

Adam Boduch

React and React Native

Use React and React Native to build applications for desktop browsers, mobile browsers, and even as native mobile apps



Packt

React and React Native

Use React and React Native to build applications for desktop browsers, mobile browsers, and even as native mobile apps

Adam Boduch



BIRMINGHAM - MUMBAI

React and React Native

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2017

Production reference: 1280217

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-565-8

www.packtpub.com

Credits

Author

Adam Boduch

Copy Editor

Charlotte Carneiro

Reviewer

August Marcello III

Project Coordinator

Sheejal Shah

Commissioning Editor

Edward Gordon

Proofreader

Safis Editing

Acquisition Editor

Nitin Dasan

Indexer

Aishwarya Gangawane

Content Development Editor

Onkar Wani

Graphics

Jason Monteiro

Technical Editor

Prashant Mishra

Production Coordinator

Shantanu Zagade

About the Author

Adam Boduch has been involved with large-scale JavaScript development for nearly 10 years. Before moving to the front end, he worked on several large-scale cloud computing products, using Python and Linux. No stranger to complexity, Adam has practical experience with real-world software systems, and the scaling challenges they pose.

He is the author of several JavaScript books, including Flux Architecture, and is passionate about innovative user experiences and high performance.

Adam would like to acknowledge August Marcello III for all of his technical expertise and hard work that went into reviewing this book. Thanks buddy.

About the Reviewer

August Marcelllo III is a highly passionate software engineer with nearly two decades of experience in the design, implementation, and deployment of modern client-side web application architectures in the enterprise. An exclusive focus on delivering compelling SaaS-based user experiences throughout the Web ecosystem has proven both personally and professionally rewarding. His passion for emerging technologies in general, combined with a particular focus on forward-thinking JavaScript platforms, have been a primary driver in his pursuit of technical excellence. When he's not coding, he could be found trail running, mountain biking, and spending time with family and friends.

Many thanks to Chuck, Mark, Eric, and Adam, who I have had the privilege to work with and learn from. I'm grateful to my family, friends, and the experiences I have been blessed to be a part of.

www.PacktPub.com

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

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

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



<https://www.packtpub.com/mapt>

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

Why subscribe?

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786465655>.

If you'd like to join our team of regular reviewers, you can email us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

For Melissa, Jason, Simon, and Kevin

Table of Contents

Preface	1
<hr/>	
Chapter 1: Why React?	9
What is React?	9
React is just the view	10
Simplicity is good	11
Declarative UI structure	12
Time and data	13
Performance matters	14
The right level of abstraction	15
Summary	17
Chapter 2: Rendering with JSX	18
<hr/>	
What is JSX?	18
Hello JSX	18
Declarative UI structure	19
Just like HTML	20
Built-in HTML tags	20
HTML tag conventions	21
Describing UI structures	22
Creating your own JSX elements	23
Encapsulating HTML	23
Nested elements	25
Namespaced components	26
Using JavaScript expressions	29
Dynamic property values and text	29
Mapping collections to elements	30
Summary	32
Chapter 3: Understanding Properties and State	33
<hr/>	
What is component state?	33
What are component properties?	34
Setting component state	35
Initial component state	35
Setting component state	37
Merging component state	39

Passing property values	41
Default property values	41
Setting property values	42
Stateless components	45
Pure functional components	45
Defaults in functional components	47
Container components	48
Summary	50
Chapter 4: Event Handling – The React Way	51
Declaring event handlers	51
Declaring handler functions	52
Multiple event handlers	52
Importing generic handlers	53
Event handler context and parameters	56
Auto-binding context	56
Getting component data	57
Inline event handlers	59
Binding handlers to elements	60
Synthetic event objects	61
Event pooling	62
Summary	64
Chapter 5: Crafting Reusable Components	65
Reusable HTML elements	65
The difficulty with monolithic components	66
The JSX markup	67
Initial state and state helpers	68
Event handler implementation	70
Refactoring component structures	73
Start with the JSX	73
Implementing an article list component	74
Implementing an article item component	76
Implementing an add article component	78
Making components functional	80
Rendering component trees	82
Feature components and utility components	83
Summary	84
Chapter 6: The React Component Lifecycle	85
Why components need a lifecycle	85

Initializing properties and state	86
Fetching component data	87
Initializing state with properties	91
Updating state with properties	94
Optimize rendering efficiency	98
To render or not to render	98
Using metadata to optimize rendering	101
Rendering imperative components	103
Rendering jQuery UI widgets	103
Cleaning up after components	107
Cleaning up asynchronous calls	107
Summary	111
Chapter 7: Validating Component Properties	112
 Knowing what to expect	112
 Promoting portable components	113
 Simple property validators	114
Basic type validation	114
Requiring values	117
Any property value	120
 Type and value validators	122
Things that can be rendered	122
Requiring specific types	125
Requiring specific values	127
 Writing custom property validators	130
 Summary	132
Chapter 8: Extending Components	133
 Component inheritance	133
Inheriting state	134
Inheriting properties	136
Inheriting JSX and event handlers	139
 Composition with higher-order components	142
Conditional component rendering	143
Providing data sources	145
 Summary	149
Chapter 9: Handling Navigation with Routes	150
 Declaring routes	150
Hello route	150
Decoupling route declarations	152

Parent and child routes	154
Handling route parameters	156
Resource IDs in routes	156
Optional parameters	161
Using link components	164
Basic linking	164
URL and query parameters	165
Lazy routing	167
Summary	172
Chapter 10: Server-Side React Components	173
What is isomorphic JavaScript?	173
The server is a render target	173
Initial load performance	174
Sharing code between the backend and frontend	175
Rendering to strings	176
Backend routing	178
Frontend reconciliation	182
Fetching data	183
Summary	189
Chapter 11: Mobile-First React Components	190
The rationale behind mobile-first design	190
Using react-bootstrap components	192
Implementing navigation	193
Lists	197
Forms	202
Summary	209
Chapter 12: Why React Native?	210
What is React Native?	210
React and JSX are familiar	211
The mobile browser experience	212
Android and iOS, different yet the same	213
The case for mobile web apps	213
Summary	214
Chapter 13: Kickstarting React Native Projects	215
Using the React Native command-line tool	215
iOS and Android simulators	219
Xcode	219

Genymotion	220
Running the project	221
Running iOS apps	222
Running Android apps	224
Summary	227
Chapter 14: Building Responsive Layouts with Flexbox	228
Flexbox is the new layout standard	228
Introducing React Native styles	229
Building flexbox layouts	232
Simple three column layout	232
Improved three column layout	235
Flexible rows	239
Flexible grids	241
Flexible rows and columns	244
Summary	248
Chapter 15: Navigating Between Screens	249
Screen organization	249
Navigators, scenes, routes, and stacks	250
Responding to routes	250
Navigation bar	255
Dynamic scenes	258
Jumping back and forth	263
Summary	268
Chapter 16: Rendering Item Lists	269
Rendering data collections	269
Sorting and filtering lists	273
Fetching list data	282
Lazy list loading	285
Summary	288
Chapter 17: Showing Progress	289
Progress and usability	289
Indicating progress	289
Measuring progress	293
Navigation indicators	298
Step progress	302
Summary	306
Chapter 18: Geolocation and Maps	307

Where am I?	307
What's around me?	312
Annotating points of interest	313
Plotting points	314
Plotting overlays	315
Summary	320
Chapter 19: Collecting User Input	321
Collecting text input	321
Selecting from a list of options	326
Toggling between off and on	332
Collecting date/time input	336
Summary	342
Chapter 20: Alerts, Notifications, and Confirmation	343
Important information	343
Getting user confirmation	344
Success confirmation	344
Error confirmation	354
Passive notifications	359
Activity modals	366
Summary	369
Chapter 21: Responding to User Gestures	371
Scrolling with our fingers	371
Giving touch feedback	374
Swipeable and cancellable	379
Summary	386
Chapter 22: Controlling Image Display	387
Loading images	387
Resizing images	390
Lazy image loading	395
Rendering icons	400
Summary	404
Chapter 23: Going Offline	405
Detecting the state of the network	405
Storing application data	409
Synchronizing application data	414
Summary	422
Chapter 24: Handling Application State	423

Information architecture and Flux	423
Unidirectionality	423
Synchronous update rounds	424
Predictable state transformations	424
Unified information architecture	425
Implementing Redux	426
Initial application state	427
Creating the store	428
Store provider and routes	429
The App component	430
The Home component	434
State in mobile apps	438
Scaling the architecture	439
Summary	440
Chapter 25: Why Relay and GraphQL?	441
Yet another approach?	441
Verbose vernacular	442
Declarative data dependencies	443
Mutating application state	444
The GraphQL backend and microservices	446
Summary	446
Chapter 26: Building a Relay React App	447
TodoMVC and Relay	447
The GraphQL schema	448
Bootstrapping Relay	453
Adding todo items	455
Rendering todo items	458
Completing todo items	460
Summary	463
Index	464

Preface

About the book

I never had any interest in developing mobile apps. I used to believe strongly that it was the Web, or nothing, that there was no need for more yet more applications to install on devices that are already overflowing with apps. Then React Native happened. I was already writing React code for web applications and loving it. It turns out that I wasn't the only developer that balked at the idea of maintaining several versions of the same app using different tooling, environments, and programming languages. React Native was created out of a natural desire to take what works well from a web development experience standpoint (React), and apply it to native app development. Native mobile apps offer better user experiences than web browsers. It turns out I was wrong, we do need mobile apps for the time being. But that's okay, because React Native is a fantastic tool. This book is essentially my experience as a React developer for the Web and as a less experienced mobile app developer. React native is meant to be an easy transition for developers who already understand React for the Web. With this book, you'll learn the subtleties of doing React development in both environments. You'll also learn the conceptual theme of React, a simple rendering abstraction that can target anything. Today, it's web browsers and mobile devices. Tomorrow, it could be anything.

What this book covers

This book covers the following three parts:

- React: Chapter 1 to 11
- React Native: Chapter 12 to 23
- React Architecture: Chapter 23 to 26

Part I: React

Chapter 1, *Why React?*, covers the basics of what React really is, and why you want to use it.

Chapter 2, *Rendering with JSX*, explains that JSX is the syntax used by React to render content. HTML is the most common output, but JSX can be used to render many things, such as native UI components.

