new language features to JavaScript like destructuring, computed property keys, classes, arrow functions, block-scoped variables, and much more.

> There has been a lot of irritation around the name of the new JavaScript language version, which is why next to ECMAScript 2015, one encounters any of the following labels around the web: *ECMAScript.next*, *ECMAScript 6*, *ECMAScript Harmony*, and, to simply sum it up, *ES6*. ES6 is still the most widely used term and probably your best friend when searching the web for information on this topic. In this book I will use the short form of the final term that was decided on recently: *ES2015*.
>
> Tip: there is a live ES2015 to ES5 transpiler at https://babeljs.io/repl/.

The specification of the new ES2015 language version isn't 100% final yet, but that's not stopping people from writing ES2015 to ES5 transpilers, which is why the following ES2015 code in our *index.ios.js* file works:

```
var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  TextInput,
  View,
} = React;
```

Here, 5 different variables are assigned a value at once, from the variable *React*. This is called a *destructuring assignment*. Here is a more simple example that shows how it works:

```
var fruits = {banana: "A banana", orange: "An orange", apple: "An apple"};

var { banana, orange, apple } = fruits;
```

This will assign the value "A banana" to the variable *banana*, the value "An orange" to the variable *orange*, and "An apple" to the variable *apple*. The transpiled code looks like this:

```
var fruits = { banana: "A banana", orange: "An orange", apple: "An apple" };

var banana = fruits.banana;
var orange = fruits.orange;
var apple = fruits.apple;
```

Thus we can follow that the *React* variable is an object with keys like *AppRegistry*, *StyleSheet* etc., and we can use this shorthand notation to create and assign variables with the same name all at once.

There is yet another part of our code in *index.ios.js* that needs to be transpiled, on the last line:

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

becomes

```
AppRegistry.registerComponent('HelloWorld', function()  {return HelloWorld;});
```

It's another nifty shorthand notation in ES2015, the *arrow function*, which simplifies anonymous function declaration. There's a bit more to it than just saving keystrokes, we will get to this later.

Summary

Ok, let's recap:

- We have seen that architecturally, our React Native applications are native Objective-C programs that set up an *RCTRootView*, which can be seen as a kind of container where JavaScript code can be executed. This container also allows the JavaScript code to bind to native iOS UI elements, which results in a very fluid user interface.
- In order to get our JavaScript application code together with the React and React Native JavaScript library loaded into the RCTRootView, the *React packager* is used. This io.js application is a transpiler (it converts JSX and ES2015 into ES5 JavaScript), a bundler (it creates one single large script from our own code files and the React library scripts) and a webserver (it provides the transpiled and bundled code via HTTP).
- We learned about some of the modern language approaches React Native takes in regards to JavaScript code, like JSX, destructuring assignments, and the arrow function.

We are now prepared to explore serious application development with React Native. We will learn the details of working with JSX, StyleSheets, ES2015, UIKit elements and much more while doing so.

Creating a first real app: BookBrowser

Playing around with our *Hello World* example wasn't really satisfying. We will now build a new app from scratch that provides useful, real-world functionality.

The app we are going to build will be called *BookBrowser*. It is a small mobile app that acts as an interface to the *Google Books* database. It allows to search for specific books and provides detailed information about the books it finds.

Preparing the new application

In order to start with a clean slate, we need to create a new React Native project called *BookBrowser*. Stop the iOS Simulator, and close the *HelloWorld* project in Xcode®. In case you started the React packager yourself on the command line using *npm start*, make sure that you stop the running process by hitting **Ctrl-C** in the according Terminal window.

> The React packager is a web server that listens on TCP port 8081, and we cannot start another packager process for our new project on that port while another process still occupies it.

On the command line, leave the *HelloWorld* project folder and create the new project by running `react-native init BookBrowser`.

> Consider always running `npm update -g react-native` before creating new projects. React Native is still a very young project that releases new versions regularly.
>
> You may also encounter the scenario where you are working on an application, and the React Native team releases a new version of the framework. You can find out about this by keeping an eye on https://github.com/facebook/react-native/releases/latest.
>
> When a new release takes place and you want to upgrade your current app to the new release, proceed as follows:
>
> - Go to the root folder of your project
> - Open *package.json* and change the version number for *react-native* in the *dependencies* section
> - Run `npm install`

- In Xcode®, stop the running app by hitting ⌘-. (that's the command key and a period), and then hit **shift-⌘-K** to clean all compiled sources

You can now run the app again, and it will compile with the updated framework files.

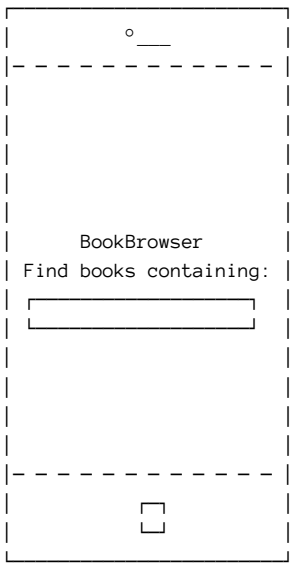
When *react-native* has finished building the project structure, open the newly created *index.ios.js* file in your JavaScript editor, and strip this file to a bare minimum, like this:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
} = React;

var BookBrowser = React.createClass({
  render: function() {
    return (
      <View>
      </View>
    );
  }
});

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

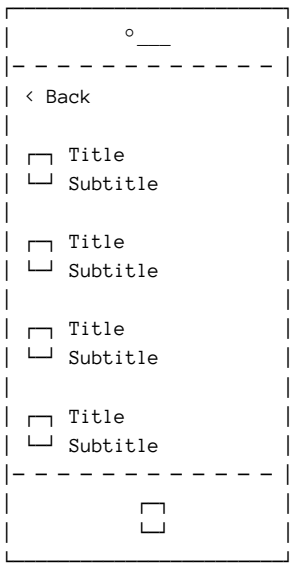
This gives us a "bare minimum" React Native app in the sense that you can't remove another element without ending up with a broken application. From this, we can start. One take away from this is that we can't have an application without a view. We even need a view even if we don't *view* anything.

Planning the app

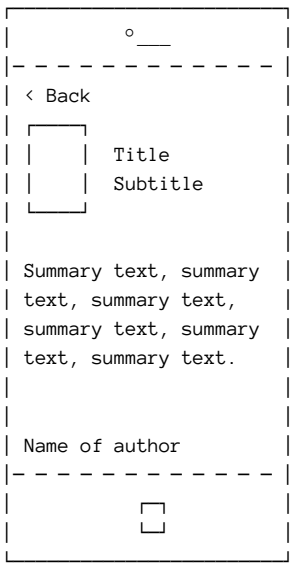
How should the final app look? When starting the app, we should be presented with a search screen which allows to enter text, in order to find books that contain this text in their title.



Starting the search should result in a scrollable results list showing rows of found books with a thumbnail of the cover, the title, and the subtitle:



Finally, the app allows to tap on a result, which makes the app navigate to a book detail screen that displays further information about the selected title:



We likely want to show some kind of activity indicator and a “Please wait…” message during long-running operations (e.g. when querying the *Google Books* API for search results).

With that said, we need to implement the following elements:

- Three screens, one for each step in the workflow
- A mechanism that switches between screens, automatically as well as user-driven
- Several text elements
- A text input field
- Image elements
- A scrollable list whose elements react to tap events
- A mechanism that fetches information (results and details) from the *Google Books* API
- Activity indicators (like the spinning wheel)

Let’s see what building blocks React Native provides for us:

- Screens can be realized by building UIs via JSX and styling them using *React.StyleSheet*.
- Text can be added to these UIs with the *Text* JSX element
- Text inputs are added with the *TextInput* element
- Images are place on the UI with the *Image* element
- Lists are built with the *ListView* element
- Tap events can be handled using *TouchableHighlight* element
- We can build a hierarchy of screens including navigation handling using the *NavigatorIOS* element

- Using React Native's implementation (or rather, polyfill) of the *fetch* abstraction, we can query resources from the web

Looks like we are going to create a lot of JSX structures. These will be accompanied by and combined with some plain JavaScript mechanisms that take care of handling user touch events, resource fetching, and wiring things up.

Building the first screen

Let's start by building the first screen with the search UI right in our bare *index.ios.js* file.

First, we add the *BookBrowser* "headline":

```
1  'use strict';
2
3  var React = require('react-native');
4  var {
5    AppRegistry,
6    View,
7  } = React;
8
9  var BookBrowser = React.createClass({
10   render: function() {
11     return (
12       <View>
13         <Text>
14           BookBrowser
15         </Text>
16       </View>
17     );
18   }
19 });
20
21 AppRegistry.registerComponent('BookBrowser', () => BookBrowser);
```

Reload this in the iOS simulator with **⌘-R**, and you get - a not very subtle red error screen talking about a *getInvalidGlobalUseError*. Why? We are now using another JSX element, *Text*, which is transpiled to

```
React.createElement(Text, null,
  "BookBrowser"
)
```

As you can see, *React.createElement* references an entity named *Text*, but this is not available in the current scope. We need to make it available by assigning it from the *React* object, using the destructuring assignment in line 4:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
  Text,
} = React;

var BookBrowser = React.createClass({
  render: function() {
    return (
      <View>
        <Text>
          BookBrowser
        </Text>
      </View>
    );
  }
});

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

Ok, this makes the app work error-free, but the text we added is located in the upper left corner of the UI. We need to make use of React Native's styling facilities if we want to build a UI that looks reasonable.

Styling UIs with stylesheets

We already encountered UI stylesheets in our *Hello World* application. The semantics and syntax of React Native stylesheets resemble CSS, but it's only an approximation. React Native does not include a CSS compiler or anything like that. Stylesheets are plain JavaScript objects whose keys and attributes are used by React Native in order to apply style settings to native UI elements in a certain way. This styling is limited to the means that the several iOS UIKit elements provide.

JSX structures combined with stylesheets are the React Native equivalent to the Xcode® Interface Builder. This means no point-and-click and no *What You See Is What You Get*.

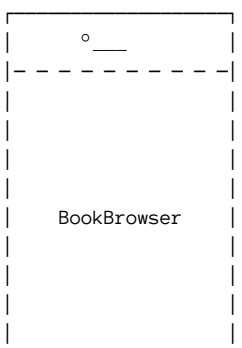
However, thanks to the HTML-and-CSS-like metaphor of JSX and stylesheets, we can approach UI interface building with React Native very much like we approach building web pages.

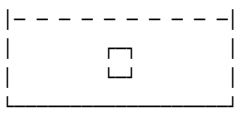
Ok, let's use stylesheet definitions to place our text element in a more pleasant place on the screen.

```
1  'use strict';
2
3  var React = require('react-native');
4  var {
5    AppRegistry,
6    View,
7    Text,
8    StyleSheet,
9  } = React;
10
11 var BookBrowser = React.createClass({
12   render: function() {
13     return (
14       <View style={styles.container}>
15         <Text>
16           BookBrowser
17         </Text>
18       </View>
19     );
20   }
21 });
22
23 var styles = StyleSheet.create({
24   container: {
25     flex: 1,
26     flexDirection: 'column',
27     justifyContent: 'center',
28     alignItems: 'center'
29   }
30 });
31
32 AppRegistry.registerComponent('BookBrowser', () => BookBrowser);
```

> Note how line 8 introduces the *StyleSheet* object before using it on line 23.

With this, the string *BookBrowser* is now placed at the center of the screen, horizontally as well as vertically:





We didn't add that much code, but I need to back up a bit and explain what forces are at work here.

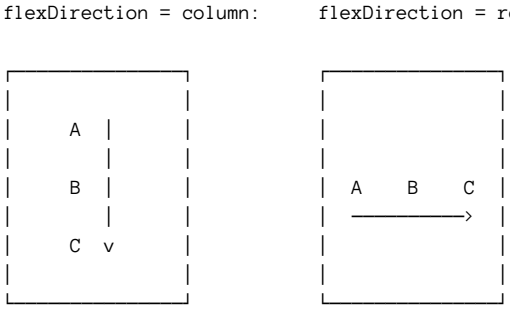
The Flexbox system explained

One of the central mechanics that are utilized when styling React Native app UIs is that of the *Flexbox* layout system. Another CSS metaphor that is translated to React Native UI layouting, this system allows to define how elements within a container are arranged. Not that this wasn't already possible with traditional CSS mechanics, but the semantics of Flexbox feel a lot more natural and a lot less hacky (`align: 0 auto` anyone?).

Using this system is one of the many great ideas in React Native.

The container in our case is our outer-most (and, as of now, only) *View* element. We attach it the *styles.container* definition by setting the *style* attribut on its JSX node.

