

THE EXPERT'S VOICE® IN JAVASCRIPT

React Native for iOS Development

Harness the power of React and JavaScript
to build stunning iOS applications

Akshat Paul
Abhishek Nalwaya

Apress®

React Native for iOS Development



Akshat Paul

Abhishek Nalwaya

Apress®

React Native for iOS Development

Copyright © 2016 by Akshat Paul and Abhishek Nalwaya

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-1396-4

ISBN-13 (electronic): 978-1-4842-1395-7

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Louise Corrigan

Development Editor: James Markham

Technical Reviewer: Bruce Wade

Editorial Board: Steve Anglin, Pramila Balen, Louise Corrigan, Jim DeWolf, Jonathan Gennick,

Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott,

Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing

Coordinating Editor: Melissa Maldonado

Copy Editor: Mary Behr and April Rondeau

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Contents at a Glance

About the Authors..... ix

About the Technical Reviewer xi

Acknowledgments xiii

Introduction xv

■ Chapter 1: Learning the Basics: A Whistle-Stop Tour of React..... 1

■ Chapter 2: The Simplest Program: Hello World with React Native 19

■ Chapter 3: Canvas, Brush, and Paint: Working with the User Interface 41

■ Chapter 4: Flux: Solving Problems Differently..... 75

■ Chapter 5: Device Capabilities..... 95

■ Chapter 6: Communicating with Servers..... 135

■ Chapter 7: React Native Supplements 149

Index..... 169

Contents

About the Authors..... ix

About the Technical Reviewer xi

Acknowledgments xiii

Introduction xv

■ Chapter 1: Learning the Basics: A Whistle-Stop Tour of React..... 1

 Why React? 2

 Virtual DOM 2

 One-Way Data Flow 4

 Installation and Setup 6

 Introduction to Components..... 7

 JSX 8

 Deep-Dive Into Components..... 12

 Properties 12

 State..... 14

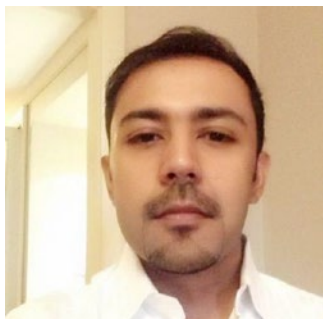
 Summary..... 17

■ Chapter 2: The Simplest Program: Hello World with React Native	19
What Is React Native?	20
React Native Prerequisites	20
Installation.....	20
Installing Node and npm.....	21
Installing Watchman	21
Installing the React Native Package	21
Updating React Native	21
Your First App	22
Creating a Basic Skelton	22
It's Not a UIWebView.....	28
Enabling Live Reload	29
Why Is React Native Different?.....	29
The Anatomy of a React Native Application.....	30
Debugging	32
Reload.....	33
Debugging in Chrome	34
Debugging in Safari	35
Showing the FPS Monitor	35
The Inspect Element.....	36
Starting Profiling.....	38
Summary	39
■ Chapter 3: Canvas, Brush, and Paint: Working with the User Interface	41
NavigatorIOS.....	42
Flexbox	45
Flex-direction.....	48
Flex	50
Adding Images	54
TouchableHighlight.....	58
Routing to a Component.....	61

ListView	67
ScrollView	73
Summary	74
■ Chapter 4: Flux: Solving Problems Differently	75
MVC Pattern	76
MVC Problem	76
Flux.....	77
Success of Flux	79
Flux Deep Dive	79
The Dispatcher	79
The Need for Dispatcher [dispatch() and waitFor()]	79
Stores	80
Actions.....	80
Flux with ReactJS Example	81
Flux with React Native Example	87
Summary	93
■ Chapter 5: Device Capabilities	95
GeoLocation.....	95
Reviewing the GeoLocationMap Code	97
Adding Annotation on Map.....	98
Displaying the Latitude and Longitude of the Present Location	100
AsyncStorage	108
Reviewing the AsyncStorage Code	111
Native Alert.....	114
Reviewing the NativeAlert Code	117
Extending the NativeAlert Example	119
Reviewing the Extended NativeAlert Example Code.....	123

WebView.....	124
Reviewing the WebView Code	125
Animations	126
Reviewing the Animation Code.....	132
Summary	134
■ Chapter 6: Communicating with Servers.....	135
XMLHttpRequest.....	135
WebSocket	136
Fetch	137
Getting Data from a Server.....	139
Saving Data to a Server.....	143
Summary.....	147
■ Chapter 7: React Native Supplements	149
Reflux	149
Differences from Flux	150
Redux	160
Debug on Device	161
Popular Modules for React Native	166
react-native-fbsdk.....	166
react-native-scrollable-tab-view.....	166
react-native-webpack-server.....	166
react-native-side-menu.....	166
Where to Go from Here.....	167
Index.....	169

About the Authors



Akshat Paul is a developer and author of the book “RubyMotion iOS Development Essentials”. He has extensive experience of mobile and web development and has delivered many enterprise and consumer applications over the years.

In other avatars, Akshat frequently speaks at conferences and meetups on various technologies. He has given talks at RubyConf India, RubyMotion #Inspect conference in Brussels and was keynote speaker at technology leadership events in Bangkok. Besides writing code, Akshat spends time with his family, is an avid reader and obsessive about healthy eating.



Abhishek Nalwaya is an author of the book, “RubyMotion iOS Development Essentials”. He is a Ruby enthusiast and loves to participate regularly at Ruby and Ruby on Rails meetup groups. He works for a management consulting firm. He has spoken at many conferences, meetups like RubyConf India and the RubyMotion #inspect conference. Besides programming, Abhishek loves to run a few miles and cook healthy food.

About the Technical Reviewer



Bruce Wade is the founder of Warply Designed Inc. (www.warplydesigned.com), a company specializing in using game technology for real-world applications. He has more than 16 years of software development experience with a strong focus on 2D/3D animation and interactive applications, primarily using Apple technology.



Acknowledgments

We would like to thank our families and friends, who saw us through this book, provided support, talked things over, read, wrote, and offered comments, without which conceiving this book wouldn't have been possible.

Also, we would like to thank Melissa, Louise, James, the entire team at Apress, and especially Jeffrey Pepper who allowed us to quote their remarks and assisted in the editing, proofreading, and design of this book. Writing a book is a long and arduous journey, but you all made it so easy for us.

Introduction

As the title suggests, this is an introduction to React Native. React has become a leading technology for building fast, responsive native iOS apps.

You will begin by understanding the path breaking concepts of React, which makes it distinctive. React is a JavaScript library for creating user interfaces and yet is much more. You will set up React Native and begin exploring the anatomy of React Native apps. You will learn about the React “way of thinking” and “write once and read everywhere” and how that works. You will learn why the DOM can impede the speed of any usage heavy or large app and how React offers an incredible and simple solution in Virtual DOM to speed your applications. You’ll also learn about flux architecture, how it differs from MVC, and how you can include it in your React Native project to solve problems differently and efficiently. You will learn to create stunning user interfaces and interact with the various device capabilities. You will then boost your development capabilities by including some popular packages already developed by the React Native community that will helps you write less but do more. Finally, you’ll learn to how write test cases and submit your application to App Store. Using real-world examples with an example-driven approach, you will learn by doing and have a running app at the end of each chapter.

In all, we hope that you will find this a useful and quick read that will get you started with creating responsive and efficient apps.

Programming Code

The programming code for the examples in this book are located in a zip file that may be updated from time to time. This file may be found in the Source Code/Errata tab on the book’s page at [Apress.com/9781484213964](https://www.apress.com/9781484213964).

Chapter 1

Learning the Basics: A Whistle-Stop Tour of React

“The journey of a thousand miles begins with one step.”

—Lao Tzu

Before you embark on your React Native journey, you must know a little bit about React(also known as ReactJS or React.js). In this chapter, you will quickly look at the core concepts of React, which will help you to work on React Native. React is different from most popular web technologies, and you will learn why as you move forward in this chapter. Its core concepts will really open up your mind to a new way of thinking if you have spent a considerable amount of time with traditional frameworks; this new way of thinking is sometimes called *the React way of thinking*. You may have heard the phrase “*write once, run everywhere*” but dismissed it as nearly impossible due to the proliferation of form factors (web, mobile, tablets). React has a different guiding principle: “*learn once, write anywhere.*” Wow, that seems quite different, and liberating. So you’ll begin this first chapter with a quick tour of React, which will help you get ready for React Native. If you have an elementary knowledge of React, you may skip this chapter and move on to Chapter [2](#).

React is a JavaScript library for creating user interfaces. It was built in a combined effort by Facebook and Instagram teams. React was first introduced to the world in 2013, and has taken it by storm, with community-wide acceptance and the benefit of being the technology at the heart of Facebook. According to official documentation, some consider React Native as the V in MVC because React Native makes no assumptions about rest of the technology stack used. You may use whatever technology you wish and you can create a single section of your app with React Native; you may also conveniently make changes in an already created application.

Why React?

But do we need another JavaScript library in a world full of js libraries and frameworks? There is hardly a month that goes by without a new js framework introduced.

React came into existence because its creators were faced with a significant problem: how to build large applications where data changes frequently. This problem occurs in almost any real-world application and React was created from the ground up to solve it. As you know, many popular frameworks are MVC or MV* but here's a point to be noted and re-iterated: React is not an MV* framework. It's just a library for building composable user interfaces for UI components whose data changes over time. Unlike popular js frameworks, React does not use templates or HTML directives. React builds user interfaces by breaking the UI into many components. That's it—nothing else. This means that React uses the full features of programming languages to build and render views.

The following are some of the advantages of choosing React for your next project:

- *React uses JavaScript extensively:* Traditionally the views in HTML are separated from the functionality in JavaScript. With React, components are created and there is one monolithic section where JavaScript has intimate knowledge of your HTML.
- *Extendable and maintainable:* Components are formed by a unified markup with its view logic, which actually makes the UI easy to extend and maintain.
- *Virtual DOM:* React applications are blazing fast. The credit for this goes to the virtual DOM and its diffing algorithm.
- *One-way data flow:* Two-way data binding is a great idea, but in real-world applications it produces more pain than benefits. One of the common drawbacks with two-way data binding is that you have no idea how your data gets updated. With one-way data flow, things are simple: you know exactly where data is mutating, which makes it easier to maintain and test your app.

In order to have a strong foundation of a new technology, it's necessary to understand its core concepts. In the next section, you will explore a few unique concepts of React, which will take you one step closer to understanding this amazing technology.

Virtual DOM

In all web applications one of the most expensive operations from which an app suffers is mutating the DOM. To solve this problem, React maintains a virtual representation of the DOM (as shown in Figure 1-1), which is called Virtual DOM or VDOM. Along with a diffing algorithm, React Native is able to compute the delta against the actual DOM and only update the part of the DOM that is changed. The amount of change is therefore less, which leads to a blazing fast application. In the beginning of your application you might not see it, but as your project balloons to insane complexity (which usually happens in real-world apps) you will begin to see the benefits of a snappy experience for users.

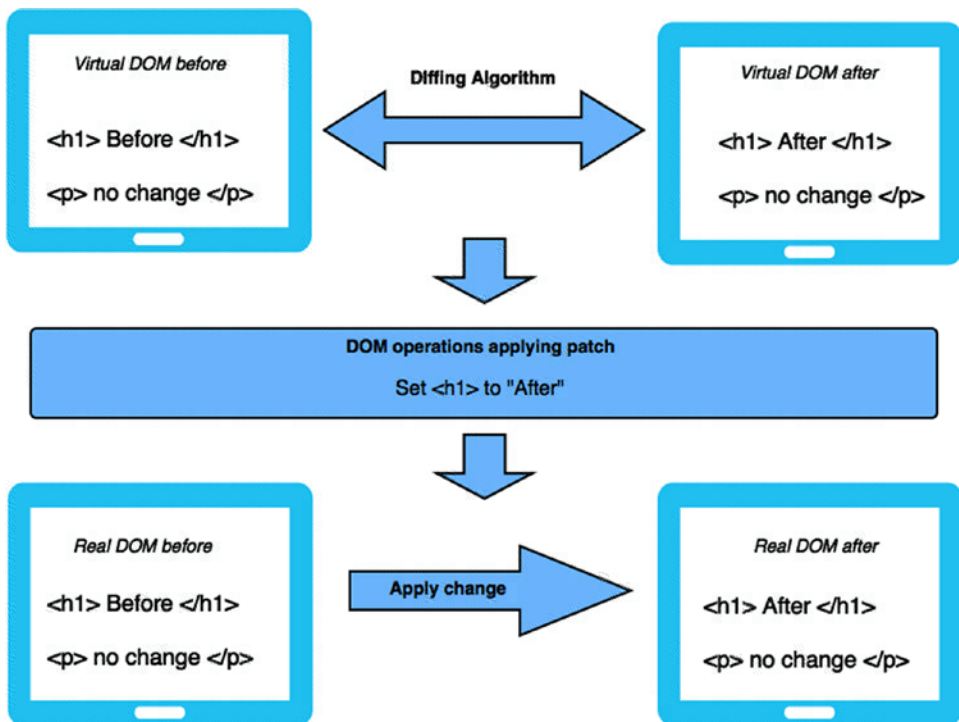


Figure 1-1. Virtual DOM and diffing algorithm operations

Manual DOM manipulation is messy, and keeping track of the previous state of the DOM is very hard. As shown in Figure 1-1, React solves this problem by keeping two copies of a virtual DOM. Next, a diffing algorithm is applied on these two virtual DOMs, which essentially checks for the changes that occurred and returns a stream of DOM operations. These DOM operations are then applied to the actual browser DOM.

Let's now understand in terms of components how a virtual DOM works. In React, every component has a state; this state is likely observable. Whenever there is a change in state, React essentially knows that this change requires a re-render. So when the application state changes, it generates a new VTree; once again the diff algorithm shared the DOM paths for required changes, as shown in Figure 1-2. This results in keeping manual DOM manipulation to the minimum.

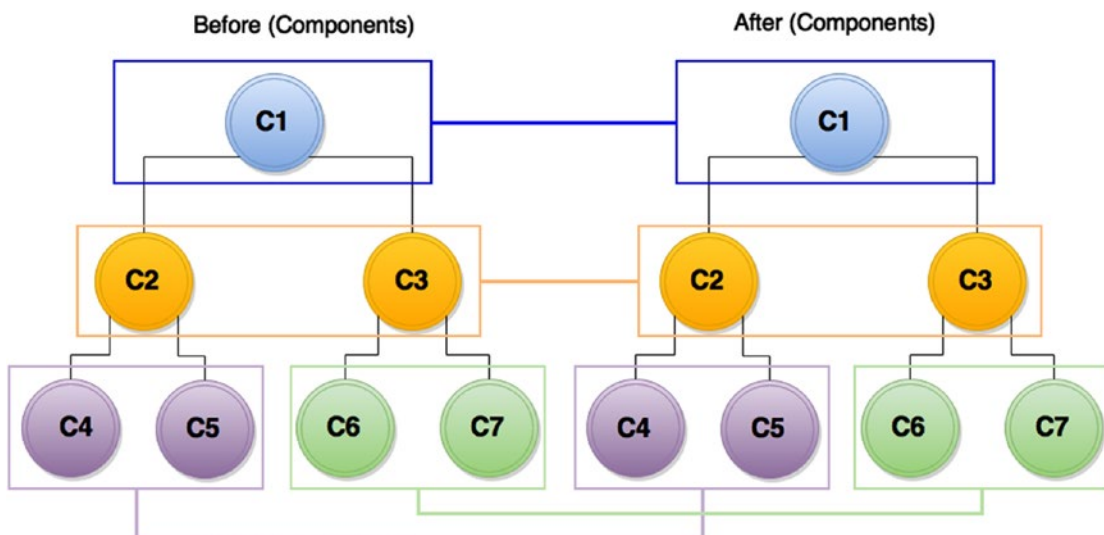


Figure 1-2. Components with virtual DOM

This feature of virtual DOM is not just important but a killer feature of React. DOM access is super slow, and humbly speaking, the world has made it worse by hitting the DOM again and again in most of the applications. To make your application fast, you should touch the DOM as little as possible, and this is beautifully handled by the implementation of virtual DOM. You won't notice this with a small and trivial application, but once your app grows to having thousands of DOM elements all trying to get updated, React will not let your performance feel the impact.

One-Way Data Flow

React is primarily the V in a MVC pattern but before you dive into the idea of one-way data flow in React, you must understand the challenges of MVC frameworks. One of the biggest challenges of a MVC framework is managing the view. As you know, the view component of the MVC framework is mainly the DOM representation. It is simple when you write code that interacts with the DOM, but it is very complicated for the framework to handle various DOM manipulations.

Traditional MVC views generally encompass a lot of heavy UI, and as the data changes even for a tiny element, it eventually re-renders the app again, and the cycle continues. The reason for this is that typically most of these MVC frameworks follow two-way data binding (see Figure 1-3).

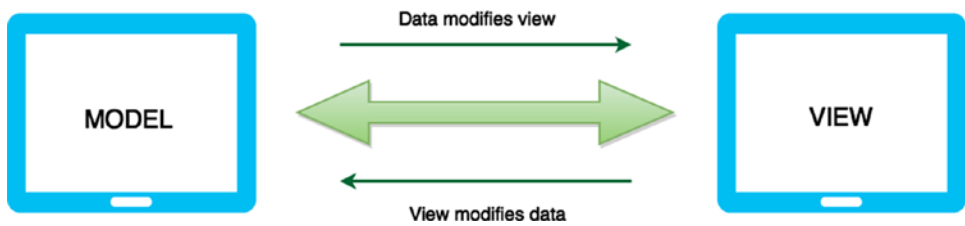


Figure 1-3. Two-way data binding

In JavaScript, data changes in memory and it is bound to a view in the UI, which means that when data is modified in JavaScript, which is in memory, the data will be changed in the UI as well. In return, when data changes in the UI (that is, the DOM) by clicking a button or any other event, it gets updated in memory also, keeping the two in sync. In theory, this works flawlessly and the idea is romantically perfect. However, in real-world applications, problems arise when you have a fairly complex and large application with multiple views representing data in one of your models. As you add more models and more views this two-way data binding ends up as spaghetti with every change in data added to the pot, which sometimes even ends up in an infinite event loop where one view updates a model, which in turn updates a view, and so on, as shown in Figure 1-4.

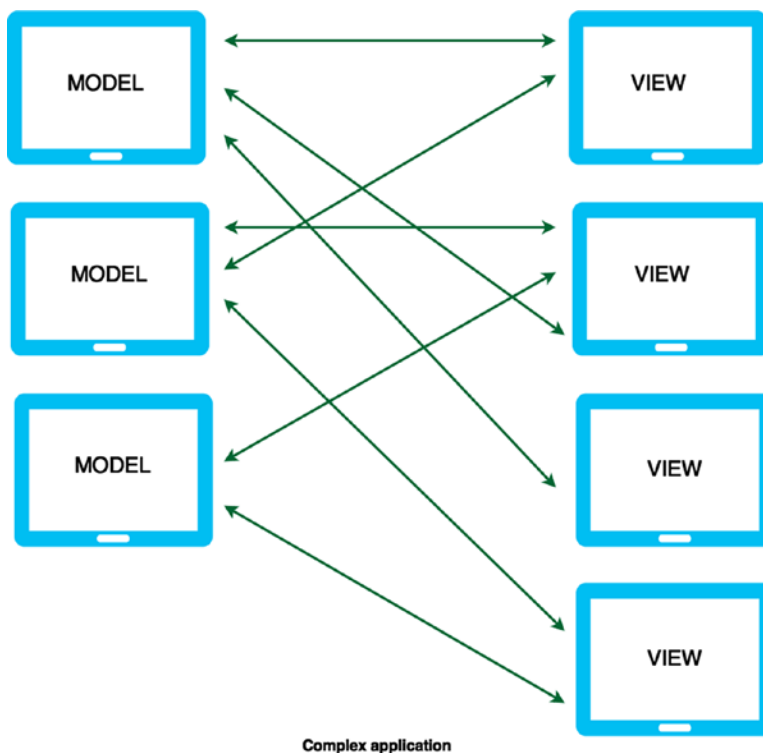


Figure 1-4. Unwanted spaghetti relationship

Another issue with this system is that making changes comes at a very high cost. When you introduce a new developer to an application that is this complex, it's tough to understand the impact one change might cause in this abyss of spaghetti relationships.

React follows one-way data flow to keep things simple, as shown in Figure 1-5. It is based on the concept of separation of concerns (SoC). This is a design principle in computer science in which an application or program is divided into distinct sections, each addressing a single or specific concern. The value of this design principle is that it simplifies development to create a maintainable and scalable application. This leads to modularized code where an individual section can be reused, developed, and modified independently. This makes so much sense and is indeed an example of intelligent thinking.

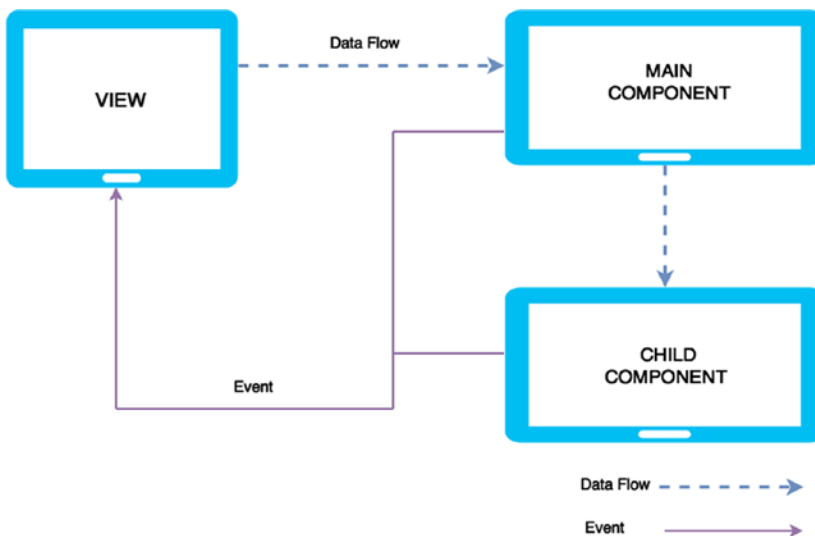


Figure 1-5. React Native's one-way data flow

Installation and Setup

In order to understand practical examples, you must first set up your environment to run your React examples. The easiest way to start working with React is using JSFiddle examples.

- React JSFiddle with JSX (you will learn about JSX shortly): <https://jsfiddle.net/reactjs/69z2wepo/>
- React JSFiddle: <https://jsfiddle.net/reactjs/5vjqabv3/>

Another way is to download the complete starter kit to even work offline: <http://facebook.github.io/react/downloads/react-0.13.3.zip>

You may also install react-tools using npm: `npm install -g react-tools`.

Another useful instrument to boost your productivity with React Native is React Developer Tools, which is a very useful Chrome extension that allows you to inspect React component hierarchy in the Chrome browser:

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>

For convenience, you will be using the following setup for your examples in this chapter: Facebook cdn for both React Native and JSX transformer.

react.js: <http://fb.me/react-0.13.3.js>

JSX transformer: <http://fb.me/JSXTransformer-0.13.3.js>

Simply refer to them in your code. Next, you will install the node packages browserify, watchify, and babelify, which will help you transform and keep your JSX in appropriate js files:

```
$ npm install react browserify watchify babelify --save-dev
$ watchify -t babelify ./your-jsx-file.jsx -o ./final-js-file.js -v
```

That's great! To serve the application, you will be using the simple python server (SimpleHTTPServer) in the root of the project folder via the following command:

```
$ python -m SimpleHTTPServer
```

Simple python server will serve your examples by default on the 8000 port.

Now that you are done with your setup, you can quickly start to understand the concepts of React.

Introduction to Components

React is all about components. Anything and everything you do in React is based on reusable components. In other words, with React you only do one thing, and that is build components. In previous sections, we discussed separation of concerns. The whole concept of these components is that they are totally encapsulated, making them easy to test and reuse.

Creating reusable components is a work of art and React Native provides many features for you. You will do a deep dive into them soon, but first let's create a simple "Hello World" component.

Create a `hello_world.html` file and paste the following code into it:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="http://fb.me/react-0.13.3.js"></script>
    <script src="http://fb.me/JSXTransformer-0.13.3.js"></script>
  </head>
```

```
<body>
  <div id="intro"></div>
  <script type="text/jsx">
    React.render(
      <h1>Hello, world </h1>, document.getElementById('intro')
    );
  </script>
</body>
</html>
```

That was simple. Your code is residing in HTML tags. JavaScript resides in script tags. We do see something different here in type ‘text/jsx’; this is JSX which we will explain in detail in upcoming sections. We also see a keyword ‘React’ which is the entry point into the React library. Inside of which there are H1 tags with the required text, along with the target where the text appears (in this case, `document.getElementById('intro')`). Also, this React code can live in a separate file and can be referenced in the HTML. Let’s create a separate file named `src/helloworld.js`.

```
React.render(
  <h1>Hello, world </h1>,
  document.getElementById("intro")
);
```

And now let’s reference it in your `hello_world.html` file:

```
<script type="text/jsx" src="src/hello-world.js"></script>
```

That was simple. But what is JSX? Let’s look into JSX before we go full throttle into different aspects of components.

JSX

React does not require you to use JSX. However, JSX does make life simpler. Its XML-like syntax helps define tree structures with attributes very easily. XML has benefits like balancing open and closed tags. This makes a large tree easier to read than function calls or object literals.

Let’s look at a simple example, first without JSX to see how things get simplified as we gradually include JSX.

```
<!DOCTYPE HTML>
<html lang='en'>
  <head>
    <meta charset="UTF-8">
    <title>Example without JSX</title>
    <script src=" http://fb.me/react-0.13.3.js"></script>
  </head>
  <body>
```

```
<script >

var Appo = React.createClass({
  render:function(){
    return React.createElement("h1",null, "Hello Dear")
  }
});

React.render(React.createElement(Appo), document.body);
</script>
</body>
</html>
```

In this example you use `React.createElement`. You first pass a string representing the HTML tag, the second is your property, and the third is the child, which is inner HTML in this case.

Next, you want to render your component in `React.render` so you pass the App component and target where it has to be loaded as `document.body`. The output is similar to Figure 1-6.



Figure 1-6. Output in `React.render`

Next, let's look at one more method, which is deprecated and no longer used. (The React ecosystem is developing so fast that things are already getting deprecated; however, having a look at the past shows how far they have come in making things easier in such a short period of time.)