Chapter 3, *Understanding Properties and State*, shows how properties are passed to components, and how state re-renders components when it changes.

Chapter 4, *Event Handling—The React Way*, explains that events in React are specified in JSX. There are subtleties with how React processes events, and how your code should respond to them.

Chapter 5, *Crafting Reusable Components*, shows that components are often composed using smaller components. This means that you have to properly pass data and behaviour to child components.

Chapter 6, *The React Component Lifecycle*, explains how React components are created and destroyed all the time. There are several other lifecycle events that take place in between where you do things such as fetch data from the network.

Chapter 7, *Validating Component Properties*, shows that React has a mechanism that allows you to validate the types of properties that are passed to components. This ensures that there are no unexpected values passed to your component.

Chapter 8, *Extending Components*, provides an introduction to the mechanisms used to extend React components. These include inheritance and higher order components.

Chapter 9, *Handling Navigation with Routes*, navigation is an essential part of any web application. React handles routes declaratively using the `react-router` package.

Chapter 10, *Server-Side React Components*, discusses how React renders components to the DOM when rendered in the browser. It can also render components to strings, which is useful for rendering pages on the server and sending static content to the browser.

Chapter 11 *Mobile-First React Components*, explains that mobile web applications are fundamentally different from web applications designed for desktop screen resolutions. The `react-bootstrap` package can be used to build UIs in a mobilefirst fashion.

Part II: React Native

Chapter 12, *Why React Native?*, shows that React Native is React for mobile apps. If you've already invested in React for web applications, then why not leverage the same technology to provide a better mobile experience?

Chapter 13, *Kickstarting React Native Projects*, discusses that nobody likes writing boilerplate code or setting up project directories. React Native has tools to automate these mundane tasks.

Chapter 14, *Building Responsive Layouts with Flexbox*, explains why the Flexbox layout model is popular with web UI layouts using CSS. React Native uses the same mechanism to layout screens.

Chapter 15, *Navigating Between Screens*, discusses the fact that while navigation is an important part of web applications, mobile applications also need tools to handle how a user moves from screen to screen.

Chapter 16, *Rendering Item Lists*, shows that React Native has a list view component that's perfect for rendering lists of items. You simply provide it with a data source, and it handles the rest.

Chapter 17, *Showing Progress*, explains that progress bars are great for showing a determinate amount of progress. When you don't know how long something will take, you use a progress indicator. React Native has both of these components.

Chapter 18, *Geolocation and Maps*, shows that the `react-native-maps` package provides React Native with mapping capabilities. The Geolocation API that's used in web applications is provided directly by React Native.

Chapter 19, *Collecting User Input*, shows that most applications need to collect input from the user. Mobile applications are no different, and React Native provides a variety of controls that are not unlike HTML form elements.

Chapter 20, *Alerts, Notifications, and Confirmation*, explains that alerts are for interrupting the user to let them know something important has happened, notifications are unobtrusive updates, and confirmation is used for getting an immediate answer.

Chapter 21, *Responding to User Gestures*, discusses how gestures on mobile devices are something that's difficult to get right in the browser. Native apps, on the other hand, provide a much better experience for swiping, touching, and so on. React Native handles a lot of the details for you.

Chapter 22, *Controlling Image Display*, shows how images play a big role in most applications, either as icons, logos, or photographs of things. React Native has tools for loading images, scaling them, and placing them appropriately.

Chapter 23, *Going Offline*, explains that mobile devices tend to have volatile network connectivity. Therefore, mobile apps need to be able to handle temporary offline conditions. For this, React Native has local storage APIs.

Part III: React Architecture

Chapter 24, *Handling Application State*, discusses how application state is important for any React application, web or mobile. This is why understanding libraries such as Redux and Immutable.js is important.

Chapter 25, *Why Relay and GraphQL?*, explains that Relay and GraphQL, used together, is a novel approach to handling state at scale. It's a query and mutation language, plus a library for wrapping React components.

Chapter 26, *Building a Relay React App*, shows that the real advantage of Relay and GraphQL is that your state schema is shared between web and native versions of your application.

What you need for this book

- A code editor
- A modern web browser
- NodeJS

Who this book is for

This book is written for any JavaScript developer—beginner or expert—who wants to start learning how to put both of Facebook's UI libraries to work. No knowledge of React is needed, though a working knowledge of ES2015 will help you follow along better.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Instead of setting the actual `Modal` component to be transparent, we set the transparency in the `backgroundColor`, which gives the look of an overlay."

A block of code is set as follows:

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

import styles from './styles';