We enable the Flexbox system for this container by setting *flex* to *1* in the stylesheet. Then, we define a flex direction (or rather, flow direction) for the elements within that container. In this case, it's *column*, which makes elements flow vertically from top to bottom. The other one is *row*, which makes elements flow horizontally from left to right. Here is a simple visualization:

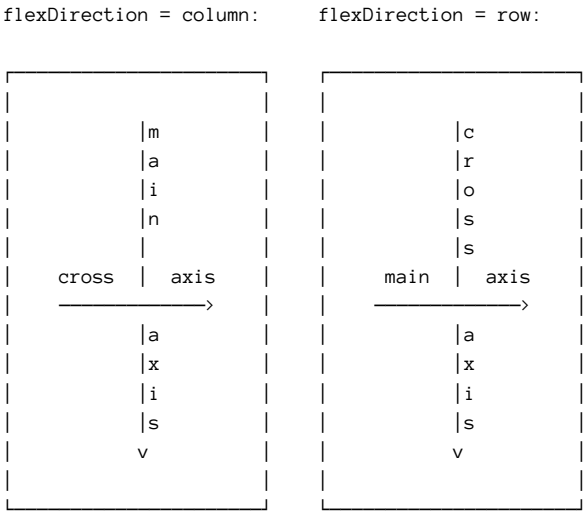


> The Flexbox system in React Native is an approximation of the "real" Flexbox system of CSS. It is not complete and doesn't provide everything that the original provides. E.g., in CSS you can define two other flow directions, *column-reverse* and *row-reverse*. As of this writing, this is not possible with React Native.

Conceptually, what *flexDirection* defines is the so-called *main axis*. It's the orientation and direction along which elements in the container "flow". If you define the *flexDirection* as *row*, and you add

three elements *A*, *B* and *C* to the container, these elements will be rendered on the screen next to each other from left to right, i.e., *A B C*.

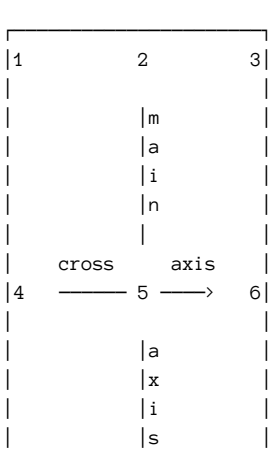
If our *flexDirection* is *column*, then our main axis runs from top to bottom. As a result, the so called *cross axis* then runs from left to right, and vice versa:



Now the other two parameters we defined come into play: *justifyContent* and *alignItems*. Their value defines how elements are aligned on the main axis (via justifyContent) and on the cross axis (via alignItems).

With both parameters set to *center* in our case, and the container filling the whole screen, our element ends up in the middle of the screen, horizontally as well as vertically.

The most useful settings for both parameters are *flex-start*, *center*, and *flex-end* (there are more, but we will not discuss them now). Let's see how these work in our case:



```
|          v          |
|                     |
|7         8         9|
└─────────────────────┘
```

Here are the parameter settings that result in the location of elements 1 to 9:

Element	Location	justifyContent	alignItems
1	top left	flex-start	flex-start
2	top center	flex-start	center
3	top right	flex-start	flex-end
4	middle left	center	flex-start
5	middle center	center	center
6	middle right	center	flex-end
7	bottom left	flex-end	flex-start
8	bottom center	flex-end	center
9	bottom right	flex-end	flex-end

The key to understand where an element ends up on the screen is to visualize where the main axis and the cross axis run (which depends on the value of *flexDirection*) - probably visualize them as actual arrows like in the ascii art diagrams in this book.

A justifyContent value of *flex-start* means that the element is placed at the start of the arrow, *center* means it's placed half-way between the start and the end of the arrow, and *flex-end* means it's placed at the end of the arrow, at the arrowhead. The same applies for the cross axis and the placement definition via *alignItems*.

According to the mental model we have built around the mechanics of the Flexbox system, another JSX text element right after the one for "BookBrowser" should end up below that. Thus, this code:

```
1  'use strict';
2
3  var React = require('react-native');
4  var {
5    AppRegistry,
6    View,
7    Text,
8    StyleSheet,
9  } = React;
10
11 var BookBrowser = React.createClass({
12   render: function() {
13     return (
14       <View style={styles.container}>
15         <Text>
```

```
16            BookBrowser
17          </Text>
18          <Text>
19            Find books containing:
20          </Text>
21        </View>
22      );
23    }
24  });
25
26  var styles = StyleSheet.create({
27    container: {
28      flex: 1,
29      flexDirection: 'column',
30      justifyContent: 'center',
31      alignItems: 'center'
32    }
33  });
34
35  AppRegistry.registerComponent('BookBrowser', () => BookBrowser);
```

will result in this layout:

> In case we set flexDirection to *row*, the "BookText" element would still be placed at the center of the screen, but the next JSX element we add would end up right to "BookText", not below it. You can verify this by simply setting flexDirection from *column* to *row* in your code.

Adding text input

We can now add the third element of our start screen, the text input field. The *TextInput* JSX element takes care of this:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
  Text,
  TextInput,
  StyleSheet,
} = React;

var BookBrowser = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text>
          BookBrowser
        </Text>
        <Text>
          Find books containing:
        </Text>
        <TextInput/>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center'
  }
});

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

Switch to the iOS simulator, hit ⌘-**R**, and - bummer. Looks like nothing has changed at all. And visually, that is indeed true. The new element *is* there, but it isn't visible, because it doesn't have any styling. A text input without styling is a text input that simply doesn't take up any space.

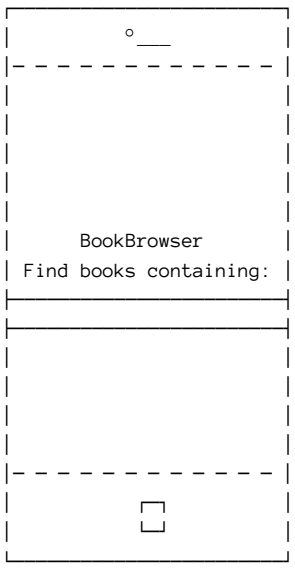
In order to make it visible, add the following stylesheet definition below the *container* block:

```
textInput: {
  height: 30,
  borderWidth: 1,
}
```

Then, apply this definition to the element in the JSX:

```
<TextInput style={styles.textInput}/>
```

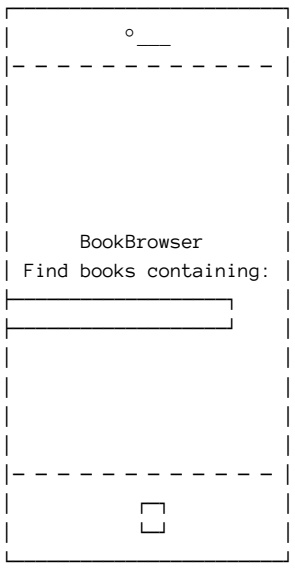
After another reload, the UI layout starts to make sense:



Well, to a certain degree at least. The input box takes up the whole width of the screen. Now, one could assume that setting a *width* in the stylesheet would solve this easily, but the result is not what one would expect:

```
textInput: {
  height: 30,
  width: 300,
  borderWidth: 1,
}
```

results in



With all flex orientations set to *center*, why does the input field end up on the left? My honest answer is: I have no idea. It might even be a bug in React Native. The solution, however, is to not set a width at all, but instead make the element keep some distance using margins:

```
textInput: {
  height: 30,
  borderWidth: 1,
  marginLeft: 60,
  marginRight: 60
}
```

Great. This doesn't exactly give us an award winning layout, but at least a working, correct one.

Now, I'm not a designer and cannot teach you app layout design, but here is a more sophisticated stylesheet which results in a screen that I would consider "nice-looking":

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
  Text,
  TextInput,
  StyleSheet,
} = React;

var BookBrowser = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
```

```
16          <Text style={styles.headline}>
17            BookBrowser
18          </Text>
19          <Text style={styles.label}>
20            Find books containing:
21          </Text>
22          <TextInput
23            placeholder="e.g. JavaScript or Mobile"
24            style={styles.textInput}/>
25        </View>
26      );
27    }
28  });
29
30  var styles = StyleSheet.create({
31    container: {
32      flex: 1,
33      flexDirection: 'column',
34      justifyContent: 'center',
35      alignItems: 'center',
36      backgroundColor: '#5AC8FA',
37    },
38    headline: {
39      fontSize: 36,
40      fontWeight: 'bold',
41      color: '#FFF',
42      marginBottom: 28,
43    },
44    label: {
45      fontSize: 24,
46      fontWeight: 'normal',
47      color: '#FFF',
48      marginBottom: 8,
49    },
50    textInput: {
51      borderColor: '#8E8E93',
52      borderWidth: 0.5,
53      backgroundColor: '#FFF',
54      height: 40,
55      marginLeft: 60,
56      marginRight: 60,
57      padding: 8,
58    }
59  });
60
61  AppRegistry.registerComponent('BookBrowser', () => BookBrowser);
```

Some notes on these changes.

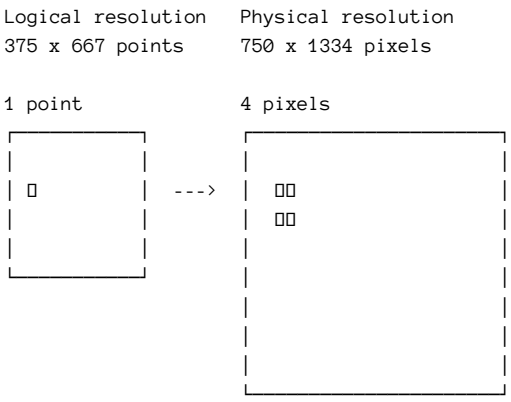
Using HTML and CSS, we can simply set text properties like *color* or *font-size* on a parent, non-text element like a *div*, and all text elements contained within that div will have these properties applied. This is *not* the case with React Native. Each JSX text element needs to be styled directly, which is why I created a *headline* and *label* style definition and applied it to the pertinent JSX *Text* elements.

Just like in HTML5, text input fields can have a placeholder text (line 23) which is presented to the user as long as no text is put into the field.

Of points and pixels

In our stylesheet, we are working with numerical values whenever we set size attributes like height, margins, or border widths. The unit of these values is *points*, and for every iPhone after the 3GS, one point does *not* equal one pixel. Retina displays have a very high physical screen resolution (750 x 1334 pixels in case of the iPhone 6), but we don't address these pixels directly. Instead, using the point system, we address a logical screen resolution, and it's device-dependent how many physical pixels end up on the physical device screen when painting an element to the logical screen using points.

In case of non-retina devices (like the iPhone 3GS), the translation is 1:1, that is, one point is drawn as one pixel. In case of the iPhone 6 Retina display, the translation is 1:2, that is, an element with a width of 1 point and a height of 1 point is rendered using a total of 4 pixels:



The *borderWidth* of the text input field is set to 0.5 - on the iPhone 6, it is therefore rendered with a width of 1 physical pixel.

While this system might seem limiting at first, it really eases app development for multiple devices a lot, because the logical resolution approach gives us a common coordinate system independent of actual screen resolution. If we had to work with the physical resolution, we had to handle device differences ourselves. This would probably feel a bit like this pseudo-code:

```
if (device == 'iPhone 6') {
    margin = 40;
}
if (device == 'iPhone 3GS') {
    margin = 20;
}
```

> See http://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions for a very nice overview of iOS devices and their respective logical and physical resolutions.

Handling events

Ok, enough layout magic, at least for now. We need to build some logic into our app - when entering text and submitting the input, we need to query the Google Books API for titles containing the entered text, and we need to present a list of matching books to the user.

That's three distinct steps we need to take. First, we need to react to the event of the text input being submitted, or finished. As we will see, this can be done implicitly - we don't need to put a "Go!" button into the UI. The input field itself is configurable enough for our needs.

Here are the changes we need to introduce in order to make our text input more intelligent:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
  Text,
  TextInput,
  StyleSheet,
} = React;

var BookBrowser = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.headline}>
          BookBrowser
        </Text>
        <Text style={styles.label}>
          Find books containing:
        </Text>
        <TextInput
          placeholder="e.g. JavaScript or Mobile"
          returnKeyType="search"
```

```
25            enablesReturnKeyAutomatically="true"
26            onEndEditing={ event => console.log(event.nativeEvent.text) }
27            style={styles.textInput}/>
28        </View>
29      );
30    }
31  });
32
33  var styles = StyleSheet.create({
34    container: {
35      flex: 1,
36      flexDirection: 'column',
37      justifyContent: 'center',
38      alignItems: 'center',
39      backgroundColor: '#5AC8FA',
40    },
41    headline: {
42      fontSize: 36,
43      fontWeight: 'bold',
44      color: '#fff',
45      marginBottom: 28,
46    },
47    label: {
48      fontSize: 24,
49      fontWeight: 'normal',
50      color: '#fff',
51      marginBottom: 8,
52    },
53    textInput: {
54      borderColor: '#8E8E93',
55      borderWidth: 0.5,
56      backgroundColor: '#fff',
57      height: 40,
58      marginLeft: 60,
59      marginRight: 60,
60      padding: 8,
61    }
62  });
63
64  AppRegistry.registerComponent('BookBrowser', () => BookBrowser);
```

The changes are on line 24, 25 and 26. We added three new properties to the TextInput element: *returnKeyType*, *enablesReturnKeyAutomatically*, and *onEndEditing*.