```
<!DOCTYPE HTML>
<html lang='en'>
  <head>
    <meta charset="UTF-8">
    <title>Old world example</title>
    <script src="http://fb.me/react-0.13.3.js"></script>
  </head>
```

```

<body>
<script>
  var Appo = React.createClass({
    render:function(){
      return React.DOM.h1(null, "Hello Past")
    }
  });

  React.render(Appo(), document.body);
</script>
</body>
</html>

```

This example uses the `React.DOM` pattern where you specify the tag to be used (in this case, `h1`); the first parameter is the property and the second is a string, which is used as your inner HTML. When rendering with `React.render` you pass your component function and target where it is to be loaded (in this case, the document body). If you have latest version of React, you will get the warning and error shown in Figure 1-7 in your console.

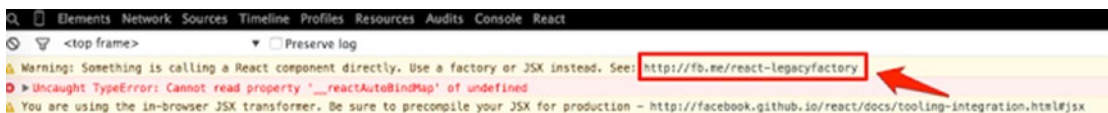


Figure 1-7. DevTools will show this warning

Visit <http://fb.me/react-legacyfactory> to find another way besides JSX called factory to wrap your component before calling it. This also reminds you that it is time to use JSX.

Let's now see how the same example looks when using JSX (see also Figure 1-8):

```

<!DOCTYPE HTML>
<html lang='en'>
<head>
  <meta charset="UTF-8">
  <title>JSX me like !</title>
  <script src="http://fb.me/react-0.13.3.js"></script>
  <script src="http://fb.me/JSXTransformer-0.13.3.js"></script>
</head>
<body>
  <!-- with JSX -->
  <script type="text/jsx">
    var App = React.createClass({
      render:function(){
        return <h1>Hi from JSX</h1>
      }
    });
    React.render(<App />, document.body);
  </script>
</body>
</html>

```



Figure 1-8. Output using JSX

Here, we have included JSX transformer, the in-browser transformer. Your component `App` has `h1` tags. Next, you render it using `React.render`, passing your component in JSX syntax and the second argument as the target, which is `document.body`. That's it.

Now separate your JSX code into a separate JSX file and convert it into pure JavaScript to see what's happening. For that, let's create a folder named `src` and place your JSX code into the `hello.jsx` file:

```
var Appo = React.createClass({
  render:function(){
    return React.DOM.h1(null, "Hello Past")
  }
});
React.render(Appo(), document.body);
```

Next, let's add a `dist` folder where you can keep your converted js file example, `bundle.js`. After this, reference your `bundle.js` file in your HTML file (in this case, `jsx_example.html`).

As discussed earlier, you should install the node packages `browserify`, `babelify`, and `watchify` in order to use them to convert your JSX into js. This can be done using the following command:

```
$ npm install react browserify watchify babelify --save-dev
```

Now, convert your `hello.jsx` into `bundle.js` with the following command:

```
$ watchify -t babelify ./src/hello.jsx -o ./dist/bundle.js -v
```

The result is the following `bundle.js` file:

```
var Appo = React.createClass({
  displayName: "Appo",
  render: function render() {
    return React.DOM.h1(null, "Hello Past");
  }
});
React.render(Appo(), document.body);
```

Although JSX is not required explicitly with React, it is preferred. It lets you create JavaScript objects using HTML syntax. Components serve two purposes: templates and display logic. Therefore, markup and code are tied intimately together. Display logic often is quite complex and to express it using template languages does become difficult. The best way to solve this problem is to generate HTML and components from JavaScript itself. JSX solves these problems with its HTML-type syntax by creating React tree nodes.

You can keep the JSX source code intact for better maintainability; just keep it in a separate folder of JSX files. To convert JSX code to normal JavaScript, use the JSX compiler (<http://facebook.github.io/react/jsx-compiler.html>) either manually or via a build script. It is possible to serve JSX files in your production environment, but React will give you console notices about reduced performance.

Deep-Dive Into Components

In next section, you will explore the vital concepts of *components*, which will help you work with them easily.

Properties

React has something similar to attributes in HTML: properties. You can use them to initialize your component, like so:

```
React.render(<App myProp="Hi from prop" />, document.getElementById('container'))
```

Here you are initializing the App component with a prop named `myProp`, which is targeted for the id of `'container'`. You can access this prop in JSX via interpolation in the following way. Let's create a component App:

```
var App = React.createClass({
  render: function(){
    var txt = this.props.txt
    return (
      <div>
        <h1> {myProp} </h1>
      </div>
    );
  }
});
React.render(<App myProp="Hi from prop" />, document.getElementById('container'))
```


If you refresh your browser, you will see a message from the property for your inner HTML. Please use the following code for your HTML file (the results are shown in Figure 1-9):

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <div id="container"></div>
  </body>
  <script src="http://fb.me/react-0.13.3.js"></script>
  <script src="http://fb.me/JSXTransformer-0.13.3.js"></script>
  <script src="dist/bundle.js"></script>
</html>
```



Figure 1-9. Refreshing your browser produces this message

As your application grows, you need to make sure your components are correctly created in the first place. In the case of a property, you can specify a kind of property with a range of validators. This ensures and validates the kind of data received. Let's look at some of them with an example:

```
var App = React.createClass({
  propTypes: {
    message: React.PropTypes.string,
    age: React.PropTypes.number
  },
  render: function(){
    // keeping props in a variable to use to often
    var message = this.props.message,
        age = this.props.age
    return (
      <div>
        <h1> {message} </h1>
        <p> My age is {age}</p>
      </div>
    );
  }
});

React.render(<App age={5} message="Hi from prop" />, document.getElementById('container'))
```

Here, the `propTypes` keyword signifies a hash of prop names and their types. There are two propTypes in your example: `string` and `number` for the `message` and `age` properties, respectively.

There are many other property types. Note that you can add `isRequired` to the end of any propTypes to make it required:

```
propTypes: {  
  //some specific JS primitive  
  arrayType: React.PropTypes.array,  
  boolType: React.PropTypes.bool,  
  funcType: React.PropTypes.func,  
  objectType: React.PropTypes.object,  
  //if a value of a prop is necessary  
  numberType: React.PropTypes.number.isRequired  
}
```

There is also a default type in properties via the keyword `getDefaultProps`. For example, in your same component, you can mention default types for your `message` and `age` properties:

```
getDefaultProps: function(){  
  return {  
    message: 'Default value of message',  
    age: 0  
  }  
},
```

State

In the last section, you learned about properties, which are static values that are passed into your component. State, on the other hand, is maintained and updated by the component. Let's understand this concept with an example:

```
var App = React.createClass({  
  getInitialState: function(){  
    return {  
      message: 'this is a default message from state',  
    }  
  },  
  render: function(){  
    return (  
      <div>  
        <h1> {this.state.message} </h1>  
      </div>  
    );  
  }  
});  
React.render(<App />, document.getElementById('container'))
```

If you run this snippet, you will see the result shown in Figure 1-10 in your browser.



Figure 1-10. Resulting message using state

Let's look at the code. In your same component, you initialize the state using the `getInitialState` keyword, in which you set up the initial state of the message:

```
getInitialState: function(){
  return {
    message: 'this is a default message from state',
  },
},
```

Next, unlike the last example, you access this state using `this.state.message`, which prints the initial text of the message state:

```
<div>
  <h1> {this.state.message} </h1>
</div>
```

Now let's add some functionality to your component. You add a text box above your message statement. As you type in the text box, the message gets updated in real time using the concept of state:

```
var App = React.createClass({
  getInitialState: function(){
    return {
      message: 'this is a default message from state',
    }
  },
  updateState: function(e){
    this.setState({message: e.target.value})
  },
},
```

```
render: function(){
  return (
    <div>
      <input type="text" onChange={this.updateState} />
      <h1> {this.state.message} </h1>

    </div>
  );
}
});

React.render(<App />, document.getElementById('container'))
```

If you execute this code in your browser, you will see the result shown in Figure 1-11.



Figure 1-11. Adding a text box above your message statement

Let's see what you added to your component. First, you introduced a function named `updateState`:

```
updateState: function(e){
  this.setState({message: e.target.value})
}
```

This new function, `updateState`, takes an event called (`e`) and updates the value of `message` state. Also, you added an input field:

```
<div>
  <input type="text" onChange={this.updateState} />
  <h1> {this.state.message} </h1>
</div>
```

The input box has an `onChange` event, which calls your custom method `updateState` whenever the state gets updated. As you type in your textbox, your printed message gets updated instantaneously.

Summary

This chapter provided a quick tour of React. Before you begin with the next chapter, let's recap what you have learned so far. You were introduced to the React library and the reasons behind its invention. Then you learned how to install and set up React. You studied the fundamentals of this technology, such as virtual DOM, one-way data flow, and JSX. You also got an introduction to components, and using states and props with components.

Now that you are equipped to code and work in the React ecosystem, the interesting path of your journey begins in the next chapter. You'll start working with React Native.

Chapter 2

The Simplest Program: Hello World with React Native

“Big things have small beginnings.”

—Prometheus

In the last chapter, you got a good overview of the React ecosystem. Now it’s time to get your hands dirty with React Native. In this chapter, you will set up your environment by installing the prerequisites and then you will create your first React Native application.

The best way to learn is through practical examples. Continuing this theme throughout the book, you will follow examples to learn React Native by programming to understand the concepts.

In this chapter, you will explore the following topics:

- An introduction to React Native
- The essentials of React Native
- The installation of React Native
- Your first application
- The anatomy of a React Native application
- How to debug your application

Note You might face a situation where different projects work on different Node versions. Therefore, it’s recommended you install NVM (node version manager) to help keep multiple node versions that can be switched between projects.

What Is React Native?

React Native is an open source platform for developing native mobile applications; it was developed largely by a team at Facebook. The cool part of working with React Native is that your program uses standard web technologies like JavaScript (JSX), CSS, and HTML, yet your application is fully native. In other words, your application is blazing fast and smooth, and it is equivalent to any native application built using traditional iOS technologies like Objective-C and Swift. However, React Native does not compromise in terms of performance and overall experience, like popular hybrid frameworks that use web technologies to build iOS apps.

React Native aims to bring the power of React, which was explained in Chapter 1, to mobile development. In the words of the React team, “learn once, write anywhere.” Working with React and React Native, you will see how many of your components built for the Web using React can be easily ported to your React Native iOS apps with little or no modification. React Native introduces a highly functional approach to constructing user interfaces, which is very different from the traditional iOS development approach.

Although React Native was built by Facebook developers, it’s an open source project. The code is available at <https://github.com/facebook/react-native>.

React Native Prerequisites

Before starting the installation, let’s first review the prerequisites for React Native:

- iOS apps can be developed only on an Apple Mac with OSX installed. You need OSX version 10.10 or above.
- You need Xcode 6 or above, which includes the iOS SDK and simulators. React Native only supports iOS7 or above. Xcode can be downloaded from the Apple App Store.
- It’s helpful if you are enrolled in the Apple iOS Developer program. If you’re not in the iOS Developer Program, you won’t be able to
 - Test applications on actual devices.
 - Access beta OS releases.
 - Test flight for beta testing.
 - Submit your app to the App Store.

Installation

Let’s do a quick, one-time setup of React Native. React Native is an assortment of JavaScript and Objective-C code, so you need tools that create, run, and debug your native application written in JavaScript. Let’s go one by one.

Installing Node and npm

Node.js is an open source platform built on Chrome's JavaScript runtime; it offers a way to easily build fast, scalable programs. Node.js allows you to run JavaScript in Terminal, and helps create modules. You install Node.js by running the following command in Terminal: `$brew install node`.

Homebrew is the recommended way to install Node. You can also download the installer from <https://nodejs.org> and install it manually.

npm is the package manager for Node.js. If you're from the iOS world, it's similar to CocoaPods.

Check your Node installation by running the following command in Terminal:

```
>> node -v  
v4.2.1
```

```
>> npm -v  
2.13.1
```

Installing Watchman

Watchman observes your files and records when they change. It can also trigger actions (such as rebuilding assets) when matching files change. For more details, visit <https://facebook.github.io/watchman/>.

You can install Watchman by running the following command:

```
$brew install watchman
```

Installing the React Native Package

React Native comes as an npm package, so use the following code to install the React Native-cli module:

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

Updating React Native

Both React Native and iOS are fast-moving frameworks. It is recommended to update them every time a new release is available. Upgrading React Native is simple. Run the following command in Terminal:

```
npm update -g react-native-cli
```


Your First App

Now that you are all charged up about React Native and have your system set up, it's time to create your first application. To keep things simple, in the beginning just follow along. Sometimes you might feel disconnected by monotonously typing in the code, but going along is enough for now. Remember that mimicry is a powerful form of learning; it's how we learned most of our skills, such as talking, reading, writing, and it is how you will learn to program with React Native. As you proceed, this method will help you understand deeply why you authored certain pieces of code.

Throughout the book, you will create one application and take it from just Hello World to a full-blown, distribution-level application, except in some places where we need to digress to explore a concept independently. So before you set it up, let's talk about the problem you plan to solve. The app you will create during the course of this book plans to solve a few *housing* problems; it will be a very primitive version of any popular property search application.

Let's call it HouseShare. It will have some rudimentary features like listings, creating an entry, geolocating a property, and many more. As you move along, you will see how various React Native features fit with your application.

That's quite a lot, but in this chapter you will just create the basic structure for your project using React Native and some Hello World code.

Creating a Basic Skelton

Fire up Terminal and type in the following command:

```
react-native init HouseShare
```

This code uses the CLI tool to construct a React Native project that is ready to build and run as-is. This command creates the basic folder structure for your React Native iOS project. Next, let's get into this directory:

```
> cd HouseShare/ios/
```

Now, click `HouseShare.xcodeproj`. This is an Xcode project file and will open up your project in Xcode. Next, let's load your application in the iOS simulator. To build your application and load it in the simulator, simply click the Run button at the top (Figure 2-1) or execute Command + R. This will compile, build, and fire up your project in the iOS simulator (Figure 2-2).



Figure 2-1. Building by clicking the Run button

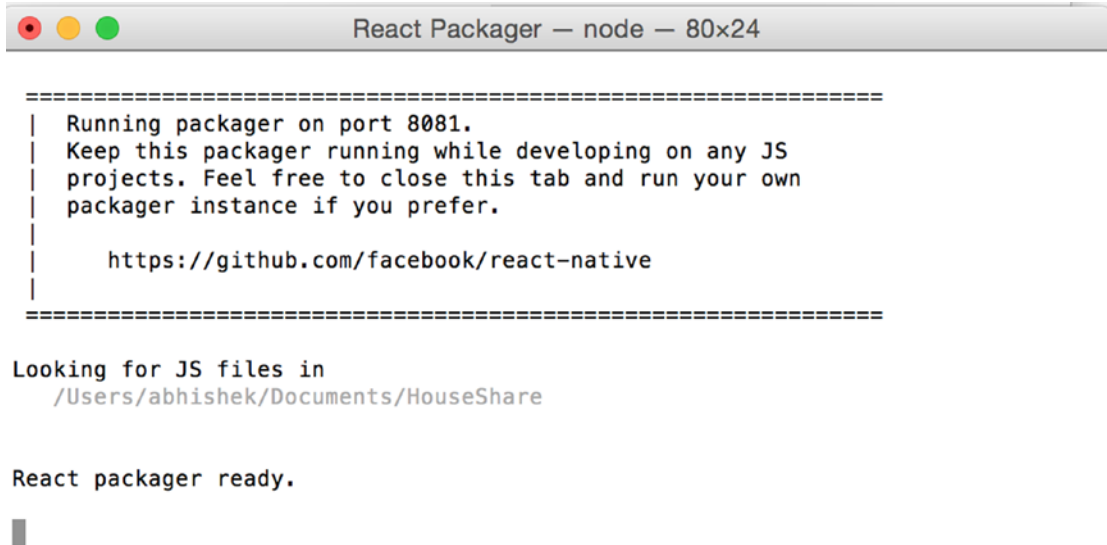


Figure 2-2. Using the iOS simulator

That was really quick. Thanks to a single command, the basic structure of your project is in place and your application is loaded in the simulator. Also note that Terminal is opened automatically when you run the application from Xcode. This is the Node package manager for React Native. If you kill this, the app will stop working.

Terminal is opened to start the React Native Packager and a server to handle the above request (Figure 2-3). The React Native Packager is responsible for reading and building the JSX (you'll look at this later) and JavaScript code.

`http://localhost:8081/index.ios.bundle`



```

=====
| Running packager on port 8081.
| Keep this packager running while developing on any JS
| projects. Feel free to close this tab and run your own
| packager instance if you prefer.
|
| https://github.com/facebook/react-native
|
=====

Looking for JS files in
/Users/abhishek/Documents/HouseShare

React packager ready.
■

```

Figure 2-3. The packager is ready

Note If Terminal is not started and the iPhone shows a red screen, run `npm start` and keep Terminal open.

Set up your project in any editor you prefer. React Native does not force you nor does it have a preference towards any specific editor, so you may continue to use Xcode as default. However, we recommend you open your project in an editor like our personal favorite, Sublime Text.

Make some changes in `index.io.js`. In fact, remove all code from the file. Now, add the following code:

```

'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}>
          HelloWorld !!
        </Text>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  welcome: {
    fontSize: 25,
    textAlign: 'center'
  }
});

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

Press Command + R in the iOS simulator and it will refresh the screen, as shown in Figure 2-4. (Note that in this book, we will use the Command key instead of the CMD key, which is equivalent.)



Figure 2-4. *The screen is refreshed*

That was quick! In a fraction of a second you can see the changes you applied. You don't need to compile the code and restart the simulator for React Native changes. If you have done any native iOS app development before, hitting Refresh to see the changes may seem like a miracle.

Now, let's understand the code. At the top of the file is the following line:

```
'use strict';
```

This enables strict mode, which adds improved error handling to the React Native JavaScript code. The fifth edition of the ECMAScript specification introduced strict mode. Strict mode makes it easier to write secure JavaScript and convert bad syntax into real errors. This is very important for debugging any example and using undeclared variables.

Next is the following line:

```
var React = require('react-native');
```

This loads the React Native module and assigns it to a React Native variable that can be used in your code. React Native uses the same module-loading technology as Node.js with the `require` function; this is roughly equivalent to linking and importing libraries in Swift.

After that, you add the following snippet:

```
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;
```

You are assigning multiple object properties to a single variable; this is called a destructuring the assignment. This cool feature was proposed in JavaScript ECMAScript 6. Although it is optional, it's very beneficial; otherwise, every time you use a component in your code, you would have to use a fully qualified name for it, such as `React.AppRegistry`, `React.StyleSheet`, and so on. This saves quite some time.

Next, you create a view:

```
var HelloWorld = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          HelloWorld !!
        </Text>
      </View>
    );
  }
});
```

React Native's basic building blocks are called components. You can use the `createClass` method to create custom component classes. This class has just one function, `render()`. The render function is responsible for what is shown on the screen. You use JavaScript syntax extensions (JSX) for rendering the UI. JSX is a JavaScript syntax extension that looks similar to XML.

Now you define the styling of your app. Here you will use Flexbox; it is similar to what CSS is to HTML. For now, you can type this code. We will explain styling in the next chapter.

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  welcome: {
    fontSize: 25,
    textAlign: 'center'
  }
});
```

You can see that this styling is very similar to CSS; you can define font size, alignment, and so on.

The last step is to define the entry point and root component of the app:

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

It's Not a UIWebView

You are using web technologies, but your app does not have a web component; it has a native component. Open Debug ► View Debugging ► Capture View Hierarchy (see Figure 2-5).

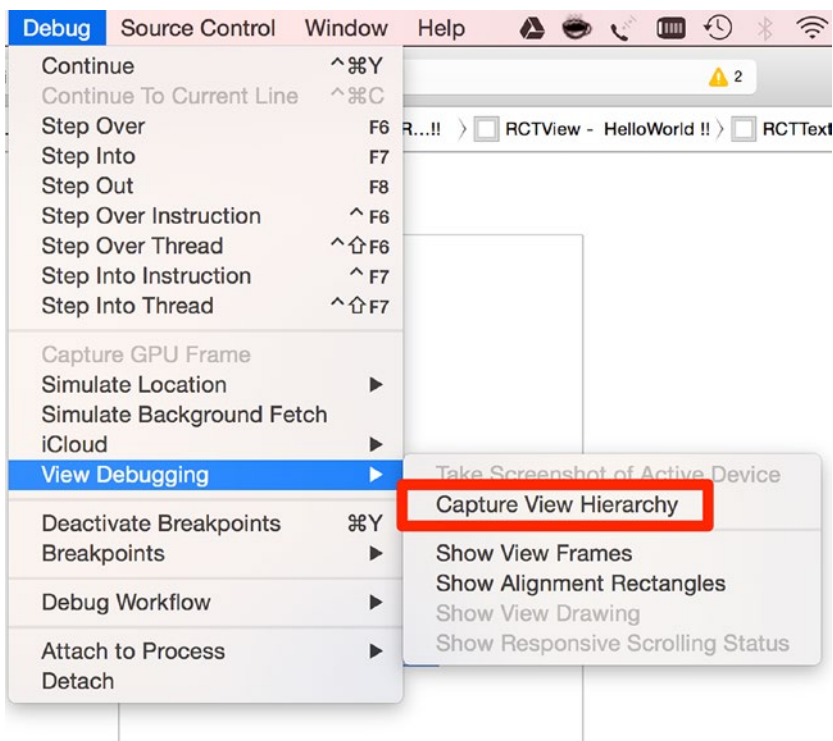


Figure 2-5. Using the native component

As you traverse through the tree of UIWindow, you'll see that there is no UIWebView in the code, and "Hello World !!" is the call of RCTText, as shown in Figure 2-6.

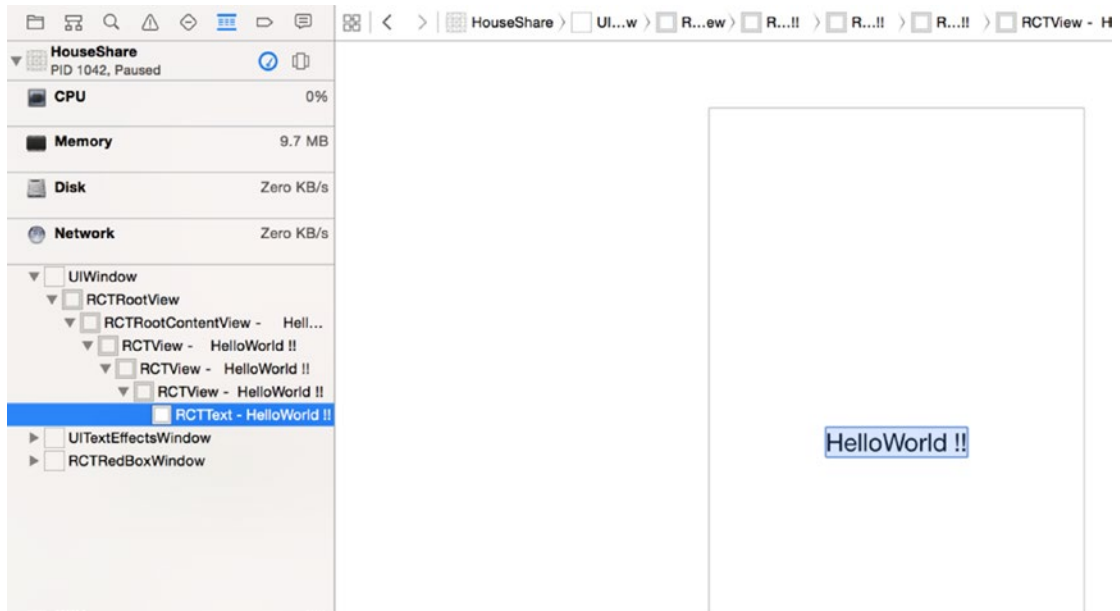


Figure 2-6. "Hello World !!" is the call of RCTText

Enabling Live Reload

Another cool feature of React Native is live reload. It reloads your application view inside the iOS simulator the moment there is a change. To activate this option, you need to access the developer menu from the application opened in the iOS simulator by pressing Ctrl + Command + Z and select the Enable Live Reload option. Now any change made in your JavaScript code will cause your app to reload automatically.

Why Is React Native Different?

Before you deep dive further into the React Native world, you must understand why there was a need for another framework to build mobile apps. We already live in a world full of frameworks and tool chains that are capable of building mobile apps. Prior to the inception of React Native, building mobile apps using web technologies was possible via two strategies:

- *WebView-based:* These frameworks use common web technologies like HTML and JavaScript, and use WebView to load the application. An example is the popular framework Phonegap.
- *Native apps using web technologies:* These frameworks again use common web technologies like HTML and JavaScript (to be precise, they imitate using JavaScript and HTML) to create native apps. An example is the popular framework Titanium Appcelerator.

Apps created using these strategies have performance issues. WebView-based apps are slow because they use the DOM, and DOM manipulations are expensive, which leads to performance issues. As stated in one of the blog posts at Flipboard (<http://engineering.flipboard.com/2015/02/mobile-web/>), “you cannot build a 60fps scrolling list view with DOM.” This is one of the fundamental problems with apps developed through this technique: although development time may be quick, you end up with a sluggish experience.

The other strategy, where the framework imitates JavaScript and HTML, and converts them to native code, has other challenges. Although the final app is native in nature, there is a basic issue during this conversion from JavaScript to native: it runs on the main thread. In these apps, you interface directly with native objects all the time, which leads to once again a slow and sluggish experience.

React Native is fundamentally different from these two approaches. It runs all layouts on separate threads, and your main thread is free to update the UI, which makes the animation and UI rendering smooth, just like 100% pure native apps.

React Native use the JavaScriptCore framework to run JavaScript. In iOS 7, Apple introduced a native Objective-C API for JavaScriptCore. This framework allows JavaScript and Objective-C to talk to each other. This means you can create and call JavaScript functions from Objective-C or call back into Objective-C from JavaScript. It all works like a charm.

React Native is different in one more aspect. As seen in your Hello World example, you write a component in JavaScript just like you would do with React, except that instead of using an HTML div, you use tags like View and Text. In the case of an iOS application, a View is basically a UIView.

The Anatomy of a React Native Application

Now let's understand the application structure that the React Native `init` command has generated. If you open the project called HouseShare, it looks like a normal Xcode project. It has the following folder structure:

```
|ios
|- HouseShare
|- HouseShare.xcodeproj
|- HouseShareTests
|android
node_modules
index.ios.js
index.android.js
package.json
```

Note The folder structure defined here might be changed or modified as the framework evolves, but the majority of the functionality remains the same.

If you open the project in Xcode, it will have a different folder structure. The “folders” in Xcode are actually groups and are not necessarily linked to a folder in Finder.

- **iOS:** The iOS folder has two folders and one file. As seen above, there is a HouseShare folder, which has all the Objective-C code, such as AppDelegate, Images.xcassets, Info.plistLaunchScreen.xib, and other files. Another folder is HouseShareTests, which is where all your test cases reside. Finally, there is your Xcode project file, HouseShare.xcodeproj, which is used to load into Xcode to build your application.
- **package.json:** This folder contains metadata about your app, and it will install all dependencies when you run the npm install. If you're familiar with Ruby, it's similar to a Gemfile.
- **node_modules:** All of the Node modules mentioned in package.json will be downloaded to this folder. This folder also contains the code for the React Native framework.
- **index.ios.js:** This is the file where you begin programming your iOS application.
- **AppDelegate.m:** This is the starting point of any iOS app.
- **Android:** React Native also supports development for Android. All your native Android code resides in this folder.
- **index.android.js:** This file is where you begin programming for an Android application.

Let's open the AppDelegate.m file from HouseShare/ios/HouseShare/AppDelegate.m:

```
#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 a device, change `localhost` to the IP address of your computer
     * (you can get this by typing `ifconfig` into Terminal and selecting the
     * `inet` value under `en0:`) 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
 * from the root of your project directory, run
 *
 * $ react-native bundle --minify
 *
 * see http://facebook.github.io/react-native/docs/runningondevice.html
 */

// jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"main"
withExtension:@"jsbundle"];

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

self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
UIViewController *rootViewController = [[UIViewController alloc] init];
rootViewController.view = rootView;
self.window.rootViewController = rootViewController;
[self.window makeKeyAndVisible];
return YES;
}

@end
```

`RCTRootView` is an Objective-C class provided by React Native, which is inherited from the iOS `UIView` Class. It takes your JavaScript code and executes it.

It also loads the `http://localhost:8081/index.ios.bundle` URL, which has your code written in `index.ios.js` and also a program added by the React Native framework.

Debugging

Debugging with React Native is in line with how we debug web apps; in short, it's really simple. To access debugging options, press `Command + D` within the loaded application from the iOS simulator. This will open a menu that provides several debugging options, as shown in Figure 2-7.

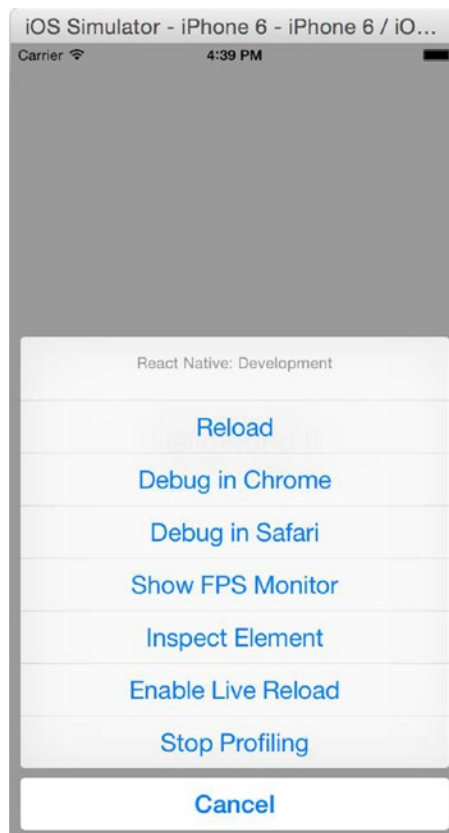


Figure 2-7. *Debugging options*

You must disable this menu for the final build because your end user should not see these options. To disable it, open the project in Xcode and select: **Product** ► **Scheme** ► **Edit Scheme** (or press **Command + <**). Then select **Run** from the menu on the left and change the **Build Configuration** to **Release**.