// Imports our own platform-independent "DatePicker"
// and "TimePicker" components.
import DatePicker from './DatePicker';
import TimePicker from './TimePicker';
```

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

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

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Again, the same principle with the `ToastAndroid` API applies here. You might have noticed that there's another button in addition to the **Show Notification** button."

Warnings or important notes appear in a box like this.



Tips and tricks appear like this.



Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

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

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

The code bundle for the book is also hosted on GitHub at

<https://github.com/PacktPublishing/React-and-React-Native>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from:

[https://www.packtpub.com/sites/default/files/downloads/ReactandReactNative_Colo
rImages.pdf](https://www.packtpub.com/sites/default/files/downloads/ReactandReactNative_ColorImages.pdf).

Errata

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

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

Piracy

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

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

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

Questions

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

1

Why React?

If you're reading this book, you might already have some idea of what React is. You also might have heard a React success story or two. If not, don't worry. I'll do my best to spare you from additional marketing literature in this opening chapter. However, this is a large book, with a lot of content, so I feel that setting the tone is an appropriate first step. Yes, the goal is to learn React and React Native. But, it's also to put together a lasting architecture that can handle everything we want to build with React today, and in the future.

This chapter starts with brief explanation of why React exists. Then, we'll talk about the simplicity that makes React an appealing technology and how React is able to handle many of the typical performance issues faced by web developers. Lastly, we'll go over the declarative philosophy of React and the level of abstraction that React programmers can expect to work with.

Let's go!

What is React?

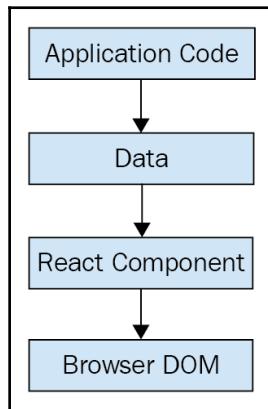
I think the one-line description of React on its homepage (<https://facebook.github.io/react>) is brilliant:

A JavaScript library for building user interfaces.

It's a library for building user interfaces. This is perfect, because as it turns out, this is all we want most of the time. I think the best part about this description is everything that it leaves out. It's not a mega framework. It's not a full-stack solution that's going to handle everything from the database to real-time updates over web socket connections. We don't actually want most of these pre-packaged solutions, because in the end, they usually cause more problems than they solve. Facebook sure did listen to what we want.

React is just the view

React is generally thought of as the *view* layer in an application. You might have used a library such as Handlebars or jQuery in the past. Just like jQuery manipulates UI elements, or Handlebars templates are inserted onto the page, React components change what the user sees. The following diagram illustrates where React fits in our frontend code:



This is literally all there is to React—the core concept. Of course there will be subtle variations to this theme as we make our way through the book, but the flow is more or less the same. We have some application logic that generates some data. We want to render this data to the UI, so we pass it to a React component, which handles the job of getting the HTML into the page.

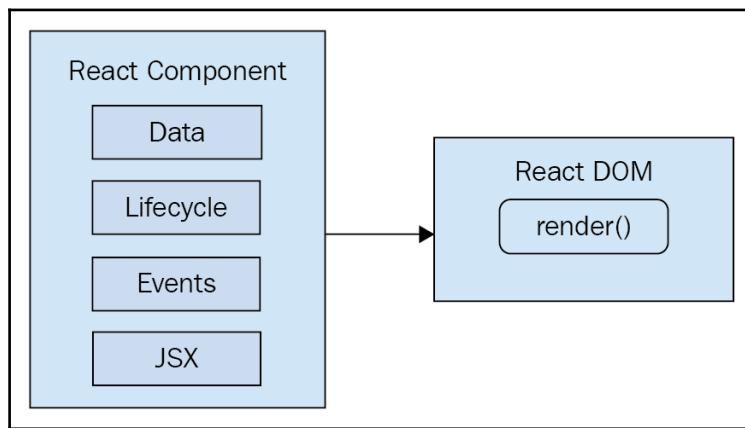
You may wonder what the big deal is, especially since at the surface, React appears to be yet another rendering technology. We'll touch on some of the key areas where React can simplify application development in the remaining sections of the chapter.



Don't worry; we're almost through the introductory stuff. Awesome code examples are on the horizon!

Simplicity is good

React doesn't have many moving parts to learn about and understand. Internally, there's a lot going on, and we'll touch on these things here and there throughout the book. The advantage to having a small API to work with is that you can spend more time familiarizing yourself with it, experimenting with it, and so on. The opposite is true of large frameworks, where all your time is devoted to figuring out how everything works. The following diagram gives a rough idea of the APIs that we have to think about when programming with React:



React is divided into two major APIs. First, there's the React DOM. This is the API that's used to perform the actual rendering on a web page. Second, there's the React component API. These are the parts of the page that are actually rendered by React DOM. Within a React component, we have the following areas to think about:

- **Data:** This is data that comes from somewhere (the component doesn't care where), and is rendered by the component
- **Lifecycle:** These are methods that we implement that respond to changes in the lifecycle of the component. For example, the component is about to be rendered
- **Events:** This is code that we write for responding to user interactions
- **JSX:** This is the syntax of React components used to describe UI structures

Don't fixate on what these different areas of the React API represent just yet. The takeaway here is that React, by nature, is simple. Just look at how little there is to figure out! This means that we don't have to spend a ton of time going through API details here. Instead, once you pick up on the basics, we can spend more time on nuanced React usage patterns.

Declarative UI structure

React newcomers have a hard time coming to grips with the idea that components mix markup in with their JavaScript. If you've looked at React examples and had the same adverse reaction, don't worry. Initially, we're all skeptical of this approach, and I think the reason is that we've been conditioned for decades by the **separation of concerns** principle. Now, whenever we see things mixed together, we automatically assume that this is bad and shouldn't happen.

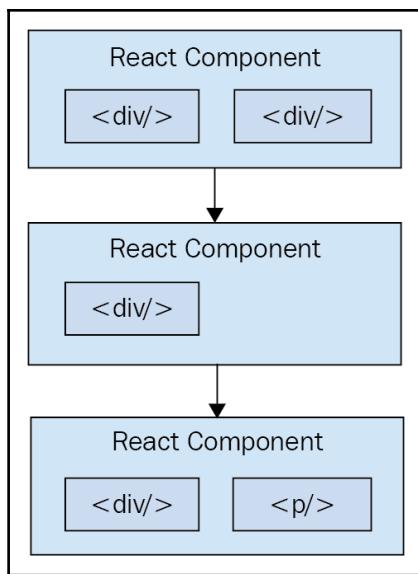
The syntax used by React components is called **JSX (JavaScript XML)**. The idea is actually quite simple. A component renders content by returning some JSX. The JSX itself is usually HTML markup, mixed with custom tags for the React components. The specifics don't matter at this point; we'll get into details in the coming chapters. What's absolutely groundbreaking here is that we don't have to perform little micro-operations to change the content of a component.

For example, think about using something like jQuery to build your application. You have a page with some content on it, and you want to add a class to a paragraph when a button is clicked. Performing these steps is easy enough, but the challenge is that there are steps to perform at all. This is called **imperative programming**, and it's problematic for UI development. While this example of changing the class of an element in response to an event is simple, real applications tend to involve more than three or four steps to make something happen.

React components don't require executing steps in an imperative way to render content. This is why JSX is so central to React components. The XML-style syntax makes it easy to describe what the UI should look like. That is, what are the HTML elements that this component is going to render? This is called **declarative programming**, and is very well suited for UI development.

Time and data

Another area that's difficult for React newcomers to grasp is the idea that JSX is like a static string, representing a chunk of rendered output. Are we just supposed to keep rendering this same view? This is where time and data come into play. React components rely on data being passed into them. This data represents the dynamic aspects of the UI. For example, a UI element that's rendered based on a Boolean value could change the next time the component is rendered. Here's an illustration of the idea:

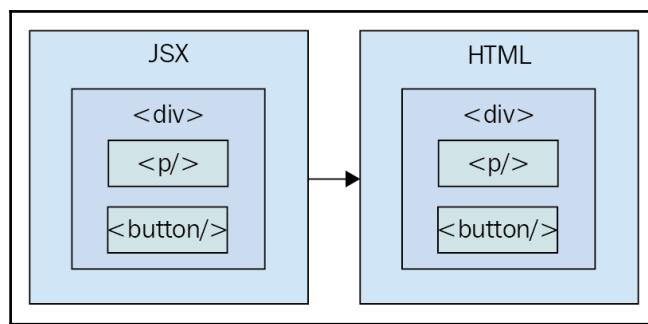


Each time the React component is rendered, it's like taking a snapshot of the JSX at that exact moment in time. As our application moves forward through time, we have an ordered collection of rendered user interface components. In addition to declaratively describing what a UI should be, re-rendering the same JSX content makes things much easier for developers. The challenge is making sure that React can handle the performance demands of this approach.

Performance matters

Using React to build user interfaces means that we can declare the structure of the UI with JSX. This is less error-prone than the imperative approach to assembling the UI piece by piece. However, the declarative approach does present us with one challenge: **performance**.

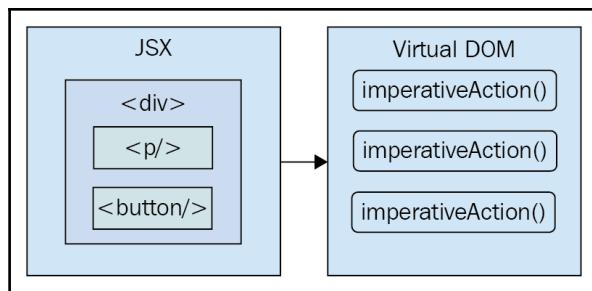
For example, having a declarative UI structure is fine for the initial rendering, because there's nothing on the page yet. So, the React renderer can look at the structure declared in JSX, and render it into the browser DOM. This is illustrated in the following diagram:



On the initial render, React components and their JSX are no different from other template libraries. For instance, Handlebars will render a template to HTML markup as a string, which is then inserted into the browser DOM. Where React is different from libraries such as Handlebars, is when data changes and we need to re-render the component. Handlebars will just rebuild the entire HTML string, the same way it did on the initial render. Since this is problematic for performance, we often end up implementing imperative workarounds that manually update tiny bits of the DOM. What we end up with is a tangled mess of declarative templates and imperative code to handle the dynamic aspects of the UI.

We don't do this in React. This is what sets React apart from other view libraries. Components are declarative for the initial render, and they stay this way even as they're re-rendered. It's what React does under the hood that makes re-rendering declarative UI structures possible.

React has something called the **virtual DOM**, which is used to keep a representation of the real DOM elements in memory. It does this so that each time we re-render a component, it can compare the new content, to the content that's already displayed on the page. Based on the difference, the virtual DOM can execute the imperative steps necessary to make the changes. So not only do we get to keep our declarative code when we need to update the UI, React will also make sure that it's done in a performant way. Here's what this process looks like:



When you read about React, you'll often see words like **diffing** and **patching**. Diffing means comparing old content with new content to figure out what's changed. Patching means executing the necessary DOM operations to render the new content.

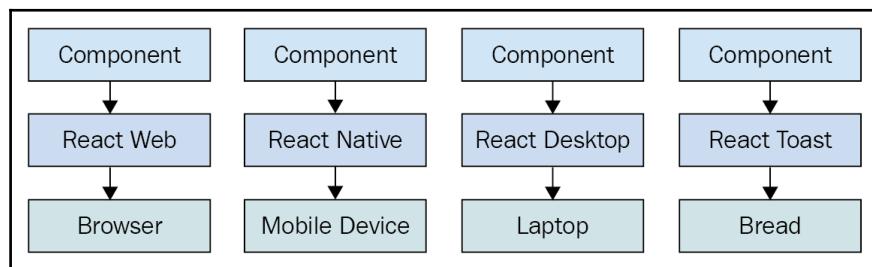
The right level of abstraction

The final topic I want to cover at a high level before we dive into React code is **abstraction**. React doesn't have a lot of it, and yet the abstractions that React implements are crucial to its success.

In the preceding section, you saw how JSX syntax translates to low-level operations that we have no interest in maintaining. The more important way to look at how React translates our declarative UI components is the fact that we don't necessarily care what the render target is. The render target happens to be the browser DOM with React. But, this is changing.

The theme of this book is that React has the potential to be used for any user interface we want to create, on any conceivable device. We're only just starting to see this with React Native, but the possibilities are endless. I personally will not be surprised when React Toast becomes a thing, targeting toasters that can singe the rendered output of JSX on to bread. The abstraction level with React is at the right level, and it's in the right place.

The following diagram gives you an idea of how React can target more than just the browser:



From left to right, we have React Web (just plain React), React Native, React Desktop, and React Toast. As you can see, to target something new, the same pattern applies:

- Implement components specific to the target
- Implement a React renderer that can perform the platform-specific operations under the hood
- Profit

This is obviously an oversimplification of what's actually implemented for any given React environment. But the details aren't so important to us. What's important is that we can use our React knowledge to focus on describing the structure of our user interface on any platform.

React Toast will probably never be a thing, unfortunately.



Summary

In this chapter, you were introduced to React at a high level. React is a library, with a small API, used to build user interfaces. Next, you were introduced to some of the key concepts of React. First, we discussed the fact that React is simple, because it doesn't have a lot of moving parts. Next, we looked at the declarative nature of React components and JSX. Then, you learned that React takes performance seriously, and that this is how we're able to write declarative code that can be re-rendered over and over. Finally, we thought about the idea of render targets and how React can easily become the UI tool of choice for all of them.

That's enough introductory and conceptual stuff for my taste. As we make our way toward the end of the book, we'll revisit these ideas, as they're important strategy-wise. For now, let's take a step back and nail down the basics, starting with JSX.

2

Rendering with JSX

This chapter will introduce you to JSX. We'll start by covering the basics: what is JSX? Then, you'll see that JSX has built-in support for HTML tags, as you would expect; so we'll run through a few examples here. After having looked at some JSX code, we'll discuss how it makes describing the structure of UIs easy for us. Then, we'll jump into building our own JSX elements, and using JavaScript expressions for dynamic content.

Ready?

What is JSX?

In this section, we'll implement the obligatory *hello world* JSX application. At this point, we're just dipping our toes into the water; more in-depth examples will follow. We'll also discuss what makes this syntax work well for declarative UI structures.

Hello JSX

Without further ado, here's your first JSX application:

```
// The "render()" function will render JSX markup and
// place the resulting content into a DOM node. The "React"
// object isn't explicitly used here, but it's used
// by the transpiled JSX source.
import React from 'react';
import { render } from 'react-dom';

// Renders the JSX markup. Notice the XML syntax
// mixed with JavaScript? This is replaced by the
// transpiler before it reaches the browser.
render(
```

```
(<p>Hello, <strong>JSX</strong></p>),  
document.getElementById('app')  
>;
```

Pretty simple, right? Let's walk through what's happening here. First, we need to import the relevant bits. The `render()` function is what really matters in this example, as it takes JSX as the first argument and renders it to the DOM node passed as the second argument.



The parentheses surrounding the JSX markup aren't strictly necessary. However, this is a React convention, and it helps us to avoid confusing markup constructs with JavaScript constructs.

The actual JSX content in this example fits on one line, and it simply renders a paragraph with some bold text inside. There's nothing fancy going on here, so we could have just inserted this markup into the DOM directly as a plain string. However, there's a lot more to JSX than what's shown here. The aim of this example was to show the basic steps involved in getting JSX rendered onto the page. Now, let's talk a little bit about declarative UI structure.



JSX is transpiled into JavaScript statements; browsers have no idea what JSX is. I would highly recommend downloading the companion code for this book from <https://github.com/PacktPublishing/React-and-React-Native>, and running it as you read along. Everything compiles automatically for you; you just need to follow the simple installation steps.

Declarative UI structure

Before we continue to plow forward with our code examples, let's take a moment to reflect on our *hello world* example. The JSX content was short and simple. It was also **declarative**, because it described what to render, not how to render it. Specifically, by looking at the JSX, you can see that this component will render a paragraph, and some bold text within it. If this were done imperatively, there would probably be some more steps involved, and they would probably need to be performed in a specific order.

So, the example we just implemented should give you a feel for what declarative React is all about. As we move forward in this chapter and throughout the book, the JSX markup will grow more elaborate. However, it's always going to describe what is in the user interface. Let's move on.

Just like HTML

At the end of the day, the job of a React component is to render HTML into the browser DOM. This is why JSX has support for HTML tags, out of the box. In this section, we'll look at some code that renders some of the available HTML tags. Then, we'll cover some of the conventions that are typically followed in React projects when HTML tags are used.

Built-in HTML tags

When we render JSX, element tags are referencing React components. Since it would be tedious to have to create components for HTML elements, React comes with HTML components. We can render any HTML tag in our JSX, and the output will be just as we'd expect. If you're not sure, you can always run the following code to see which HTML element tags React has:

```
// Prints a list of the global HTML tags
// that React knows about.
console.log(
  'available tags',
  Object.keys(React.DOM).sort()
);
```

You can see that `React.DOM` has all the built-in HTML elements that we need, implemented as React components. Now let's try rendering some of these:

```
import React from 'react';
import { render } from 'react-dom';

// React internal defines all the standard HTML tags
// that we use on a daily basis. Think of them being
// the same as any other react component.
render(
  <div>
    <button />
    <code />
    <input />
    <label />
    <p />
    <pre />
    <select />
    <table />
    <ul />
  </div>
),
document.getElementById('app')
```

) ;

Don't worry about the rendered output for this example; it doesn't make sense. All we're doing here is making sure that we can render arbitrary HTML tags, and they render as expected.



You may have noticed the surrounding `<div>` tag, grouping together all of the other tags as its children. This is because React needs a root component to render; you can't render adjacent elements, like `(<p><p><p>)` for instance.

HTML tag conventions

When we render HTML tags in JSX markup, the expectation is that we'll use lowercase for the tag name. In fact, capitalizing the name of an HTML tag will straight up fail. Tag names are case sensitive and non-HTML elements are capitalized. This way, it's easy to scan the markup and spot the built-in HTML elements versus everything else.

We can also pass HTML elements any of their standard properties. When we pass them something unexpected, a warning about the unknown property is logged. Here's an example that illustrates these ideas.

```
import React from 'react';
import { render } from 'react-dom';

// This renders as expected, except for the "foo"
// property, since this is not a recognized button
// property. This will log a warning in the console.
render(
  <button foo="bar">
    My Button
  </button>
),
  document.getElementById('app')
);

// This fails with a "ReferenceError", because
// tag names are case-sensitive. This goes against
// the convention of using lower-case for HTML tag names.
render(
  <Button />,
  document.getElementById('app')
);
```



Later on in the book, we'll cover property validation for the components that we make. This avoids silent misbehavior as seen with the `foo` property in this example.

Describing UI structures

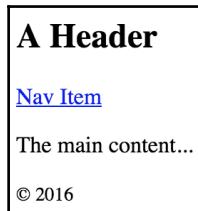
JSX is the best way to describe complex UI structures. Let's look at some JSX markup that declares a more elaborate structure than a single paragraph:

```
import React from 'react';
import { render } from 'react-dom';

// This JSX markup describes some fairly-sophisticated
// markup. Yet, it's easy to read, because it's XML and
// XML is good for concisely-expressing hierarchical
// structure. This is how we want to think of our UI,
// when it needs to change, not as an individual element
// or property.
render((
  <section>
    <header>
      <h1>A Header</h1>
    </header>
    <nav>
      <a href="item">Nav Item</a>
    </nav>
    <main>
      <p>The main content...</p>
    </main>
    <footer>
      <small>&copy; 2016</small>
    </footer>
  </section>
),
document.getElementById('app')
);
```

As you can see, there's a lot of semantic elements in this markup, describing the structure of the UI. The key is that this type of complex structure is easy to reason about, and we don't need to think about rendering specific parts of it. But before we start implementing dynamic JSX markup, let's create some of our own JSX components.

Here is what the rendered content looks like:



Creating your own JSX elements

Components are the fundamental building blocks of React. In fact, components are the vocabulary of JSX markup. In this section, we'll see how to encapsulate HTML markup within a component. We'll build examples that show you how to nest custom JSX elements and how to namespace your components.

Encapsulating HTML

The reason that we want to create new JSX elements is so that we can encapsulate larger structures. This means that instead of having to type out complex markup, we just use our custom tag. The React component returns the JSX that replaces the element. Let's look at an example now:

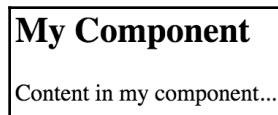
```
// We also need "Component" so that we can
// extend it and make a new JSX tag.
import React, { Component } from 'react';
import { render } from 'react-dom';

// "MyComponent" extends "Component", which means that
// we can now use it in JSX markup.
class MyComponent extends Component {
  render() {
    // All components have a "render()" method, which
    // returns some JSX markup. In this case, "MyComponent"
    // encapsulates a larger HTML structure.
    return (
      <section>
        <h1>My Component</h1>
        <p>Content in my component...</p>
      </section>
    );
  }
}
```

```
}

// Now when we render "<MyComponent>" tags, the encapsulated
// HTML structure is actually rendered. These are the
// building blocks of our UI.
render(
  <MyComponent />,
  document.getElementById('app')
);
```

Here's what the rendered output looks like:



This is the first React component that we've implemented in this book, so let's take a moment to dissect what's going on here. We've created a class called `MyComponent` that extends the `Component` class from React. This is how we create a new JSX element. As you can see in the call to `render()`, we're rendering a `<MyComponent>` element.

The HTML that this component encapsulates is returned by the `render()` method. In this case, when the JSX `<MyComponent>` is rendered by `react-dom`, it's replaced by a `<section>` element, and everything within it.



When we render JSX, any custom elements we use must have their corresponding React component within the same scope. In the preceding example, the class `MyComponent` was declared in the same scope as the call to `render()`, so everything worked as expected. Usually, we'll import components, adding them to the appropriate scope. We'll see more of this as we progress through the book.

Nested elements

Using JSX markup is useful for describing UI structures that have parent-child relationships. For example, a `` tag is only useful as the child of a `` or `` tag. Therefore, we're probably going to make similar nested structures with our own React components. For this, you need to use the `children` property. Let's see how this works; here's the JSX markup that we want to render:

```
import React from 'react';
import { render } from 'react-dom';

// Imports our two components that render children...
import MySection from './MySection';
import MyButton from './MyButton';

// Renders the "MySection" element, which has a child
// component of "MyButton", which in turn has child text.
render(
  <MySection>
    <MyButton>My Button Text</MyButton>
  </MySection>
),
document.getElementById('app')
);
```

Here, you can see that we're importing two of our own React components: `MySection` and `MyButton`. Now, if you look at the JSX that we're rendering, you'll notice that `<MyButton>` is a child of `<MySection>`. You'll also notice that the `MyButton` component accepts text as its child, instead of more JSX elements. Let's see how these components work, starting with `MySection`:

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

// Renders a "<section>" element. The section has
// a heading element and this is followed by
// "this.props.children".
export default class MySection extends Component {
  render() {
    return (
      <section>
        <h2>My Section</h2>
        {this.props.children}
      </section>
    );
  }
}
```

This component renders a standard `<section>` HTML element, a heading, and then `{this.props.children}`. It's this last construct that allows components to access nested elements or text, and to render it.



The two braces used in the preceding example are used for JavaScript expressions. We'll touch on more details of the JavaScript expression syntax found in JSX markup in the following section.

Now, let's look at the `MyButton` component:

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

// Renders a "<button>" element, using
// "this.props.children" as the text.
export default class MyButton extends Component {
  render() {
    return (
      <button>{this.props.children}</button>
    );
  }
}
```

This component is using the exact same pattern as `MySection`; take the `{this.props.children}` value, and surround it with meaningful markup. React handles a lot of messy details for us. In this example, the button text is a child of `MyButton`, which is in turn a child of `MySection`. However, the button text is transparently passed through `MySection`. In other words, we didn't have to write any code in `MySection` to make sure that `MyButton` got its text. Pretty cool, right? Here's what the rendered output looks like:



Namespaced components

The custom elements we've created so far have used simple names. Sometimes, we might want to give a component a namespace. Instead of writing `<MyComponent>` in our JSX markup, we would write `<MyNamespace.MyComponent>`. This makes it clear to anyone reading the JSX that `MyComponent` is part of `MyNamespace`.

Typically, `MyNamespace` would also be a component. The idea with **namespacing** is to have a namespace component render its child components using the namespace syntax. Let's take a look at an example:

```
import React from 'react';
import { render } from 'react-dom';

// We only need to import "MyComponent" since
// the "First" and "Second" components are part
// of this "namespace".
import MyComponent from './MyComponent';

// Now we can render "MyComponent" elements,
// and it's "namespaced" elements as children.
// We don't actually have to use the namespaced
// syntax here, we could import the "First" and
// "Second" components and render them without the
// "namespace" syntax. It's a matter of readability
// and personal taste.
render((
  <MyComponent>
    <MyComponent.First />
    <MyComponent.Second />
  </MyComponent>
),
  document.getElementById('app')
);
```

This markup renders a `<MyComponent>` element with two children. The key thing to notice here is that instead of writing `<First>`, we write `<MyComponent.First>`. Same with `<MyComponent.Second>`. The idea is that we want to explicitly show that `First` and `Second` belong to `MyComponent`, within the markup.



I personally don't depend on namespaced components like these, because I'd rather see which components are in use by looking at the `import` statements at the top of the module. Others would rather import one component and explicitly mark the relationship within the markup. There is no correct way to do this; it's a matter of personal taste.

Now let's take a look at the `MyComponent` module:

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

// The "First" component, renders some basic JSX...
class First extends Component {
  render() {
```

```
        return (
          <p>First...</p>
        );
      }
    }

// The "Second" component, renders some basic JSX...
class Second extends Component {
  render() {
    return (
      <p>Second...</p>
    );
  }
}

// The "MyComponent" component renders it's children
// in a "<section>" element.
class MyComponent extends Component {
  render() {
    return (
      <section>
        {this.props.children}
      </section>
    );
  }
}

// Here is where we "namespace" the "First" and
// "Second" components, by assigning them to
// "MyComponent" as class properties. This is how
// other modules can render them as "<MyComponent.First>"
// elements.
MyComponent.First = First;
MyComponent.Second = Second;

export default MyComponent;

// This isn't actually necessary. If we want to be able
// to use the "First" and "Second" components independent
// of "MyComponent", we would leave this in. Otherwise,
// we would only export "MyComponent".
export { First, Second };
```

You can see that this module declares `MyComponent` as well as the other components that fall under this namespace (`First` and `Second`). The idea is to assign the components to the namespace component (`MyComponent`) as class properties. There are a number of things we could change in this module. For example, we don't have to directly export `First` and `Second` since they're accessible through `MyComponent`. We also don't need to define everything in the same module; we could import `First` and `Second` and assign them as class properties. Using namespaces is completely optional, and if you use them, you should use them consistently.