The result of the first property is purely aesthetically. It changes the default return button of the iOS software keyboard from *return* to *Search*.

> In case you don't see the software keyboard when entering text into the input field, hit ⌘-K in the simulator in order to activate it.

The *enablesReturnKeyAutomatically* property is more interesting. With it, we are introducing a first tiny bit of logic into our application by utilizing an implicit UI element behaviour. Setting this property to true has the effect that the user can only submit the input field if any text was entered. If no text has been entered, the *Search* button on the software keyboard is grayed out and cannot be used.

With *onEndEditing*, things finally get interesting. We know *on* attributes from HTML: *onClick*, *onBlur* etc. React Native brings this approach to our JSX UI structure. Attributes like *onEndEditing* bind a specific event to a JavaScript expression. In this case, the event happens when a user has finished editing the text input, that is, when she hits the *Search* button on the software keyboard.

JavaScript Expressions in JSX

As said, the value of this attribute is a JavaScript expression. In fact, it's not the first time we add a JavaScript expression to our JSX. All attributes with a value in curly braces bind to an expression. For example, in a line like this

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

the *style* attribute is assigned the result of the JavaScript expression `styles.container`. That isn't a very exciting expression, of course: it's simply a reference to the attribute of an object. More complex expression are possible, though.

Instead of referencing the stylesheet, we could also define the style settings inline:

```
<View style={{
  flex: 1,
  flexDirection: 'column',
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: '#5AC8FA',
}}>
```

> The first curly brace starts the expression, the second curly brace starts the object containing the style attributes.

Want to randomly give the app one of two possible background colors? No problem. While full-on *if* statements cannot be used inside JSX JavaScript expressions, the ternary operator works just fine:

```
<View style={{
  flex: 1,
  flexDirection: 'column',
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: Math.random() > 0.5 ? '#5AC8FA' : '#4CD964',
}}>
```

Repeatedly reloading the app in the simulator will present the app with a green background 50% of the time.

But back to *onEndEditing*. What exactly is our JavaScript expression here, and what does it do?

```
onEndEditing={ event => console.log(event.nativeEvent.text) }
```

The curly braces, of course, start the expression. Then, using the ES2015 arrow function syntax (`param => expression`), we declare an anonymous function.

Remember, it's just a new shorthand notation. Functionally, it's identical to

```
function(event) { return console.log(event.nativeEvent.text); }
```

And if you once again open http://localhost:8081/index.ios.bundle and search for "*React.createElement(TextInput*", you will see that this is the ES5 code that React packager transpiled the ES2015 code to. This also means that nothing stops us from putting the ES5 version into the JSX:

```
onEndEditing={ function(event) { return console.log(event.nativeEvent.text); } }
```

But of course, the arrow function syntax makes our code much leaner.

The actual logic we trigger when the event occurs is not very exciting yet. We simply log the value of the *text* attribute of the event to the console.

Nevertheless, two things are interesting here. How does the event get here, and what does console logging mean in a React Native JavaScript environment versus console logging in a web browser JavaScript environment?

Let's look at the event first. When the user finishes editing and "submits" the text input field by tapping *Search* on the software keyboard, the event occurs. This leads to React Native creating the *event* object and passing it as the first and only parameter to the anonymous function we bind to the *onEndEditing* property.

> Because the event originates on a native iOS UI element, it first exists in the realm of compiled Objective-C code. Again, its the RCTRootView bridge that takes care of connecting native code and JavaScript code. It transforms the iOS event into a JavaScript event, so to speak. React Native wraps the UI event into a *SyntheticEvent* (that's a class React Native ships with). The same happens in the classical React library used for web development: browser events get translated into React SyntheticEvents.

In case you are curious, this is how the event object looks like:

```
{
  "dispatchConfig": {
    "phasedRegistrationNames": {
      "bubbled":"onEndEditing",
      "captured":"onEndEditingCapture"
    }
   },
   "dispatchMarker":".r[1]{TOP_LEVEL}[0].2",
   "nativeEvent": {
      "target":7,
      "text":"deedde"
   },
   "target":7,
   "currentTarget":".r[1]{TOP_LEVEL}[0].2",
   "timeStamp":1430810753006,
   "_dispatchIDs":".r[1]{TOP_LEVEL}[0].2"
}
```

Once the event has been wrapped into a usable form and has been passed down into our own code, we can react to it and do something useful with it. Right now, we log it. Were do log messages end up?

Logging and debugging

Well, it depends. React Native kindly redirects them into Xcode® land, i.e., they are received by Xcode® just like debug messages from Objective-C applications are. Xcode® shows these messages in the so-called *Console*. You can make the Console visible in Xcode® by hitting `shift-⌘-C`. It appears at the bottom of the Xcode® window - you might want to pull that area of the Xcode® interface a bit larger in order to comfortably read log messages.

If now you enter text into the input field and submit by clicking on “Search”, a message will appear in the log window:

```
2015-05-05 21:30:20.580 [info][tid:com.facebook.React.JavaScript] "Hello World"
```

You will also receive helpful warning messages from React Native itself, like this one:

```
2015-05-05 21:30:14.933 [warn][tid:com.facebook.React.JavaScript] "Warning: Failed propType: Invalid prop `ena\
blesReturnKeyAutomatically` of type `string` supplied to `TextInput`, expected `boolean`. Check the render met\
hod of `BookBrowser`."
```

Whoops, let’s fix that. We pass a string to the *enablesReturnKeyAutomatically* attribute of our TextInput field, but it expects a boolean. How can we pass a boolean? That’s simple, we just use a JavaScript expression:

```
<TextInput
  placeholder="e.g. JavaScript or Mobile"
  returnKeyType="search"
  enablesReturnKeyAutomatically={true}
  onEndEditing={ event => console.log(event.nativeEvent.text) }
  style={styles.textInput}/>
```

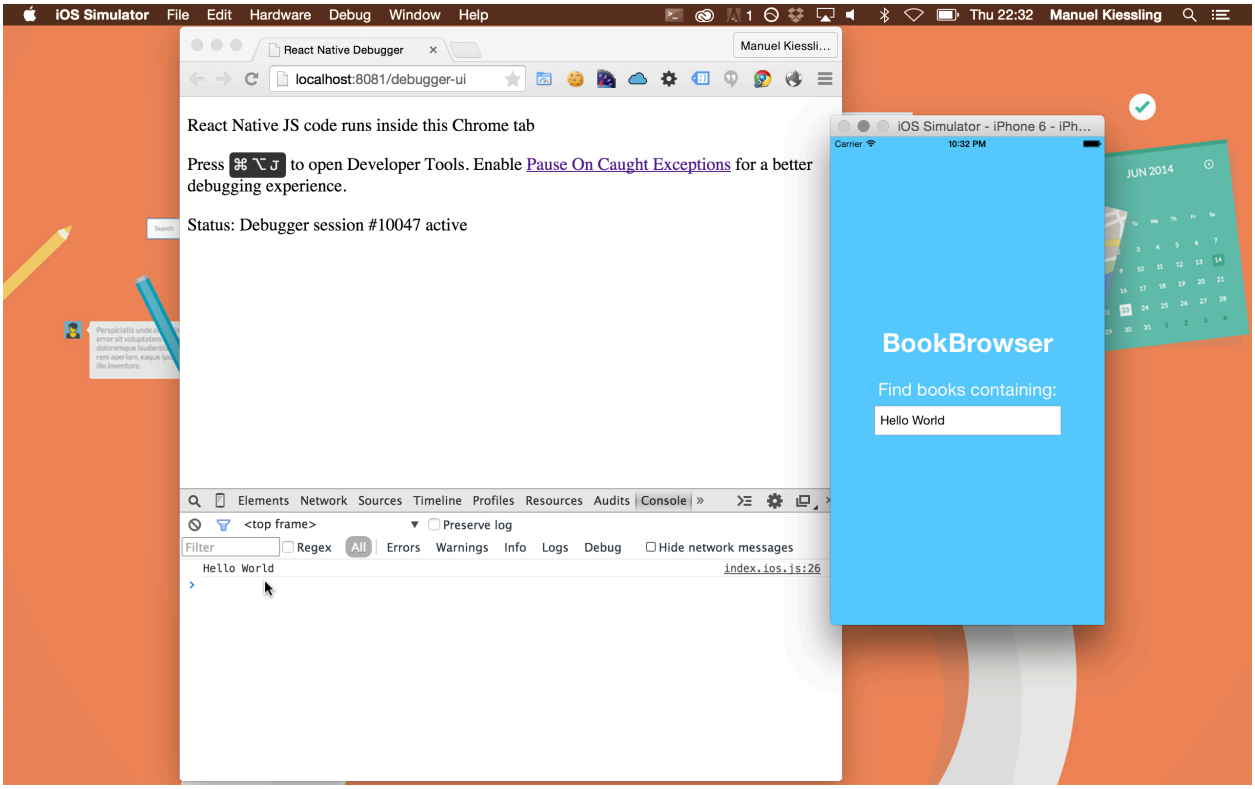
Reading log messages in Xcode® does the job, but its not really convenient. Good news: you can get them into the development console of your browser, just as you would while developing for the web.

Here's how:

- Launch Google Chrome
- Run the app in the simulator
- In the simulator, choose *Shake Gesture* from the *Hardware* menu (no, shaking your monitor won't do)
- Within the app, a menu is shown. Click *Enable Chrome Debugging*

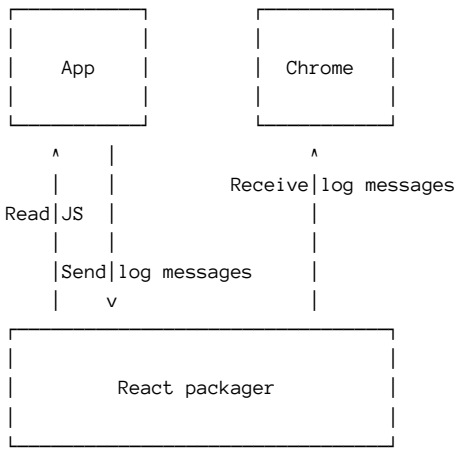
Now a new tab is opened in Chrome, which tells us what to do: Hit **⌘-⌥-J** (that's cmd-alt-J), and the Chrome Developer Tools are opened. There, in the *Console* view, we can now see the debug log messages from our React Native app.

The result of this setup should look like this:



A working React Native - Google Chrome debug setup

Once again it's the React packager that makes things possible. Our app holds a connection to the packager webserver on port 8081 and sends its log messages there, and the page that opened in Chrome is also pointing at this server in order to retrieve the messages and pipe them into the Developer Tools console.



Note that when Chrome debugging is enabled, debug messages will be sent *only* to Chrome, and will now longer appear inside Xcode®.

In addition to logging with *console.log*, you can also log messages with other severities: use *console.debug*, *console.info*, *console.warn* and *console.error*. Also, you can log objects in a more structured way. In our example, replace

```
console.log(event.nativeEvent.text)
```

with

```
console.dir(event.nativeEvent)
```

on line 26 and the Developer Toolbar console will allow you to browse the structure and content of the *event.nativeEvent* object - something that Xcode® can't do at all.

Live reloading

When you opened the debug menu in the simulator using the virtual shake gesture, you probably noticed another menu item: *Enable Live Reload*. This is a really nifty feature that "classical" iOS developers can only dream of. When enabled, it's enough to simply save the *index.ios.js* file in your editor - the app will reload automatically. This really shines when working on the stylesheet - you change a value, e.g. a margin, and upon save, you immediately see the layout change in the actual app. This allows for very rapid app layouting in a very direct feedback loop.

Setting up app screen navigation

We now have a first working screen which allows us to start a book search. When a search is started, we need to switch to a new screen where we can display the search results. Conceptually, this is like sending the user to the next page after submitting a form on a web page, but technically, it's different.

In iOS, in-app navigation uses the concept of a hierarchy of routes. We need to build this hierarchy, and instead of rendering our search screen directly, we need to make it the root node of the routes hierarchy. From there, we can start navigating.

The key to this is the *NavigatorIOS* component provided by React Native.

Right now, *BookBrowser* is a React object that renders a View component. We need to rewrite it into an object that renders a NavigatorIOS component instead, and this component needs to be set up with our search screen view.