Let's review each of the options shown in Figure 2-7.

Reload

The reload option refreshes the screen in the simulator with the latest React Native code without compiling the project again. This can be done in two ways: one, by clicking the **Reload** option in the menu, as shown in Figure 2-7, or by pressing **Command + R**. This will reload all the changes made in the JavaScript code.

Any changes made in your Swift or Objective-C files will not be reflected this since these changes require recompilation. Also, if you add any assets like images, the app needs to be restarted.

Debugging in Chrome

This is one of the best and most used options for debugging your JavaScript code written in React Native. As with web apps, you can debug your React Native application in Chrome. When you click “Debug in Chrome” it opens `http://localhost:8081/debugger-ui` in Chrome (Figure 2-8).

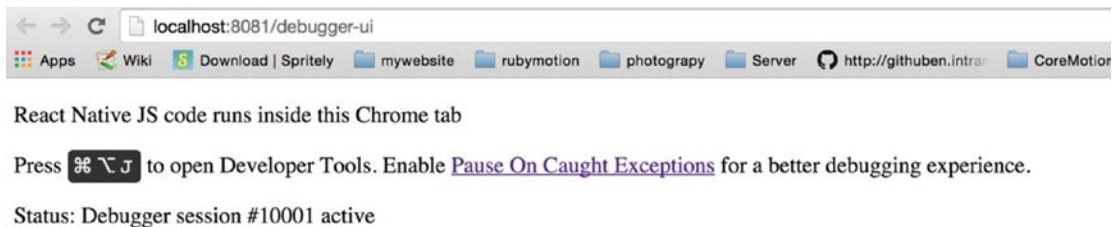


Figure 2-8. Debugging in Chrome

Install the React Developer Tools, which is a Chrome extension for debugging both your React application and React Native code. It allows you to inspect the React Native component hierarchies in the Chrome Developer Tools. To install it, please visit the Chrome webstore or go to the following URL: <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>.

Once the extension is installed, press Command + Option + J or select View ➤ Developer ➤ Developer Tools from your Chrome browser to access the developer tools console.

You will get a new tab called React in your Chrome DevTools. This shows that you the root React components that have been rendered on the page, as well as the subcomponents that they ended up rendering. You can also see props, state, components, and event listeners, as shown in Figure 2-9.

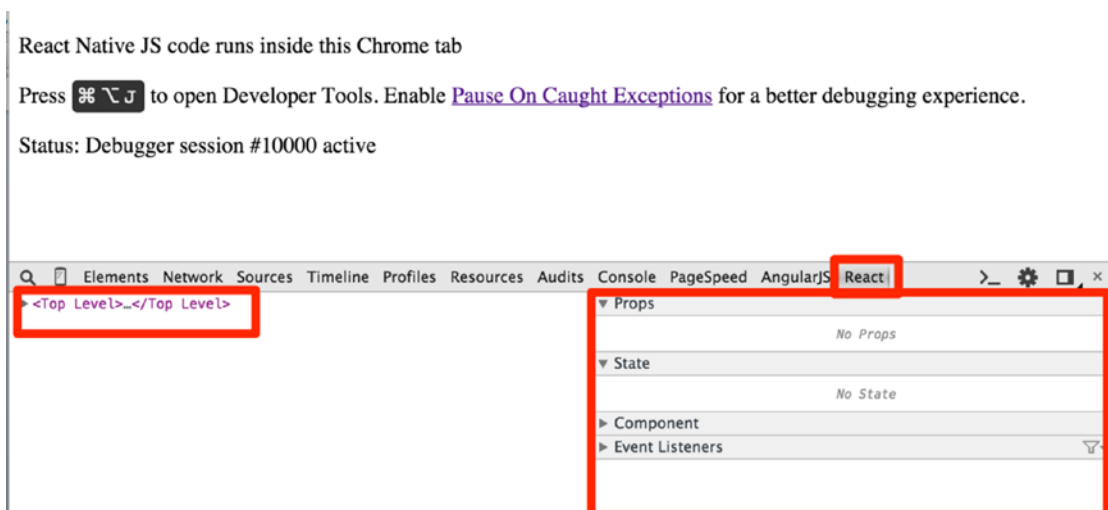
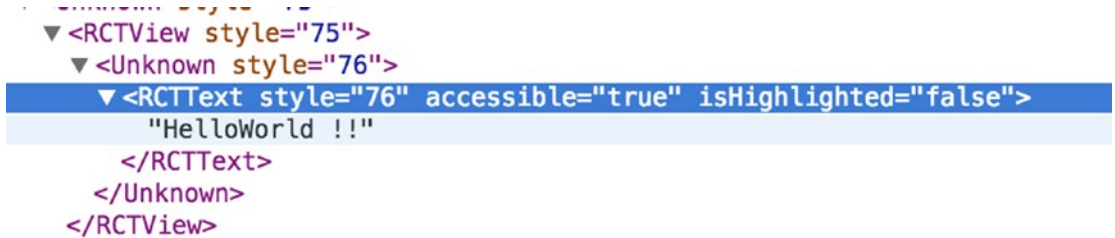


Figure 2-9. Debugging in Chrome DevTools

Look at Figure 2-10 and you can see a similar hierarchy to your Xcode: Hello World is wrapped in RCTText and that is in turn wrapped in RCTview.



```
▼ <RCTView style="75">  
  ▼ <Unknown style="76">  
    ▼ <RCTText style="76" accessible="true" isHighlighted="false">  
      "HelloWorld !!"  
    </RCTText>  
  </Unknown>  
</RCTView>
```

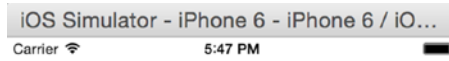
Figure 2-10. Debugging the app with the React tab in Chrome DevTools

Debugging in Safari

If you do not have Chrome, you may also use Safari for debugging, but Chrome is preferred for debugging React Native apps.

Showing the FPS Monitor

Many applications use a lot of animations and graphics. FPS (frames per second) defines the smoothness of these animations for your application; this is used extensively in gaming apps. When you select “Show FPS Monitor” in the menu, it shows a few properties for your app in the simulator (Figure 2-11). Although you might not find much use for these properties in your Hello World app, they are great for animation-intensive apps to prevent them from getting into lethargic mode, creating a bumpy user experience. See Figure 2-11.



HelloReact!!



Figure 2-11. Additional properties in the simulator

The Inspect Element

You can also inspect a React Native element from the simulator, somewhat similar to how you inspect an element in a browser, although you can't currently change live values of properties as you can in a browser. For now, you can see your style sheet properties for any object. Click the HelloReact!! text (Figure 2-12) and it will open the details of that element.



Figure 2-12. Click the text to see element details

The details of that element are shown in Figure 2-13 at the bottom left.

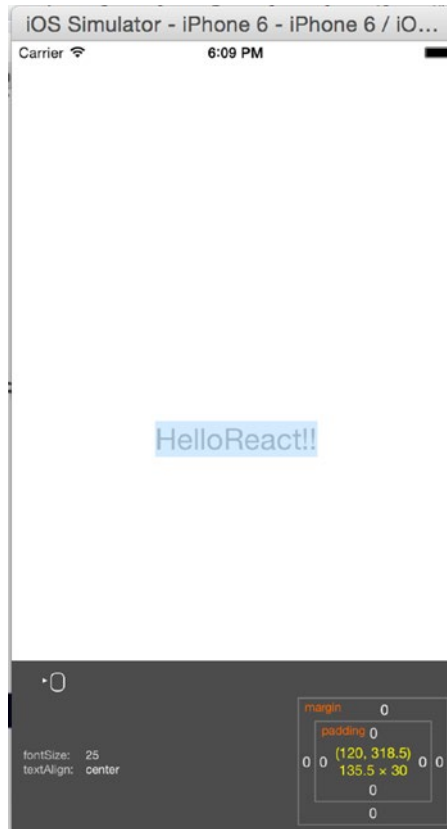


Figure 2-13. Font details

You can see that the font size for Hello World is 25 and it is aligned at the center.

Starting Profiling

Profiling is used to measure performance. The profiling session provides insight into which parts of your code are used most, how much time it takes, and which sections of your code you should improve. You can also find the time spent in each method from the number of times the profiler stopped on a method.

The following URL links to a very nice blog post that will give you a good overview of how to use profiling with Xcode: www.raywenderlich.com/23037/how-to-use-instruments-in-xcode.

Summary

In this chapter, you were introduced to React Native. You set up the React Native development environment and you wrote your first application. You also learned about the folder structure of React Native applications and how to debug. You are now all set to take a deep dive into creating a user interface with React Native for your iOS application.

In the next chapter, you will learn how to create stunning user interfaces by mastering Flexbox. You'll see how to navigate from one component to another, how to add images to your unembellished application, and how to create a `ListView` and a `ScrollView` using React Native.

Canvas, Brush, and Paint: Working with the User Interface

“User interface is the process of shifting from chaotic complexity to elegant simplicity.”

—Akshat Paul

In the previous chapter, we were introduced to React Native and created our first React Native application. Now that we have a blank skeleton for our project, we will fill it with a stunning user interface. In this chapter, we will cover the following topics:

- NavigatorIOS
- Flexbox
- Image
- TouchableHighlight
- Routing to a new component
- ListView
- ScrollView

Any experienced software professional will agree—the success of an app depends on the fact that it not only works flawlessly, but also looks great. Therefore, a great user interface makes a huge difference in the success of your app.

The layout system is a fundamental concept that needs to be mastered in order to create great applications. Let's begin by understanding how to navigate within an iOS application using React Native.

NavigatorIOS

NavigatorIOS is a React Native component for creating navigation. It wraps UIKit navigation and allows you to add a back-swipe feature to your app. NavigatorIOS manages a stack of view controllers so as to provide a drill-down interface for hierarchical content. Now that we know what NavigatorIOS does, let's implement it in our project.

First, add NavigatorIOS in the list of components in the `index.ios.js` file:

```
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  NavigatorIOS,
} = React;
```

Let's now create a component that has NavigatorIOS and call it `mainView`, and then let's load another component within `mainView`, which we will call `Home`:

```
var mainView = React.createClass ({
  render: function() {
    return (
      <NavigatorIOS
        style={styles.navigator} initialRoute={{
          title: 'House Share',
          component: Home
        }}
      />
    );
  },
});
```

`Home` component will be like home page of our application.

NavigatorIOS is a React Native component; we have also provided a title (House Share) and component name (Home) for the page using `initialRoute`. There are other options available, which are documented at <https://facebook.github.io/react-native/docs/navigators.html>.

NavigatorIOS helps with the most basic iOS routing. A route is an object that describes each view in the navigator. First, the route is provided to NavigatorIOS. In the previous code, we called component Home from our mainView. Let's add an empty Home component:

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

Let's also add some styles to our mainView component:

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  welcome: {
    fontSize: 25,
    textAlign: 'center'
  },
  navigator: {
    flex: 1
  }
});
```

Let's now update the entry point for our application to mainView:

```
AppRegistry.registerComponent('HouseShare', () => mainView);
```

Let's now build our application with Xcode and see the result, as shown in Figure 3-1.



Figure 3-1. Our application, with updated navigation bar at the top

Excellent—we can see an updated navigation bar at the top. Let's add some color to this navigation bar to make our changes more visible:

```
var mainView = React.createClass ({
  render: function() {
    return (
      <NavigatorIOS barTintColor='#48BBEC' titleTextColor= "FFFFFF" style={styles.
        navigator} initialRoute={{
          title: 'House Share',
          component: Home
        }}
      />
    );
  },
});
```

We have used `barTintColor` and `titleTextColor` to add colors to our navigation bar (see Figure 3-2).



Figure 3-2. Our toolbar now has color

We have done a little bit of styling in this section, which might be something new for you if you come from a grid-layout background. React Native uses Flexbox for styling, which we will discuss in detail in the following section.

Flexbox

While creating our layout in previous example, you must have seen the `flex` property mentioned in the styles. This is because React Native apps use the Flexbox layout model.

The React Native Flexbox layout model is inspired by CSS Flex Box Layout in CSS3. The React Native team has rewritten this feature specifically for iOS. The main idea behind the Flexbox is being able to create the layout without worrying about different screen sizes or orientation of devices. A flex container expands items to fill available free space, or shrinks them to prevent overflow. Let's get some basic knowledge of Flexbox to expedite our layout development. First, let's update the view:

```
var Home = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.topBox} />
        <View style={styles.bottomBox} />
      </View>
    );
  }
});
```

We have created one main view with a style container and two subviews with the styles `topBox` and `bottomBox`. Now, let's create the styles:

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column'
  },
  welcome: {
    fontSize: 25,
    textAlign: 'center'
  },
  navigator: {
    flex: 1
  },
  topBox: {
    flex: 2,
    backgroundColor: '#CCE5FF'
  },
  bottomBox: {
    flex: 1,
    backgroundColor: '#FFF5CC'
  }
});
```

Turn back to the simulator and refresh the view using `Command+R`.



Figure 3-3. Screen in portrait mode

Now, rotate the simulator, and you will see it automatically adjust the size of these colored boxes.

Let's change the simulator to landscape mode (see Figure 3-4). This can be done easily using Command + Right/Left arrow key (⌘+Left Arrow). You can see how the box has adjusted its size, and how the title adjusted its width, in order to use all the available space. Thanks to Flexbox, a pretty laborious task is simplified.



Figure 3-4. Screen in landscape mode

Now, let's review the flex properties `flex-direction` and `flex`.

Flex-direction

Flexbox is a single-direction layout concept. `flex-direction` allows you to define the direction in which the child elements are going to flow. It can have two values, `row` and `column`. In the previous example we used `column`.

Let's change it to `row`:

```
container: {  
  flex: 1,  
  flexDirection: 'row'  
}
```

Turn back to the simulator and refresh the view with Command+R (see Figure 3-5).

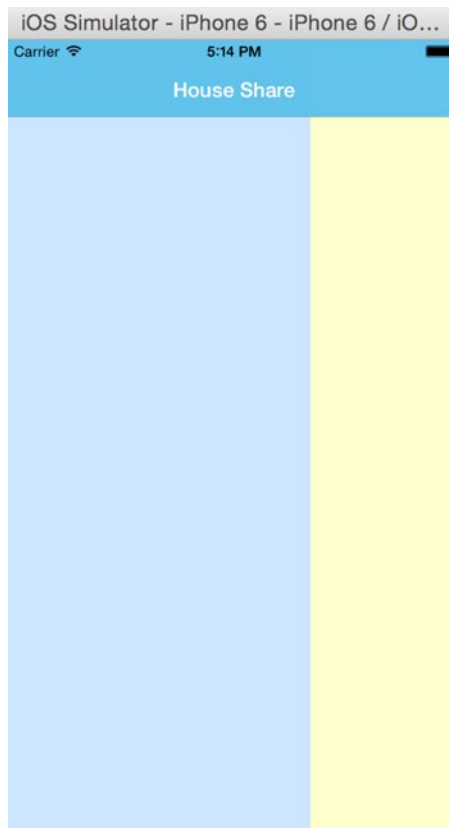


Figure 3-5. *Changing the orientation of the box*

We can see how the orientation of the box has changed. Now again change the property `flexDirection` to `column` (see Figure 3-6).

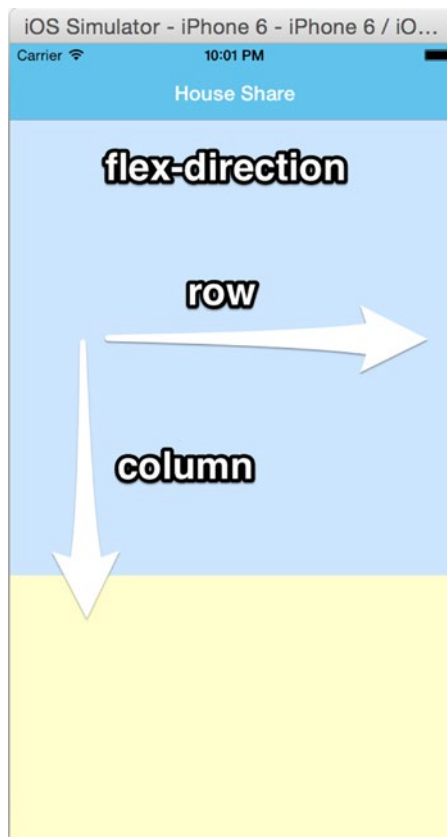


Figure 3-6. Changing the property to column

Flex

You must have seen the flex value in the stylesheet; it can be either integers or decimals, indicating the relative size of the box:

```
container: {  
  flex: 1,  
  flexDirection: 'column'  
},  
  
topBox: {  
  flex: 2,  
  backgroundColor: '#CCE5FF',  
},  
bottomBox: {  
  flex: 1,  
  backgroundColor: '#FFFFCC'  
}
```

And our view says:

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

So flex defines the size percentage for the box. We can see that the container has two views inside, topBox and bottomBox, with flex values of 2 and 1 respectively (see Figure 3-7).

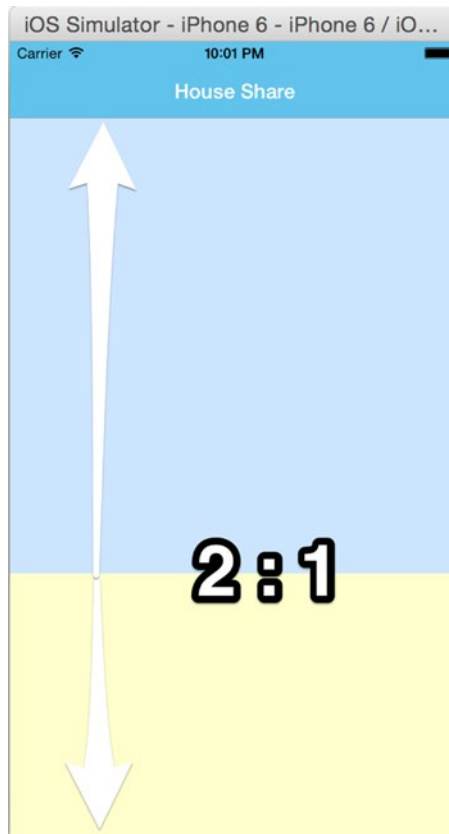


Figure 3-7. Container in 2:1 ratio

Now, update the view and add one topBox view inside container view:

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

Refresh the view. The container has three views now: topBox, bottomBox, and then topBox again (see Figure 3-8).

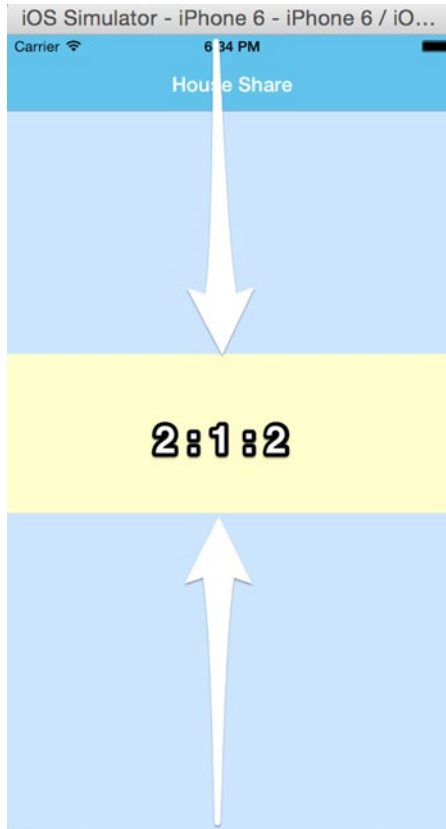


Figure 3-8. Container with three views

This will divide the view into a 2:1:2 ratio, since their flex values are in the ratio 2:1:2.

To get better sense of how this works, let's change the flex values and see how it changes our screen. Let's change the flex value of topBox to 1.

Let's update the CSS to:

```
container: {  
  flex: 1,  
  flexDirection: 'column'  
},
```

```
topBox: {  
  flex: 1,  
  backgroundColor: '#CCE5FF',  
},  
bottomBox: {  
  flex: 1,  
  backgroundColor: '#FFF5CC'  
}
```

Refresh the view to see the changes, as shown in Figure 3-9.

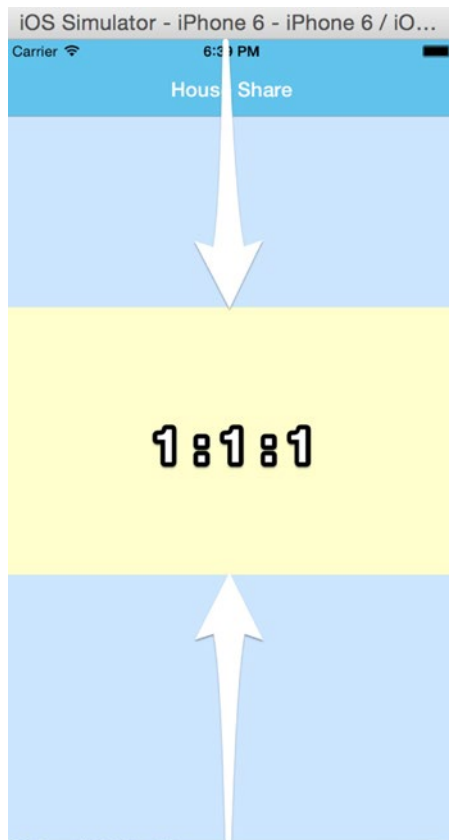


Figure 3-9. View in ratio of 1:1:1

We can see that now the screen is divided in a ratio of 1:1:1, since the flex values of the views are in a ratio of 1:1:1. With Flexbox, it is easy to create layouts that can resize according to screen size and orientation. This is just introduction to Flexbox; we will explain more properties throughout the book as and when needed. You can also find more options at <https://facebook.github.io/react-native/docs/flexbox.html>.

Adding Images

React Native has a built-in component, `Image`, which will help us to display images, including network images, temporary local images, and also images from local disk, such as the camera roll. To start, we will display local images.

Open the `Image.xcassets` file in Xcode (see Figure 3-10). Click on the `+` button at the bottom.

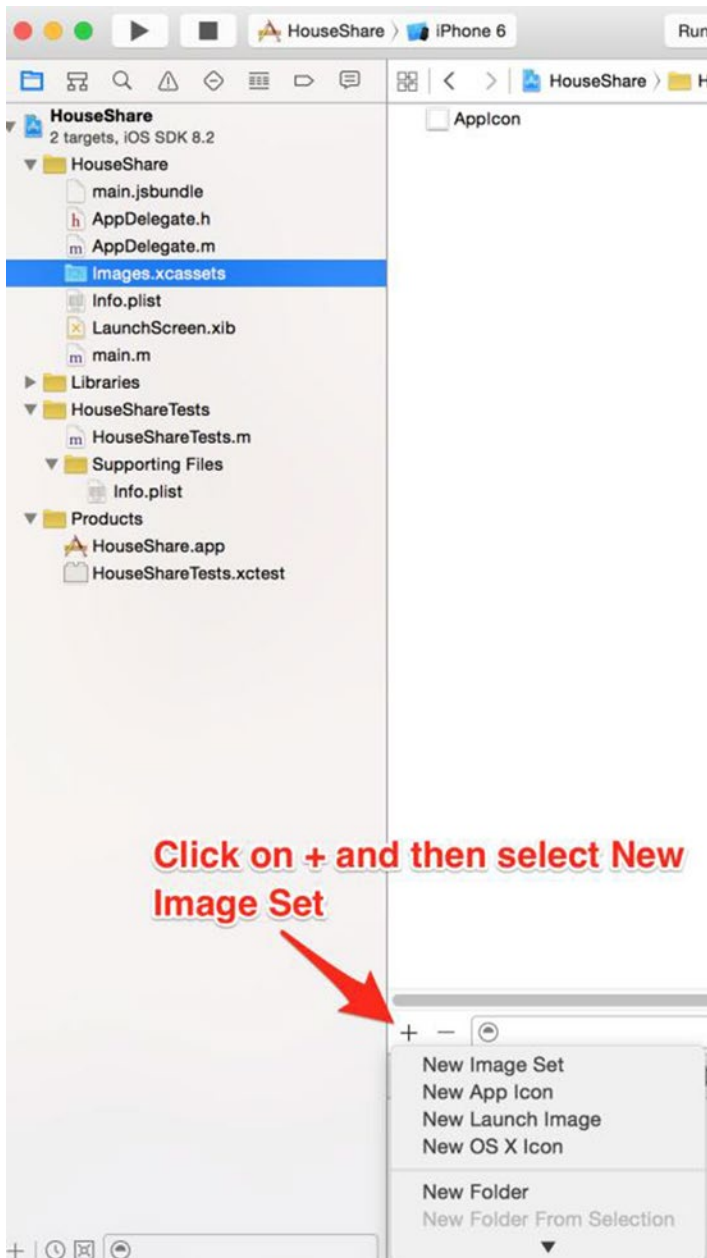


Figure 3-10. *Image.xcassets* in Xcode

Name the image set “home” and drag the home images to the boxes on the screen (see Figure 3-11).

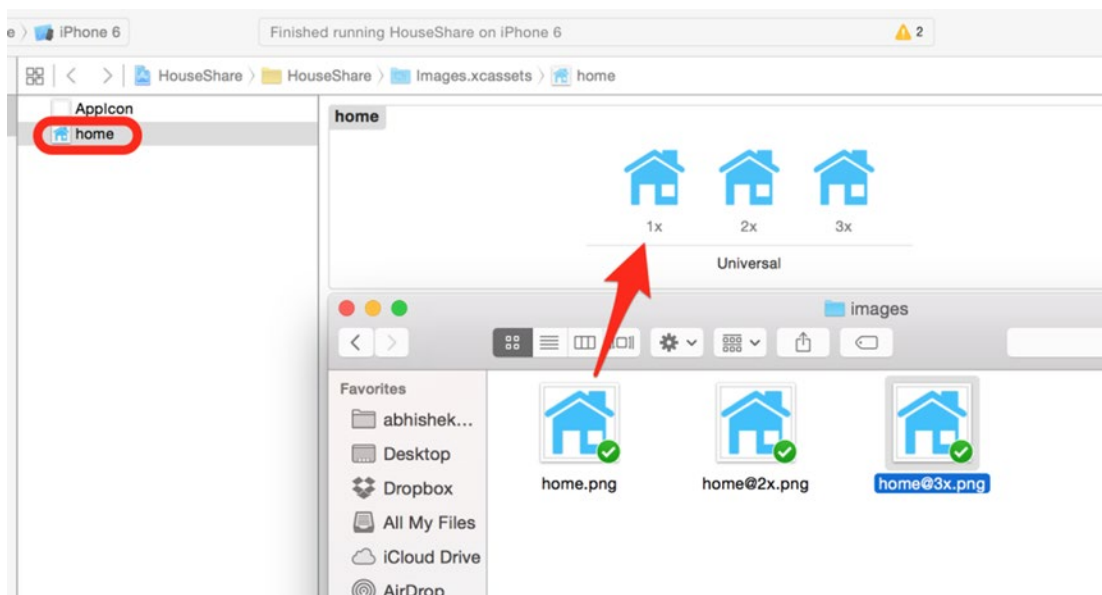


Figure 3-11. Image set “home”

The use of xcassets is the new standard after iOS 7. The asset catalog manages your app’s images; iOS has different image resolutions and it groups them together. When building, Xcode compiles this image catalog into the most efficient bundle for final distribution.

Now that we have added an image to our project, let’s add it to our component. First, add the Image component to our list of components:

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

Let’s now add the image to our view:

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

```

    <View style={styles.bottomBox} >
        <Image source={require('image!home')} style={styles.image}/>
    </View>
    <View style={styles.topBox} >
    </View>
</View>
);
}
});

```

`require('image!home')` is pointing to the asset catalog of the image with the name `home`.