Using JavaScript expressions

As we saw in the preceding section, JSX has special syntax that lets us embed JavaScript expressions. Any time we render JSX content, expressions in the markup are evaluated. This is the dynamic aspect of JSX content, and in this section, you'll learn how to use expressions to set property values and element text content. You'll also learn how to map collections of data to JSX elements.

Dynamic property values and text

Some HTML property or text values are static, meaning that they don't change as the JSX is re-rendered. Other values, the values of properties or text, are based on data that's found elsewhere in the application. Remember, React is just the view layer. Let's look at an example so that you can get a feel for what the JavaScript expression syntax looks like in JSX markup:

```
import React from 'react';
import { render } from 'react-dom';

// These constants are passed into the JSX
// markup using the JavaScript expression syntax.
const enabled = false;
const text = 'A Button';
const placeholder = 'input value...';
const size = 50;

// We're rendering a "<button>" and an "<input>"
// element, both of which use the "{}" JavaScript
// expression syntax to fill in property, and text
// values.
render(
  <section>
```

```
<button disabled={!enabled}>{text}</button>
<input placeholder={placeholder} size={size} />
</section>
),
document.getElementById('app')
);
```

Anything that is a valid JavaScript expression can go in between the braces: `{ }`. For properties and text, this is often a variable name or object property. Notice in this example, that the `!enabled` expression computes a boolean value. Here's what the rendered output looks like:



```
A Button input value...
```



If you're following along with the downloadable companion code, which I strongly recommend doing, try playing with these values, and seeing how the rendered HTML changes.

Mapping collections to elements

Sometimes we need to write JavaScript expressions that change the structure of our markup. In the preceding section, we looked at using JavaScript expression syntax to dynamically change the property values of JSX elements. What about when we need to add or remove elements based on a JavaScript collection?



Throughout the book, when I refer to a JavaScript **collection**, I'm referring to both plain objects and arrays. Or more generally, anything that's iterable.

The best way to control JSX elements dynamically is to map them from a collection. Let's look at an example of how this is done:

```
import React from 'react';
import { render } from 'react-dom';

// An array that we want to render as a list...
const array = [
  'First',
  'Second',
  'Third',
```

```
];
// An object that we want to render as a list...
const object = {
  first: 1,
  second: 2,
  third: 3,
};

render((
  <section>
    <h1>Array</h1>

    { /* Maps "array" to an array of "<li>"s.
       Note the "key" property on "<li>".
       This is necessary for performance reasons,
       and React will warn us if it's missing. */ }
    <ul>
      {array.map(i => (
        <li key={i}>{i}</li>
      )))
    </ul>
  <h1>Object</h1>

  { /* Maps "object" to an array of "<li>"s.
       Note that we have to use "Object.keys()"
       before calling "map()" and that we have
       to lookup the value using the key "i". */ }
  <ul>
    {Object.keys(object).map(i => (
      <li key={i}>
        <strong>i:</strong>{object[i]}
      </li>
    )))
  </ul>
</section>
),
document.getElementById('app')
);
```

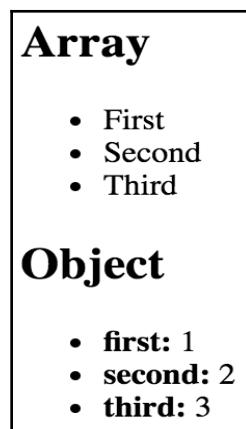
The first collection is an array called `array`, populated with string items. Moving down to the JSX markup, you can see that we're making a call to `array.map()`, which will return a new array. The mapping function is actually returning a JSX element (``), meaning that each item in the array is now represented in the markup.



The result of evaluating this expression is an array. Don't worry; JSX knows how to render arrays of elements.

The object collection uses the same technique, except we have to call `Object.keys()` and then map this array. What's nice about mapping collections to JSX elements on the page is that we can drive the structure of React components based on collection data. This means that we don't have to rely on imperative logic to control the UI.

Here's what the rendered output looks like:



Summary

In this chapter, you learned the basics about JSX, including its declarative structure and why this is a good thing. Then, you wrote some code to render some basic HTML and learned about describing complex structures using JSX.

Next, you spent some time learning about extending the vocabulary of JSX markup by implementing your own React components, the fundamental building blocks of your UI. Finally, you learned how to bring dynamic content into JSX element properties, and how to map JavaScript collections to JSX elements, eliminating the need for imperative logic to control UI display.

Now that you have a feel for what it's like to render UIs by embedding declarative XML in your JavaScript modules, it's time to move on to the next chapter where we'll take a deeper look at component properties and state.

3

Understanding Properties and State

React components rely on JSX syntax, which is used to describe the structure of the UI. JSX will only get us so far—we need data to fill in the structure of our React components. The focus of this chapter is component data, which comes in two varieties: *properties* and *state*.

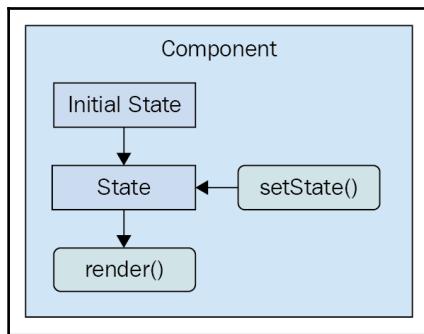
We'll start things off by defining what is meant by properties and state. Then, we'll walk through some examples that show you the mechanics of setting component state, and passing component properties. Toward the end of this chapter, we'll build on your newfound knowledge of props and state and introduce functional components and the container pattern.

What is component state?

React components declare the structure of a UI element using JSX. But this is only part of the story. Components need data if they are to be useful. For example, your component JSX might declare a `` that maps a JavaScript collection to `` elements. Where does this collection come from?

State is the dynamic part of a React component. This means that you can declare the initial state of a component, which changes over time.

Imagine that we're rendering a component where a piece of state is initialized to an empty array. Later on, this array is populated with data. This is called a **change in state**, and whenever we tell a React component to change its state, the component will automatically re-render itself. The process is visualized here:



The state of a component is something that either the component itself can set, or other pieces of code, outside of the component. Now we'll look at component properties and how they differ from component state.

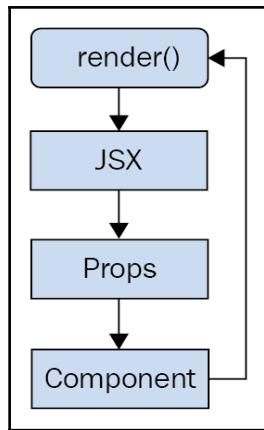
What are component properties?

Properties are used to pass data into your React components. Instead of calling a method with new state as the argument value, properties are passed only when the component is rendered. That is, we pass property values to JSX elements.



In the context of JSX, properties are called **attributes**, probably because that's what they're called in XML parlance. In this book, properties and attributes are synonymous with one another.

Properties are different than state because they're not something that's changed after the initial render of the component. If a property value has changed, and we want to re-render the component, then we have to re-render the JSX that was used to render it in the first place. The React internals take care of making sure this is done efficiently. Here's an illustration of rendering and re-rendering a component using properties:



This looks a lot different than a stateful component. The real difference is that with properties, it's often a parent component that decides when to render the JSX. The component doesn't actually know how to re-render itself. As we'll see throughout this book, this type of top-down flow is easier to predict than state that changes all over the place.

With the introductory explanations out of the way, let's make sense of these two concepts by writing some code.

Setting component state

In this section, you're going to write some React code that sets the state of components. First, you'll learn about the initial state—this is the default state of a component. Next, you'll learn how to change the state of a component, causing it to re-render itself. Finally, you'll see how new state is merged with existing state.

Initial component state

The initial state of the component isn't actually required, but if your component uses state, it should be set. This is because if the component JSX expects certain state properties to be there and they aren't, then the component will either fail or render something unexpected. Thankfully, it's easy to set the initial component state.

The initial state of a component should always be an object with one or more properties. For example, you might have a component that uses a single array as its state. This is fine, but just make sure that you set the initial array as a property of the state object. Don't use an array as the state. The reason for this is simple: consistency. Every react component uses a plain object as its state.