To do so, we first need to rename the variable that holds our search screen View, like so:

```
var SearchScreen = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.headline}>
          BookBrowser
        </Text>
        <Text style={styles.label}>
          Find books containing:
        </Text>
        <TextInput
          placeholder="e.g. JavaScript or Mobile"
          returnKeyType="search"
          enablesReturnKeyAutomatically={true}
          onEndEditing={ event => console.log(event.nativeEvent.text) }
          style={styles.textInput}/>
      </View>
    );
  }
});
```

We can now recreate the *BookBrowser* object, making it carry a NavigatorIOS component that references the *SearchScreen* object as its initial route:

```
var BookBrowser = React.createClass({
  render: function() {
    return (
      <NavigatorIOS
        initialRoute={{
          component: SearchScreen,
          title: 'Search',
        }}
        style={styles.navContainer}
      />
    );
  }
});
```

Not too complicated, is it? Instead of rendering a *<View>* component, we render a *<NavigatorIOS>* component, and we pass our *SearchScreen* object via the *initialRoute* attribute. We also define a title for this route, which becomes the headline of our screen.

Like all React Native view components, the *<NavigatorIOS>* component needs a style, and of course we need to declare it before accessing it, which results in the following code:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  View,
  Text,
  TextInput,
  StyleSheet,
  NavigatorIOS,
} = React;

var SearchScreen = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.headline}>
          BookBrowser
        </Text>
        <Text style={styles.label}>
          Find books containing:
        </Text>
        <TextInput
          placeholder="e.g. JavaScript or Mobile"
          returnKeyType="search"
          enablesReturnKeyAutomatically={true}
          onEndEditing={ event => console.log(event.nativeEvent.text) }
          style={styles.textInput}/>
      </View>
    );
  }
});

var BookBrowser = React.createClass({
  render: function() {
    return (
      <NavigatorIOS
        initialRoute={{
          component: SearchScreen,
          title: 'Search',
        }}
        style={styles.navContainer}
      />
    );
  }
});

var styles = StyleSheet.create({
  navContainer: {
    flex: 1,
  },
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
```

```
    backgroundColor: '#5AC8FA',
  },
  headline: {
    fontSize: 36,
    fontWeight: 'bold',
    color: '#fff',
    marginBottom: 28,
  },
  label: {
    fontSize: 24,
    fontWeight: 'normal',
    color: '#fff',
    marginBottom: 8,
  },
  textInput: {
    borderColor: '#8E8E93',
    borderWidth: 0.5,
    backgroundColor: '#fff',
    height: 40,
    marginLeft: 60,
    marginRight: 60,
    padding: 8,
  }
});

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

Now that we began introducing a logical structure into our code, we can as well start to give it more physical structure, too. It makes sense to keep each of our screens in its own file, and keep *index.ios.js* as the entry point for the application that sets up the navigation structure.

Start by creating a new file called *SearchScreen.js*, in the root folder of the project:

```
'use strict';

var React = require('react-native');
var {
  View,
  Text,
  TextInput,
  StyleSheet,
  } = React;

var SearchScreen = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.headline}>
          BookBrowser
        </Text>
        <Text style={styles.label}>
          Find books containing:
        </Text>
        <TextInput
```

```
22            placeholder="e.g. JavaScript or Mobile"
23            returnKeyType="search"
24            enablesReturnKeyAutomatically={true}
25            onEndEditing={ event => console.log(event.nativeEvent.text) }
26            style={styles.textInput}/>
27        </View>
28      );
29    }
30  });
31
32  var styles = StyleSheet.create({
33    container: {
34      flex: 1,
35      flexDirection: 'column',
36      justifyContent: 'center',
37      alignItems: 'center',
38      backgroundColor: '#5AC8FA',
39    },
40    headline: {
41      fontSize: 36,
42      fontWeight: 'bold',
43      color: '#fff',
44      marginBottom: 28,
45    },
46    label: {
47      fontSize: 24,
48      fontWeight: 'normal',
49      color: '#fff',
50      marginBottom: 8,
51    },
52    textInput: {
53      borderColor: '#8E8E93',
54      borderWidth: 0.5,
55      backgroundColor: '#fff',
56      height: 40,
57      marginLeft: 60,
58      marginRight: 60,
59      padding: 8,
60    }
61  });
62
63  module.exports = SearchScreen;
```

This file only contains what is needed to render the search screen: We import the React Native components that are used in the screen, we define the *SearchScreen* object with the JSX that describes the structure of this screen, and we declare the stylesheet object that is referenced from the JSX.

Finally, we *export* the SearchScreen object, which allows to load and reference it from other files via *require*. Which is what we do in our now much shorter *index.ios.js*:

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  NavigatorIOS,
} = React;

var SearchScreen = require('./SearchScreen');

var BookBrowser = React.createClass({
  render: function() {
    return (
      <NavigatorIOS
        initialRoute={{
          component: SearchScreen,
          title: 'Search',
        }}
        style={styles.navContainer}
      />
    );
  }
});

var styles = StyleSheet.create({
  navContainer: {
    flex: 1,
  },
});

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

If you restart the app, nothing has changed. Code now comes from two different source files, but the React packager resolves this for us. It recognizes that *index.ios.js* requires a file named *SearchScreen.js*, and pulls the content of this file into the bundle it serves at http://localhost:8081/index.ios.bundle.

Navigating to a second screen

Now that we laid the foundation for an app with several screens, let's add a second screen and see how we can navigate the user from one screen (the Search) to the next (the Results).

To do so, create a third file named *ResultsScreen.js*. Fill it with some basic content:

```
'use strict';

var React = require('react-native');
var {
  View,
  Text,
  StyleSheet,
  } = React;

var ResultsScreen = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.label}>
          This is the results screen
        </Text>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
  },
  label: {
    fontSize: 24,
    fontWeight: 'normal',
    color: '#fff',
  },
});

module.exports = ResultsScreen;
```

The whole logic that brings us from the Search screen to the Results screen lives in the *SearchScreen.js* file. We need to extend it with three parts: We need to require the ResultsScreen code, we need to add a function that triggers the navigation to the next screen, and we need to wire this function to the event that fires when the text input field is submitted. The result looks like this:

```
'use strict';

var React = require('react-native');
var {
  View,
  Text,
  TextInput,
  StyleSheet,
  } = React;

var ResultsScreen = require('./ResultsScreen');

var SearchScreen = React.createClass({

  gotoResultsScreen: function() {
    this.props.navigator.push({
      title: 'Results',
      component: ResultsScreen,
    });
  },

  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.headline}>
          BookBrowser
        </Text>
        <Text style={styles.label}>
          Find books containing:
        </Text>
        <TextInput
          placeholder="e.g. JavaScript or Mobile"
          returnKeyType="search"
          enablesReturnKeyAutomatically={true}
          onEndEditing={ event => this.gotoResultsScreen() }
          style={styles.textInput}/>
      </View>
    );
  }

});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
  },
  headline: {
    fontSize: 36,
    fontWeight: 'bold',
    color: '#fff',
    marginBottom: 28,
  },
```

```
57    label: {
58      fontSize: 24,
59      fontWeight: 'normal',
60      color: '#fff',
61      marginBottom: 8,
62    },
63    textInput: {
64      borderColor: '#8E8E93',
65      borderWidth: 0.5,
66      backgroundColor: '#fff',
67      height: 40,
68      marginLeft: 60,
69      marginRight: 60,
70      padding: 8,
71    }
72  });
73
74  module.exports = SearchScreen;
```

Let's look at each change.

Line 11 is simple: We need the *ResultsScreen* object, therefore we must require the code file that declares it.

Line 15 introduces the *gotoResultsScreen* function - it's where the interesting stuff happens. In it, we call the *push()* function of object *navigator*, which is part of the *props* attribute of our SearchScreen class.

Line 35 is straight-forward again - we simply trigger the new function instead of logging the event.

Switch to the simulator and reload in order to see the result of these seemingly small changes. We have introduced full-fledged screen navigation! Upon hitting enter or tapping the *Search* button on the software keyboard, we navigate to the Results screen via a simple but nice animation, the title of the screen changes to "Results", and we get a *<Search>* button in the upper left corner which allows us to navigate back to the search screen. Not bad.

The question, of course, is: what is the *props* attribute, and how did it receive the *navigator* object?

Component properties - props

First off, every React Native component has properties, or props. The props are the options of a React Native component that are set from the outside and are passed into the component.

For example, props are used by a parent component to set up an initial configuration for a child component. This is what happens in our case: The *BookBrowser* component defines a *NavigatorIOS* JSX element, which declares *SearchScreen* to be its child via the *initialRoute* parameter. Behind the scenes, the *NavigatorIOS* element passes itself to the *SearchScreen* component, where it becomes available as the *navigator* prop.

Props can also be set explicitly. In case of *NavigatorIOS*, this is done through the *passProbs* attribute of the *initialRoute* object:

```
<NavigatorIOS
  initialRoute={{
    component: SearchScreen,
    title: 'Search',
    passProps: { placeholder: 'javascript' },
  }}
  style={styles.navContainer}
/>
```

With this, our *SearchScreen* component now has another prop named *placeholder*, with a value of *javascript*. Within the *SearchScreen* component, we can access and use it like so:

```
<TextInput
  placeholder={this.props.placeholder}
  returnKeyType="search"
  enablesReturnKeyAutomatically={true}
  onEndEditing={ event => this.gotoResultsScreen() }
  style={styles.textInput}/>
```

Of course, we can also access it outside the context of our JSX, as we did with the *navigator* prop inside our *gotoResultsScreen* function.

Passing static values as props is only mildly interesting. It's much more useful when passing dynamic values - the props mechanism is the perfect way to pass the search phrase from the Search screen to the Results screen. To do so, change the *gotoResultsScreen* function as follows, in file *SearchScreen.js* on lines 15 to 21:

```
gotoResultsScreen: function(searchPhrase) {
  this.props.navigator.push({
    title: 'Results',
    component: ResultsScreen,
    passProps: { 'searchPhrase': searchPhrase }
  });
},
```

and on line 36, pass the text input content to the function:

```
onEndEditing={ event => this.gotoResultsScreen(event.nativeEvent.text) }
```

With this, our ResultsScreen component receives a new prop named *searchPhrase*, which we can use in the render JSX in file *ResultsScreen.js* like this:

```
<View style={styles.container}>
  <Text style={styles.label}>
    This is the results screen
  </Text>
  <Text style={styles.label}>
    You searched for: {this.props.searchPhrase}
  </Text>
</View>
```

Reload once again, and the result is that not only can we navigate between screens, we can also pass data between them.

Retrieving data over the network with the fetch API

We can now concentrate on the application logic of the results screen. We need to start an actual book search in order to make the screen useful. When we receive results, we need to present them to the user, and we need to display an error message if no results could be found.

In order to implement this, we need to introduce several new elements. We will learn about the *fetch* pattern, component state, the *ListView*, *Image* and *TouchableHighlight* components, and we will see how the render process can be split into multiple parts that are interchanged dynamically.

Let's start with the part that fetches data from the *Google Books* API. Our interaction with this API is really simple. All we need to do is make a GET request against the *https://www.googleapis.com/books/v1/volumes* endpoint, which must be parameterized with a search phrase that filters results to titles matching the phrase. The response is a JSON encoded list of books. You can give it a try by pointing your browser at https://www.googleapis.com/books/v1/volumes?q=javascript&langRestrict=en&maxResults=40, which returns a list of upt to 40 english books related to JavaScript.

We need to make this request from our application, passing the user submitted search for the *q* parameter of the endpoint.

Because we need to correctly encode the parameter, a little helper function is in order:

```
var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};
```

This will go into ResultsScreen.js later.

Now to the act of fetching. No big deal in web applications nowadays with tools like jQuery, but remember, this is not the web, and we are not in a browser. Luckily, JavaScript recently got a very nice abstraction for loading content over the web, the *fetch* API. It's a huge improvement over XHR, and it allows to express fetching data from remote web services in a very concise way:

```
fetch(url).then(function(response) {
  return response.json();
}).then(function(jsonData) {
  console.dir(jsonData);
}).catch(function(error) {
  console.dir(error);
});
```

As we can use ES2015 features like arrow functions in React Native, we can shorten that even more:

```
fetch(url)
  .then(response => response.json())
  .then(jsonData => console.log(jsonData))
  .catch(error   => console.log(error));
```

fetch works asynchronously, and handles success cases as well as errors. If everything works fine, we receive an HTTP response, which is passed to the first *then* block, and we can pull the JSON payload from the response, which is passed to the second *then* block, where we can handle the received JSON data as we see fit.

In case of network problems or other errors, the *catch* block triggers and receives an error object, and we have the chance to handle this case, too.

