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 2017-02-07


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# 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 35% done. It was last updated on June 15, 2015. The code in this book has been tested to work with React Native v0.5.0.

> 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®, Node.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
- Node.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 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 Node.js

*Node.js* is a server-side JavaScript runtime environment. React Native ships with some helper tools that are written for Node.js. Also, we use the Node.js package manager, *NPM*, to install the React Native command line tool itself.

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

```
brew install node
```

# Installing Watchman

React Native uses *Watchman* to 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 Node.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!
```

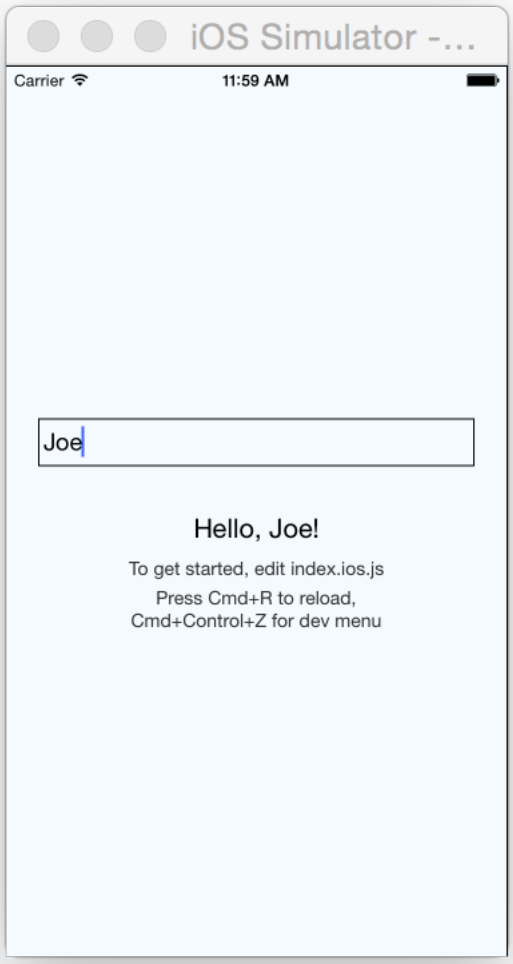
Then switch back to the iOS Simulator window and hit **⌘-R**. The UI of our app will refresh and display the new text.

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 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*.

## 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 Node.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.