Let's turn our attention to some code now. Here's a component that sets an initial state object:

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

export default class MyComponent extends Component {
  // The initial state is set as a simple property
  // of the component instance.
  state = {
    first: false,
    second: true,
  }

  render() {
    // Gets the "first" and "second" state properties
    // into constants, making our JSX less verbose.
    const { first, second } = this.state;

    // The returned JSX uses the "first" and "second"
    // state properties as the "disabled" property
    // value for their respective buttons.
    return (
      <main>
        <section>
          <button disabled={first}>First</button>
        </section>
        <section>
          <button disabled={second}>Second</button>
        </section>
      </main>
    );
  }
}
```

If you look at the JSX that's returned by `render()`, you can actually see the state values that this component depends on—`first` and `second`. Since we've set these properties up in the initial state, we're safe to render the component, and there won't be any surprises. For example, we could render this component only once, and it would render as expected, thanks to the initial state:

```
import React from 'react';
```

```
import { render } from 'react-dom';

import MyComponent from './MyComponent';

// "MyComponent" has an initial state, nothing is passed
// as a property when it's rendered.
render(
  (<MyComponent />),
  document.getElementById('app')
);
```

Here's what the rendered output looks like:



Setting the initial state isn't very exciting, but it's important nonetheless. Let's make the component re-render itself when the state is changed.

Setting component state

Let's create a component that has some initial state. We'll then render this component, and update its state. This means that the component will be rendered twice. Let's take a look at the component so that you can see what we're working with here:

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

export default class MyComponent extends Component {
  // The initial state is used, until something
  // calls "setState()", at which point the state is
  // merged with this state.
  state = {
    heading: 'React Awesomesauce (Busy)',
    content: 'Loading...',
  }

  render() {
    const { heading, content } = this.state;

    return (
      <main>
        <h1>{heading}</h1>
        <p>{content}</p>
      </main>
    );
  }
}
```

```
) ;  
}  
}
```

As you can see, the JSX of this component depends on two state properties—heading and content. The component also sets the initial values of these two state properties, which means that it can be rendered without any unexpected gotchas. Now, let's look at some code that renders the component, and then re-renders it by changing the state:

```
import React from 'react';  
import { render } from 'react-dom';  
  
import MyComponent from './MyComponent';  
  
// The "render()" function returns a reference to the  
// rendered component. In this case, it's an instance  
// of "MyComponent". Now that we have the reference,  
// we can call "setState()" on it whenever we want.  
const myComponent = render(  
  (<MyComponent />),  
  document.getElementById('app')  
) ;  
  
// After 3 seconds, set the state of "myComponent",  
// which causes it to re-render itself.  
setTimeout(() => {  
  myComponent.setState({  
    heading: 'React Awesomesauce',  
    content: 'Done!',  
  });  
, 3000);
```

The component is first rendered with its default state. However, the interesting spot in this code is the `setTimeout()` call. After 3 seconds, it uses `setState()` to change the two state property values. Sure enough, this change is reflected in the UI. Here's what the initial state looks like when rendered:



React Awesomesauce (Busy)

Loading...

Here's what the rendered output looks like after the state change:



This example highlights the power of having declarative JSX syntax to describe the structure of the UI component. We declare it once, and update the state of the component over time to reflect changes in the application as they happen. All the DOM interactions are optimized and hidden from view. Pretty cool, huh?

In this example, we actually replaced the entire component state. That is, the call to `setState()` passed in the same object properties found in the initial state. But, what if we only want to update part of the component state?

Merging component state

When you set the state of a React component, you're actually merging the state of the component with the object that you pass to `setState()`. This is useful because it means that you can set part of the component state while leaving the rest of the state as it is. Let's look at an example now. First, a component with some state:

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

export default class MyComponent extends Component {
  // The initial state...
  state = {
    first: 'loading...',
    second: 'loading...',
    third: 'loading...',
  }

  render() {
    const { state } = this;

    // Renders a list of items from the
    // component state.
    return (
      <ul>
        {Object.keys(state).map(i => (
          <li key={i}>
            <strong>i:</strong> {state[i]}
          </li>
        ))}
      </ul>
    );
  }
}
```

```
        ) ) )
      </ul>
    );
}
}
```

This component renders the keys and values of its state. Each value defaults to `loading...`, because we don't yet know the value. Let's write some code that sets the state of each state property individually:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

// Stores a reference to the rendered component...
const myComponent = render(
  <MyComponent />,
  document.getElementById('app')
);

// Change part of the state after 1 second...
setTimeout(() => {
  myComponent.setState({ first: 'done!' });
}, 1000);

// Change another part of the state after 2 seconds...
setTimeout(() => {
  myComponent.setState({ second: 'done!' });
}, 2000);

// Change another part of the state after 3 seconds...
setTimeout(() => {
  myComponent.setState({ third: 'done!' });
}, 3000);
```

The takeaway from this example is that you can set individual state properties on components. It will efficiently re-render itself. Here's what the rendered output looks like for the initial component state:

- **first:** loading...
- **second:** loading...
- **third:** loading...

Here's what the output looks like after two of the `setTimeout()` callbacks have run:

- **first:** done!
- **second:** done!
- **third:** loading...

Passing property values

Properties are like state—they are data that get passed into components. However, properties are different from state in that they're only set once, when the component is rendered. In this section, we'll look at *default property values*. Then, we'll look at *setting property values*. After this section, you should be able to grasp the differences between component state and properties.

Default property values

Default property values work a little differently than default state values. They're set as a class property called `defaultProps`. Let's take a look at a component that declares default property values:

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

export default class MyButton extends Component {
  // The "defaultProps" values are used when the
  // same property isn't passed to the JSX element.
  static defaultProps = {
    disabled: false,
    text: 'My Button',
  }

  render() {
    // Get the property values we want to render.
    // In this case, it's the "defaultProps", since
    // nothing is passed in the JSX.
    const { disabled, text } = this.props;

    return (
      <button disabled={disabled}>{text}</button>
    );
  }
}
```

So, why not just set the default property values as an instance property, like we do with default state? The reason is that *properties are immutable*, and there's no need for them to be kept as an instance property value. State, on the other hand, changes all the time, so the component needs an instance level reference to it.

You can see that this component sets default property values for `disabled` and `text`. These values are only used if they're not passed in through the JSX markup used to render the component. Let's go ahead and render this component without any JSX properties, to make sure that the `defaultProps` values are used.

```
import React from 'react';
import { render } from 'react-dom';

import MyButton from './MyButton';

// Renders the "MyButton" component, without
// passing any property values.
render(
  (<MyButton />),
  document.getElementById('app')
);
```

The same principle of always having a default state applies with properties. You want to be able to render components without having to know in advance what the dynamic values of the component are. Now, we'll turn our attention to setting property values on React components.

Setting property values

First, let's create a couple components that expect different types of property values.



In Chapter 7, *Validating Component Properties*, we'll go into more detail on validating the property values that are passed to components.

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

export default class MyButton extends Component {

  // Renders a "<button>" element using values
  // from "this.props".
  render() {
    const { disabled, text } = this.props;
```

```
        return (
          <button disabled={disabled}>{text}</button>
        );
      }
}
```

This simple button component expects a Boolean `disabled` property and a string `text` property. Let's create one more component that expects an array property value:

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

export default class MyList extends Component {
  render() {

    // The "items" property is an array.
    const { items } = this.props;

    // Maps each item in the array to a list item.
    return (
      <ul>
        {items.map(i => (
          <li key={i}>{i}</li>
        ))}
      </ul>
    );
  }
}
```

As you can see, we can pass just about anything we want as a property value via JSX, just as long as it's a valid JavaScript expression. Now let's write some code to set these property values:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';

// The two components we're passing props to
// when they're rendered.
import MyButton from './MyButton';
import MyList from './MyList';

// This is the "application state". This data changes
// over time, and we can pass the application data to
// components as properties.
const appState = {
  text: 'My Button',
  disabled: true,
  items: [
    'First',
```

```
'Second',
'Third',
],
};

// Defines our own "render()" function. The "renderJSX()" function is from "react-dom" and does the actual rendering. The reason we're creating our own "render()" function is that it contains the JSX that we want to render, and so we can call it whenever there's new application data.
function render(props) {
  renderJSX((
    <main>
      { /* The "MyButton" component relies on the "text" and the "disabled" property. The "text" property is a string while the "disabled" property is a boolean. */ }
      <MyButton
        text={props.text}
        disabled={props.disabled}
      />

      { /* The "MyList" component relies on the "items" property, which is an array. Any valid JavaScript data can be passed as a property. */ }
      <MyList items={props.items} />
    </main>
  ),
  document.getElementById('app')
);
}

// Performs the initial rendering...
render(appState);

// After 1 second, changes some application data, then calls "render()" to re-render the entire structure.
setTimeout(() => {
  appState.disabled = false;
  appState.items.push('Fourth');
  render(appState);
}, 1000);
```

The `render()` function looks like it's creating new React component instances every time it's called. Well, React is smart enough to figure out that these components already exist, and that it only needs to figure out what the difference in output will be with the new property values. React is very powerful—have I alluded to this yet?

Another takeaway from this example is that we have an `appState` object that holds onto the state of the application. Pieces of this state are then passed into components as properties, when the components are rendered. State has to live somewhere, and in this case, we've moved it outside of the component. We'll build on this topic in the next section, when we implement stateless functional components.

Stateless components

The components you've seen so far in this book have been classes that extend the base `Component` class. It's time to learn about **functional components** in React. In this section, you'll learn what a pure functional component is by implementing one. Then, we'll cover setting default property values for stateless functional components.

Pure functional components

A functional React component is just what it sounds like—a function. Picture the `render()` method of any React component that you've seen. This method, in essence, is the component. The job of a functional React component is to return JSX, just like a class-based React component. The difference is that this is all a functional component can do. It has no state and no lifecycle methods.

Why would we want to use functional components? It's a matter of simplicity more than anything else. If your component depends on some properties to render some JSX and does nothing else, then why bother with a class when a function is simpler?

A **pure function** is a function without side effects. That is to say, called with a given set of arguments, the function always produces the same output. This is relevant for React components because, given a set of properties, it's easier to predict what the rendered content will be.