Let's now see how we can integrate this pattern into our Results screen. We need to extend file *ResultsScreen.js* by adding the *buildUrl* function, add the code that does the fetching, and trigger the fetching operation when the screen is loaded. The natural place for the latter is the *componentDidMount* function that every React Native component automatically triggers when it is put on the screen.

```
1  'use strict';
2
3  var React = require('react-native');
4  var {
5    View,
6    Text,
7    StyleSheet,
8    } = React;
9
10 var buildUrl = function(q) {
11   return 'https://www.googleapis.com/books/v1/volumes?q='
12     + encodeURIComponent(q)
13     + '&langRestrict=en&maxResults=40';
14 };
15
16 var ResultsScreen = React.createClass({
17
18     componentDidMount: function() {
19       this.fetchResults(this.props.searchPhrase);
20     },
21
22     fetchResults: function(searchPhrase) {
```

```
23        fetch(buildUrl(searchPhrase))
24          .then(response => response.json())
25          .then(jsonData => console.dir(jsonData))
26          .catch(error => console.dir(error));
27      },
28
29      render: function() {
30        return (
31        <View style={styles.container}>
32          <Text style={styles.label}>
33            This is the results screen
34          </Text>
35          <Text style={styles.label}>
36            You searched for: {this.props.searchPhrase}
37          </Text>
38        </View>
39      );
40    }
41  });
42
43  var styles = StyleSheet.create({
44    container: {
45      flex: 1,
46      flexDirection: 'column',
47      justifyContent: 'center',
48      alignItems: 'center',
49      backgroundColor: '#5AC8FA',
50    },
51    label: {
52      fontSize: 24,
53      fontWeight: 'normal',
54      color: '#fff',
55    },
56  });
57
58  module.exports = ResultsScreen;
```

The additions are on lines 10-12 and lines 16-25.

> Because the *buildUrl* function provides generic functionality that isn't specific to the UI, I've made it a generic function in the scope of the file, while the *fetchResults* function will be very specific to its UI component, which is why I've made it a method of the component.

When running the updated app with Chrome Debugging enabled, you will see a results object with an array of found books in the *items* array in the Developer Tools *Console* tab after starting a search.

> Let's take a moment to look at the data and its structure that we receive from the Google Books API:

```
{
   kind: "books#volumes",
   totalItems: 958,
 ▾ items: [
    ▾ {
         kind: "books#volume",
         id: "PXa2bby0oQ0C",
         etag: "3pv7hRXCH5Q",
         selfLink: https://www.googleapis.com/books/v1/volumes/PXa2bby0oQ0C,
       ▾ volumeInfo: {
            title: "JavaScript: The Good Parts",
            subtitle: "The Good Parts",
          ▸ authors: [ … ],
            publisher: "\"O'Reilly Media, Inc.\"",
            publishedDate: "2008-05-08",
            description: "Most programming languages contain good and bad parts, but JavaScript has more
            than its share of the bad, having been developed and released in a hurry before it could be
            refined. This authoritative book scrapes away these bad features to reveal a subset of
            JavaScript that's more reliable, readable, and maintainable than the language as a whole-a
            subset you can use to create truly extensible and efficient code.",
          ▸ industryIdentifiers: [ … ],
          ▸ readingModes: { … },
            pageCount: 172,
            printType: "BOOK",
          ▸ categories: [ … ],
            averageRating: 4.5,
            ratingsCount: 61,
            maturityRating: "NOT_MATURE",
            allowAnonLogging: false,
            contentVersion: "0.6.6.0.preview.3",
          ▾ imageLinks: {
               smallThumbnail: http://bks8.books.google.de/books/content?id=PXa2bby0oQ0C&
               printsec=frontcover&img=1&zoom=5&edge=curl&source=gbs_api,
               thumbnail: http://bks8.books.google.de/books/content?id=PXa2bby0oQ0C&printsec=frontcover&
               img=1&zoom=1&edge=curl&source=gbs_api
            },
            language: "en",
            previewLink: http://books.google.de/books?id=PXa2bby0oQ0C&printsec=frontcover&dq=javascript&
            hl=&cd=1&source=gbs_api,
            infoLink: http://books.google.de/books?id=PXa2bby0oQ0C&dq=javascript&hl=&source=gbs_api,
            canonicalVolumeLink: http://books.google.de/books/about/JavaScript_The_Good_Parts.html?hl=&
            id=PXa2bby0oQ0C
         },
       ▸ saleInfo: { … },
       ▸ accessInfo: { … },
       ▸ searchInfo: { … }
      },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … },
    ▸ { … }
   ]
}
```


The Google Books API response structure

As you can see, it's relatively straightforward. We receive an object with an attribute *items*, which is an array of objects, one for each found book.

Each item object has several attributes - for now, we are mostly interested in `volumeInfo.title`, `volumeInfo.subtitle`, and `volumeInfo.imageLinks.smallThumbnail`.

Of course we don't want to log the results, we want to display a list of books on the screen.

The best way to present the results is by using a *ListView* element. A ListView is the scrollable table of items we all know from iOS apps like *Music* or *Mails*.

Component state

With the introduction of a ListView, we also need to introduce *state* into our application. While *props* are used to initialize components, allowing us to pass data from one component to another (e.g. from the search text input field to the search results screen), we need state in order to manage data that exists *and changes* during the lifetime of a component.

In our case, the list of books we retrieve from the Google Books API makes up the state of the ListView that is used to present that list to the user. When we switch to the results screen, this list is empty, and it is filled with the results from the API call once that call is finished.

One other piece of state we need to manage in the results screen code is the information about the status of the API fetch operation itself - in order to create a good user experience, we should display a *Loading, please wait...* message while results are retrieved, because on mobile devices in mobile networks, the operation might take some time. The underlying state - are we currently loading, or are we done loading? - needs to be managed. Depending on the loading status, we either render the loading message, or we present the results.

Let's start with that.

State can be *initialized*, it can be *changed*, and it can be *read*. All of this is done in the following code:

```
'use strict';

var React = require('react-native');
var {
  View,
  Text,
  StyleSheet,
  } = React;

var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};

var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
      };
    },

    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
```

```
        .then(response => response.json())
        .then(jsonData => {
          this.setState({ isLoading: false });
          console.dir(jsonData);
        })
        .catch(error => console.dir(error));
    },

    render: function() {
      if (this.state.isLoading) {
        return this.renderLoadingMessage();
      } else {
        return this.renderResults();
      }
    },

    renderLoadingMessage: function() {
      return (
        <View style={styles.container}>
          <Text style={styles.label}>
            Searching for "{this.props.searchPhrase}".
          </Text>
          <Text style={styles.label}>
            Please wait...
          </Text>
        </View>
      );
    },

    renderResults: function() {
      return (
        <View style={styles.container}>
          <Text style={styles.label}>
            Finished searching.
          </Text>
        </View>
      );
    },

});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
  },
  label: {
    fontSize: 24,
    fontWeight: 'normal',
    color: '#fff',
  },
});

```

```
85  module.exports = ResultsScreen;
```

On line 17, we add the function *getInitialState*. This is where we can set starting values for all state variables of the ResultsScreen component. In this case, we create a state variable named *isLoading*, which defaults to *true* because when the screen appears, we have yet to load the books matching the users search.

On line 31, in the anonymous function that is triggered when the *fetch* operation is done, we change the value of the state variable to *false*.

Now we can react to the value of our state variable - we want to render different contents depending on the value of *isLoading*.

To do so, two new dedicated rendering methods are created: on line 45, *renderLoadingMessage* has the JSX that displays a waiting message to the user. On line 58, *renderResults* has been created - it's still just a placeholder for the actual results list which we will build in a moment.

Our actual *render* method on line 37 now simply refers to one of these depending on the value of *this.state.isLoading*, our state variable. As a result, when initiating a search, we first see the *Please wait...* message, which is replaced with the placeholder results view when loading has been finished.

> The *Please wait...* message will disappear really quickly if you have a very fast internet connection. In this case, you can fake a longer load time by utilizing a timeout:
>
> ```
> fetchResults: function(searchPhrase) {
>   fetch(buildUrl(searchPhrase))
>     .then(response => response.json())
>     .then(jsonData => {
>       setTimeout(() => {
>         this.setState({isLoading: false});
>       }, 2000);
>       console.dir(jsonData);
>     })
>     .catch(error => console.dir(error));
> },
> ```

The important takeaway here is that these state variables are not just ordinary variables. They come with underlying mechanisms - our application *reacts* to changes in their value!

This becomes apparent when we try to achieve the same functionality with a plain old variable:

```
'use strict';

var React = require('react-native');
var {
  View,
  Text,
  StyleSheet,
  } = React;

var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};

var isLoading = true;

var ResultsScreen = React.createClass({
    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
        .then(response => response.json())
        .then(jsonData => {
          isLoading = false;
          console.dir(jsonData);
        })
        .catch(error => console.dir(error));
    },

    render: function() {
      if (isLoading) {
        return this.renderLoadingMessage();
      } else {
        return this.renderResults();
      }
    },

    renderLoadingMessage: function() {
      return (
        <View style={styles.container}>
          <Text style={styles.label}>
            Searching for "{this.props.searchPhrase}".
          </Text>
          <Text style={styles.label}>
            Please wait...
          </Text>
        </View>
      );
    },

    renderResults: function() {
      return (
        <View style={styles.container}>
```

```
57            <Text style={styles.label}>
58              Finished searching.
59            </Text>
60          </View>
61        );
62      },
63
64  });
65
66  var styles = StyleSheet.create({
67    container: {
68      flex: 1,
69      flexDirection: 'column',
70      justifyContent: 'center',
71      alignItems: 'center',
72      backgroundColor: '#5AC8FA',
73    },
74    label: {
75      fontSize: 24,
76      fontWeight: 'normal',
77      color: '#fff',
78    },
79  });
80
81  module.exports = ResultsScreen;
```

On line 16, we define the variable, on line 27, we change its value, and on line 34, we read it.

But if you run this version of the code, the results screen will never switch to the "Finished searching" view. React simply doesn't have an eye on the *isLoading* variable, because it's not part of its state.

Showing results with a ListView

Ok, let's take the final step - instead of the placeholder message, we want to display a scrollable list with the results of our search. So let's tackle the ListView component.

The JSX part of the ListView component typically looks like this:

```
<ListView
  dataSource={this.state.dataSource}
  renderRow={this.renderBook}
  style={styles.listView}
/>
```

With these three attributes, the ListView is able to do its job. Like other React Native UI elements, a ListView has a *style* attribute. *dataSource* references a variable that holds all items of the list, one for each row. *renderRow* references a rendering function that generates the view for one row of the list - it is called for each row that is currently visible on the screen, and the item from the dataSource list that belongs to that row is passed as a parameter:

```
    dataSource item #1

    dataSource item #2

    dataSource item #3
┌─────────────────────────┐
|   dataSource item #4    | -> renderRow(item #4)
|                         |
|   dataSource item #5    | -> renderRow(item #5)
|                         |
|   dataSource item #6    | -> renderRow(item #6)
|                         |
|   dataSource item #7    | -> renderRow(item #7)
|                         |
|   dataSource item #8    | -> renderRow(item #8)
└─────────────────────────┘
    dataSource item #9
```

In the above illustration, the ListView holds a dataSource list with 9 items, but only items 4-8 are currently within the borders of the device screen, and thus the renderRow function is called for these items only. This way even a list with several thousand items can be rendered to the screen efficiently.

The dataSource variable isn't a simple array - it's a specialized object that needs to be created by creating an instance of *ListView.DataSource*. As with our *isLoading* state variable, this needs to be done in the *getInitialState* method:

```
getInitialState: function() {
  return {
    isLoading: true,
    dataSource: new ListView.DataSource({
      rowHasChanged: (row1, row2) => row1 !== row2,
    }),
  };
},
```

As you can see, we pass a *rowHasChanged* method definition. The ListView calls this in order to find out if the content of a row has changed, and it only re-renders rows that did change, making the process more efficient.

In the *fetchResults* function, where we currently only log the results of the API call, we can now use the results to update the dataSource state variable:

```
fetchResults: function(searchPhrase) {
  fetch(buildUrl(searchPhrase))
    .then(response => response.json())
    .then(jsonData => {
      this.setState({
        isLoading: false,
        dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
      });
    })
    .catch(error => console.dir(error));
},
```

Again, *dataSource* isn't a simple array, and we therefore cannot simply assign it a new value - we need to do this via its *cloneWithRows* method.