Let's also specify the width and height for this image:

```

image: {
  width: 70,
  height: 70
},

```

Since we have made changes in Xcode, we have to restart the simulator. Stop the application and rebuild the code. We can see that the house image is shown on the screen (see Figure 3-12).

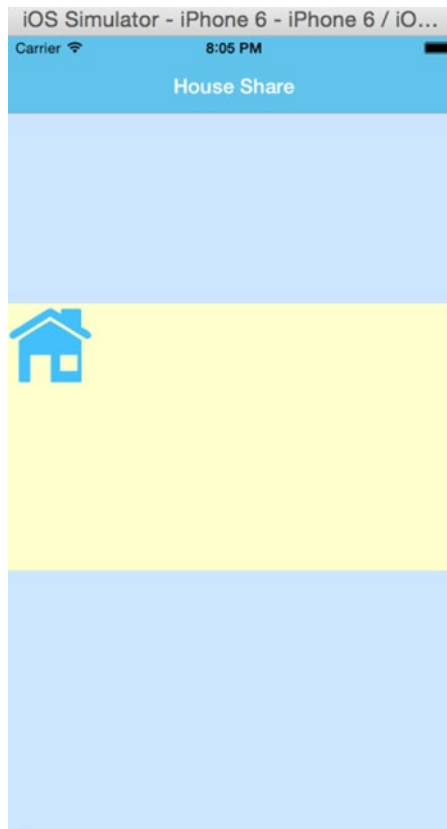


Figure 3-12. Now we have a house image

Now, let's add some styling to it by updating `bottomBox` with the following style:

```
bottomBox: {  
  flex: 1,  
  backgroundColor: '#FFFFCC',  
  alignItems: 'center',  
  justifyContent: 'center',  
},
```

`alignItems` and `justifyContent` define the default behavior for how flex items are laid out along the cross axis and main axis respectively. Since we want to show this in the center, we have updated these values to `center`.

Refresh the screen, and you will find that the house image is centered in the view (see Figure 3-13).

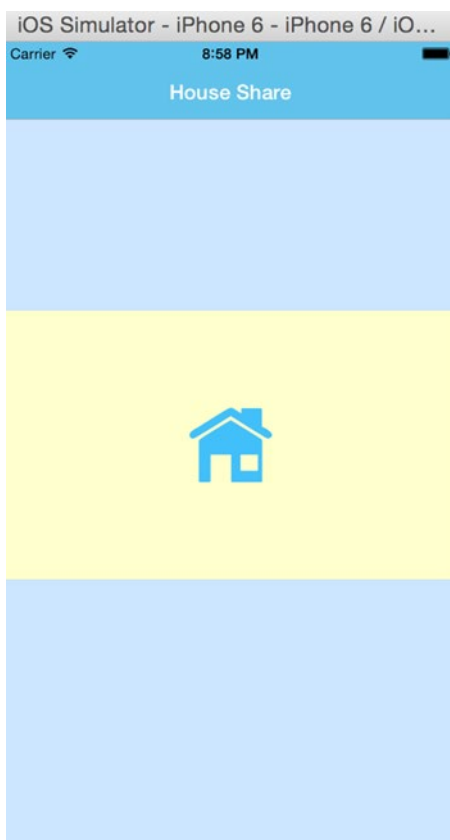


Figure 3-13. *The house is now centered*

We can also give any server image URL as the source, and the `Image` component will take care of loading it from the network. We will do this in a later part of this chapter.

TouchableHighlight

Touch is a fundamental way to interact with an iOS application. `TouchableHighlight` is a React Native component that helps us create buttons that give a proper response in the event of touch. These are not actually buttons, but the React Native team decided that it was easier to construct buttons directly in JavaScript instead of using `UIButton`. The buttons in your app use `TouchableHighlight`, which is not a button but works like a button. Let us understand this with an example.

Let's add the `TouchableHighlight` component to our code:

```
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  NavigatorIOS,
  TouchableHighlight
} = React;
```

Update the view and add two `TouchableHighlight` buttons:

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

        <View style={styles.topBox} >
          <TouchableHighlight style={styles.button}
            underlayColor='#99d9f4'>
            <Text style={styles.buttonText}>Show Houses</Text>
          </TouchableHighlight>
          <TouchableHighlight style={styles.button}
            underlayColor='#99d9f4'>
            <Text style={styles.buttonText}>Add House</Text>
          </TouchableHighlight>
        </View>
      </View>
    );
  }
});
```

And of course we need stylesheets for our buttons:

```
button: {
  flex: 1,
  backgroundColor: '#48BBEC',
  borderColor: '#48BBEC',
  borderWidth: 1,
```

```
borderRadius: 8,  
alignSelf: 'stretch',  
justifyContent: 'center',  
margin: 10  
},  
buttonText: {  
  fontSize: 18,  
  color: 'white',  
  alignSelf: 'center'  
}
```

Refresh the app in the iOS simulator. We will see two buttons on the screen in our third block (as shown in Figure 3-14).

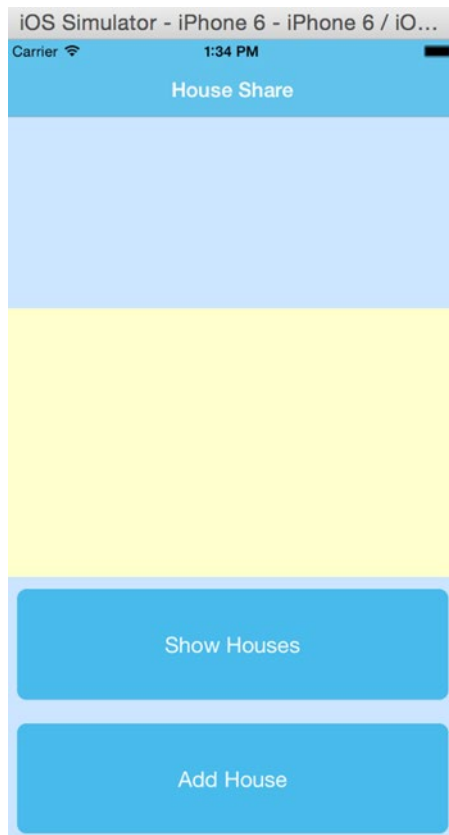


Figure 3-14. Screen with two buttons at bottom

Let's continue building our page by adding one more view in which to list the housing options. This will be done by clicking on the show house page, which will redirect to another component. Replace the following code for the Home component:

```
var Home = React.createClass({
  _handleListProperty: function() {
    console.log('Button clicked successfully');
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.topBox} />
        <View style={styles.bottomBox} />

        <View style={styles.topBox} >
          <TouchableHighlight
            style={styles.button}
            onPress= {this._handleListProperty}
            underlayColor='#99d9f4'
          >
            <Text style={styles.buttonText}>List properties</Text>
          </TouchableHighlight>
          <TouchableHighlight style={styles.button}
            underlayColor='#99d9f4'>
            <Text style={styles.buttonText}>Add House</Text>
          </TouchableHighlight>
        </View>
      </View>
    );
  }
});
```

Let us review what we have done here; we have added an `onPress` attribute to our `TouchableHighlight` component for the List Properties section. Whenever someone presses the List Properties button, it calls the function `_handleListProperty`.

If you build your application and open up the console in your developer tools, every time you click on the List Properties button you will see the message “Button clicked successfully” printed in the console.

Next, we will create our `ListProperty` component. But first, let's refactor our code and keep our components in separate files. Create an `App` folder that has a `Components` subfolder, which is where we will keep our React Native components. Inside the `Components` folder, create the file `Home.js` and place its respective components there. We will then require them in our `index.ios.js`, which will now look like this:

```
'use strict';
var React = require('react-native');
var Home = require('./App/Components/Home');
var {
  AppRegistry,
  StyleSheet,
```

```

    Text,
    View,
    NavigatorIOS,
    TouchableHighlight
  } = React;
  var mainView = React.createClass ({
    render: function() {
      return (

        <NavigatorIOS barTintColor='#48BBEC' titleTextColor= "#FFFFFF"
        style={styles.navigator} initialRoute={{
          title: 'House Share',
          component: Home
        }}/>
      );
    },
  });

  var styles = StyleSheet.create({
    navigator: {
      flex: 1
    }
  });
  AppRegistry.registerComponent('HouseShare', () => mainView);

```

Routing to a Component

In React Native, you will be building many components and will be routing back and forth between them. We must have a way to do that. In this section, we will learn to route from one component to another. Before we create our component `ListProperty`, we need to navigate the route to this component. This can be done by modifying our `_handleListProperty` function in the `Home` component. Replace `_handleListProperty` function with the following code at `./App/Components/Home.js`:

```

_handleListProperty: function() {
  this.props.navigator.push({
    title: "List Properties",
    component: ListProperty
  })
},

```

Here, `navigator.push` navigates forward to a new route; in this case it's `ListProperty`.

Let us now create a `ListProperty` component by creating a file `ListProperty.js` in the `./App/Components/` folder. Add the following code in your `ListProperty.js` file:

```

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

```

```
var ListProperty = React.createClass({
  render: function() {
    return (
      <View />
    );
  }
});
module.exports = ListProperty;
```

Refresh your application and click the List Properties button, which will bring you to the empty view shown in Figure 3-15.

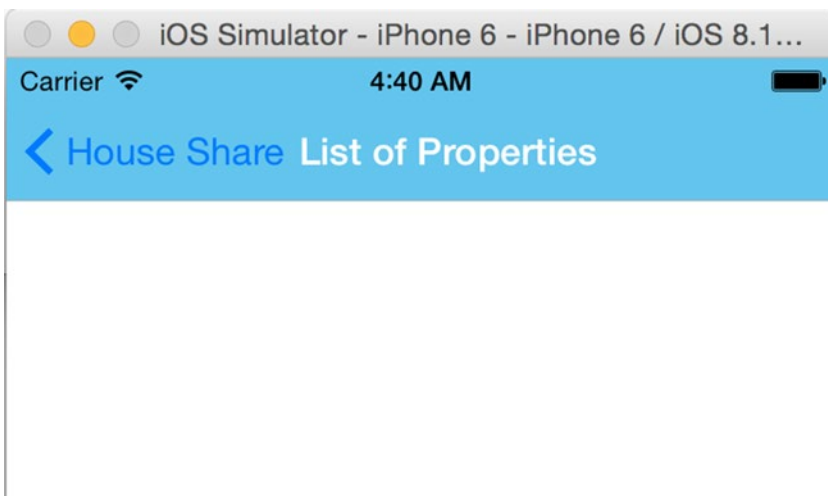


Figure 3-15. Empty view with traversing option

In left corner, you will see an option to traverse back to the previous component. NavigatorIOS maintains this back stack by which to go back to the previous component. Let's now work on this empty component. The idea is to create a table view having a list of properties, each of which has an image thumbnail to the left side. The rest of the details should appear next to it.

To keep things simple for this chapter, we will mock up data instead of pulling it from some external service (later, you will learn how to pull the same data from an external API). With this data, we will show the name of the property, its address, and a thumbnail picture. This will look something like Figure 3-16.



Figure 3-16. Property name and address

Add the following code to your ListProperty component in the file `./App/Components/ListProperty.js`:

```
var React = require('react-native');

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

var MOCK_DATA = [
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/ols'}},
];

var ListProperty = React.createClass({
  render: function() {
    var property = MOCK_DATA[0]
    return (
      <View style={styles.container}>
        <Image
          source={{uri: property.images.thumbnail}}
          style={styles.thumbnail}/>
        <View style={styles.rightContainer}>
          <Text style={styles.name}>{property.name}</Text>
          <Text style={styles.address}>{property.address}</Text>
        </View>
      </View>
    );
  }
});
```

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  thumbnail: {
    width: 53,
    height: 81,
  },
  rightContainer: {
    flex: 1,
  },
  name: {
    fontSize: 20,
    marginBottom: 8,
    textAlign: 'center',
  },
  address: {
    textAlign: 'center',
  },
});
module.exports = ListProperty;
```

Let's refresh our application in the iOS simulator to see the changes (see Figure 3-17).

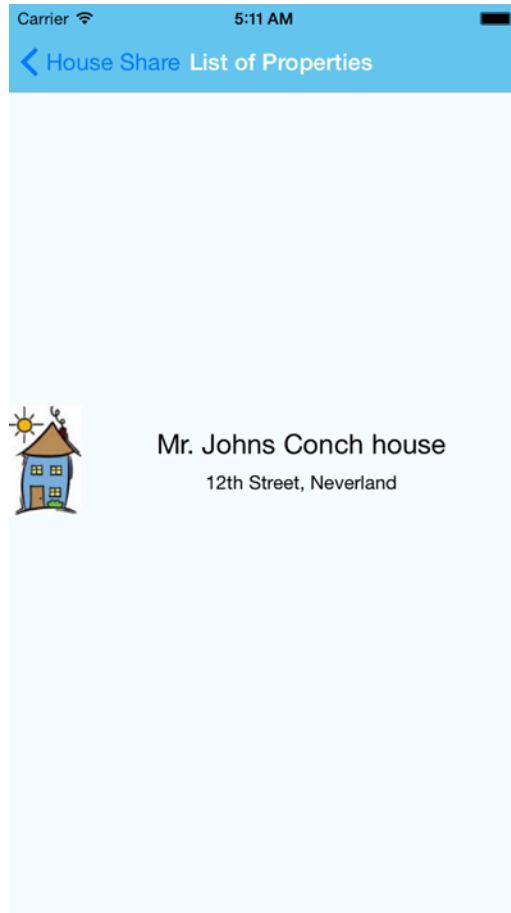


Figure 3-17. Thumbnail image with property name and address

Let us review what we have done here:

```
var React = require('react-native');

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

We first specified all components to be used in this section. We once again added the `Image` component, which will be used to load our image—not from a bundled image but, as promised earlier, from an image URL.

Next, we added some mock data:

```
var MOCK_DATA = [
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/ols'}},
];
```

Here, we added name, address, and image URL for our property thumbnail. We will be adding more values to this mock data as we move forward with the example.

Next, we created our ListProperty component:

```
var ListProperty = React.createClass({
  render: function() {
    var property = MOCK_DATA[0]
    return (
      <View style={styles.container}>
        <Image
          source={{uri: property.images.thumbnail}}
          style={styles.thumbnail}/>
        <View style={styles.rightContainer}>
          <Text style={styles.name}>{property.name}</Text>
          <Text style={styles.address}>{property.address}</Text>
        </View>
      </View>
    );
  }
});
```

Our ListProperty component has a variable property, which has data from the MOCK_DATA array. We then used a View component to create a view containing the thumbnail image within an Image component. In the Image component we have the property source, which can have the values http address, local file path, or the name of a static image resource.

Using a Text component we have specified the name and address from the mock data:

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  thumbnail: {
    width: 53,
    height: 81,
  },
  rightContainer: {
    flex: 1,
  },
});
```

```
name: {
  fontSize: 20,
  marginBottom: 8,
  textAlign: 'center',
},
address: {
  textAlign: 'center',
},
});

module.exports = ListProperty;
```

Finally, we added the styles and exported our `ListProperty` component.

ListView

In previous section, we populated one element of an array by specifying the index of the element. In this section, we will populate a list of data using `ListView`. Before we embark on that, let's learn a bit more about the `ListView` component.

A `ListView` is a component designed for populating vertically scrolling lists of dynamic data. The minimal steps are to create a `ListView` data source, populate it with an array of data similar to the native `TableView` data source, and then instantiate a `ListView` component with that data source and a `renderRow` callback. This takes a blob from the data array and returns a renderable component, which will be shown.

`ListView` looks very similar to `TableView`, but the implementation doesn't actually use `TableView`. Rather, it uses `ScrollView` behind the scenes. Features like swipe to delete, reordering, and so on can't be used directly through `ListView`.

Replace the following code in your `ListProperty.js`:

```
var React = require('react-native');

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

var MOCK_DATA = [
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/o15'}}},
  {name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
    'http://hmp.me/o16'}}},
  {name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
    'http://hmp.me/o17'}}},
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images:
    {thumbnail: 'http://hmp.me/o15'}}},
];
```

```
{name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
'http://hmp.me/ol6'}},
{name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
'http://hmp.me/ol7'}},
{name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
'http://hmp.me/ol5'}},
{name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
'http://hmp.me/ol6'}},
{name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
'http://hmp.me/ol7'}}
];

var ListProperty = React.createClass({

  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      dataSource: ds.cloneWithRows(MOCK_DATA),
    };
  },

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

  renderProperty: function(property) {
    return (
      <View style={styles.container}>
        <Image
          source={{uri: property.images.thumbnail}}
          style={styles.thumbnail}/>
        <View style={styles.rightContainer}>
          <Text style={styles.name}>{property.name}</Text>
          <Text style={styles.address}>{property.address}</Text>
        </View>
      </View>
    );
  },
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
```

```
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    thumbnail: {
        width: 63,
        height: 91,
    },
    rightContainer: {
        flex: 1,
    },
    name: {
        fontSize: 20,
        marginBottom: 8,
        textAlign: 'center',
    },
    address: {
        textAlign: 'center',
    },
    listView: {
        paddingTop: 20,
        backgroundColor: '#F5FCFF',
    },
});
```

```
module.exports = ListProperty;
```

Refresh your application in the iOS simulator to see the updated view, as shown in Figure 3-18.



Figure 3-18. Scrollable addresses

Great! Now we have a list of properties that we can scroll through. Let us review the implementation now:

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

We have once again specified what all components will be used for in this section. There is a new component added—`ListView`. If you have seen React Native documentation besides `ListView` to implement similar functionality, it may have been `ScrollView`. `ListView` is, however, much better than simply rendering all of these elements or putting them in a `ScrollView`. This is because, despite React Native being fast, rendering a very big list

of elements could be slow. Similar to `TableView`, `ListView` implements the rendering list of elements so that you only display the ones shown on screen; those already rendered but that are now off screen are removed from the native-view hierarchy, which makes the rendering smooth and fast.

Moving forward, we have updated our `MOCK_DATA`:

```
var MOCK_DATA =[
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/o15'}}},
  {name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
    'http://hmp.me/o16'}}},
  {name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
    'http://hmp.me/o17'}}},
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/o15'}}},
  {name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
    'http://hmp.me/o16'}}},
  {name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
    'http://hmp.me/o17'}}},
  {name: 'Mr. Johns Conch house', address: '12th Street, Neverland', images: {thumbnail:
    'http://hmp.me/o15'}}},
  {name: 'Mr. Pauls Mansion', address: '625, Sec-5, Ingsoc', images: {thumbnail:
    'http://hmp.me/o16'}}},
  {name: 'Mr. Nalwayas Villa', address: '11, Heights, Oceania', images: {thumbnail:
    'http://hmp.me/o17'}}}
];
```

In this code we added more entries in order to create a scrollable view. Now, let's look at the changes we made in our `ListProperty` component:

```
var ListProperty = React.createClass({
  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      dataSource: ds.cloneWithRows(MOCK_DATA),
    };
  },
  render: function() {
    return (
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderProperty}
        style={styles.listView}/>
    );
  },
  renderProperty: function(property) {
    return (
      <View style={styles.container}>
        <Image
          source={{uri: property.images.thumbnail}}
          style={styles.thumbnail}/>
      </View>
    );
  }
});
```

```
        <View style={styles.rightContainer}>
          <Text style={styles.name}>{property.name}</Text>
          <Text style={styles.address}>{property.address}</Text>
        </View>
      </View>
    );
  },
});
```

Here, we set the component specification of `getInitialState`:

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows(MOCK_DATA),
  };
},
```

`getInitialState` is invoked only once before the component is mounted. The return value will be used as the initial value of `this.state`.

Next, we modified the render function so that, once we have our data, it renders a `ListView` instead of a single entry:

```
render: function() {
  return (
    <ListView
      dataSource={this.state.dataSource}
      renderRow={this.renderProperty}
      style={styles.listView}/>
  );
},
renderProperty: function(property) {
  return (
    <View style={styles.container}>
      <Image
        source={{uri: property.images.thumbnail}}
        style={styles.thumbnail}/>
      <View style={styles.rightContainer}>
        <Text style={styles.name}>{property.name}</Text>
        <Text style={styles.address}>{property.address}</Text>
      </View>
    </View>
  );
},
});
```

You'll notice we used `dataSource` from `this.state`, which is already set up by `getInitialState`. We also called the `renderProperty` function to set up every row with an image, name, and address. Finally, we added some styles:

```
listView: {
  paddingTop: 20,
  backgroundColor: '#F5FCFF',
},
```

ScrollView

Although we are not using `ScrollView` in our House Share application, to populate a list we can use `ScrollView` just like we used `ListView`. `ScrollView` is one of the most versatile and useful controls in iOS, as it is a great way to list content that is greater in size than the screen size.

We can add a basic `ScrollView` by using the following code:

```
var scrollview = React.createClass({
  getInitialState: function() {
    return {
      values: values
    };
  },
  _renderRow: function(value, index) {
    return (
      <View
        style={styles.row}
        key={index}
      >
        <Text>{value + " <----- Slide the row "</Text>
      </View>
    )
  },
  render: function() {
    return (
      <View style={styles.container}>
        <ScrollView style={styles.outerScroll}>
          {this.state.values.map(this._renderRow, this)}
        </ScrollView>
      </View>
    );
  }
});
```

We will set up `getInitialState` for `ScrollView` with the following:

```
var values = [1,2,3,4]
```

We can map those values to the `_renderRow` function, which will just return a basic view, with some text. This is basic `ScrollView`; if we want to scroll horizontally and we want to lock that direction, we can do so with the following:

```
<ScrollView
  horizontal={true}
  directionalLockEnabled={true}
>
```

There are lots of other options available with `ScrollView`; for documentation and examples, you may visit the following URL: <https://facebook.github.io/react-native/docs/scrollview.html>.

Summary

In this chapter, we learned some of the fundamentals that are essential for creating a stunning user experience. We covered the following:

- `NavigatorIOS` for back-swipe functionality across app
- Flexbox layout model
- `TouchableHighlight`, a wrapper for making views respond properly to touches
- Routing to another component
- Using `ListView` for efficient scrolling of vertical lists
- Using `ScrollView` for listing content larger than the screen size.

In the next chapter, we will learn how to solve problem in a different way by using Flux. We will not only discover how to work with flux patterns, but also how it's different than the ubiquitous MVC pattern. We will also create a simple flux app with React and port it in React Native.

Flux: Solving Problems Differently

“Simplicity is prerequisite for reliability.”

—Dijkstra

Flux is an application architecture that Facebook introduced to the world and uses for building client-side applications. It complements React Native composable view components by utilizing a unidirectional data flow. It's more of a pattern than a proper framework, and one can start using Flux immediately without an excess load of code. Before we delve into the details, it is important to know of the popular pattern MVC, which is commonly used, and how it differs from Flux. In this chapter we will learn about the following topics:

- MVC pattern
- MVC problem
- Flux
- Flux deep dive
- Flux example with ReactJS
- Flux example with React Native

MVC Pattern

Historically, the MVC pattern separates code into three distinct parts: model, view, and controller. The main purpose of this pattern is to isolate the representation of information from user interaction. Let's understand each of these parts individually:

- *Model*: Manages the behavior and data of application
- *View*: Representation layer of the model in the user interface
- *Controller*: Takes user input and makes necessary manipulations to the model, causing the view to get updated

MVC Problem

MVC is a very popular pattern to design applications, but it comes with its own set of problems. The more complex your source code becomes, the more complex things get. Figure 4-1 shows the simplest implementation of MVC, which works pretty well with small applications. But as our application grows, it demands new features, so there should be room to accommodate more models and views.

MVC



Figure 4-1. Basic MVC implementation

Figure 4-2 shows what happens when the models and views increase.

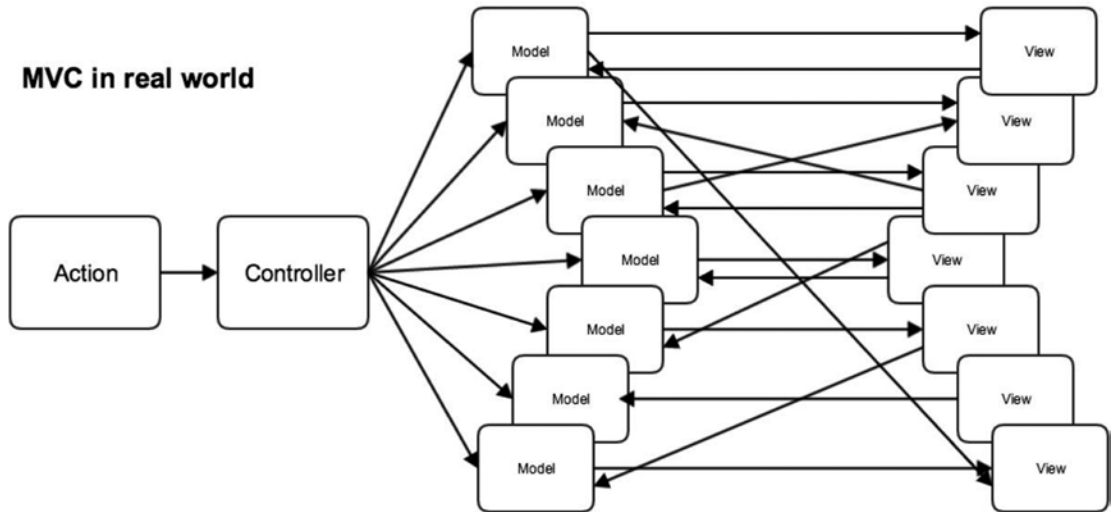


Figure 4-2. An explosion of arrows

With so many models and views interacting with each other, when we track a model it triggers a view, and the view triggers another model, and this goes on like spaghetti, which many times ends up in an infinite loop. The worst part is that it's really difficult to debug code in such a situation, which eventually makes the system fragile. Well, Facebook faced similar setbacks and solved this problem with a new pattern called Flux!

Flux

Flux abjures MVC in favor of a unidirectional data flow. Flux works well, since the single-direction data flow makes it easy to understand and modify an application as it grows and becomes more complicated. Earlier, we found that two-way data bindings lead to cascading updates, where change in one data model leads to updates in to another data model, making it very difficult to predict what would change as the result of a single user interaction. As shown in Figure 4-3, Flux applications have three major parts: the dispatcher, the stores, and the views (where we use React Native components). These should not be compared with the Model View Controller elements of the MVC pattern.

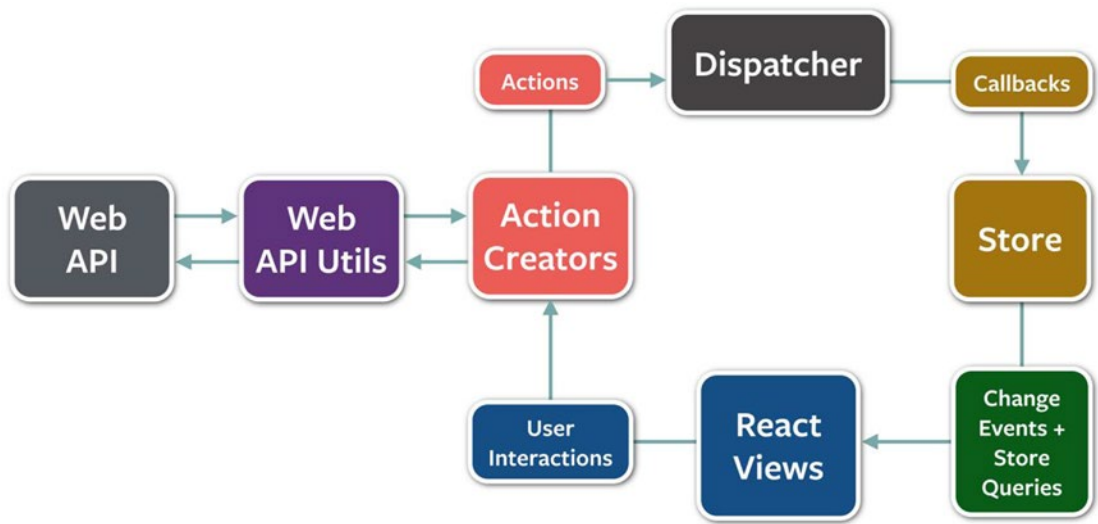


Figure 4-3. Three major components of Flux

Image source: <https://github.com/facebook/flux/blob/master/docs/img/flux-diagram-white-background.png>

Though controllers do exist in a Flux application, they are controller views, where views are found at the top of the hierarchy that retrieves data from the stores and forwards it to their children. Looking at Figure 4-4, the most important part of the Flux architecture is the dispatcher, which is a singleton that directs the flow of data and ensures that updates do not cascade.

FLUX

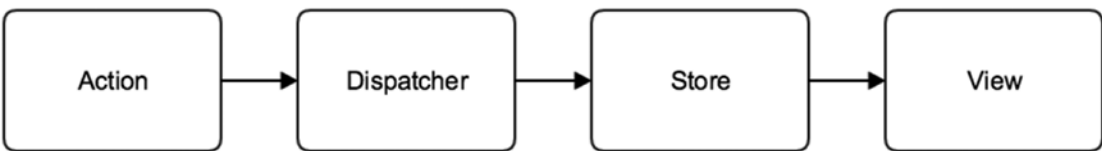


Figure 4-4. Dispatcher directs the flow of data

As an application grows, the dispatcher becomes more vital, as it is responsible for managing dependencies between stores by invoking the registered callbacks in a specific order.

When a user interacts with a React Native view, the view sends an action (usually represented as a JavaScript object with some fields) through the dispatcher, which notifies the various stores that hold the application’s data and business logic. When the stores change state, they notify the views that something has been updated. This works especially

well with React Native’s declarative model, which allows the stores to send updates without specifying how to transition views between states.

In short, Flux offers the following benefits:

- Improves data consistency
- Makes it easier to pinpoint the bugs
- Offers more meaningful unit tests; since all the states of a module are there in same place, we can test a module independently
- Contains predictable code

Success of Flux

One of Facebook’s most popular features was chat. However, it was extremely buggy and had the most negative user feedback. The new chat system that Facebook implemented uses Flux, and it now has a seamless experience; you can look at example chat code in a Facebook React Native example at the following URL:

<https://github.com/facebook/flux/tree/master/examples/flux-chat>.

Flux Deep Dive

In this section, we’ll take a closer look at some of Flux’s core concepts.

The Dispatcher

The dispatcher is the central hub that manages all data flow in a Flux application. It is essentially a registry of callbacks into the stores and has no real intelligence of its own; it is a simple mechanism for distributing the actions to the stores. Each store registers itself and provides a callback. When an action creator provides the dispatcher with a new action, all stores in the application receive the action via the callbacks in the registry. Dispatcher also acts like a traffic controller. If it gets an action while the data layer is still processing, it can reject the action, which is a good constraint to have. It is a guarantee that you will know when your action starts and what changes it makes to the data layer, as there are cascading effects that build up in between—you are indeed in full control of your system.

The Need for Dispatcher [`dispatch()` and `waitFor()`]

As an application grows, dependencies across different stores also increase. Imagine we have a situation where store A needs store B to update itself first, so that it can itself know how to update. We need the dispatcher to invoke the callback for store B, and finish that callback, before moving forward with store A. To assert this dependency, store A needs to communicate with the dispatcher to first complete the action updating store B. The dispatcher provides this functionality through the `waitFor()` method.

The `dispatch()` method provides a simple, synchronous iteration through the callbacks, invoking each in turn. When `waitFor()` is encountered within one of the callbacks, execution of that callback stops and `waitFor()` provides us with a new iteration cycle over the dependencies. After the entire set of dependencies has been fulfilled, the original callback then continues to execute.

Further, the `waitFor()` method may be used in different ways for different actions, within the same store's callback. In one case, store A might need to wait for store B. But in another case, it might need to wait for store C. Using `waitFor()` within the code block that is specific to an action allows us to have fine-grained control of these dependencies.

Problems arise, however, if we have circular dependencies. That is, if store A needs to wait for store B, and store B needs to wait for store A, we could wind up in an endless loop. The dispatcher now available in the Flux repo protects against this by throwing an informative error to alert the developer that this problem has occurred. The developer can then create a third store and resolve the circular dependency.

Stores

Stores contain the application state and logic. Their role is somewhat similar to that of a model in a traditional MVC, but they manage the state of many objects—they do not represent a single record of data like ORM models do. More than simply managing a collection of ORM-style objects, stores manage the application state for a particular domain within the application.

As mentioned earlier, a store registers itself with the dispatcher and provides it with a callback. This callback receives the action as a parameter. Within the store's registered callback, a `switch` statement based on the action's type is used to interpret the action and to provide the proper hooks into the store's internal methods. This allows an action to result in an update to the state of the store, via the dispatcher. After the stores are updated, they broadcast an event declaring that their state has changed, so the views may query the new state and update themselves.

Actions

When new data enters the system, whether through a person interacting with the application or through a web API call, that data is packaged into an *action*—an object literal containing the new fields of data and a specific action type. We often create a library of helper methods called *action creators* that not only create the action object, but also pass the action to the dispatcher.

Different actions are identified by a type attribute. When all of the stores receive the action, they typically use this attribute to determine if and how they should respond to it. In a Flux application, both stores and views control themselves; external objects do not act upon them. Actions flow into the stores through the callbacks they define and register, not through setter methods.

Letting the stores update themselves eliminates many entanglements typically found in MVC applications, where cascading updates between models can lead to unstable state and make accurate testing very difficult. The objects within a Flux application are highly decoupled and adhere very strongly to the Law of Demeter, the principle that each object within a system should know as little as possible about the other objects in the system. This results in software that is more maintainable, adaptable, testable, and easier for new engineering team members to understand.

Flux with ReactJS Example

In this section we will create a simple ReactJS application with Flux. We are keeping this example to a minimum with the purpose of understanding the theoretical explanation in the previous section via an application implementation.

You can refer to the `flux-with-react` source provided and follow along or you can create a brand new ReactJS application. Inside the root directory of `flux-with-react`, you will need to install the required packages.

This example is a web application in which we will be using couple of npm modules. Initialize your project with the following:

```
$ npm init
```

This will initialize your project; fill in the required information. Next, let's install some node modules, which will be helpful for our project:

```
$ npm install --save-dev react reactify object-assign gulp gulp-browserify gulp-concat  
es6-promise flux
```

Out of all of these packages, the most important ones are `eact`, which is the JavaScript library for ReactJS; `reactify`, which helps convert ES6 syntax to ES5 syntax constructs; `gulp`, which builds in system help for automating painful tasks during the development workflow; and lastly `flux`, Facebook's npm module to implement Flux architecture.

Next, create `gulpfile.js` and add the following tasks to it:

```
var gulp = require('gulp');  
var browserify = require('gulp-browserify');  
var concat = require('gulp-concat');  
  
gulp.task('browserify', function() {  
  gulp.src('src/js/main.js')  
    .pipe(browserify({transform:'reactify'}))  
    .pipe(concat('main.js'))  
    .pipe(gulp.dest('dist/js'));  
});
```

```
gulp.task('copy', function() {
  gulp.src('src/index.html')
    .pipe(gulp.dest('dist'));
});

gulp.task('default',['browserify', 'copy'], function() {
  return gulp.watch('src/**/*.*', ['browserify', 'copy'])
});
```

Here, we have two tasks: `browserify` and `copy`. What `browserify` does is grab, transform, and concatenate `main.js` (where our application code resides) and place it in our distribution folder, which is `dist`. The next task, `copy`, copies `index.html` to the `dist` folder.

We have another default task which runs both these tasks and also watches continuously to see if there are any changes happening to files related to these tasks.

Next, let's create our folder structure, which should support a Flux architecture. Create the folder structure shown in Figure 4-5 with empty files inside your root folder:



Figure 4-5. New Flux file structure

Now, let's begin creating our application from scratch. Add the following code in your file `index.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Flux example with ReactJS</title>
  </head>
  <body>
    <div id="main" class="container"></div>
    <script src="js/main.js"></script>
  </body>
</html>
```

Not much is happening in this file; it is the html file where we load our `main.js`, which is our entry point to all our JavaScript code. The id `main` is the target where our component will be loaded. Next, use the following code in `src/main.js`:

```
var React = require('react');
var App = require('./components/app');

React.render(
  <App />,
  document.getElementById('main')
);
```

Here in `src/main.js`, we use the variable `React` to grab `react`, and we use the variable `App` to grab our component from `./components/app.js`. We render the `App` component—where all the magic of Flux is happening—on the target id `main`. Next, we create our component by adding the following code in `src/components/app.js`:

```
var React = require('react');
var AppActions = require('../actions/app-actions');
var AppStore = require('../stores/app-store');

var AppTitle = React.createClass({
  onClickHandler:function(){
    AppActions.addElement('element added');
  },
  render:function(){
    return (
      <div className="wrapper">
        <h3 onClick={this.onClickHandler}>Open console and Click Here </h3>
      </div>
    )
  }
});

module.exports = AppTitle;
```

Once again, we have few require statements that grab react, actions, and stores. Next, using `createClass` we create an `AppTitle` component that has a render method with a `div`, which has an `h3` tag with an `onClick` event. We also have another function, `onClickHandler`, which is fired with an `onClick` event whenever the `h3` tag is clicked.

Let's take a closer look at the `onClickHandler` function:

```
onClickHandler:function(){
  AppActions.addElement('element added');
},
```

The `AppActions` here is a fire and forget, which has one action to perform, `addElement` (which we will investigate when we get into actions), and some data to pass along. Let's now open `src/actions/app-actions.js` and paste the following code in it:

```
var AppDispatcher = require('../dispatcher/app-dispatcher');
var AppConstants = require('../constants/app-constants');

var AppActions = {
  addElement: function(param){
    AppDispatcher.handleAction({
      actionType:AppConstants.ADD_ELEMENT,
      description: param
    })
  }
}
module.exports = AppActions
```

Once again we have some required items in the beginning—dispatcher and some constants. If you remember, in our `App` component code we called the `addElement` function inside `onClickHandler`. That function is described here:

```
addElement: function(param){
  AppDispatcher.handleAction({
    actionType:AppConstants.ADD_ELEMENT,
    description: param
  })
}
```

The `addElement` function has an object description that it receives as a parameter and is then passed to the dispatcher. We also have an `actionType` described here, which we received from `AppConstants`. In `src/constants/app-constants.js` add the following code:

```
module.exports = {
  ADD_ELEMENT: 'ADD_ELEMENT'
};
```

Here, we have created a few constants that can be reused across the application. Let's move to dispatcher and add following code to `src/dispatcher/app-dispatcher.js`:

```
var Dispatcher = require('flux').Dispatcher;
var assign = require('object-assign');

var AppDispatcher = assign(new Dispatcher(), {
  handleAction: function(action) {
    this.dispatch({
      source: 'TEST_ACTION',
      action: action
    });
  }
});

module.exports = AppDispatcher;
```

Here, we have used the `Dispatcher` variable to wrap dispatcher from the `flux` module. If you remember, in `app-actions.js` we had called the function `handleAction`; here we have described what it does. From the action we had to send an object, which is here used as a parameter, which is further wrapped and given context as `TEST_ACTION`. This is then broadcasted using `this.dispatch`. This broadcasted object is listened to by stores. Let's add the following code to `src/stores/app-stores.js`:

```
var AppDispatcher = require('../dispatcher/app-dispatcher');
var EventEmitter = require('events').EventEmitter;
var assign = require('object-assign');

var CHANGE_EVENT = 'change';

var AppStore = assign({}, EventEmitter.prototype, {
  emitChange: function() {
    this.emit(CHANGE_EVENT);
  }
});

AppDispatcher.register(function(payload){
  console.log(payload);
  return true;
});

module.exports = AppStore;
```

To get started with stores, we will use node's `EventEmitter`. We need an `EventEmitter` to broadcast the change event to our control views. However, in our example, even if we keep our `emit` function empty, it won't work since there is nothing changing on our control views. Let's now review the important part here: where all the stuff is happening:

```
AppDispatcher.register(function(payload){
  console.log(payload);
  return true;
});
```

The store registers to `AppDispatcher` and takes the payload—which is the object broadcasted by the dispatcher. In this case, we log this payload on the console, so that every time you click on the `h3` tag on screen you will see the payload logged on the console. Let's create a build:

```
$ gulp
```

Open `dist/index.html` along with your developer tools in your browser. Click “Open console and Click Here” and you will see the result shown in Figure 4-6.

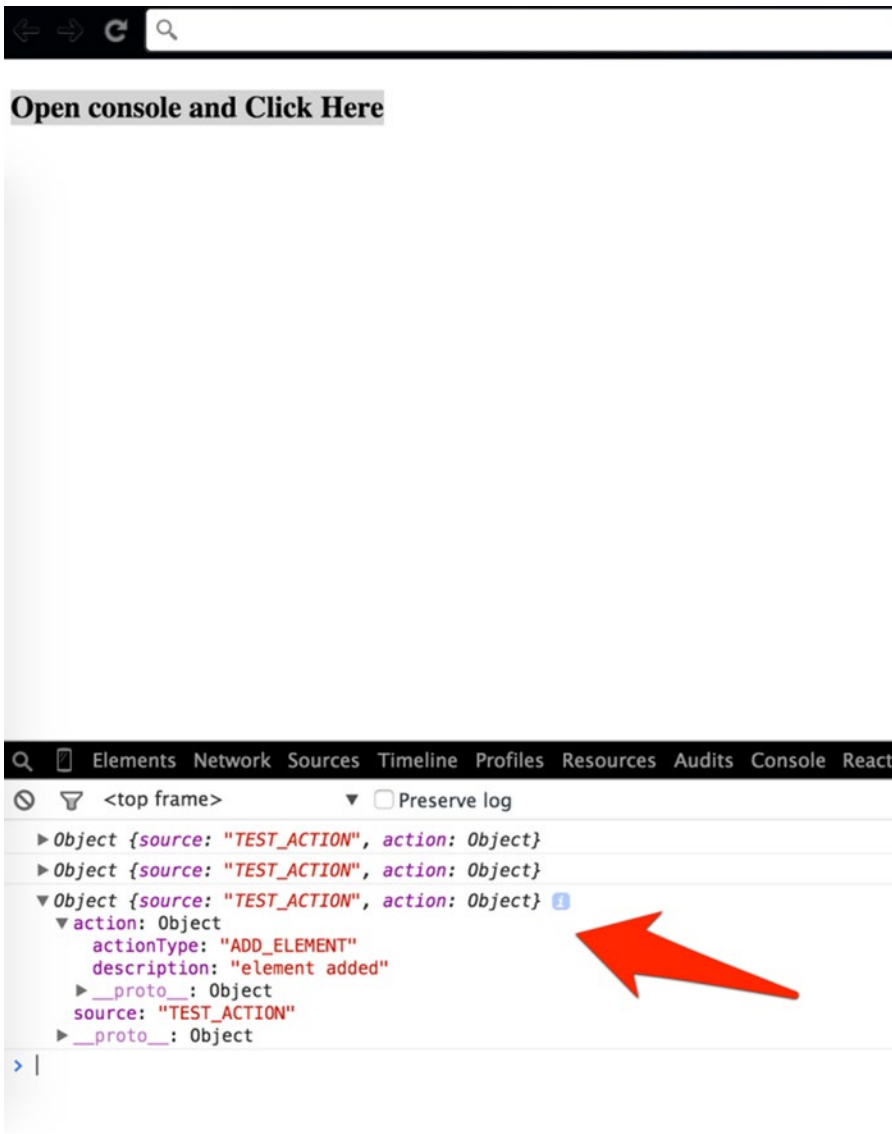


Figure 4-6. Result of `$ gulp build`

Now that we have a good understanding of Flux and how it works with React it's time to use all of this knowledge to implement the same task with React Native.

Flux with React Native Example

In the previous section, we learned how to use Flux with ReactJS. The beauty of React Native is that we can use the concepts we learned with ReactJS to implement similar tasks for an iOS application with React Native and Flux. In this section, we will create a classic ToDo application with Flux and React Native.

To begin, let's set up our environment by creating a new React Native application:

```
$ react-native init FluxTodo
```

Although we have named our application FluxTodo, it does not mean we are even close to building a Flux app. We need to add some npm modules in order to begin building our app. Let's add the following node modules to our application:

```
$ npm install --save-dev events key-mirror object-assign flux
```

Next, in order to keep things simple, we will keep all of our code within `index.ios.js`; however, it's a good practice to keep actions, dispatcher, stores and components separately.

Let's begin by requiring a dispatcher and grabbing a few important modules that are necessary for our application. Add the following code in `index.ios.js`:

```
'use strict';
var React = require('react-native');
var ReactPropTypes = React.PropTypes;
var assign = require('object-assign');
var keyMirror = require('key-mirror');
var EventEmitter = require('events').EventEmitter;

var Dispatcher = require('flux').Dispatcher;
var AppDispatcher = new Dispatcher();
```

Here, we have grabbed dispatcher and created a new instance, AppDispatcher, which we will later use with our actions.

Next, add the constants we will be using in our application:

```
var TodoConstants = keyMirror({
  TODO_CREATE: null,
  TODO_COMPLETE: null,
  TODO_DESTROY: null,
  TODO_UNDO_COMPLETE: null
});
```

Let's now add actions for our Todo application to `index.ios.js`:

```
var TodoActions = {
  create: function(text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },
  toggleComplete: function(todo) {
    var id = todo.id;
    if (todo.complete) {
      AppDispatcher.dispatch({
        actionType: TodoConstants.TODO_UNDO_COMPLETE,
        id: id
      });
    } else {
      AppDispatcher.dispatch({
        actionType: TodoConstants.TODO_COMPLETE,
        id: id
      });
    }
  },
  destroy: function(id) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_DESTROY,
      id: id
    });
  }
};
```

Here we have created `TodoActions`, which is a function to add new entries to your to do list by assigning an `actionType=TodoConstants.TODO_CREATE`. Similarly, we have `actionTypes` for `TODO_UNDO_COMPLETE`, `TODO_COMPLETE`, and `TODO_DESTROY`. Let us now add logic for these operations using the store. Add the following code to `index.ios.js`:

```
var CHANGE_EVENT = 'change';

var _todos = {};

function create(text) {
  var id = (+new Date() + Math.floor(Math.random() * 999999)).toString(36);
  _todos[id] = {
    id: id,
    complete: false,
    text: text
  };
}

function update(id, updates) {
  _todos[id] = assign({}, _todos[id], updates);
}
```

```
function destroy(id) {
  delete _todos[id];
}
var TodoStore = assign({}, EventEmitter.prototype, {
  getAll: function() {
    var todos = [];
    for(var key in _todos) {
      todos.push(_todos[key]);
    }
    return todos;
  },
  emitChange: function() {
    this.emit(CHANGE_EVENT);
  },
  addChangeListener: function(callback) {
    this.on(CHANGE_EVENT, callback);
  },
  removeChangeListener: function(callback) {
    this.removeListener(CHANGE_EVENT, callback);
  }
});
```

```
AppDispatcher.register(function(action) {
  var text;
  switch(action.actionType) {
    case TodoConstants.TODO_CREATE:
      text = action.text.trim();
      if (text !== '') {
        create(text);
        TodoStore.emitChange();
      }
      break;
    case TodoConstants.TODO_UNDO_COMPLETE:
      update(action.id, {complete: false});
      TodoStore.emitChange();
      break;
    case TodoConstants.TODO_COMPLETE:
      update(action.id, {complete: true});
      TodoStore.emitChange();
      break;
    case TodoConstants.TODO_DESTROY:
      destroy(action.id);
      TodoStore.emitChange();
      break;
    default:
  }
});
```

Finally, let's add our components and some styling. Also register the MainSection component:

```
var MainSection = React.createClass({
  propTypes: {
    todos: React.PropTypes.object.isRequired
  },
  render: function() {
    return (
      <View>
        <ListView dataSource={this.props.todos} renderRow={this.renderItem} />
      </View>
    );
  },
  renderItem: function(todo) {
    return <TodoItem todo={todo} />;
  }
});

var TodoItem = React.createClass({
  render: function() {
    var todo = this.props.todo;
    var todoItemStyle;
    todoItemStyle = (todo.complete) ? styles.TODO_ITEM_DONE : styles.TODO_ITEM;
    return (
      <View style={todoItemStyle}>
        <Text style={styles.TEXT}>{todo.text}</Text>
        <Text onPress={() => this._onToggleComplete(todo)}>[Complete]</Text>
        <Text onPress={() => this._onDestroy(todo)}>[Delete]</Text>
      </View>
    );
  },
  _onToggleComplete: function(todo) {
    TodoActions.toggleComplete(todo);
  },
  _onDestroy: function(todo) {
    TodoActions.destroy(todo.id);
  }
});

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

```

var TodoTextInput = React.createClass({
  getInitialState: function() {
    return {
      value: ''
    }
  },
  render: function() {
    return (
      <View>
        <TextInput
          style={styles.TODOTextInput}
          onChangeText={(text) => this.setState({value: text})}
          onBlur={this._save}
          placeholder={'What needs to be done?'}
          value={this.state.value}
        />
      </View>
    );
  },
  _save: function() {
    var text = this.state.value;
    if(text) TodoActions.create(text);
    this.setState({
      value: ''
    });
  }
});

```

And, finally, let's add styles:

```

var styles = StyleSheet.create({
  TodoApp: {
    padding: 20,
    paddingTop: 40
  },
  TodoItem: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF',
    height: 58
  },
  TodoItemDone: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF',
    height: 58,
    opacity: .3
  },
});

```

```
text: {
  flex: 1,
  textAlign: 'left',
  fontSize: 16
},
TodoTextInput: {
  height: 40,
  backgroundColor: '#EEEEEE',
  padding: 10,
  fontSize: 16
}
});
```

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

It's time to build our application in Xcode to test what we have done. Fire up your Xcode and build FluxTodo application. You will see following screen as shown in Figure 4-7.

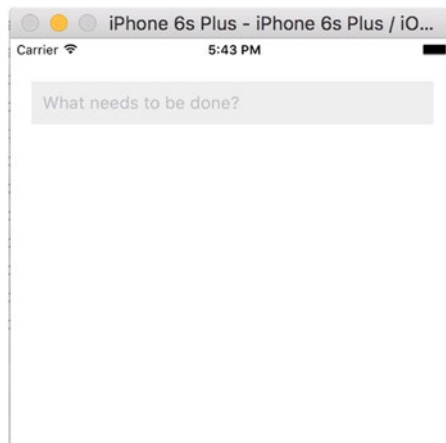


Figure 4-7. MainSection of the todo application to add todo tasks

We can add any todo task in the text box which will be immediately populated below the text box as shown in Figure 4-8.

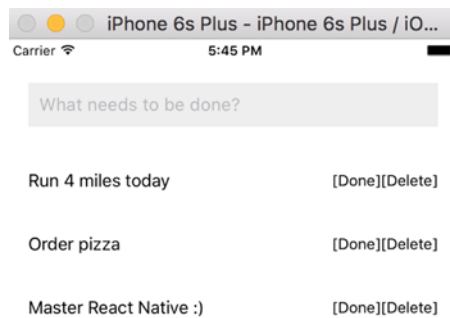


Figure 4-8. *Populated the todo list*

We also have the option to mark a task as done or delete it. If a task is deleted, it is removed from the list and when a task is marked as done it gets faded as shown in Figure 4-9.

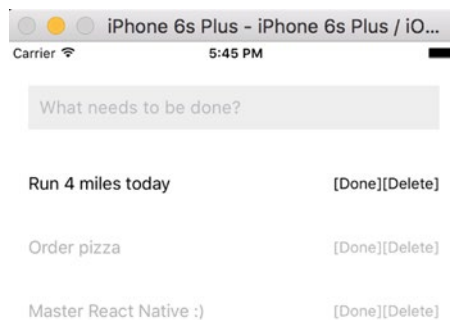


Figure 4-9. *The done and deletes tasks from the populated list*

Summary

In this chapter, we have learned about Flux patterns and how Flux differs from and solves problems fundamentally differently than traditional MVC patterns. We also deep-dove into Flux core concepts, followed by looking at examples of using it in a project. We concluded by creating a React Native application with Flux.

Chapter 5

Device Capabilities

An iOS device is not just limited to making phone calls; it's one of the most advanced pieces of machinery ever invented. And the real power lies in the various device capabilities. In this chapter we will learn about the following device capabilities:

- `Geolocation`
- `AsyncStorage`
- `Native Alert`
- `WebView`
- `Animations`

Geolocation

In this section, you will learn how to use iOS location services with a React Native application. Location services are used very often in many popular apps, especially in travel, navigation, cab sharing and the list is endless. This single feature significantly improves the user experience and the silver lining is it's very easily implemented.

The Geolocation property is enabled in React Native by default when we setup a project with `react-native init`. Let's create an application to implement this capability:

```
$react-native init GeolocationMap
```

Update `index.ios.js` with the following code:

```
'use strict';

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



```
Text,
View,
MapView
} = React;

var GeoLocationMap = React.createClass({
  getInitialState: function() {
    return {
      region: {
        latitude: 40.712784,
        longitude: -74.005941,
        latitudeDelta: 10,
        longitudeDelta: 10,
      },
    };
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.desc}>
          <Text> Maps </Text>
        </View>
        <MapView
          style={styles.map}
          region={this.state.region}
        />
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: '#F5FCFF',
    alignItems: 'stretch'
  },
  desc: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center'
  },
  map: {
    flex: 5,
  },
});

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

Now build your application in Xcode for the first time to see the results shown in Figure 5-1.



Figure 5-1. GeoLocation MapView

Reviewing the GeoLocationMap Code

Let's now understand what we have done in this part of our program.