Let's look at a functional component now:

```
import React from 'react';

// Exports an arrow function that returns a
// "<button>" element. This function is pure
// because it has no state, and will always
// produce the same output, given the same
// input.
export default ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
);
```

Concise, isn't it? This function returns a `<button>` element, using the properties passed in as arguments (instead of accessing them through `this.props`). This function is pure because the same content is rendered if the same `disabled` and `text` property values are passed. Now let's see how to render this component:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';

// "MyButton" is a function, instead of a
// "Component" subclass.
import MyButton from './MyButton';

// Renders two "MyButton" components. We only need
// the "first" and "second" properties from the
// props argument by destructuring it.
function render({ first, second }) {
  renderJSX((
    <main>
      <MyButton
        text={first.text}
        disabled={first.disabled}
      />
      <MyButton
        text={second.text}
        disabled={second.disabled}
      />
    </main>
  ),
  document.getElementById('app')
);
}

// Reders the components, passing in property data.
render({
  first: {
```

```
        text: 'First Button',
        disabled: false,
    },
second: {
    text: 'Second Button',
    disabled: true,
},
});
```

As you can see, there's zero difference between class-based and function-based React components, from a JSX point of view. The JSX looks exactly the same whether the component was declared using class or function syntax.



The convention is to use arrow function syntax to declare functional React components. However, it's perfectly valid to declare them using traditional JavaScript function syntax, if that's better suited to your style.

Here's what the rendered HTML looks like:



Defaults in functional components

Functional components are lightweight; they don't have any state or lifecycle. They do, however, support some **metadata** options. For example, we can specify the default property values of functional components the same way we would with a class-based component. Here's an example of what this looks like:

```
import React from 'react';

// The functional component doesn't care if the property
// values are the defaults, or if they're passed in from
// JSX. The result is the same.
const MyButton = ({ disabled, text }) => (
    <button disabled={disabled}>{text}</button>
);

// The "MyButton" constant was created so that we could
// attach the "defaultProps" metadata here, before
// exporting it.
MyButton.defaultProps = {
    text: 'My Button',
```

```
        disabled: false,  
    };  
  
export default MyButton;
```

The `defaultProps` property is on a function instead of a class. When React encounters a functional component with this property, it knows to pass in the defaults if they're not provided via JSX.

Container components

In this final section of the chapter, we're going to cover the concept of **container components**. This is a common React pattern, and it brings together many of the concepts that you've learned about state and properties.

The basic premise of container components is simple: don't couple data fetching with the component that renders the data. The container is responsible for fetching the data and passing it to its child component. It contains the component responsible for rendering the data.

The idea is that you should be able to achieve some level of **substitutability** with this pattern. For example, a container could substitute its child component. Or, a child component could be used in a different container. Let's see the container pattern in action, starting with the container itself:

```
import React, { Component } from 'react';  
  
import MyList from './MyList';  
  
// Utility function that's intended to mock  
// a service that this component uses to  
// fetch it's data. It returns a promise, just  
// like a real async API call would. In this case,  
// the data is resolved after a 2 second delay.  
function fetchData() {  
    return new Promise((resolve) => {  
        setTimeout(() => {  
            resolve([  
                'First',  
                'Second',  
                'Third',  
            ]);  
        }, 2000);  
    });  
}
```

```
}

// Container components usually have state, so they
// can't be declared as functions.
export default class MyContainer extends Component {

    // The container should always have an initial state,
    // since this will be passed down to child components
    // as properties.
    state = { items: [] }

    // After the component has been rendered, make the
    // call to fetch the component data, and change the
    // state when the data arrives.
    componentDidMount() {
        fetchData()
            .then(items => this.setState({ items }));
    }

    // Renders the containee, passing the container
    // state as properties, using the spread operator: "...".
    render() {
        return (
            <MyList {...this.state} />
        );
    }
}
```

The job of this component is to fetch data and to set its state. Any time the state is set, `render()` is called. This is where the *child component* comes in. The state of the container is passed to the child as properties. Let's take a look at the `MyList` component next:

```
import React from 'react';

// A stateless component that expects
// an "items" property so that it can render
// a "<ul>" element.
export default ({ items }) => (
    <ul>
        {items.map(i => (
            <li key={i}>{i}</li>
        ))}
    </ul>
);
```

Nothing much to it; a simple functional component that expects an `items` property. Let's see how the container component is actually used:

```
import React from 'react';
import { render } from 'react-dom';

import MyContainer from './MyContainer';

// All we have to do is render the "MyContainer"
// component, since it looks after providing props
// for its children.
render(
  (<MyContainer />),
  document.getElementById('app')
);
```

We'll go into more depth on container component design in Chapter 5, *Crafting Reusable Components*. The idea of this example was to give you a feel for the interplay between state and properties in React components.

When you load the page, you'll see the following content rendered after the 3 seconds it takes to simulate an HTTP request:

- First
- Second
- Third

Summary

In this chapter, you learned about state and properties in React components. We started off by defining and comparing the two concepts. Then, you implemented several React components and manipulated their state. Next, you learned about properties by implementing code that passed property values from JSX to the component. Finally, you were introduced to the concept of a container component, used to decouple data fetching from rendering content.

In the next chapter, you'll learn about handling user events in React components.

4

Event Handling – The React Way

The focus of this chapter is event handling. You've probably already seen more than one approach to handling events in JavaScript applications. React has yet another approach: declaring event handlers in JSX. We'll get things going by looking at how event handlers for particular elements are declared in JSX. Then, you'll learn about binding handler context and parameter values. Next, we'll implement inline event handler functions in our JSX markup.

We'll then discuss how React actually maps event handlers to DOM elements under the hood. Finally, you'll learn about the synthetic events that React passes to event handler functions, and how they're pooled for performance purposes.

Declaring event handlers

The differentiating factor with event handling in React components is that it's **declarative**. Contrast this with something like jQuery, where you have to write imperative code that selects the relevant DOM elements and attaches event handler functions to them.

The advantage with the declarative approach of event handlers in JSX markup is that they're part of the UI structure. Not having to track down code that assigns event handlers is mentally liberating.

In this section, we'll write a basic event handler, so you can get a feel for the declarative event handling syntax found in React applications. Then, we'll look at using generic event handler functions.

Declaring handler functions

Let's take a look at a basic component that declares an event handler for the click event of an element:

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

export default class MyButton extends Component {

  // The click event handler, there's nothing much
  // happening here other than a log of the event.
  onClick() {
    console.log('clicked');
  }

  // Renders a "<button>" element with the "onClick"
  // event handler set to the "onClick()" method of
  // this component.
  render() {
    return (
      <button onClick={this.onClick}>
        {this.props.children}
      </button>
    );
  }
}
```

As you can see, the event handler function, `this.onClick()`, is passed to the `onClick` property of the `<button>` element. By looking at this markup, it's clear what code is going to run when the button is clicked.

See the official React documentation for the full list of supported event property names:
<https://facebook.github.io/react/docs/>.

Multiple event handlers

What I really like about the declarative event handler syntax in JSX is that it's easy to read when there's more than one handler assigned to an element. Sometimes, for example, there are two or three handlers for an element. Imperative code is difficult to work with for a single event handler, let alone several of them. When an element needs more handlers, it's just another JSX attribute. This scales well from a code maintainability perspective:

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

export default class MyInput extends Component {
```

```
// Triggered when the value of the text input changes...
onChange() {
  console.log('changed');
}

// Triggered when the text input loses focus...
onBlur() {
  console.log('blured');
}

// JSX elements can have as many event handler
// properties as necessary.
render() {
  return (
    <input
      onChange={this.onChange}
      onBlur={this.onBlur}
    />
  );
}
}
```

This `<input>` element could have several more event handlers, and the code would be just as readable.

As you keep adding more event handlers to your components, you'll notice that a lot of them do the same thing. It's time to start thinking about how to share generic handler functions across components.

Importing generic handlers

Any React application is likely going to have the same event handling functionality for different components. For example, in response to a button click, the component should sort a list of items. It's these types of super generic behaviors that belong in their own modules so that several components can share them. Let's implement a component that uses a generic event handler function:

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

// Import the generic event handler that
// manipulates the state of a component.
import reverse from './reverse';

export default class MyList extends Component {
  state = {
```

```
    items: ['Angular', 'Ember', 'React'],
}

// Makes the generic function specific
// to this component by calling "bind(this)".
onReverseClick = reverse.bind(this)

render() {
  const {
    state: {
      items,
    },
    onReverseClick,
  } = this;

  return (
    <section>
      { /* Now we can attach the "onReverseClick" handler
          to the button, and the generic function will
          work with this component's state. */}
      <button onClick={onReverseClick}>Reverse</button>
      <ul>
        {items.map((v, i) => (
          <li key={i}>{v}</li>
        )))
      </ul>
    </section>
  );
}
}
```

Let's walk through what's going on here, starting with the imports. We're importing a function called `reverse()`. This is the generic event handler function that we're using with our `<button>` element. When it's clicked, we want the list to reverse its order.

You can see that we're creating an `onReverseClick` property in this class. This is created using `bind()` to bind the context of the generic function to this component instance.

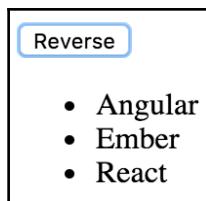
Finally, looking at the JSX markup, you can see that the `onReverseClick()` function is used as the handler for the button click.

So how does this work, exactly? We have a generic function that somehow changes the state of this component because we bound a context to it? Well, pretty much, yes, that's it. Let's look at the generic function implementation now:

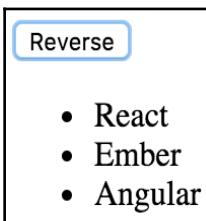
```
// Exports a generic function that changes the
// state of a component, causing it to re-render
// itself.
export default function reverse() {
  this.setState(this.state.items.reverse());
}
```

Pretty simple! Obviously, this function depends on a `this.state` property and an `items` array within the state. However, this is simple to do. The key is that the state that this function works with are generic; an application could have many components with an `items` array in its state.

Here's what our rendered list looks like:



As expected, clicking the button causes the list to sort, using our generic `reverse()` event handler:



Now we'll take a deeper look at binding the context of event handler functions, as well as binding their parameters.

Event handler context and parameters

In this section, we'll examine React components that automatically bind their event handler contexts and how you can pass data into event handlers. Having the right context is important for React event handler functions, because they usually need access to properties or state of the component. Being able to parameterize event handlers is also important, because they don't pull data out of DOM elements.

Auto-binding context

The components you've implemented so far in this book have used the **ES2015** class style declaration. This is where you declare a class that extends the base React Component class. When you do this, however, any event handler methods in the component will need to be manually bound to the component context. For example, if you need access to `this.props`, `this` needs to be a reference to the component.

You can use the `React.createClass()` function to declare a component and have its method contexts **auto bind** to the component. In other words, there's no need to call `bind()` on your callback functions. Let's see an example of this in action:

```
import React from 'react';

export default React.createClass({


  // This event handler requires access to the
  // component properties, but it doesn't need
  // to explicitly bind it's context, because
  // "createClass()" components do this automatically.
  onClick() {
    console.log('clicked',
      `${this.props.children}`);
  },


  // Renders a button with a bound event handler.
  render() {
    return (
      <button onClick={this.onClick}>
        {this.props.children}
      </button>
    );
  },
});
```

This looks a lot like a class declaration. In fact, this used to be the only supported approach for declaring React components. This component in particular handles a button click event. The `onClick()` method needs access to the component because it references `this.props.children`.

So, you might want to use this function to declare your components if you have a lot of code that manually binds the context of your event handler functions.