Now that we have taken care of initializing and updating the ListView dataSource, we can replace the existing *renderResults* method with one that renders the ListView, and add the *renderBook* method that the ListView uses:

```
renderResults: function() {
  return (
    <ListView
      dataSource={this.state.dataSource}
      renderRow={this.renderBook}
      style={styles.listView}
      />
  );
},

renderBook: function(book) {
  return (
    <View style={styles.row}>
      <Text style={styles.title}>
        {book.volumeInfo.title}
      </Text>
      <Text style={styles.subtitle}>
        {book.volumeInfo.subtitle}
      </Text>
    </View>
  );
}
```

Last but not least, we should add some minimal styling for the new components:

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
  },
  label: {
    fontSize: 24,
    fontWeight: 'normal',
    color: '#fff',
  },
  listView: {
  },
  row: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
    paddingTop: 20,
    paddingBottom: 20,
    paddingLeft: 20,
    paddingRight: 20,
    marginTop: 1,
  },
  title: {
    fontSize: 20,
    fontWeight: 'bold',
    color: '#fff',
  },
  subtitle: {
    fontSize: 16,
    fontWeight: 'normal',
    color: '#fff',
  }
});
```

With all of these changes in place, the *ResultsScreen.js* file now looks like this:

```
'use strict';

var React = require('react-native');
var {
  View,
  ListView,
  Text,
  StyleSheet,
  } = React;

var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
```

```
};

var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
        dataSource: new ListView.DataSource({
          rowHasChanged: (row1, row2) => row1 !== row2,
        }),
      };
    },

    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
        .then(response => response.json())
        .then(jsonData => {
          this.setState({
            isLoading: false,
            dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
          });
          console.dir(jsonData.items);
        })
        .catch(error => console.dir(error));
    },

    render: function() {
      if (this.state.isLoading) {
        return this.renderLoadingMessage();
      } else {
        return this.renderResults();
      }
    },

    renderLoadingMessage: function() {
      return (
        <View style={styles.container}>
          <Text style={styles.label}>
            Searching for "{this.props.searchPhrase}".
          </Text>
          <Text style={styles.label}>
            Please wait...
          </Text>
        </View>
      );
    },

    renderResults: function() {
      return (
        <ListView
          dataSource={this.state.dataSource}
          renderRow={this.renderBook}
          style={styles.listView}
```

```
71            />
72        );
73      },
74
75      renderBook: function(book) {
76        return (
77          <View style={styles.row}>
78            <Text style={styles.title}>
79              {book.volumeInfo.title}
80            </Text>
81            <Text style={styles.subtitle}>
82              {book.volumeInfo.subtitle}
83            </Text>
84          </View>
85        );
86      }
87
88  });
89
90  var styles = StyleSheet.create({
91    container: {
92      flex: 1,
93      flexDirection: 'column',
94      justifyContent: 'center',
95      alignItems: 'center',
96      backgroundColor: '#5AC8FA',
97    },
98    label: {
99      fontSize: 24,
100     fontWeight: 'normal',
101     color: '#fff',
102   },
103   listView: {
104   },
105   row: {
106     flex: 1,
107     flexDirection: 'column',
108     justifyContent: 'center',
109     alignItems: 'center',
110     backgroundColor: '#5AC8FA',
111     paddingTop: 20,
112     paddingBottom: 20,
113     paddingLeft: 20,
114     paddingRight: 20,
115     marginTop: 1,
116   },
117   title: {
118     fontSize: 20,
119     fontWeight: 'bold',
120     color: '#fff',
121   },
122   subtitle: {
123     fontSize: 16,
124     fontWeight: 'normal',
125     color: '#fff',
126   }
```

```
127 });
128
129 module.exports = ResultsScreen;
```

> Note line 6, which defines the ListView variable.

This marks an important step on our way to a fully working app. We can now search for books and get a list of matched titles as a result.

Displaying images

Let's try to improve the UI a bit by adding the image of the book cover next to each title and subtitle. This requires the following changes:

Below line 7, add the *Image* component variable:

```
var {
  View,
  ListView,
  Text,
  Image,
  StyleSheet,
  } = React;
```

In the *renderBook* method on line 76 and following, add an *<Image>* element that references the thumbnail URL from the API results, and put the title and subtitle text elements into a new View block that references the *rightContainer* style:

```
renderBook: function(book) {
  return (
    <View style={styles.row}>
      <Image
        style={styles.thumbnail}
        source={{uri: book.volumeInfo.imageLinks.smallThumbnail}}
        />
      <View style={styles.rightContainer}>
        <Text style={styles.title}>
          {book.volumeInfo.title}
        </Text>
        <Text style={styles.subtitle}>
          {book.volumeInfo.subtitle}
        </Text>
       </View>
    </View>
  );
}
```

On line 111 and following, insert a style definition for *rightContainer*, change the *flexDirection* of the *row* style from *column* to *row*, and remove the *paddingTop*, *-Bottom*, and *-Left* attributes:

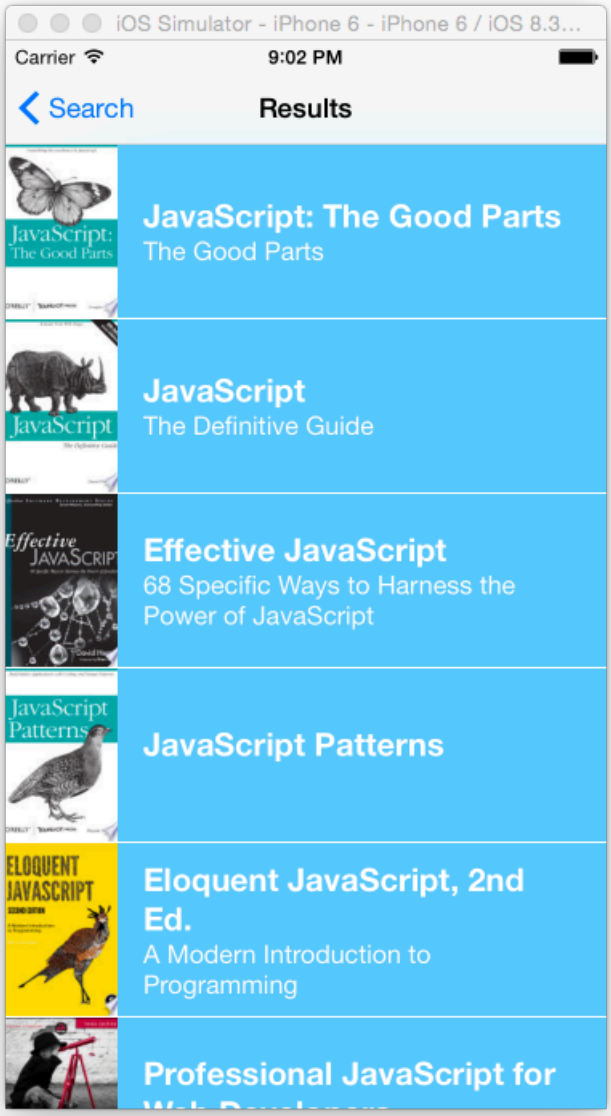
```
rightContainer: {
  flex: 1,
},
row: {
  flex: 1,
  flexDirection: 'row',
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: '#5AC8FA',
  paddingRight: 20,
  marginTop: 1,
},
```

> `flex: 1` is all it needs for *rightContainer* because the default values for *flexDirection*, *justifyContent* and *alignItems* are sufficient.

Below line 133, add the style definition for *thumbnail*:

```
thumbnail: {
  width: 70,
  height: 108,
  marginRight: 16,
}
```

With these changes in place, your results screen should look like this:



The final results screen

Addding a book details screen

With this, on to the third screen, where we display details for the book the user selected from the list.

Again, we will utilize the NavigatorIOS mechanics to bring the user from the search results to the book details screen. Because we already got all the information for each book from the API call, we can simply pass the book information from the results screen to the book details screen as a prop.

To do so, we first create another method on the ResultsScreen object called *showBookDetails*:

```
showBookDetails: function(book) {
  this.props.navigator.push({
    title: book.volumeInfo.title,
    component: BookDetails,
    passProps: {book}
  });
}
```

The concept put to use here is nothing new - we push a new screen on the navigator route hierarchy, and we pass the book object as a prop. The *BookDetails* component is not yet in place, we will create it in a moment.

The *showBookDetails* method needs to be called when the user taps on a row in the results list. In order to handle touch events in our app, we need the *TouchableHighlight* component. Using the *<TouchableHighlight>* element, we simply wrap the screen element that should be touchable, and connect it to our *showBookDetails* method. The most logical place to do this is the *renderBook* JSX definition:

```
renderBook: function(book) {
  return (
    <TouchableHighlight onPress={() => this.showBookDetails(book)}>
      <View style={styles.row}>
        <Image
          style={styles.thumbnail}
          source={{uri: book.volumeInfo.imageLinks.smallThumbnail}}
          />
        <View style={styles.rightContainer}>
          <Text style={styles.title}>
            {book.volumeInfo.title}
          </Text>
          <Text style={styles.subtitle}>
            {book.volumeInfo.subtitle}
          </Text>
        </View>
      </View>
    </TouchableHighlight>
  );
},
```

As you can see, we made the connection between the *<TouchableHighlight>* element and the *showBookDetails* method by defining an anonymous function call on the *onPress* attribute, and we pass the *book* object that was previously passed into the *renderBook* method. Last but not least, we need to define the *TouchableHighlight* component at the beginning of the file, and we need to require the yet-to-be-written code file that contains the *BookDetails* component - with this, the updated *ResultsScreen.js* file looks like this:

```
'use strict';

var React = require('react-native');
var {
  View,
  ListView,
  Text,
  Image,
  TouchableHighlight,
  StyleSheet,
  } = React;

var BookDetails = require('./BookDetails');

var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};

var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
        dataSource: new ListView.DataSource({
          rowHasChanged: (row1, row2) => row1 !== row2,
        }),
      };
    },

    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
        .then(response => response.json())
        .then(jsonData => {
          this.setState({
            isLoading: false,
            dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
          });
          console.dir(jsonData.items);
        })
        .catch(error => console.dir(error));
    },

    render: function() {
      if (this.state.isLoading) {
        return this.renderLoadingMessage();
      } else {
        return this.renderResults();
      }
    },

    renderLoadingMessage: function() {
```

```
57        return (
58          <View style={styles.container}>
59            <Text style={styles.label}>
60              Searching for "{this.props.searchPhrase}".
61            </Text>
62            <Text style={styles.label}>
63              Please wait...
64            </Text>
65          </View>
66        );
67      },
68
69      renderResults: function() {
70        return (
71          <ListView
72            dataSource={this.state.dataSource}
73            renderRow={this.renderBook}
74            style={styles.listView}
75            />
76        );
77      },
78
79      renderBook: function(book) {
80        return (
81          <TouchableHighlight onPress={() => this.showBookDetails(book)}>
82            <View style={styles.row}>
83              <Image
84                style={styles.thumbnail}
85                source={{uri: book.volumeInfo.imageLinks.smallThumbnail}}
86                />
87              <View style={styles.rightContainer}>
88                <Text style={styles.title}>
89                  {book.volumeInfo.title}
90                </Text>
91                <Text style={styles.subtitle}>
92                  {book.volumeInfo.subtitle}
93                </Text>
94               </View>
95            </View>
96          </TouchableHighlight>
97        );
98      },
99
100     showBookDetails: function(book) {
101       this.props.navigator.push({
102         title: book.volumeInfo.title,
103         component: BookDetails,
104         passProps: {book}
105       });
106     }
107
108 });
109
110 var styles = StyleSheet.create({
111   container: {
112     flex: 1,
```

```
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
  },
  label: {
    fontSize: 24,
    fontWeight: 'normal',
    color: '#fff',
  },
  listView: {
  },
  row: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#5AC8FA',
    paddingRight: 20,
    marginTop: 1,
  },
  rightContainer: {
    flex: 1,
  },
  title: {
    fontSize: 20,
    fontWeight: 'bold',
    color: '#fff',
  },
  subtitle: {
    fontSize: 16,
    fontWeight: 'normal',
    color: '#fff',
  },
  thumbnail: {
    width: 70,
    height: 108,
    marginRight: 16,
  }
});

module.exports = ResultsScreen;
```

The last missing piece is the *BookDetails* screen. Here is the full code, which is explained in detail afterwards:

```
'use strict';

var React = require('react-native');
var {
  View,
  ScrollView,
  Text,
  Image,
  StyleSheet,
  } = React;

var BookDetails = React.createClass({
    render: function() {
      console.log(this.props.book.volumeInfo.description);
      return (
        <ScrollView>
          <View style={styles.topContainer}>
            <Image
              style={styles.thumbnail}
              source={{uri: this.props.book.volumeInfo.imageLinks.smallThumbnail}}
              />
            <View style={styles.titlesContainer}>
              <Text style={styles.title}>
                {this.props.book.volumeInfo.title}
              </Text>
              <Text style={styles.subtitle}>
                {this.props.book.volumeInfo.subtitle}
              </Text>
            </View>
          </View>
          <View style={styles.middleContainer}>
            <Text style={styles.description}>
              {this.props.book.volumeInfo.description}
            </Text>
          </View>
          <View style={styles.bottomContainer}>
            <Text style={styles.author}>
              Author: {this.props.book.volumeInfo.authors[0]}
            </Text>
          </View>
        </ScrollView>
      );
    }
});

var Dimensions = require('Dimensions');
var windowSize = Dimensions.get('window');

var styles = StyleSheet.create({
  topContainer: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'flex-start',
    alignItems: 'flex-start',
    backgroundColor: '#5AC8FA',
  },
```