```
var {  
  AppRegistry,  
  StyleSheet,  
  Text,  
  View,  
  MapView  
} = React;
```

Here we are first including the React Native component `MapView`, which is pivotal to accessing the location services. Next we create our Geolocation component which will utilize `MapView`:

```
var GeoLocationMap = React.createClass({
  getInitialState: function() {
    return {
      region: {
        latitude: 40.712784,
        longitude: -74.005941,
        latitudeDelta: 10,
        longitudeDelta: 10,
      },
    };
  },
  render: function() {
    return (
      <View style={styles.container}>
        <View style={styles.desc}>
          <Text> Maps </Text>
        </View>
        <MapView
          style={styles.map}
          region={this.state.region}
        />
      </View>
    );
  }
});
```

Here, we have set the initial state for the region with certain latitude and longitude parameters, which will be later set when we render the function with the `MapView` component. In the `MapView` component, we are using the `region` prop, which is supplied with latitude, longitude, `longitudeDelta` and `latitudeDelta`. These should always be numbers(integer or float) and they help us plot a specific region on the map. Finally we have added some style with `Flex` and registered our component.

Adding Annotation on Map

Next, let's add annotations to our application, and update `getInitialState` with the new state annotations, having parameters like latitude, longitude, title and subtitle:

```
getInitialState: function() {
  return {
    region: {
      latitude: 40.712784,
      longitude: -74.005941,
      latitudeDelta: 10,
      longitudeDelta: 10,
    },
  },
}
```

```
        annotations: [{
            latitude: 40.72052634,
            longitude: -73.97686958312988,
            title: 'New York',
            subtitle: 'This is cool!'
        }],
    };
},
```

Now update the MapView component with the new prop called *annotations*:

```
<MapView
    style={styles.map}
    region={this.state.region}
    annotations= {this.state.annotations}
/>
```

Besides the mentioned values, the annotations prop can have the following additional parameters:

```
annotations [{latitude: number, longitude: number, animateDrop: bool, title: string,
subtitle: string, hasLeftCallout: bool, hasRightCallout: bool, onLeftCalloutPress: function,
onRightCalloutPress: function, id: string}] #
```

Let's refresh and see the changes shown in Figure 5-2.



Figure 5-2. MapView with added parameters

Displaying the Latitude and Longitude of the Present Location

In this final part of our Geolocation application, we will display our present latitude and longitude on the view. In the previous example, we had a constant location; in this part, we will fly to our current location in real-time. Now that sounds like something exciting, so let's start building it. There are two ways to check for the current location on our maps. One is to simply add `showsUserLocation={true}` to your `MapView` component.

Another way is to use `NSLocationWhenInUseUsageDescription` geolocation. We will need to update `info.plist` to add this key, but it is enabled by default when you create a project with `react-native init`.

Replace your `index.ios.js` with the following code:

```
'use strict';

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

```

    View,
    MapView
  } = React;

var GeoLocationMap = React.createClass({
  getInitialState: function() {
    return {
      region: {
        latitude: 40.712784,
        longitude: -74.005941,
        latitudeDelta: 10,
        longitudeDelta: 10,
      },
      annotations: [{
        latitude: 40.72052634,
        longitude: -73.97686958312988,
        title: 'New York',
        subtitle: 'This is cool!'
      }],
    };
  },
  componentDidMount: function() {
    navigator.geolocation.getCurrentPosition(
      (initialPosition) => this.setState({initialPosition}),
      (error) => alert(error.message),
      {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
    );
    this.watchID = navigator.geolocation.watchPosition((lastPosition) => {
      this.setState({latitude: lastPosition.coords.latitude});
      this.setState({longitude: lastPosition.coords.longitude});
      var newRegion = {
        latitude: lastPosition.coords.latitude,
        longitude: lastPosition.coords.longitude,
        latitudeDelta: 10,
        longitudeDelta: 10,
      }
      this.setState({ region: newRegion});

      this.setState({ annotations: [{
        latitude: lastPosition.coords.latitude,
        longitude: lastPosition.coords.longitude,
        title: 'Current Location',
        subtitle: 'You are here'
      }]});
    });
  },
});

```

```
componentWillUnmount: function() {
  navigator.geolocation.clearWatch(this.watchID);
},
render: function() {
  return (
    <View style={styles.container}>
      <View style={styles.desc}>
        <Text>
          latitude: {this.state.latitude}
        </Text>
        <Text>
          longitude: {this.state.longitude}
        </Text>
      </View>
      <MapView
        style={styles.map}
        region={this.state.region}
        annotations= {this.state.annotations}
      />
    </View>
  );
}
```

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: '#F5FCFF',
    alignItems: 'stretch'
  },
  desc: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center'
  },
  map: {
    flex: 5,
  },
});
```

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

Now build our application to load it on the iOS simulator shown in Figure 5-3.

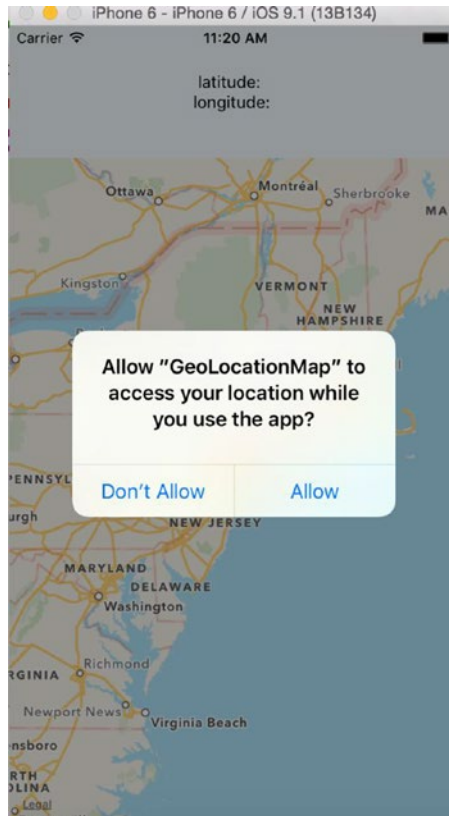


Figure 5-3. Access location prompt

If we allow this request, we will fly to the location we mentioned in our code; in this case it's California (Figure 5-4).

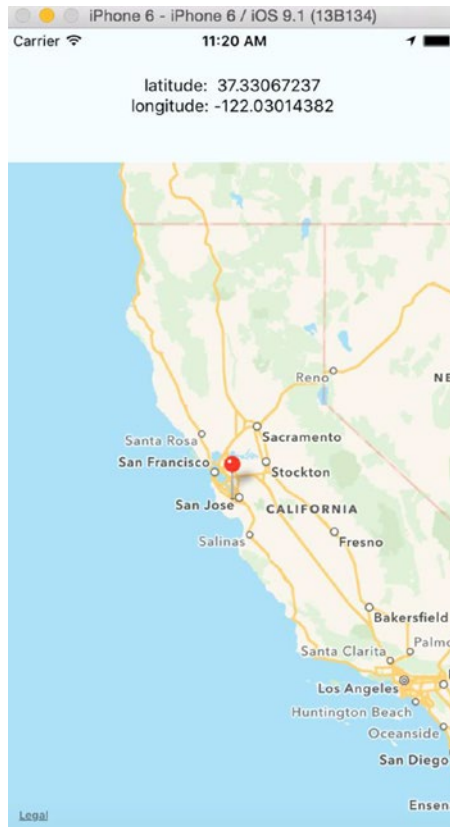


Figure 5-4. Fly to a specified location in the code

There is one extra step when you debug a GeoLocation application. As mentioned, in this example we will fly to our present location. We have saved that requirement so that it can be accomplished now. Open the Debug menu using Ctrl + Command + z and select the “Debug in Chrome” option as shown in Figure 5-5.

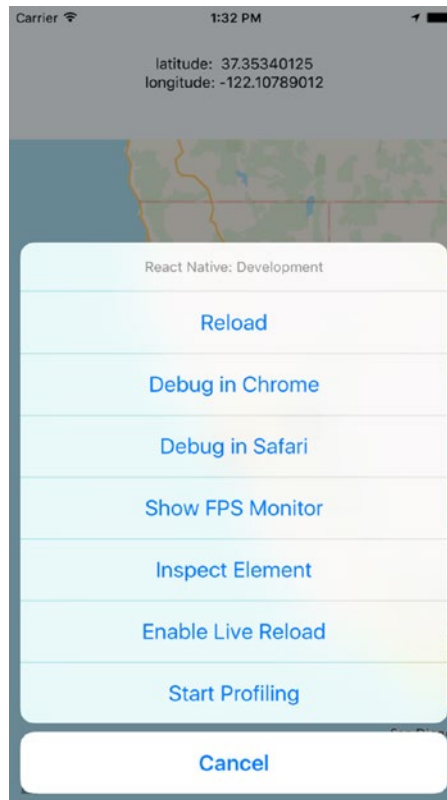


Figure 5-5. Select debug in chrome option

Once you click “Debug in Chrome” you will get a popup to allow your location to be served to the iOS simulator application as shown in Figure 5-6.

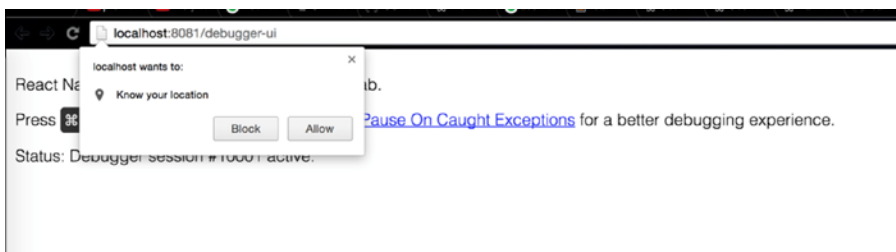


Figure 5-6. Allow the present location to be shared with the iOS simulator

Once you allow this, you will move to your current location in the app(Figure 5-7).



Figure 5-7. Current location is now shown

Reviewing the GeoLocation Code for the Present Location

Let's review what we have programmed in this example.

```
componentDidMount: function() {
  navigator.geolocation.getCurrentPosition(
    (initialPosition) => this.setState({initialPosition}),
    (error) => alert(error.message),
    {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
  );
  this.watchID = navigator.geolocation.watchPosition((lastPosition) => {
    this.setState({latitude: lastPosition.coords.latitude});
    this.setState({longitude: lastPosition.coords.longitude});
    var newRegion = {
      latitude: lastPosition.coords.latitude,
      longitude: lastPosition.coords.longitude,
      latitudeDelta: 10,
      longitudeDelta: 10,
    }
  });
}
```

Here, in `componentDidMount`, we first get the current position; by default, the iOS simulator gives us a default value which you can change using `debug ► location`. If `navigator.geolocation.getCurrentPosition` does get the value, then we set various states and their parameters. In this example, we have latitude and longitude description which are at the top of the view and are set using `this.setState({latitude :})` and `this.setState({longitude:})`. And the map below it is plotted with `newRegion`.

```

    this.setState({ region: newRegion});

    this.setState({ annotations: [{
      latitude: lastPosition.coords.latitude,
      longitude: lastPosition.coords.longitude,
      title: 'Current Location',
      subtitle: 'You are here'
    }]});

  });
},

```

We use React's superb power to rerender the component whenever the state changes. With the above code snippet, we will rerender `newRegion` with updated coordinates.

```

componentWillUnmount: function() {
  navigator.geolocation.clearWatch(this.watchID);
},

```

With `componentWillUnmount`, we will clear `navigator.geolocation` in case we move to some other part of the application.

```

render: function() {
  return (
    <View style={styles.container}>
      <View style={styles.desc}>
        <Text>
          latitude: {this.state.latitude}
        </Text>
        <Text>
          longitude: {this.state.longitude}
        </Text>
      </View>
      <MapView
        style={styles.map}
        region={this.state.region}
        annotations= {this.state.annotations}
      />
    </View>
  );
}
});

```

Finally, we render our components on the view with the latitude and longitude description having its respective style and the updated MapView component with its respective style, region mapping and annotations.

AsyncStorage

AsyncStorage is a key-value based storage system. It can be easily implemented and is globally available to the app. This persistence system is simple and asynchronous, and also a recommended way to store data.

Let's create an AsyncStorage example application; to do so, execute the following command:

```
$react-native init AsyncStorage
```

Great now lets add the following code in our index.ios.js

```
'use strict';

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

var STORAGE_KEY = '@AsyncStorageExample:someKey';

var AsyncStorageExample = React.createClass({
  getInitialState: function() {
    return {
      messages: '',
      textInputMessage: ''
    };
  },
  componentDidMount() {
    AsyncStorage.getItem(STORAGE_KEY).then((value) => {
      this.addMessage(value);
    }).done();
  },
  addMessage(message) {
    this.setState({messages: message});
  },
```

```

updateStorage () {
  AsyncStorage.setItem(STORAGE_KEY, this.state.textInputMessage);
  AsyncStorage.getItem(STORAGE_KEY).then((value) => {
    this.addMessage(value);
  }).done();
},

render: function() {
  return (
    <View style={styles.container}>
      <View style={styles.form}>
        <TextInput style={styles.textField}
          onChangeText={({textInputMessage}) => this.setState({textInputMessage})}
          value={this.state.textInputMessage}/>
        <TouchableHighlight style={styles.button} onPress={this.updateStorage}>
          <Text> Update Storage </Text>
        </TouchableHighlight>
      </View>
      <View style={styles.message}>
        <Text> Text from local storage:</Text>
        <Text>{this.state.messages} </Text>
      </View>
    </View>
  );
}
});

```

```

var styles = StyleSheet.create({
  container: {
    flex: 1,

    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  form: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  textField: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    width: 180},
  message: {
    flex: 1,
    alignItems: 'center'
  },
});

```

```
button: {  
  backgroundColor: '#05A5D1',  
  marginTop: 10,  
  height: 40,  
  width: 180,  
  alignItems: 'center',  
  justifyContent: 'center'  
}  
});
```

```
AppRegistry.registerComponent('AsyncStorage', () => AsyncStorageExample);
```

Let's build our application in Xcode to see the results. You can enter the text in a text box as shown in Figure 5-8 and then click "Update Storage."

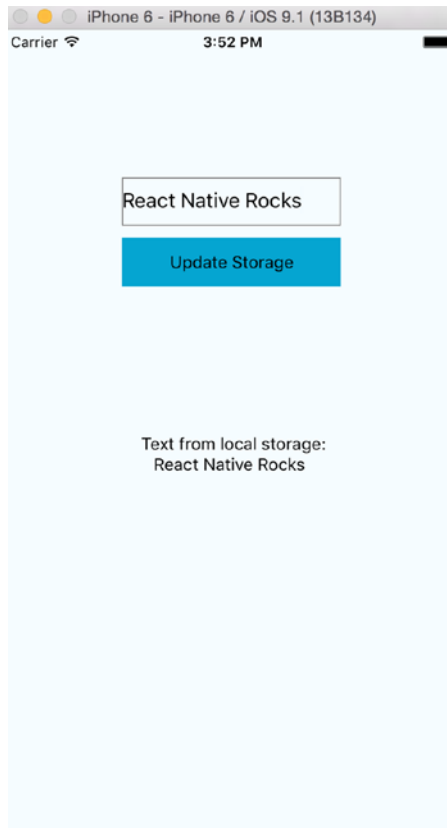


Figure 5-8. Storage is updated

Once that is done refresh for the result shown in Figure 5-9.

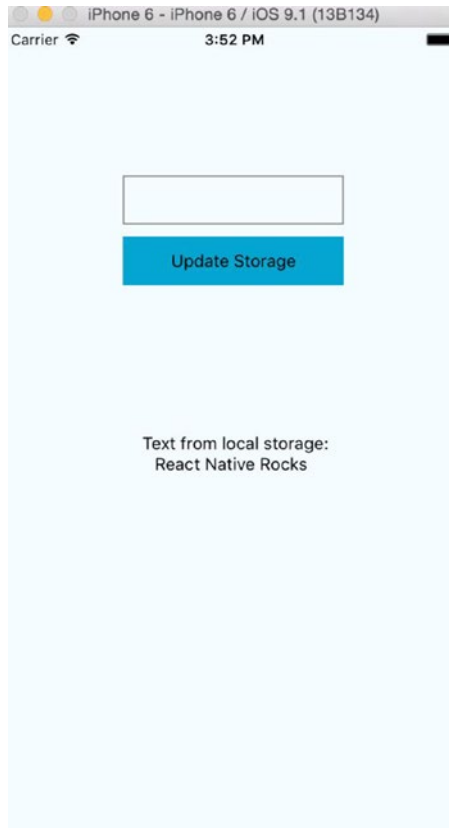


Figure 5-9. Text from the async storage mechanism

This time the text below “Text from local storage” is coming from the async storage mechanism, which we have set in place.

Reviewing the AsyncStorage Code

In this example, we have included the AsyncStorage default component in our list of components to be used for this app.

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



```
TouchableHighlight,  
  AsyncStorage  
} = React;
```

```
var STORAGE_KEY = '@AsyncStorageExample:someKey';
```

We will be using this AsyncStorage React component within our AsyncStorageExample component. Above, we have also specified a key, which we will use, with our Async Storage.

Inside our AsyncStorageExample component we have set up `getInitialState` and `componentDidMount` methods and also, created `addMessage` & `updateStorage`. Let's discuss them one by one.

```
getInitialState: function() {  
  return {  
    messages: '',  
    textInputMessage: ''  
  };  
},
```

In `getInitialState` we have specified blank values for `messages` & `textInputMessages`, which we will keep updating as and when their state changes.

```
componentDidMount() {  
  AsyncStorage.getItem(STORAGE_KEY).then((value) => {  
    this.addMessage(value);  
  }).done();  
},
```

With `componentDidMount` AsyncStorage the `getItem` method is loaded with the `addMessage` value. This is invoked only at the time of initial rendering and is responsible for showing the text below 'text from local storage', once we have updated the storage and refreshed the app again.

```
addMessage(message) {  
  this.setState({messages: message});  
},
```

The method `addMessage` updates the message state with a new updated value whenever it is triggered.

```
updateStorage () {  
  AsyncStorage.setItem(STORAGE_KEY, this.state.textInputMessage);  
  AsyncStorage.getItem(STORAGE_KEY).then((value) => {  
    this.addMessage(value);  
  }).done();  
},
```

Update storage updates Async Storage values, which are persisted permanently.

```
render: function() {
  return (
    <View style={styles.container}>
      <View style={styles.form}>
        <TextInput style={styles.textField}
          onChangeText={(textInputMessage) => this.setState({textInputMessage})}
          value={this.state.textInputMessage}/>
        <TouchableHighlight style={styles.button} onPress={this.updateStorage}>
          <Text> Update Storage </Text>
        </TouchableHighlight>
      </View>
      <View style={styles.message}>
        <Text> Text from local storage:</Text>
        <Text>{this.state.messages} </Text>
      </View>
    </View>
  );
}
```

With the above code, we set up various sections of our AsyncStorageExample component. Here, we can change a text input field to update the textInputMessage state. We also have an onPress prop for the TouchableHighlight component, which calls the updateStorage method and persists the values permanently. In the end, we display the saved message by accessing the present state of the message.

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  form: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  textField: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
    width: 180},
  message: {
    flex: 1,
    alignItems: 'center'
  },
  button: {
    backgroundColor: '#05A5D1',
    marginTop: 10,
```

```
    height: 40,  
    width: 180,  
    alignItems: 'center',  
    justifyContent: 'center'  
  }  
});
```

```
AppRegistry.registerComponent('AsyncStorage', () => AsyncStorageExample);
```

Finally, we set up a user interface style with some self-explanatory Flex settings and register our `AsyncStorageExample` component.

Native Alert

Alerts are used to give important information to application users. In the case of iOS, only after selecting the option in the alert view, can we proceed further to use the app. Optionally, we can provide a list of buttons. Tapping any button will fire the respective `onPress` callback and dismiss the alert. By default, there is only one button.

Let's create a project to understand more about native alert:

```
$react-native init NativeAlert
```

React Native provides a component `AlertIOS` for creating Alert boxes. Let's create a Button that will open an alert box when clicked.

Update `index.ios.js` with the following code:

```
'use strict';  
  
var React = require('react-native');  
var {  
  AppRegistry,  
  StyleSheet,  
  Text,  
  View,  
  TouchableHighlight,  
  AlertIOS  
} = React;  
  
var NativeAlert = React.createClass({  
  render: function() {  
    return (  
      <View style={styles.container}>  
        <TouchableHighlight style={styles.wrapper}  
          onPress={() => AlertIOS.alert(  
            'Alert Title',  
            'Alert Message'  
          )}>  
      </TouchableHighlight>  
    </View>  
  )  
  }  
});
```

```
        <View style={styles.button}>
          <Text style={styles.buttonText}>Click me !!</Text>
        </View>
      </TouchableHighlight>
    </View>
  );
}
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'FFFFFF',
  },
  button: {
    backgroundColor: '#659EC7',
    padding: 10,
    margin: 10
  },
  buttonText: {
    color: 'FFFFFF'
  }
});

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

Let's build the application and test it in the simulator. Figure 5-10 shows the result.

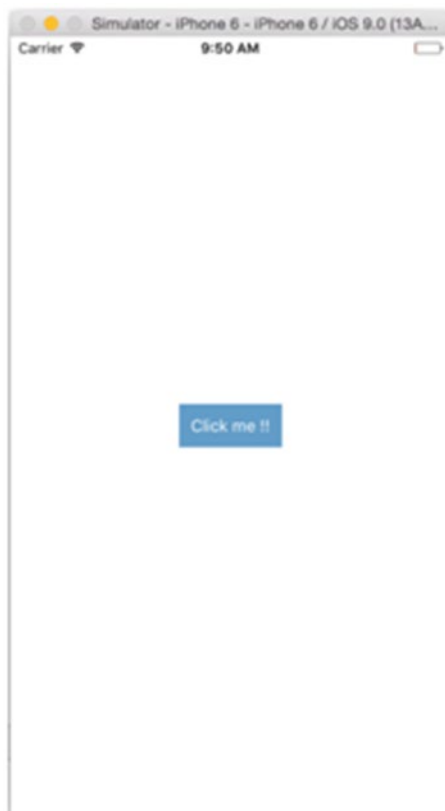


Figure 5-10. This button will open an alert box when clicked

Tap the “Click me !!” button to see an alert box as shown in Figure 5-11.

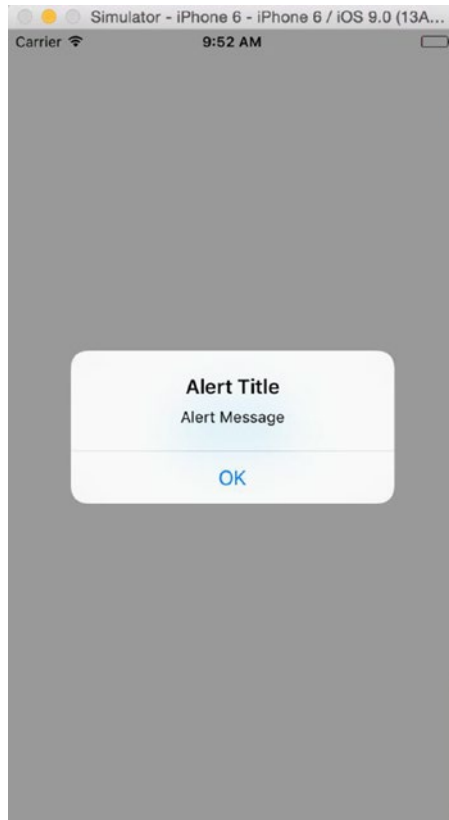


Figure 5-11. The Alert box appears

Reviewing the NativeAlert Code

After we created a new NativeAlert project, we created a new component NativeAlert:

```
var NativeAlert = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <TouchableHighlight style={styles.wrapper}
          onPress={() => AlertIOS.alert(
            'Alert Title',
            'Alert Message'
          )}>
        </TouchableHighlight>
      </View>
    );
  }
});
```

```
        <View style={styles.button}>
          <Text style={styles.buttonText}>Click me !!</Text>
        </View>
      </TouchableHighlight>
    </View>
  );
}
```

In the component `NativeAlert`, we have used `onPress` callback. The method `alert`, passes the string 'Alert Title' and 'Alert Message', which produces an alert box having a title, message and a button. `AlertIOS` provides two methods: `alert` and `prompt`.

```
static alert(title: string, message?: string, buttons?: Array<{ text: ?string; onPress?:  
?Function; }>, type?: string)
```

```
static prompt(title: string, value?: string, buttons?: Array<{ text: ?string; onPress?:  
?Function; }>, callback?: Function)
```

Finally, we have added some style in the following segment and registered our root component with `AppRegistry`.

```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'FFFFFF',
  },
  button: {
    backgroundColor: '#659EC7',
    padding: 10,
    margin: 10
  },
  buttonText: {
    color: 'FFFFFF'
  }
});
```

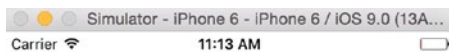
```
AppRegistry.registerComponent('NativeAlert', () => NativeAlert);
```

Extending the NativeAlert Example

Now let's add some more buttons to our application, replace the following code for your NativeAlert component in index.ios.js:

```
var NativeAlert = React.createClass({
  getInitialState: function(){
    return{
      textForButton: 'Button text will come here'
    }
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TouchableHighlight style={styles.wrapper}
          onPress={() => AlertIOS.alert(
            'Alert Title',
            'Alert Message'
          )}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>Click me !!</Text>
          </View>
        </TouchableHighlight>
        <TouchableHighlight style={styles.wrapper}
          onPress={() => AlertIOS.alert(
            'Alert Title',
            'Alert Message',
            [
              {text: 'Button 1', onPress: () => this.setState({textForButton: 'Button 1 clicked'})},
              {text: 'Button 2', onPress: () => this.setState({textForButton: 'Button 2 clicked'})}
            ]
          )}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>Alert with Buttons !!</Text>
          </View>
        </TouchableHighlight>
        <Text> {this.state.textForButton} </Text>
      </View>
    );
  }
});
```


Let's refresh our view to see the changes in Figure 5-12.



Click me !!

Alert with Buttons !!

Button text will come here

Figure 5-12. “Alert with Buttons !!” has been added

Click “Alert with Buttons!!” to see the result shown in Figure 5-13.

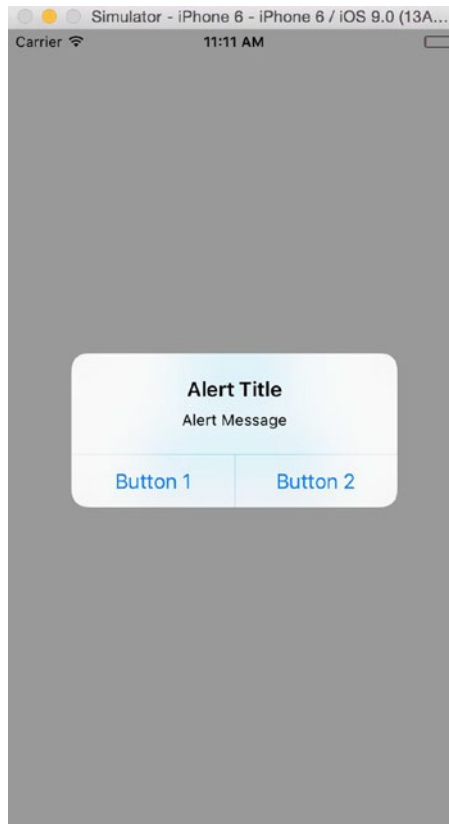


Figure 5-13. *Select Button 1 or 2*

In Figure 5-14, the home page shows which button is clicked.

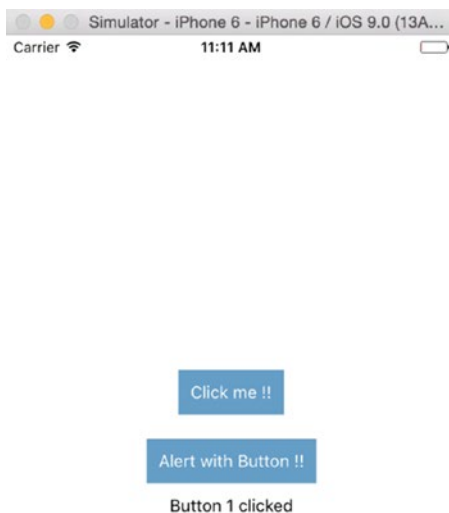


Figure 5-14. *Button 1 was clicked*

Reviewing the Extended NativeAlert Example Code

Now let's understand what we have done in this updated NativeAlert component:

```
var NativeAlert = React.createClass({
  getInitialState: function(){
    return{
      textForButton: 'Button text will come here'
    }
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TouchableHighlight style={styles.wrapper}
          onPress={() => AlertIOS.alert(
            'Alert Title',
            'Alert Message'
          )}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>Click me !!</Text>
          </View>
        </TouchableHighlight>
        <TouchableHighlight style={styles.wrapper}
          onPress={() => AlertIOS.alert(
            'Alert Title',
            'Alert Message',
            [
              {text: 'Button 1', onPress: () => this.setState({textForButton: 'Button 1 clicked'})},
              {text: 'Button 2', onPress: () => this.setState({textForButton: 'Button 2 clicked'})}
            ]
          )}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>Alert with Buttons !!</Text>
          </View>
        </TouchableHighlight>
        <Text> {this.state.textForButton} </Text>
      </View>
    );
  }
});
```

Here, we have set our `getInitialState` method with a state `textForButton`, which we will later get updated with the button we have clicked. Tapping the “Alert with Buttons!!” button fires an `onPress` call back which uses the `alert` method of `AlertIOS` to set title, message and buttons for our alert box. In this part of the `NativeAlert` component we have two buttons whose ‘`textForButton`’ state is updated with required text on execution of `onPress` callback.

WebView

In this section, we will create a shell for loading any URL in WebView with React Native. Let's begin with generating the application structure:

```
$ react-native init WebviewShell
```

Next, open up file `index.ios.js` and replace its code with the following:

```
'use strict';

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

var WebviewShell = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <WebView url={'https://www.facebook.com/'} />
      </View>
    );
  }
});

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

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

Let's build our WebviewShell application with Xcode and load in iOS simulator to see the results as shown in Figure 5-15.

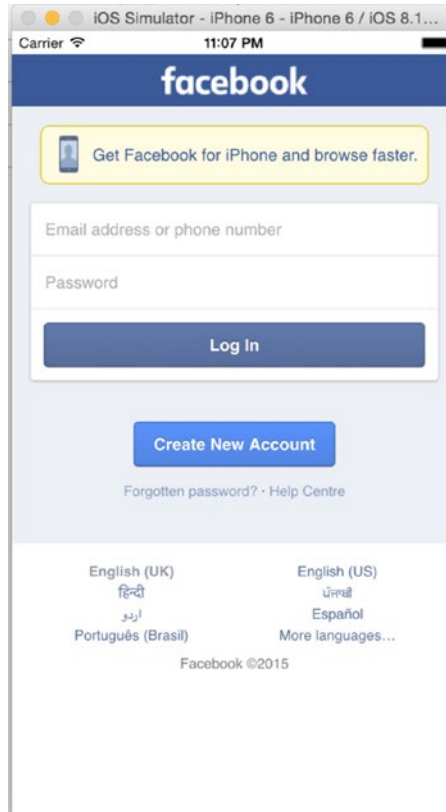


Figure 5-15. *WebView with a URL*

Reviewing the WebView Code

In this example, we have created a component WebviewShell that returns a View. The following code creates a view with our desired URL loaded in the WebView.

```
var WebviewShell = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <WebView url={'https://www.facebook.com/'} />
      </View>
    );
  }
});
```

Animations

Fluid, meaningful animations are essential in creating a stunning user experience. In this section, we will achieve this using React Native animation APIs. Animations with React Native revolve around two complementary systems: *LayoutAnimation* for animating global layout transactions, and *Animated* for more granular control over specific properties.