```
  thumbnail: {
    width: 70,
    height: 108,
    marginRight: 16,
  },
  titlesContainer: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'flex-start',
    alignItems: 'flex-start',
    backgroundColor: '#5AC8FA',
    width: windowSize.width - 86,
    paddingTop: 8,
    paddingRight: 8,
  },
  title: {
    fontSize: 20,
    fontWeight: 'bold',
    color: '#fff',
  },
  subtitle: {
    fontSize: 16,
    fontWeight: 'normal',
    color: '#fff',
  },
  middleContainer: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'flex-start',
    alignItems: 'flex-start',
    backgroundColor: '#fff',
    margin: 16,
  },
  description: {
    fontFamily: 'Times',
    fontSize: 16,
    fontWeight: 'normal',
    color: '#000',
    marginBottom: 8,
  },
  bottomContainer: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'flex-start',
    alignItems: 'flex-start',
    backgroundColor: '#5AC8FA',
    padding: 8,
  },
  author: {
    fontSize: 16,
    fontWeight: 'normal',
    color: '#fff',
  },
});

module.exports = BookDetails;
```

Because this screen is relatively simple in that it only renders a static UI with contents from a passed-in prop, the code shouldn't be too surprising by now.

However, two details need some explanation. First, we introduce yet another UI component, the *ScrollView*. Because the description text of a book can be longer than the available screen space, we need to wrap the whole UI into a special view that makes the UI scrollable - the ScrollView.

Second, within the UI, we display a slightly larger thumbail with the title and subtitle next to it on the right. In contrast to very same layout within our ListView rows on the results screen, React Native doesn't handle this as expected without our support. The problem is that titles and subtitles that are longer than the available horizontal screen space won't break to the next line automatically. You can demonstrate this if you delete line 68 and then, in the app, select a book with a long title.

In order to get the layout right, we need to calculate the available space for the View that wraps the title and subtitle text (70 points for the thumbnail plus 16 points for its margin), and assign the calculated space as the width of this view, via the *titleContainer* style definition.

To do so, we require the *Dimensions* component which can be used to retrieve the width and height of the UI screen (which of course varies between different iPhone and iPad versions).

> Investigation whether this is a bug in React Native or not is currently ongoing, and depending on the outcome, this part of the book might receive an update in the future.

Conclusion

And with this, we have created a fully operational mobile app with remote API communication, multiple screen navigation, touch handling, props and state, and much more.

But let's face it: There is still much work to be done - for now, our code very much depends on "happy path" scenarios where the user doesn't try out strange things like entering search terms that don't yield a result, where the network always works, the device is never rotated, and so on.

Thus, our next mission is to look into edge cases and make our app robust enough for actual use.

Improving the BookBrowser app

In order to make our app more robust, we need to look at the edge cases that might occur, and we need to handle those gracefully.

The first and most obvious case the user might run into is that no network connection is available, and therefore, the request to the Google Books API will fail.

Right now, this case has two effects: within the app itself, nothing happens - the user remains stuck on the "Please wait..." screen, without being notified about a problem (however, navigating back to the search screen is still possible), and in Chrome Debug mode, a message like the following is logged:

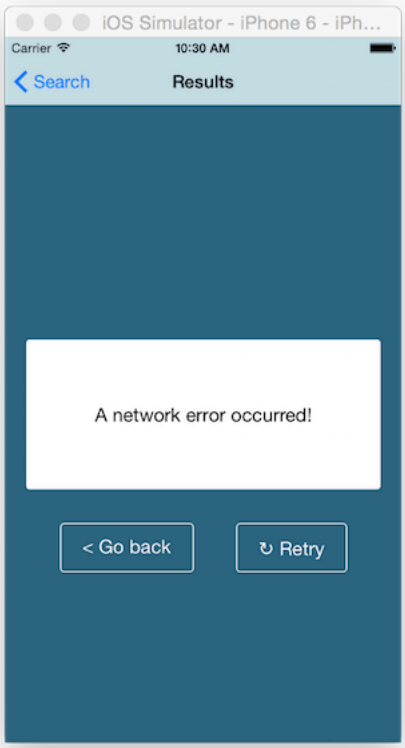
```
TypeError: Network request failed
  at xhr.onload (http://localhost:8081/index.ios.bundle:11755:18)
    at XMLHttpRequestBase.$XMLHttpRequestBase_sendLoad (http://localhost:8081/index.ios.bundle:11405:7)
      ...
```

Let's see how and where in the code we can handle that.

In file *ResultsScreen.js*, on line 45, the method *catch* is used to define a function that is called if an error occurs. Currently, we only use this to log the error. We need to integrate this error case into our application workflow.

How exactly do we want to inform the user about the fact that an error occured? We could simply add another text element to the screen, like this:

But that's not very sophisticated. An error scenario is a typical use case for a *modal*, that is, an overlay on the screen that shows the error message, like this:



A network error modal in the BookBrowser app

Also, deciding for a modal is a good excuse to learn how to add third-party React Native components written by others.

Adding a third-party modal component

Brent Vatne has released a modal component at (https://github.com/brentvatne/react-native-modal). Let's see how to integrate this into our application.

First, we need to declare his package as a dependency to our project. Because Brent released his code as an NPM package, this is simple. Just run

```
npm install react-native-modal --save
```

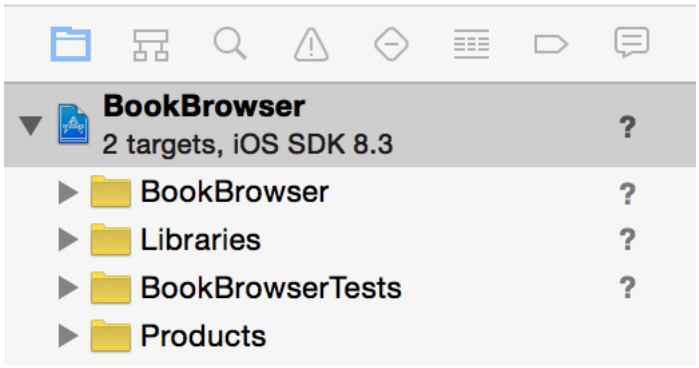
while in the root folder of the BookBrowser project.

In case you haven't worked with NPM before, note that the above *npm install* command pulls Brent's code (and the dependencies of *his* code) into the *node_modules* folder of our project. The *–save* switch updates our *package.json* file, which now declares two dependencies, *react-native* and *react-native-modal*:

```
{
  "name": "BookBrowser",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node_modules/react-native/packager/packager.sh"
  },
  "dependencies": {
    "react-native": "^0.8.0",
    "react-native-modal": "^0.3.8"
  }
}
```

His code is now available to the JavaScript side of our code, but we need to take some extra steps to make the parts of his code that are related to Xcode® known to our Xcode® setup:

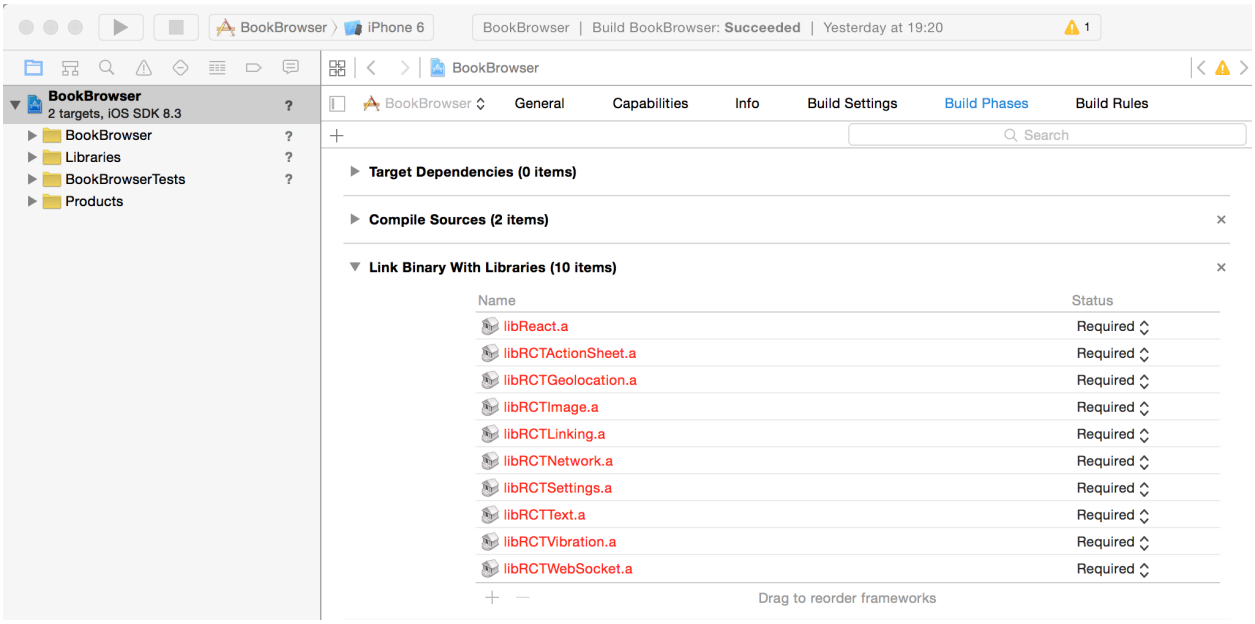
- Switch to Xcode®
- Hit ⌘-1 or select *View* ▸ *Navigators* ▸ *Show Project Navigator*
- In the leftmost pane of the main Xcode® window, fold out the contents below "BookBrowser* by clicking on the triangular arrow



The Project Navigator with subitems fold out

- In the list of folders that appears, right-click on *Libraries* and select *Add Files to "Book-Browser"...*
- In the file browser that opens, navigate to *node_modules/react-native-modal*, and select file *RNModal.xcodeproj*

- Now, in the leftmost pane, select the first entry (the one that reads *BookBrowser* and has a blue icon in front of it)
- In the center pane, switch from *General* to *Build Phases*
- In the area below, fold out *Link Binary With Libraries*, and click the plus sign below the file list



The Build Phases pane

- In the file browser that opens, select *libRNModal.a*

That's it - the *Modal* component is now integrated with the Xcode® part of our *BookBrowser* project, and can be used from within the JavaScript part of the app. Which is what we are doing next.

Using the new modal component

To get a basic modal setup working, we need to add four elements to our JavaScript code: We need to import the *react-native-modal* package into our code, a *<Modal>* JSX element is needed to wrap the view elements that should become visible when the modal is shown, a *showErrorModal* state parameter is needed which allows us to switch the modal between being visible and invisible, and the *fetch* operation needs to be extended in order to make the modal visible if and when a network error occurs.

All of this is done in file *ResultsScreen.js*, because there network operations are executed and it provides the view where we need to inform the user if an error occurs.

First, we import the modal package, at the beginning of the file:

```
var Modal = require('react-native-modal');
```

We then initialize the *showErrorModal* parameter as a state variable on the *ResultsScreen* component:

```
var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
        showErrorModal: false,
        dataSource: new ListView.DataSource({
          rowHasChanged: (row1, row2) => row1 !== row2,
        }),
      };
    },
```

We now extend the *catch* branch of our *fetch* block, switching the modal to visible if an error occurs:

```
fetch(buildUrl(searchPhrase))
  .then(response => response.json())
  .then(jsonData => {
    this.setState({
      isLoading: false,
      dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
    });
    console.dir(jsonData.items);
  })
  .catch(error => {
    this.setState({
      showErrorModal: true
    });
  });
```

Last but not least, we extend the view that is returned by *renderLoadingMessage* with the modal JSX element, which contains the error message:

```
renderLoadingMessage: function() {
  return (
    <View style={styles.container}>
      <Text style={styles.label}>
        Searching for "{this.props.searchPhrase}".
      </Text>
      <Text style={styles.label}>
        Please wait...
      </Text>
      <Modal isVisible={this.state.showErrorModal}>
        <Text>A network error occurred!</Text>
      </Modal>
    </View>
  );
},
```

With all these changes in place, the complete file looks like this:

```
'use strict';

var React = require('react-native');
var Modal = require('react-native-modal');

var {
  View,
  ListView,
  Text,
  Image,
  TouchableHighlight,
  StyleSheet,
  } = React;

var BookDetails = require('./BookDetails');

var buildUrl = function(q) {
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};

var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
        showErrorModal: false,
        dataSource: new ListView.DataSource({
          rowHasChanged: (row1, row2) => row1 !== row2,
        }),
      };
    },