The *Animated* library is designed in a way to make a variety of amazing animations and interactions with your app with simplicity and high performance. The Animated library focuses on configurable transforms in between, having declarative relationships between input and output, and easy start-stop methods, which control animations based on time.

Let's understand this library with an example by creating an AnimationExample React Native project:

```
$react-native init AnimationExample
```

Add the following code to your index.ios.js:

```
'use strict';

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

var AnimationExample = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <View
          style={{
            backgroundColor: '#DC143C',
            flex: 1
          }} />
        <View
          style={{
            backgroundColor: '#1E90FF',
            flex: 1,
          }} />
      </View>
    );
  }
});
```

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

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

Build the project with Xcode to load it on the iOS simulator. Figure 5-16 shows two colorful boxes.

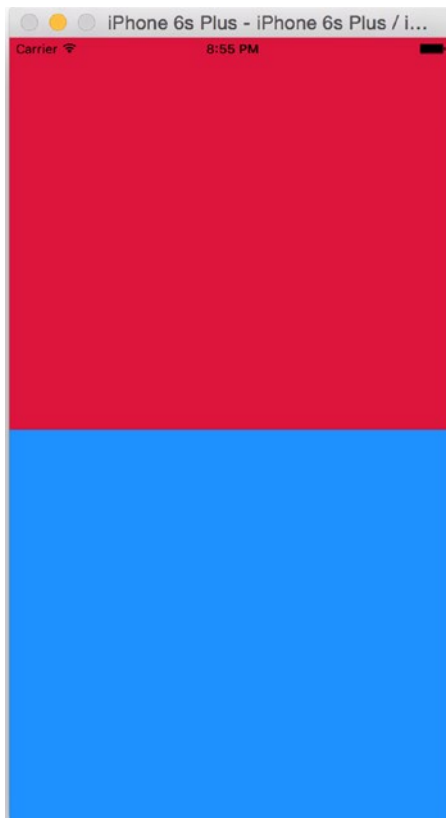


Figure 5-16. The project now has a red and blue box

Now let's add some animation. Update your `index.ios.js` with the following code:

```
'use strict';

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

var {
  AppRegistry,
  StyleSheet,
  Text,
  View,

  Animated
} = React;

var AnimationExample = React.createClass({
  getInitialState: function() {
    return {
      bounceValue: new Animated.Value(0)
    };
  },
  componentDidMount() {
    this.state.bounceValue.setValue(1.5);
    Animated.spring(
      this.state.bounceValue,
      {
        toValue: 0.8,
        friction: 1,
      }
    ).start();
  },
  render: function() {
    return (
      <View style={styles.container}>
        <Animated.View
          style={{
            backgroundColor: '#DC143C',
            flex: 1,
            transform: [
              {scale: this.state.bounceValue},
            ]
          }} />
        <View
          style={{
            backgroundColor: '#1E90FF',
            flex: 1,
          }} />
      </View>
    );
  }
});
```

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

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

Let's refresh to see the changes (Figure 5-17).

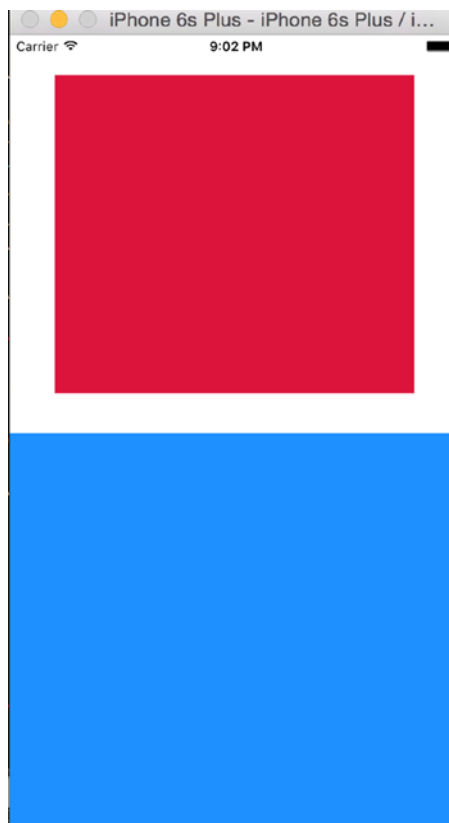


Figure 5-17. The red box in the upper half now has a bounce effect in it

Great, let's now add some animation to our bottom half section. Update `index.ios.js` with the following code:

```
'use strict';

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

var AnimationExample = React.createClass({
  getInitialState: function() {
    return {
      bounceValue: new Animated.Value(0),
      fadeAnim: new Animated.Value(0)
    };
  },
  componentDidMount() {
    this.state.bounceValue.setValue(1.5);
    Animated.spring(
      this.state.bounceValue,
      {
        toValue: 0.8,
        friction: 1,
      }
    ).start();
    Animated.timing(
      this.state.fadeAnim,
      {
        toValue: 1,
        duration: 2000,
      },
    ).start();
  },
  render: function() {
    return (
      <View style={styles.container}>
        <Animated.View
          style={{
            backgroundColor: '#DC143C',
            flex: 1,
            transform: [
              {scale: this.state.bounceValue},
            ]
          }} />
        <Animated.View
          style={{
```

```

        backgroundColor: '#1E90FF',
        flex: 1,
        opacity: this.state.fadeAnim,

      }} />
    </View>

  );
}
});

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

AppRegistry.registerComponent('AnimationExample', () => AnimationExample);

```

Figure 5-18 shows the changes.

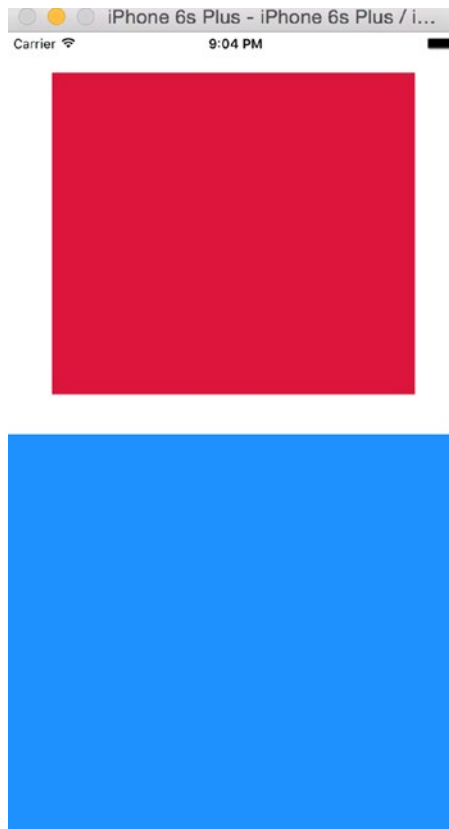


Figure 5-18. Now there is a bounce in the upper half of the screen and fading animation in the bottom half

Let's now get into our code and see what we have done here.

Reviewing the Animation Code

In the first part of this example, we have created the `AnimationExample` component, which has a split, having two sections, which we will use for animations later on. We have not used any default React Native component yet.

Next, we have added components necessary for use in our example.

```
'use strict';

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

Here, we have added an `Animated` React Native default component, which is necessary for animations in various segments of our example. Let's now understand how we have shaped our `AnimationExample` component with two different animations.

```
var AnimationExample = React.createClass({
  getInitialState: function() {
    return {
      bounceValue: new Animated.Value(0),
      fadeAnim: new Animated.Value(0)
    };
  },
},
```

With `getInitialState`, we are setting up an initial state for `bounceValue`, which will be used for the bounce effect in the upper half, and `fadeAnim`, which will add fading animation to the bottom half. Initially both are set to 0.

```
  componentDidMount() {
    this.state.bounceValue.setValue(1.5);
    Animated.spring(
      this.state.bounceValue,
      {
        toValue: 0.8,
        friction: 1,
      }
    );
  }
},
```

```

    ).start();
    Animated.timing(
      this.state.fadeAnim,
      {
        toValue: 1,
        duration: 2000,
      },
    ).start();
  },
},

```

With `componentDidMount`, we are setting the value of `bounceValue` to have a spring effect. We are also setting up a value for `fadeAnim` so that its fading effect has certain timing. Both start immediately at the time the application loads.

```

render: function() {
  return (
    <View style={styles.container}>
      <Animated.View
        style={{
          backgroundColor: '#DC143C',
          flex: 1,
          transform: [
            {scale: this.state.bounceValue},
          ]
        }} />
      <Animated.View
        style={{
          backgroundColor: '#1E90FF',
          flex: 1,
          opacity: this.state.fadeAnim,

          }} />
    </View>

  );
}
});

```

Next, we render both our animations on a specific section of the app. In the first container, we set the bounce effect and in second, we have fading animation.

```

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

```

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

Lastly, we have some more Flex styling and register our AnimationExample component.

As discussed earlier animations with React Native have two system Animated and layout animations. We have gone in details on animated, as an exercise, explore layout animation with the same example.

Summary

In this chapter, we learned about capabilities of your iOS device beyond just user interface. We learned how to use Geolocation and loading maps for your app, AsyncStorage to persist data, Native alerts to share important info in your app, WebView to load HTML5 content and lastly animations.

In next chapter we will learn how to interact with a backend server because probably no real world application is complete without connecting to a backend.

Chapter 6

Communicating with Servers

“Communication is everyone’s panacea for everything.”

—Tom Peters

After learning about the powers of device capabilities via multiple examples, it’s time to get back to your housing application. So far, you have been populating your app locally with some dummy data, but no application can survive without communicating with a server. In this chapter, you will learn how to interact with network APIs. You will explore the following topics:

- XMLHttpRequest
- WebSocket
- Fetch
- Getting data from a server
- Posting data to a server

Earlier you were getting all the data from a dummy data object, which was static within your application. It’s a rare chance any production application will entirely work with static data. Happily, React Native provides many ways to interact with network APIs. The following sections cover the ways the network stack is supported in React Native.

XMLHttpRequest

XMLHttpRequest is an API that provides the ability to transfer data between a client and a server. It provides an easy way to retrieve data from a URL without having to do a full-page refresh. In React Native, the XMLHttpRequest API is applied on top of the iOS networking APIs. A notable difference from the Web is the security model; you can read from any website on the Internet since there is no concept of CORS.


```
var xhttp= new XMLHttpRequest();
xhttp.onreadystatechange = (e) => {
  if (xhttp.readyState !== 4) {
    return;
  }

  if (xhttp.status === 200) {
    console.log('success', xhttp.responseText);
  } else {
    console.warn('error');
  }
};

xhttp.open('GET', 'https://example.com/myendpoint.php');
request.send();
```

Using XMLHttpRequest is quite tedious. But since it is compatible with the browser API, it lets you use third-party libraries directly from npm like Parse. For more information on this API, please refer to its documentation at <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>.

WebSocket

WebSocket is a protocol that provides full-duplex communication channels over a single TCP connection.

```
var ws = new WebSocket('ws://example.com/path');

ws.on('open', function() {
  // connection opened
  ws.send('example data');
});

ws.on('message', function(e) {
  // a message received
  console.log(e.data);
});

ws.on('error', function(e) {
  // an error occurred
  console.log(e.message);
});

ws.on('close', function(e) {
  // connection closed
  console.log(e.code, e.reason);
});
```

Fetch

Fetch is a popular networking API. It was created by a standard committee and has well-defined requests, responses, and the process to bind them. The following is an example of a post request with fetch:

```
fetch('https://example.com/endpoint/', {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'yourValue',
    secondParam: 'otherValue',
  })
})
```

Fetch returns a promise, which is processed using then and catch.

```
fetch('https:// example.com/endpoint')
  .then((response) => response.text())
  .then((responseText) => {
    console.log(responseText);
  })
  .catch((error) => {
    console.warn(error);
  });
```

Now that you know how to interact with network APIs, let's use one of the options, fetch, to get and post data to a server. In order to keep things simple, we have hosted a simple backend server with restful APIs that you can consume for your application.

We will be using following URLs to get and post data to a backend server. You can use curl to see the response you get from making a request to these URLs.

To get an initial seed list of properties:

```
$curl 'http://www.akshatpaul.com/list-all-properties'
[
{
name: "Mr. Johns Conch house",
address: "12th Street, Neverland",
images: {
thumbnail: "http://hmp.me/ol5"
}
},
{
name: "Mr. Pauls Mansion",
address: "625, Sec-5, Ingsoc",
images: {
thumbnail: "http://hmp.me/ol6"
}
},
]
```

```
{
  name: "Mr. Nalwayas Villa",
  address: "11, Heights, Oceania",
  images: {
    thumbnail: "http://hmp.me/ol7"
  }
},
{
  name: "Mr. Johns Conch house",
  address: "12th Street, Neverland",
  images: {
    thumbnail: "http://hmp.me/ol5"
  }
},
{
  name: "Mr. Pauls Mansion",
  address: "625, Sec-5, Ingsoc",
  images: {
    thumbnail: "http://hmp.me/ol6"
  }
},
{
  name: "Mr. Nalwayas Villa",
  address: "11, Heights, Oceania",
  images: {
    thumbnail: "http://hmp.me/ol7"
  }
},
{
  name: "Mr. Johns Conch house",
  address: "12th Street, Neverland",
  images: {
    thumbnail: "http://hmp.me/ol5"
  }
},
{
  name: "Mr. Pauls Mansion",
  address: "625, Sec-5, Ingsoc",
  images: {
    thumbnail: "http://hmp.me/ol6"
  }
},
{
  name: "Mr. Nalwayas Villa",
  address: "11, Heights, Oceania",
  images: {
    thumbnail: "http://hmp.me/ol7"
  }
}
}
```

To get list of properties that users have saved:

```
$curl 'http://www.akshatpaul.com/list-properties'
```

To post data to the server in order to save a property:

```
url: 'http://www.akshatpaul.com/properties'
```

Getting Data from a Server

First, you need some fixed data from our server to populate the list of properties. Insert the following code into the `ListProperty.js` component:

```
var React = require('react-native');

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

var REQUEST_URL = 'http://www.akshatpaul.com/list-all-properties';

var ListProperty = React.createClass({

  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      dataSource: ds,
      loaded: false
    };
  },

  componentDidMount: function() {
    this.fetchData();
  },
  fetchData: function() {
    fetch(REQUEST_URL)
      .then((response) => response.json())
      .then((responseData) => {
        console.log(responseData);
        this.setState({
          house: responseData,
          dataSource: this.state.dataSource.cloneWithRows(responseData),
          loaded: true
        });
      })
  })
});
```

```
        .catch((error) => {
        AlertIOS.alert('No Donut for you');
        loaded: true
    })
    .done();
},

render: function() {
    if (!this.state.loaded) {
        return this.renderLoadingView();
    }
    return (
        <ListView
            dataSource={this.state.dataSource}
            renderRow={this.renderProperty}
            style={styles.listView}/>
    );
},

renderProperty: function(property) {
    return (
        <View style={styles.container}>
            <Image
                source={{uri: property.images.thumbnail}}
                style={styles.thumbnail}/>
            <View style={styles.rightContainer}>
                <Text style={styles.name}>{property.name}</Text>
                <Text style={styles.address}>{property.address}</Text>
            </View>
        </View>
    );
},

renderLoadingView: function() {
    return (
        <View style={styles.container}>
            <Text>
                Loading houses...
            </Text>
        </View>
    );
},

});

var styles = StyleSheet.create({
    container: {
        flex: 1,
        flexDirection: 'row',
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
},
```

```
    thumbnail: {
      width: 63,
      height: 91,
    },
    rightContainer: {
      flex: 1,
    },
  },
  name: {
    fontSize: 20,
    marginBottom: 8,
    textAlign: 'center',
  },
  address: {
    textAlign: 'center',
  },
  listView: {
    paddingTop: 70,
    backgroundColor: '#F5FCFF',
  },
});

module.exports = ListProperty;
```

Now, build or refresh the application. Figure 6-1 shows it loaded on an iOS simulator.



Figure 6-1. Populating the app with static data fetched from a server

Here you use the request URL to get the initial seed data not from a client but from a server. Inside the `componentDidMount` method you make a request using `fetch`:

```
fetchData: function() {
  fetch(REQUEST_URL)
    .then((response) => response.json())
    .then((responseData) => {
      console.log(responseData);
      this.setState({
        house: responseData,
        dataSource: this.state.dataSource.cloneWithRows(responseData),
        loaded: true
      });
    })
    .catch((error) => {
      AlertIOS.alert('No Donut for you');
      loaded: true
    })
    .done();
}
```

This returns a promise that is processed using `then` and `catch`.

Saving Data to a Server

In your housing application, you have an option to add new properties to your back-end application. You used this feature earlier to learn how to submit data to a backend API. Add following code into your `AddProperty.js` component:

```
var React = require('react-native');

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

var AddProperty = React.createClass({
  getInitialState: function() {
    return {
      name: "",
      address: ""
    };
  },
  _onPressButtonPOST: function() {
    fetch("http://www.akshatpaul.com/properties", {method: "POST", body:
      JSON.stringify({property: {name: this.state.name, address: this.state.address}})})
      .then((responseData) => {
        AlertIOS.alert(
          "Created"
        )
      })
      .done();
  },
  render: function() {
    return (
      <View style={styles.container}>

        <TextInput style={styles.textBox} placeholder='name' onChangeText={(name) =>
          this.setState({name})} value={this.state.name} />
        <TextInput style={styles.textBox} placeholder='address' onChangeText={(address) =>
          this.setState({address})} value={this.state.address} />
        <TouchableHighlight style={styles.button}
          onPress= {this._onPressButtonPOST}
```



```
        underlayColor='#99d9f4'>
        <Text style={styles.buttonText}>Add House</Text>
    </TouchableHighlight>
</View>
);
},
});
```

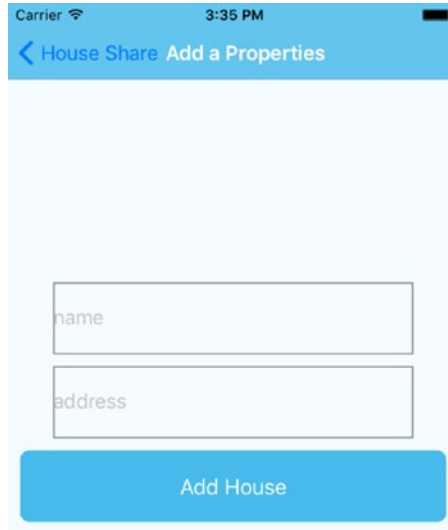
```
var styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center',

    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  textBox: {
    width: 300,
    height: 60,
    borderColor: 'gray',
    borderWidth: 1,
    alignSelf: 'center',
    marginTop: 10,
  },
  button: {

    height: 60,
    backgroundColor: '#48BBEC',
    borderColor: '#48BBEC',
    borderWidth: 1,
    borderRadius: 8,
    alignSelf: 'stretch',
    justifyContent: 'center',
    margin: 10
  },
  buttonText: {
    fontSize: 18,
    color: 'white',
    alignSelf: 'center'
  }
});
```

```
module.exports = AddProperty;
```

Refresh to test this functionality. The result is shown in Figure 6-2.



Carrier 3:35 PM

< House Share Add a Properties

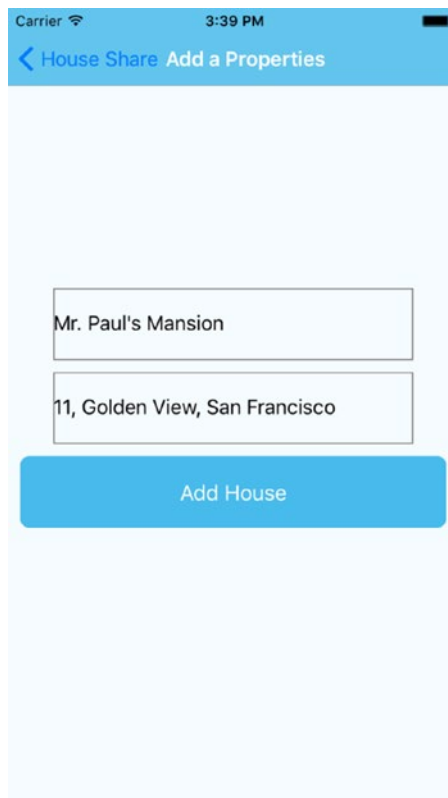
name

address

Add House

Figure 6-2. Form to submit record

Next, add some values and post it to our server (Figure 6-3).



Carrier 3:39 PM

< House Share Add a Properties

Mr. Paul's Mansion

11, Golden View, San Francisco

Add House

Figure 6-3. Values are added to the server.

When this data is saved, you get an alert box as confirmation, as shown in Figure 6-4.

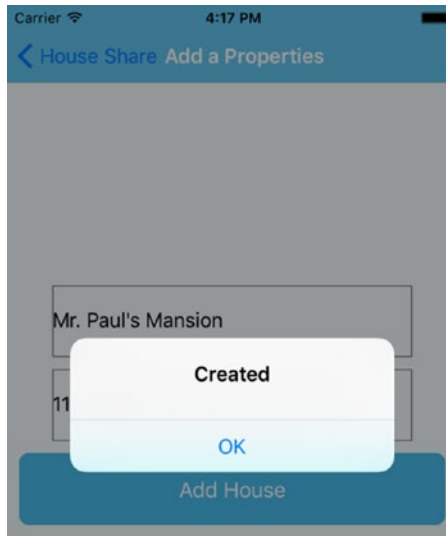


Figure 6-4. Confirmation of successful submission

Let's update the `ListProperty.js` component with a URL that can fetch the user-added property list. Update the following variable in `ListProperties.js`:

```
var REQUEST_URL = 'http://www.akshatpaul.com/list-properties'
```

If you `curl` this URL, you will get following JSON of user added properties:

```
$curl 'http://www.akshatpaul.com/list-properties'
[
{
  name: "Mr. Paul's Mansion",
  address: "11, Golden View, San Francisco",
  images: {
    thumbnail: "http://hmp.me/ol7"
  }
}
]
```

Note This API shows data submitted by various readers of this book. Your data set may differ.

Refresh the app and go to the “List of Properties” section (Figure 6-5).

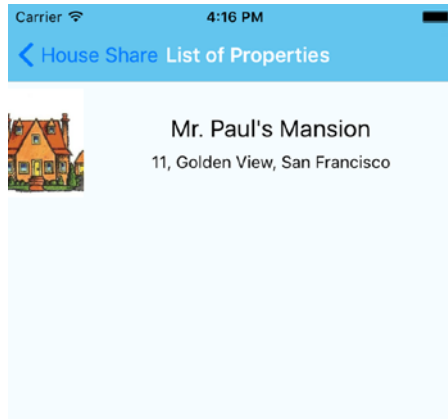


Figure 6-5. Output showing Mr. Paul's Mansion address

Now add a URL for post requests:

```
var REQUEST_URL = 'http://www.akshatpaul.com/properties';
```

Next, modify the `onPressButtonPOST` function:

```
_onPressButtonPOST: function() {
  fetch(REQUEST_URL, {method: "POST", body: JSON.stringify({property:
    {name: this.state.name, address: this.state.address}})})
    .then((responseData) => {
      AlertIOS.alert(
        "Created"
      )
    })
    .done();
}
```

Here, you make a post request using `fetch` and serialize your JSON, which returns a promise and, upon success, issues an alert box with a “Created” message.

Summary

In this chapter, you learned about various network APIs that are reimplemented from the ground up by the React Native team. You also learned various options like `XMLHttpRequest`, `WebSocket`, and `Fetch`. Since no application is complete without making server calls, you added this capability into your housing application and learned how to get data from a server, add new records, and save them to a server.

In the final chapter of this book, you will learn about popular Node packages that you can use with React Native, specifically awesome React libraries like `Reflux` and `Redux`, which make development slightly simple and faster.

Chapter 7

React Native Supplements

“A good way to stay flexible is to write less code.”

—Pragmatic Programmer

By now, you should be comfortable building apps using React Native and have become accustomed to all the basics as well as many advanced features of the React Native framework. In this final chapter, we will learn about a few supplements, which may not be necessary but are very useful in certain situations and reduce our work significantly. In this chapter we will cover the following topics:

- Reflux
- Redux
- Debugging on a device
- Popular node modules
- Where to go from here

Reflux

We learned about Flux architecture in Chapter 4. We have some other ways to implement unidirectional data flow, including a lean approach to flux called *Reflux*. It's a simple library for unidirectional data flow inspired by React's Flux. As shown in Figure 7-1, a Reflux pattern has actions and stores. Actions initiate new data, which is passed through data stores, which are then passed to view components and in turn passed back to actions. If a view component has an event that makes changes to data stores, it needs to send a signal to the stores via the actions available.

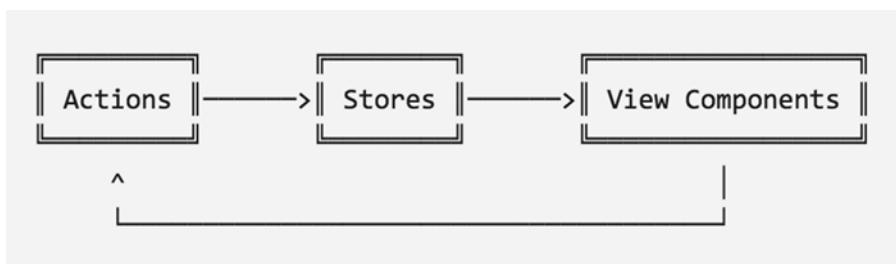


Figure 7-1. Reflux unidirectional flow

Differences from Flux

Reflux has actually refactored Flux to be closer to Functional Reactive Programming (FRP). The following are a few differences between Reflux and Flux:

- The singleton dispatcher is not present, letting every action act like a dispatcher instead.
- Because actions are listenable, the stores listen to them. Stores don't need to have big switch statements that do static type checking (of action types) with strings.
- Stores can listen to other stores; it is possible to create stores that can aggregate data further, similar to a map/reduce.
- Action creators are not needed, because Reflux actions are functions that will pass on the payload they receive to anyone listening to them.
- `waitFor` is replaced to handle serial and parallel data flows. Aggregate data stores may listen to other stores in serial and to joins for joining listeners in parallel.

In order to understand Reflux better, let's create a simple `ToDo` application just like we did earlier in Chapter 4. Let's generate a React Native application:

```
$ react-native init RefluxToDo
```

Our basic project structure is now ready. Let's add a Reflux node module to our project:

```
$ cd RefluxToDo
$ npm install reflux --save-dev
```

Now that we have Reflux added to our project, let's create a structure that we can use with Reflux. First, create a root folder named `Apps` that contains the subfolders `Actions`, `Components`, and `Stores`, which hold the files shown in Figure 7-2.

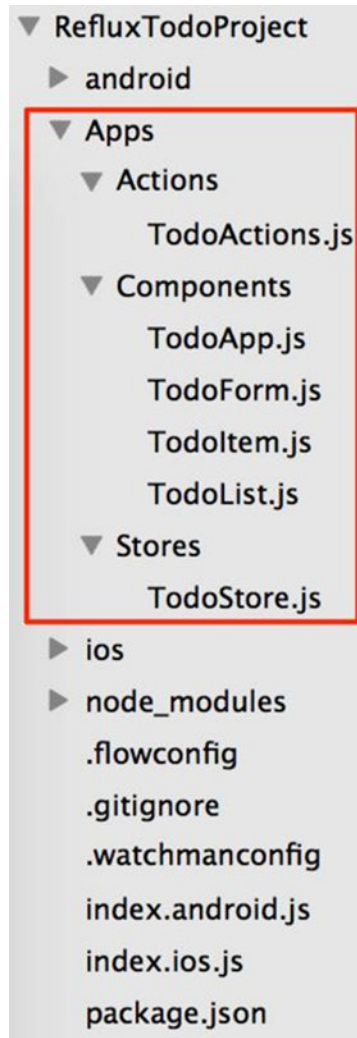


Figure 7-2. Root folder *Apps* with subfolders and files

Unlike in Flux patterns, with Reflux we only have actions and stores, as we dropped dispatcher altogether. We will keep various components in our Component folder so as to keep our `index.ios.js` file lean. These files are empty for now, so let's add the required code in each of them via the following steps.

1. Start with Actions. Open Apps/Actions/ToDoActions.js and add the following code:

```
'use strict';

var Reflux = require('reflux');

module.exports = Reflux.createActions([
  'todoCreate',
  'todoUpdate',
  'todoComplete',
  'todoUndoComplete',
  'todoDestroy',
]);
```

2. Add the following code to Stores, which resides in Apps/Stores/Todostore.js:

```
'use strict';

var Reflux = require('reflux');
var _ = require("underscore");

module.exports = Reflux.createStore({
  listenables: [require('../Actions/ToDoActions')],

  todos: {},

  onTodoCreate: function(text) {
    var id = (+new Date() + Math.floor(Math.random() * 999999)).toString(36);
    this.todos[id] = {
      id: id,
      complete: false,
      text: text,
    };
    this.trigger(null);
  },
  onTodoUpdate: function(id, updates) {
    this.todos[id] = _.extend({}, this.todos[id], updates);
    this.trigger(null);
  },
  onTodoComplete: function(id) {
    this.onTodoUpdate(id, {complete: true});
  },
  onTodoUndoComplete: function(id) {
    this.onTodoUpdate(id, {complete: false});
  },
});
```



```

onTodoDestroy: function(id) {
  delete this.todos[id];
  this.trigger(null);
},

getAll: function() {
  return _.values(this.todos);
},
});

```

3. Create the components that will interact with Actions, which in turn interacts with Stores. Let's add the following code to `Apps/Components/ToDoForm.js`:

```

ToDoForm.js
'use strict';

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

var TodoActions = require('../Actions/ToDoActions');
var TodoStore = require('../Stores/ToDoStore');

module.exports = React.createClass({
  propTypes: {
    value: React.PropTypes.string,
  },

  getInitialState: function() {
    return {
      value: '',
    };
  },

  render: function() {
    return (
      <View style={styles.header}>
        <TextInput
          style={styles.textInput}
          onChangeText={(text) => this.setState({value: text})}
          onBlur={this._save}
          placeholder='What needs to be done?'
          value={this.state.value}
        />
      </View>
    );
  },
});

```

```
_save: function() {
  var text = this.state.value;
  if (text) {
    TodoActions.todoCreate(text);
  }
}

this.setState({
  value: ''
});
},
});
```

```
var styles = StyleSheet.create({
  header: {
    marginTop: 21,
  },
  textInput: {
    height: 40,
    backgroundColor: '#EEEEEE',
    padding: 10,
    fontSize: 16
  },
});
```

4. Add the following code to the `TodoItem` component in `Apps/Components/TodoItem.js`:

```
TodoItem.js
'use strict';

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

var TodoActions = require('../Actions/TodoActions');

module.exports = React.createClass({
  render: function() {
    var todo = this.props.todo;
    var styleTodoItemComplete = (todo.complete) ? styles.todoItemComplete : null;
    return (
      <View>
        <View style={[styles.todoItem, styleTodoItemComplete]}>
          <Text style={styles.text}>{todo.text}</Text>
          <Text style={styles.text}>{todo.complete}</Text>
        </View>
      </View>
    );
  }
});
```

```

      <Text onPress={() => this._onToggleComplete(todo)}>|Mark as done|</Text>
      <Text onPress={() => this._onDestroy(todo)}> Delete|</Text>
    </View>
    <View style={styles.separator} />
  </View>
);
},

_onToggleComplete: function(todo) {
  if (todo.complete) {
    TodoActions.todoUndoComplete(todo.id);
  } else {
    TodoActions.todoComplete(todo.id);
  }
},
_onDestroy: function(todo) {
  TodoActions.todoDestroy(todo.id);
}
});

var styles = StyleSheet.create({

todoItem: {
  flex: 1,
  flexDirection: 'row',
  justifyContent: 'center',

alignItems: 'center',
  backgroundColor: '#FFFFFF',
  padding: 10,
  height: 58,
},
todoItemComplete: {
  opacity: 0.3,
},
text: {
  flex: 1,
  textAlign: 'left',
  fontSize: 16,
},
separator: {
  height: 1,
  backgroundColor: '#CCCCC',
},
});

```

5. Add the following code to `TodoList.js`, which is found in `Apps/Components/TodoList.js`:

```
Todolist.js
'use strict';

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

var TodoStore = require('../Stores/TodoStore');
var TodoItem = require('../TodoItem');

module.exports = React.createClass({
  mixins: [Reflux.listenTo(TodoStore, 'handlerTodoUpdate')],

  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      todoDataSource: ds.cloneWithRows(TodoStore.getAll()),
    };
  },

  handlerTodoUpdate: function(err) {
    if (err) {
      return
    }
    this.setState({
      todoDataSource: this.state.todoDataSource.cloneWithRows(TodoStore.getAll()),
    });
  },

  render: function() {
    return (
      <ListView
        dataSource={this.state.todoDataSource}
        renderRow={(rowData) => <TodoItem todo={rowData} />}
      />
    );
  },
});

var styles = StyleSheet.create({
  list: {
    flex: 1,
    backgroundColor: '#0FF',
  },
});
```

```

index.ios.js
'use strict';

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

var TodoApp = require('./Apps/Components/TodoApp');

AppRegistry.registerComponent('TodoProject', () => TodoApp);

```

6. Add the following code to `TodoApp.js`, found in `Apps/Components/TodoApp.js`:

```

TodoApp.js
'use strict';

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

var TodoForm = require('./TodoForm');
var TodoList = require('./TodoList');

module.exports = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <TodoForm />
        <TodoList />
      </View>
    );
  },
});

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

```

7. In order to use the components just created, update `index.ios.js` with the following code:

```
'use strict';

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

var TodoApp = require('./Apps/Components/TodoApp');

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

With these steps complete, we can now build our application in Xcode in order to run through our application. Once the application gets loaded, the home screen in [Figure 7-3](#) appears. Here we can add any item to the To Do list (“What needs to be done?”).

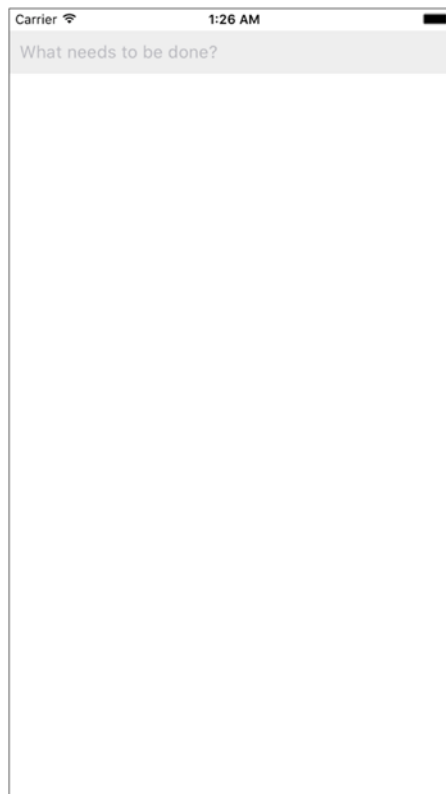


Figure 7-3. *To do list is ready to be populated with user input*

Once added, the items are listed one after another with options “Mark as done” or “Delete” displayed on the right side of each row, as shown in Figure 7-4.

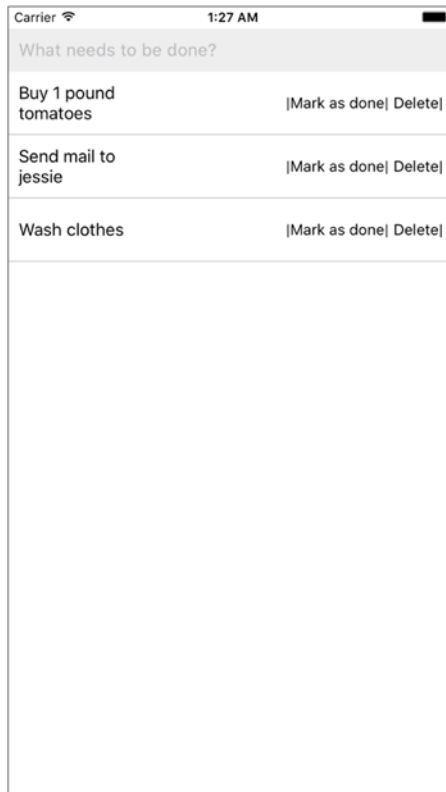


Figure 7-4. Each item in the populated list now has two options

When “Mark as done” is clicked, the row becomes faded, indicating this item is done. Clicking this option again will unfade the row. Click on “Delete” to remove items from the list entirely (see Figure 7-5).



Figure 7-5. Two items have been marked as done

Redux

Redux is a predictable-state container for JavaScript apps, and it has become very popular, especially in the React community. Though you can use the Redux concept in React, AngularJS, or any framework. Redux is open source, with Dan Abramov as its lead developer. Redux makes state mutations predictable by imposing certain controls on how and when updates can be made to state.

Until now, we have seen that state is used a lot in React Native apps, and the state value is always changing. When our actual app grows, handling these state changes can become unpredictable. Utilizing React with Flux is a good method by which to solve issues using mutation and synchronicity, as React has removed asynchrony from the view-layer and direct-DOM manipulation. However, managing the state of data is left up to the developer. Redux has solved the problem of managing state using three principles:

- Single source of truth
- State is read-only
- Mutations are written as pure functions

Single Source of Truth

The store of your application is stored in an object tree within a single store. A single-state tree has all the state changes done. This makes it easy to create universal apps, since we know all actions the client app has performed are done. It also makes debugging easier, since we can track all of the state changes.

State Is Read-only

The only way to change the state is to emit an action—an object describing what happened. Thus, we can't directly update state; it can only be updated through actions. View and network calls can't update the state directly.

Mutations Are Written as Pure Functions

To specify how the state tree is transformed by actions, you write pure reducers. Reducers will take old state and actions and return a new state. Reducers are normal JavaScript functions. It is important to understand that reducers do not update the state, they just return the new state.

Debug on Device

So far we have been debugging and testing our application on an iOS simulator, which works pretty well most of the times, but in real-world projects we would need to load our application time and again on an iOS device to test and debug.

To debug your application on an iOS device, simply plug in your device (in our case it's an iPad). Open up Xcode and select your device instead of the simulator (see Figure 7-6).

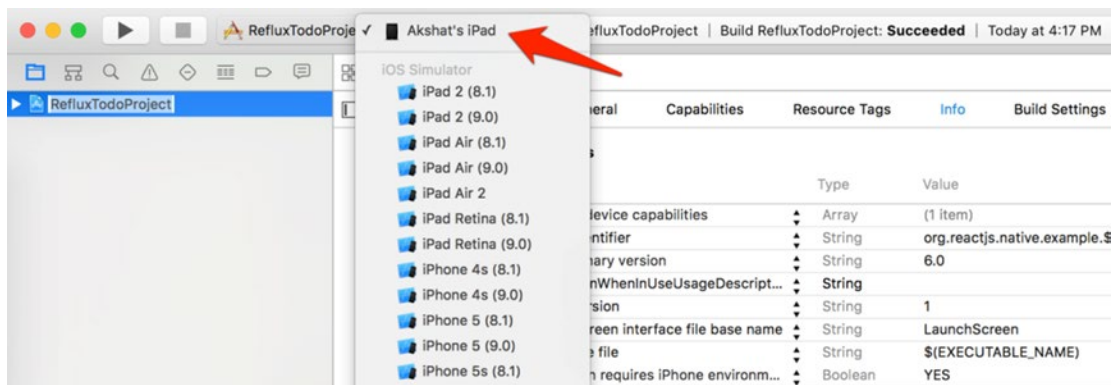


Figure 7-6. Selecting an iPad as your device

We are using our RefluxTodo application for this section, but you can use any React Native project you have built so far. Build the application to load it on the device. The application icon should appear, but we will get the error shown in Figure 7-7.

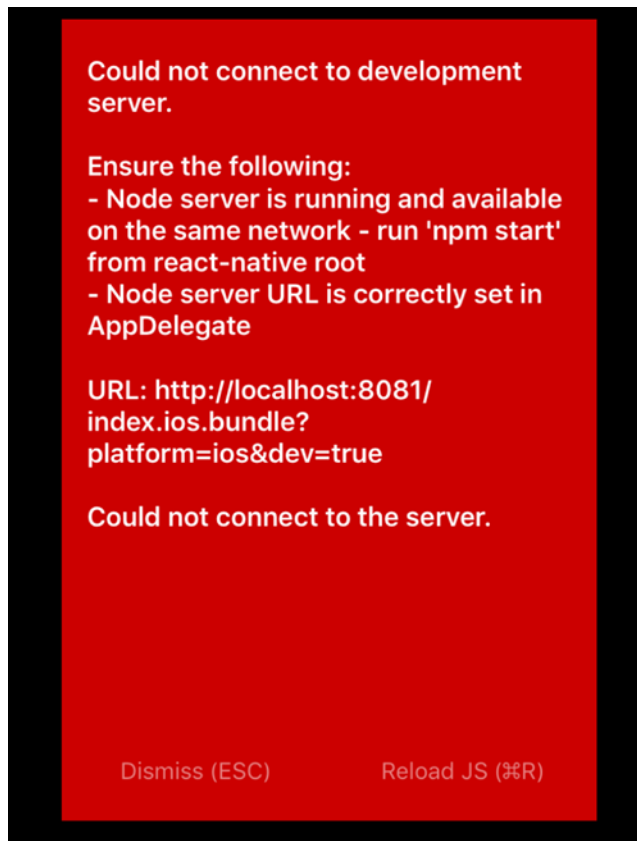


Figure 7-7. “Unable to connect” error message

The reason for this error is that our application is not able to connect to the development server. Let’s fix this by getting into our application code. Go to the AppDelegate.m file, which is usually placed in project-name/ios/project-name/AppDelegate.m, and change the following lines:

```
jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle?platform=ios&dev=true"];
```

to:

```
jsCodeLocation = [NSURL URLWithString:@"http://machine-ip-address:8081/index.ios.bundle?platform=ios&dev=true"];
```

Here, we have to replace `localhost` with our machine's IP address. Doing so allows our application to load and run properly on the device (see Figure 7-8).

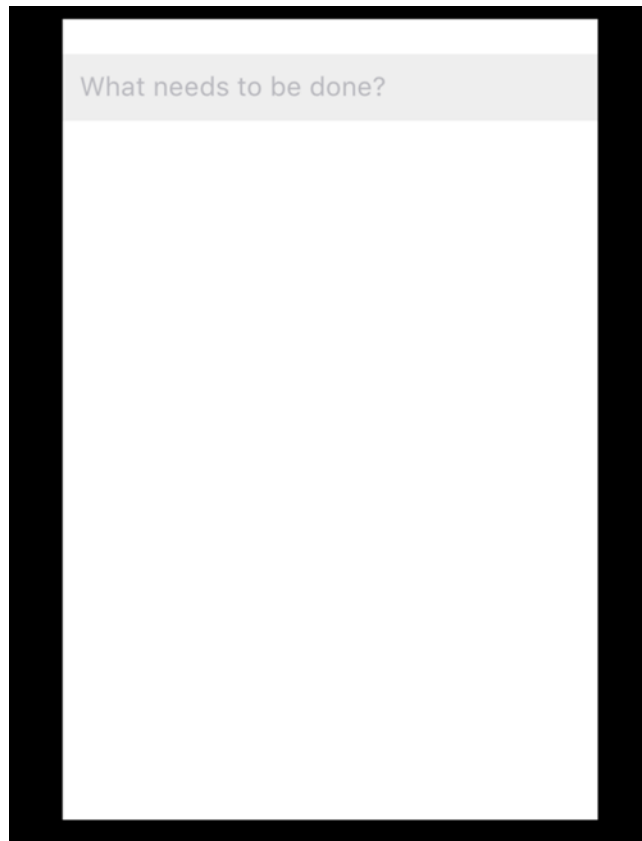


Figure 7-8. Application is now running properly

In order to access the developer menu, simply shake your device—the menu appears from the bottom of the view (see Figure 7-9).

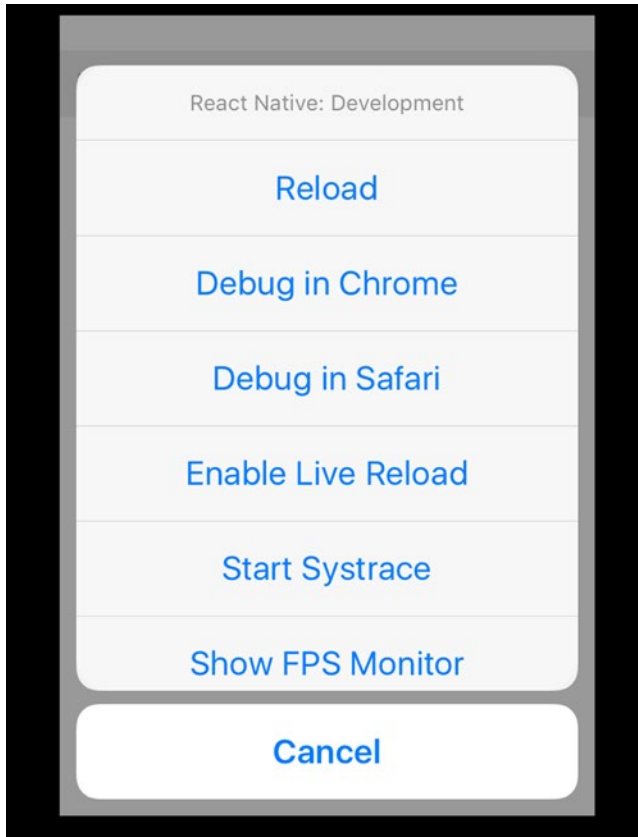


Figure 7-9. *Developer menu is now visible*

We can enable live reload, and just like in the iOS simulator, all changes will automatically be reflected on the device too. Click Debug in Chrome to see the result shown in Figure 7-10.



Figure 7-10. *WebSocket error message*

However, we get another error. In order to solve this one, we have to get back to our code and make a few changes. For this, we need to go to `RCTWebSocketExecutor.m`, which is placed in the ReactNative node module, traverse to `/your-project-name/node_module/react-native/Libraries/WebSocket/ RCTWebSocketExecutor.m`, and again we need to change our `localhost` to the machine's IP address in the following section of code:

```
From
- (instancetype)init
{
    return [self initWithURL:[RCTConvert NSURL:@"http://localhost:8081/debugger-proxy"]];
}

to
- (instancetype)init
{
    return [self initWithURL:[RCTConvert NSURL:@"http://your-machine-ip-address:8081/debugger-proxy"]];
}
```

With this change, we can debug our application just like we have so far using the simulator.

Popular Modules for React Native

The React Native community may be young, but it has already started to flourish. Like any other popular framework, lots of open-source projects have been created, which helps developers to avoid re-inventing features already developed. You can find hundreds of such node modules on GitHub; we are listing a few that can help you quickly build and deliver your next React Native application.

react-native-fbsdk

React Native FBSDK is a wrapper for iOS Facebook SDK that helps Facebook integration with React Native apps. Documented JavaScript modules help access native components like login, sharing, and share using three npm modules. You have to include three npm packages: `react-native-fbsdkcore`, `react-native-fbsdkshare`, and `react-native-fbsdklogin`—that's it.

For complete documentation or to make a contribution to this project, you may visit the following: <https://github.com/facebook/react-native-fbsdk>.

react-native-scrollable-tab-view

Fluid animations really add meaning to any feature in your application. React Native animations API are getting robust very rapidly; creating smooth animations for scrolling within tabs is a problem solved by the `react-native-scrollable-tab-view` npm package. It's a very simple JavaScript-only implementation for React Native.

For complete documentation or to make a contribution to this project, you may visit the following: <https://github.com/brentvatne/react-native-scrollable-tab-view>.

react-native-webpack-server

`react-native-webpack-server` allows you to use the webpack ecosystem with React Native. It is a development server that uses Webpack Dev Server and the React Packager to enable building React Native JavaScript bundles with webpack.

For complete documentation or to make a contribution to this project, you may visit the following: <https://github.com/mjohnston/react-native-webpack-server>.

react-native-side-menu

A very simple implementation of an expandable side menu, this package helps you add a side menu that scrolls from either the left or right side with an easy swipe gesture.

For complete documentation or to make a contribution to this project, you may visit the following: <https://github.com/react-native-fellowship/react-native-side-menu#component-props>.

Here are some other notable mentions:

- `react-native-vector-icons`
- `react-native-facebook-login`
- `react-native-google-places-autocomplete`
- `react-native-sqlite-storage`
- `react-native-tableview`
- `react-native-background-geolocation`

Where to Go from Here

Congratulations. You have learned a lot about the React Native framework. But it's just the beginning, as the React Native community and ecosystem is expanding rapidly every day. New development, innovative changes, and cutting-edge features are getting added into the framework all the time—in short, there is something new to learn every day. For the bold, it's an opportunity to embrace. So the obvious question that comes to mind is, *What's next?* The answer is, *Plenty*.

If you are looking for React Native components that are already built and are available to be used, you can visit `react.parts`, which has an exhaustive list of all available reusable modules; see <https://react.parts/native>.

For any questions or queries, you can always use stackoverflow with tag `react-native`. For really quick responses from fellow developers, you can always hop on the IRC channel for React Native at `#reactnative` on IRC: `chat.freenode.net`. You can also chat with the React Native community using discord channel *Reactiflux* at <https://discordapp.com/invite/0ZcbPKXt5bWJVmUY> (yep, the React Native community is cool enough to use chat platforms used by gamers).

React Native is a very young framework, and while working on it you might come up with suggestions that could help the framework, or maybe some stuff that is overkill for the framework. The React Native community is very open to suggestions; if you have any you can start a thread at <https://discuss.reactjs.org/>. This discussion page is also a wonderful place for learning about best practices from others and to learn from old threads. Also, it's a great place to get your question resolved by the experts. Better yet, you might just get some job offers too.

Finally, to be a master in React Native there is no better way than to simply build apps. There is no faster way to become a master of a technology than by getting a little dirty with real-world projects, solving real-life problems. Since the React Native community is still in its infancy, there is a wonderful opportunity to learn and share by creating your own modules, which can help the rest of the community, an added benefit. We hope you enjoyed reading and learning with this book and have now evolved into a React Native developer building iOS apps. Just like you, we are very excited about React Native and look forward to seeing your work make a mark in the world of iOS and React Native. Adios.

Index

■ A, B

addElement function, [84](#)

■ C

Communication, [135](#)

 fetch, [137](#)

 server

 getting data, [139](#)

 saving data, [143](#)

 WebSocket, [136](#)

 XMLHttpRequest, [135](#)

componentDidMount method, [142](#)

■ D, E

Debugging

 code implementation, [165](#)

 developer menu, [164](#)

 device selection, [161](#)

 error message, [162](#)

 jsCodeLocation, [162](#)

 localhost, [163](#)

 React Native, [32](#)

 FPS Monitor, [35](#)

 in Chrome, [34](#)

 in Safari, [35](#)

 inspect element, [36](#)

 profiling, [38](#)

 reload option, [33](#)

 WebSocket error message, [165](#)

Device capabilities

 animations

 Animated library, [126](#)

 AnimationExample component, [132](#)

 componentDidMount, [133](#)

 getInitialState, [132](#)

 LayoutAnimation, [126](#)

AsyncStorage, [108](#)

 addMessage method, [112](#)

 async storage mechanism, [111](#)

 getInitialState, [112](#)

 React component, [111](#)

 updatedStorage method, [113](#)

 Update Storage, [110](#)

GeoLocation, [95, 98](#)

 annotations, [99](#)

 latitude and longitude, [100](#)

 MapView component, [97](#)

 parameters, [99](#)

NativeAlert, [114, 119, 123](#)

 alert and prompt method, [118](#)

 textForButton state, [123](#)

WebView, [124](#)

 code review, [125](#)

 with URL, [125](#)

Dispatch() method, [80](#)

■ F

Fetch, [137](#)

Flexbox

 Flex-direction, [48](#)

 flex values, [50](#)

 landscape mode screen, [48](#)

 NavigatorIOS. NavigatorIOS

 portrait mode screen, [47](#)

 styles, [46](#)

Flux

 actions, [80](#)

 benefits, [79](#)

 components, [78](#)

 definition, [75](#)

 dispatcher, [78–79](#)

 MVC pattern, [76](#)

 MVC problem, [76](#)

Flux (*cont.*)

- ReactJS application, 81
- React Native
 - add styles, 91
 - code implementation, 87
 - MainSection, 90, 92
 - ToDo application, 88
 - todo list, 93
- stores, 80

Functional Reactive Programming (FRP), 150

G

GeoLocation, 95

H, I

`_handleListProperty` function, 60–61

J, K

JSX, 8

L

ListView, 67

M

Model, view and controller (MVC) pattern, 76

MVC problems, 76

N

NavigatorIOS

- components, 42
- mainView component, 43
- toolbar, 45
- updated navigation bar, 44

O

`onClickHandler` function, 84

P, Q

Profiling, 38

R

React, 1

- advantages, 2
- components, 7
 - HTML properties, 12
 - state, 14
- installation and setup, 6
- JSX, 8
- one-way data flow, 4, 6
- problems, 2
- spaghetti relationship, 5
- two-way data binding, 4
- VDOM, 2

- components, 4
- working principle, 3

React Native, 20

- background-geolocation, 167
- debugging, 32 (see *also* Debugging)
 - FPS Monitor, 35
 - in Chrome, 34
 - in Safari, 35
 - inspect element, 36
 - profiling, 38
 - reload option, 33
- facebook-login, 167
- FBSDK, 166
- google-places-autocomplete, 167
- Hello World application, 22
 - destructuring, 27
 - iOS simulator, 22
 - render function, 27
 - require function, 27
 - strict mode, 26
 - Xcode editor, 24

HouseShare project, 30

- Android, 31
- AppDelegate.m, 31
- index.android.js, 31
- index.ios.js, 31
- iOS folder, 31
- node_modules, 31
- package.json, 31
- RCTRootView, 32

- installation, 20
 - cli module, 21
 - Node and npm, 21
 - updates, 21
 - Watchman, 21
- JavaScriptCore framework, 30
- live reload, 29
- prerequisites, 20
- RCTText, 28
- Redux. (see Redux)
- Reflux. (see Reflux)
- scrollable-tab-view, 166
- side menu, 166
- sqlite-storage, 167
- tableview, 167
- TouchableHighlight, 58
- vector-icons, 167
- webpack-server, 166
- web technologies, 29
- WebView-based, 29
- Redux
 - definition, 160
 - principles
 - Mutations Are Written as Pure Functions, 161
 - Single Source of Truth, 161
 - State Is Read-only, 161
- Reflux
 - Actions, 152
 - components, 153
 - definition, 149
 - vs. Flux, 150
 - pattern, 150
 - populated list with options, 159
 - RefluxTodo, 150
 - Stores, 152
 - TodoApp.js, 157
 - Todoltem component, 154
 - TodoList.js, 156, 158
 - updated index.ios.js, 158

- renderProperty function, 73
- _renderRow function, 74

S

- ScrollView, 73
- Separation of concerns (SoC), 6
- Server
 - getting data, 139
 - saving data, 143

T

- TouchableHighlight, 58

U

- User interface
 - add images, 54
 - Flexbox. Flexbox
 - _handleListProperty function, 61
 - List Property component, 61
 - ListView component, 67
 - ScrollView, 73
 - TouchableHighlight, 58

V

- Virtual DOM (VDOM)
 - components, 4
 - working principle, 3
- Virtual DOM (VDOM), 2

W

- waitFor() method, 80
- WebSocket, 136

X, Y, Z

- XMLHttpRequest, 135