    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
        .then(response => response.json())
        .then(jsonData => {
          this.setState({
            isLoading: false,
            dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
          });
          console.dir(jsonData.items);
        })
        .catch(error => {
          this.setState({
            showErrorModal: true
          });
        });
    },

    render: function() {
      if (this.state.isLoading) {
```

```
57          return this.renderLoadingMessage();
58        } else {
59          return this.renderResults();
60        }
61      },
62
63      renderLoadingMessage: function() {
64        return (
65          <View style={styles.container}>
66            <Text style={styles.label}>
67              Searching for "{this.props.searchPhrase}".
68            </Text>
69            <Text style={styles.label}>
70              Please wait...
71            </Text>
72            <Modal isVisible={this.state.showErrorModal}>
73              <Text>A network error occurred!</Text>
74            </Modal>
75          </View>
76        );
77      },
78
79      renderResults: function() {
80        return (
81          <ListView
82            dataSource={this.state.dataSource}
83            renderRow={this.renderBook}
84            style={styles.listView}
85            />
86        );
87      },
88
89      renderBook: function(book) {
90        return (
91          <TouchableHighlight onPress={() => this.showBookDetails(book)}>
92            <View style={styles.row}>
93              <Image
94                style={styles.thumbnail}
95                source={{uri: book.volumeInfo.imageLinks.smallThumbnail}}
96                />
97              <View style={styles.rightContainer}>
98                <Text style={styles.title}>
99                  {book.volumeInfo.title}
100               </Text>
101               <Text style={styles.subtitle}>
102                 {book.volumeInfo.subtitle}
103               </Text>
104             </View>
105           </View>
106         </TouchableHighlight>
107       );
108     },
109
110     showBookDetails: function(book) {
111       this.props.navigator.push({
112         title: book.volumeInfo.title,
```

```
113           component: BookDetails,
114           passProps: {book}
115         });
116       }
117
118   });
119
120   var styles = StyleSheet.create({
121     container: {
122       flex: 1,
123       flexDirection: 'column',
124       justifyContent: 'center',
125       alignItems: 'center',
126       backgroundColor: '#5AC8FA',
127     },
128     label: {
129       fontSize: 24,
130       fontWeight: 'normal',
131       color: '#fff',
132     },
133     listView: {
134     },
135     row: {
136       flex: 1,
137       flexDirection: 'row',
138       justifyContent: 'center',
139       alignItems: 'center',
140       backgroundColor: '#5AC8FA',
141       paddingRight: 20,
142       marginTop: 1,
143     },
144     rightContainer: {
145       flex: 1,
146     },
147     title: {
148       fontSize: 20,
149       fontWeight: 'bold',
150       color: '#fff',
151     },
152     subtitle: {
153       fontSize: 16,
154       fontWeight: 'normal',
155       color: '#fff',
156     },
157     thumbnail: {
158       width: 70,
159       height: 108,
160       marginRight: 16,
161     }
162   });
163
164   module.exports = ResultsScreen;
```

> If you want to test these changes, you need to force the application to run into the *catch* branch of the *fetch* operation. One way to do this is to disconnect the computer that runs the application from the network. Another solution is to make the URL that is requested invalid. It is defined on line 18 of file *ResultsScreen.js*. Simply change *googleapis.com* to *googleapis.coom*, for example.

Refining the modal component

Ok, this works, but doesn't look very nice yet. Also, we can't close the modal. The most sensible solution would be to offer the user the choice between retrying the search or going back one screen.

By default, react-native-modal only offers a simple *Close* button, but it also allows to pass a freely definable React component to pass as a custom button. This is what we will do.

What does "React component" mean in this context? A React component is what we build using JSX. Thus, `<Text>A network error occurred!</Text>` is such a component. And components can be much more complex of course - for example a set of views with two buttons, like the one we need to add to the modal.

In order to provide this component to the modal, we first create a new method which returns the component:

```
renderModalButtons: function () {
   return (
     <View style={styles.modalButtonsContainer}>

       <View style={styles.modalButton}>
         <Text style={styles.modalButtonText}>&lt; Go back</Text>
       </View>

       <View style={styles.modalButton}>
         <Text style={styles.modalButtonText}>&#8635; Retry</Text>
       </View>

     </View>
   );
 },
```

Ok, that's relatively straight-forward - we return a *<View>* with two subviews for each of the two buttons, which contain the text for the respective button.

Here is the styling for this component:

```
modalButtonsContainer: {
  flex: 1,
  flexDirection: 'row',
  justifyContent: 'center',
  top: 240
},
modalButton: {
  borderColor: '#ffffff',
  borderRadius: 4,
  borderWidth: 1,
  marginLeft: 20,
  marginRight: 20,
  paddingLeft: 20,
  paddingRight: 20,
  paddingTop: 10,
  paddingBottom: 10,
},
modalButtonText: {
  fontSize: 18,
  color: '#ffffff',
}
```

This makes for a nice look, but we also need behaviour. When the user taps on the *Go back* button, we should navigate her back to the previous screen, where a search keyword can be entered.

When the *Retry* button is tapped, we should start the current search anew - maybe the network connectivity of the user's device has returned because her train left the tunnel.

Let's first write the functions that perform these operations:

```
goBack: function() {
  this.setState({
    showErrorModal: false
  });
  this.props.navigator.pop();
},

retry: function() {
  this.setState({
    showErrorModal: false
  });
  this.fetchResults();
},
```

retry is really simple - after hiding the modal, the *fetch* operation is simply triggered again.

goBack is a bit more interesting. Remember how at the top level, our UI is wrapped in a *NavigatorIOS* component, and how a reference to this component is passed as prop *navigator* to each sub-screen. When adding and navigating to a sub-screen, it is pushed onto the internal stack of screens managed by *NavigatorIOS* - thus, if we want to go back one screen, we simply *pop* from this stack.

How can we map the taps on the buttons to these functions? We already know the *TouchableHighlight* element - we use it to trigger behaviour when the user taps on a row in the list of results. This

could be used here, and it would work, but the "highlight" effect looks pretty ugly, rendering a black box beneath the button that was tapped.

But luckily, *TouchableHighlight* has a sibling called *TouchableOpacity*. From the official documentation:

> *On press down, the opacity of the wrapped view is decreased, dimming it.*

Sounds good. Of course, *TouchableOpacity* also provides the *onPress* property we need to connect our buttons with their methods:

```
renderModalButtons: function () {
  return (
    <View style={styles.modalButtonsContainer}>

      <TouchableOpacity onPress={this.goBack}>
        <View style={styles.modalButton}>
          <Text style={styles.modalButtonText}>&lt; Go back</Text>
        </View>
      </TouchableOpacity>

      <TouchableOpacity onPress={this.retry}>
        <View style={styles.modalButton}>
          <Text style={styles.modalButtonText}>&#8635; Retry</Text>
        </View>
      </TouchableOpacity>

    </View>
  );
},
```

This results in the following *ResultsScreen.js* file (note that I've also styled the modal body a bit):

```
1  'use strict';
2
3  var React = require('react-native');
4  var Modal = require('react-native-modal');
5
6  var {
7    View,
8    ListView,
9    Text,
10   Image,
11   TouchableHighlight,
12   TouchableOpacity,
13   StyleSheet,
14   } = React;
15
16 var BookDetails = require('./BookDetails');
17
18 var buildUrl = function(q) {
```

```
  return 'https://www.googleapis.com/books/v1/volumes?q='
    + encodeURIComponent(q)
    + '&langRestrict=en&maxResults=40';
};

var ResultsScreen = React.createClass({
    getInitialState: function() {
      return {
        isLoading: true,
        showErrorModal: false,
        dataSource: new ListView.DataSource({
          rowHasChanged: (row1, row2) => row1 !== row2,
        }),
      };
    },

    componentDidMount: function() {
      this.fetchResults(this.props.searchPhrase);
    },

    fetchResults: function(searchPhrase) {
      fetch(buildUrl(searchPhrase))
        .then(response => response.json())
        .then(jsonData => {
          this.setState({
            isLoading: false,
            dataSource: this.state.dataSource.cloneWithRows(jsonData.items)
          });
          console.dir(jsonData.items);
        })
        .catch(error => {
          this.setState({
            showErrorModal: true
          });
        });
    },

    goBack: function() {
      this.setState({
        showErrorModal: false
      });
      this.props.navigator.pop();
    },

    retry: function() {
      this.setState({
        showErrorModal: false
      });
      this.fetchResults();
    },

    render: function() {
      if (this.state.isLoading) {
        return this.renderLoadingMessage();
      } else {
        return this.renderResults();
```

```
75        }
76      },
77
78      renderModalButtons: function () {
79        return (
80          <View style={styles.modalButtonsContainer}>
81
82            <TouchableOpacity onPress={this.goBack}>
83              <View style={styles.modalButton}>
84                <Text style={styles.modalButtonText}>&lt; Go back</Text>
85              </View>
86            </TouchableOpacity>
87
88            <TouchableOpacity onPress={this.retry}>
89              <View style={styles.modalButton}>
90                <Text style={styles.modalButtonText}>&#8635; Retry</Text>
91              </View>
92            </TouchableOpacity>
93
94          </View>
95        );
96      },
97
98      renderLoadingMessage: function() {
99        return (
100         <View style={styles.container}>
101           <Text style={styles.label}>
102             Searching for "{this.props.searchPhrase}".
103           </Text>
104           <Text style={styles.label}>
105             Please wait...
106           </Text>
107           <Modal isVisible={this.state.showErrorModal} customCloseButton={this.renderModalButtons()}>
108             <View style={styles.modalContainer}>
109               <Text style={styles.modalBody}>A network error occurred!</Text>
110             </View>
111           </Modal>
112         </View>
113       );
114     },
115
116     renderResults: function() {
117       return (
118         <ListView
119           dataSource={this.state.dataSource}
120           renderRow={this.renderBook}
121           style={styles.listView}
122           />
123       );
124     },
125
126     renderBook: function(book) {
127       return (
128         <TouchableHighlight onPress={() => this.showBookDetails(book)}>
129           <View style={styles.row}>
130             <Image
```

```
131                 style={styles.thumbnail}
132                 source={{uri: book.volumeInfo.imageLinks.smallThumbnail}}
133                 />
134               <View style={styles.rightContainer}>
135                 <Text style={styles.title}>
136                   {book.volumeInfo.title}
137                 </Text>
138                 <Text style={styles.subtitle}>
139                   {book.volumeInfo.subtitle}
140                 </Text>
141               </View>
142             </View>
143           </TouchableHighlight>
144         );
145       },
146
147       showBookDetails: function(book) {
148         this.props.navigator.push({
149           title: book.volumeInfo.title,
150           component: BookDetails,
151           passProps: {book}
152         });
153       }
154
155   });
156
157   var styles = StyleSheet.create({
158     container: {
159       flex: 1,
160       flexDirection: 'column',
161       justifyContent: 'center',
162       alignItems: 'center',
163       backgroundColor: '#5AC8FA',
164     },
165     label: {
166       fontSize: 24,
167       fontWeight: 'normal',
168       color: '#fff',
169     },
170     listView: {
171     },
172     row: {
173       flex: 1,
174       flexDirection: 'row',
175       justifyContent: 'center',
176       alignItems: 'center',
177       backgroundColor: '#5AC8FA',
178       paddingRight: 20,
179       marginTop: 1,
180     },
181     rightContainer: {
182       flex: 1,
183     },
184     title: {
185       fontSize: 20,
186       fontWeight: 'bold',
```

```
    color: '#fff',
  },
  subtitle: {
    fontSize: 16,
    fontWeight: 'normal',
    color: '#fff',
  },
  thumbnail: {
    width: 70,
    height: 108,
    marginRight: 16,
  },
  modalContainer: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    marginTop: 40,
    marginBottom: 40,
  },
  modalBody: {
    fontSize: 18,
  },
  modalButtonsContainer: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    top: 240
  },
  modalButton: {
    borderColor: '#ffffff',
    borderRadius: 4,
    borderWidth: 1,
    marginLeft: 20,
    marginRight: 20,
    paddingLeft: 20,
    paddingRight: 20,
    paddingTop: 10,
    paddingBottom: 10,
  },
  modalButtonText: {
    fontSize: 18,
    color: '#ffffff',
  }
});

module.exports = ResultsScreen;
```

And with this, we have a working and useful modal.

> Note how even though the modal "covers" the whole UI with its transparent dark background, the *< Search* component on the navigator is still visible and touchable, resulting in the same behaviour as clicking the *< Go back* button.

If you don't want this kind of duplication, pass the `forceToFront={true}` prop on the *<Modal>* component, like this:

```
<Modal isVisible={this.state.showErrorModal} forceToFront={true} customCloseButton={this.renderModalButtons()}>
```