You can leverage ES2015 syntax to have event handler methods auto bind their context in component class declarations. We'll introduce this approach a little later on in the book.

Getting component data

In the preceding example, the event handler needed access to the component so that it could read the `children` property. In this section, we'll look at a more involved scenario where the handler needs access to component properties, as well as argument values.

We'll render a custom list component that has a click event handler for each item in the list. We'll pass the component a collection as follows:

```
import React from 'react';
import { render } from 'react-dom';

import MyList from './MyList';

// The items to pass to "<MyList>" as a property.
const items = [
  { id: 0, name: 'First' },
  { id: 1, name: 'Second' },
  { id: 2, name: 'Third' },
];

// Renders "<MyList>" with an "items" property.
render(
  (<MyList items={items} />),
  document.getElementById('app')
);
```

As you can see, each item in the list has an `id` property, used to identify the item. We'll need to be able to access this ID when the item is clicked in the UI so that the event handler can work with the item. Here's what the `MyList` component implementation looks like:

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

export default class MyList extends Component {
  constructor() {
    super();

    // We want to make sure that the "onClick()"
    // handler is explicitly bound to this component
    // as it's context.
    this.onClick = this.onClick.bind(this);
  }

  // When a list item is clicked, look up the name
  // of the item based on the "id" argument. This is
  // why we need access to the component through "this",
  // for the properties.
  onClick(id) {
    const { name } = this.props.items.find(
      i => i.id === id
    );

    console.log('clicked', `#${name}`);
  }

  render() {
    return (
      <ul>
        {/* Creates a new handler function with
          the bound "id" argument. Notice that
          the context is left as null, since that
          has already been bound in the
          constructor. */}
        {this.props.items.map(({ id, name }) => (
          <li
            key={id}
            onClick={this.onClick.bind(null, id)}
          >
            {name}
          </li>
        ))}
      </ul>
    );
  }
}
```

Here is what the rendered list looks like:

- First
- Second
- Third

We have to take care of binding the event handler context, which is done in the constructor. If you look at the `onClick()` event handler, you can see that it needs access to the component so that it can look up the clicked item in `this.props.items`. Also, the `onClick()` handler is expecting an `id` parameter. If you take a look at the JSX content of this component, you can see that we're calling `bind()` to supply the argument value for each item in the list. This means that when the handler is called in response to a click event, the `id` of the item is already provided.

This approach to parameterized event handling is quite different from prior approaches. For example, I used to rely on getting parameter data from the DOM element itself. This works well in that we only need one event handler, and it can extract the data it needs from the event argument. This approach also doesn't require setting up several new functions by iterating over a collection and calling `bind()`.

And therein lies the trade-off. React applications avoid touching the DOM, because the DOM is really just a render target for React components. If we can write code that doesn't introduce explicit dependencies to DOM elements, the code will be very portable. This is what we've done with the event handler in this example.



If you're concerned about the performance implications of creating a new function for every item in a collection, don't be. You're not going to render thousands of items on the page at a time. Benchmark your code, and if it turns out that `bind()` calls on your React event handlers are the slowest part, then you probably have a really fast application.

Inline event handlers

The typical approach to assigning handler functions to JSX properties is to use a **named** function. However, sometimes we might want to use an **inline** function. This is done by assigning an **arrow** function directly to the event property in the JSX markup:

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

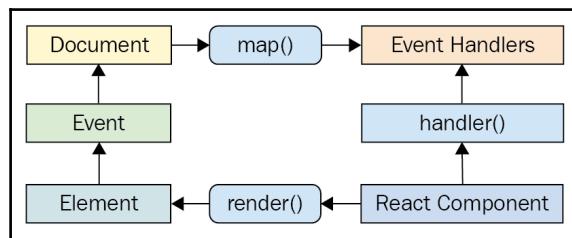
export default class MyButton extends Component {
```

```
// Renders a button element with an "onClick()" handler.  
// This function is declared inline with the JSX, and is  
// useful in scenarios where you need to call another  
// function.  
render() {  
  return (  
    <button  
      onClick={e => console.log('clicked', e)}  
    >  
      {this.props.children}  
    </button>  
  );  
}  
}
```

The main use of inlining event handlers like this is when you have a static parameter value that you want to pass to another function. In this example, we're calling `console.log()` with the `clicked` string. We could have set up a special function for this purpose outside of the JSX markup by creating a new function using `bind()`. But then we would have to think of yet another name for yet another function. Inlining is just easier.

Binding handlers to elements

When you assign an event handler function to an element in JSX, React doesn't actually attach an event listener to the underlying DOM element. Instead, it adds the function to an internal mapping of functions. There's a single event listener on the document for the page. As events bubble up through the DOM tree to the document, the React handler checks to see if any components have matching handlers. The process is illustrated here:



Why does React go to all of this trouble, you might ask? It's the same principle that we've been covering for the past few chapters; keep the declarative UI structures separated from the DOM as much as possible.

For example, when a new component is rendered, its event handler functions are simply added to the internal mapping maintained by React. When an event is triggered and it hits the document object, React maps the event to the handlers. If a match is found, it calls the handler. Finally, when the React component is removed, the handler is simply removed from the list of handlers.

None of these DOM operations actually touch the DOM. It's all abstracted by a single event listener. This is good for performance and the overall architecture (keep the render target separate from application code).

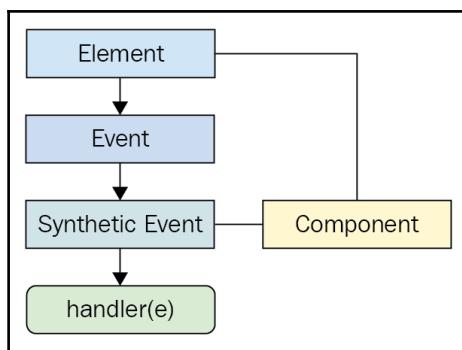
Synthetic event objects

When you attach an event handler function to a DOM element using the native `addEventListener()` function, the callback will get an event argument passed to it. Event handler functions in React are also passed an event argument, but it's not the standard Event instance. It's called `SyntheticEvent`, and it's a simple wrapper for native event instances.

Synthetic events serve two purposes in React:

- Provides a consistent event interface, normalizing browser inconsistencies with regard to event properties
- Synthetic events contain information that's necessary for propagation to work

Here's an illustration of the synthetic event in the context of a React component:



In the next section, you'll see how these synthetic events are pooled for performance reasons and the implications of this for asynchronous code.

Event pooling

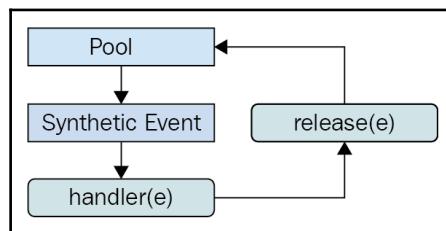
One challenge with wrapping native event instances is that this can cause performance issues. Every synthetic event wrapper that's created will also need to be garbage collected at some point, which can be expensive in terms of CPU time.



When the garbage collector is running, none of your JavaScript code is able to run. This is why it's important to be memory efficient; frequent garbage collection means less CPU time for code that responds to user interactions.

For example, if your application only handles a few events, this wouldn't matter much. But even by modest standards, applications respond to many events, even if the handlers don't actually do anything with them. This is problematic if React constantly has to allocate new synthetic event instances.

React deals with this problem by allocating a **synthetic instance pool**. Whenever an event is triggered, it takes an instance from the pool and populates its properties. When the event handler has finished running, the synthetic event instance is released back into the pool, as shown here:



This prevents the garbage collector from running frequently when a lot of events are triggered. The pool keeps a reference to the synthetic event instances, so they're never eligible for garbage collection. React never has to allocate new instances either.

However, there is one gotcha that you need to be on the lookout for. It involves accessing the synthetic event instances from asynchronous code in your event handlers. This is an issue because as soon as the handler has finished running, the instance goes back into the pool. When it goes back into the pool, all of its properties are cleared. Here's an example that shows how this can go wrong:

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

// Mock function, meant to simulate fetching
// data asynchronously from an API.
```

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve();
    }, 1000);
  });
}

export default class MyButton extends Component {
  onClick(e) {
    // This works fine, we can access the DOM element
    // through the "currentTarget" property.
    console.log('clicked', e.currentTarget.style);

    fetchData().then(() => {
      // However, trying to access "currentTarget"
      // asynchronously fails, because it's properties
      // have all been nullified so that the instance
      // can be reused.
      console.log('callback', e.currentTarget.style);
    });
  }

  render() {
    return (
      <button onClick={this.onClick}>
        {this.props.children}
      </button>
    );
  }
}
```

As you can see, the second `console.log()` is attempting to access a synthetic event property from an asynchronous callback that doesn't run until the event handler completes, which causes the event to empty its properties. This results in a warning and an undefined value.



The aim of this example is to illustrate how things can break when you write asynchronous code that interacts with events. Just don't do it!

Summary

This chapter introduced you to event handling in React. The key differentiator between React and other approaches to event handling is that handlers are declared in JSX markup. This makes tracking down which elements handle which events much simpler.

You learned that having multiple event handlers on a single element is a matter of adding new JSX properties. Next, you learned that it's a good idea to share event handling functions that handle generic behavior. Context can be important for event handler functions, if they need access to component properties or state. You learned about the various ways to bind event handler function context, and parameter values.

Then, you learned about inline event handler functions and their potential use, as well as how React actually binds a single DOM event handler to the document object. Synthetic events are an abstraction that wraps the native event, and you leaned why they're necessary and how they're pooled for efficient memory consumption.

In the next chapter, we're going to explore how to create components that are reusable for a variety purposes.

5

Crafting Reusable Components

The focus of this chapter is to show you how to implement React components that serve more than just one purpose. After reading this chapter, you'll feel confident about how to compose application features.

We'll start the chapter off with a brief look at HTML elements and how they work in terms of helping to implement features versus having a high level of utility. Then, we'll look at the implementation of a monolithic component and discuss the issues that it will cause down the road. The next section is devoted to re-implementing the monolithic component in such a way that the feature is composed of smaller components.

Finally, the chapter ends with a discussion of rendering trees of React components, and gives you some tips on how to avoid introducing too much complexity as a result of decomposing components. We'll close this final section by reiterating the concept of high-level feature components versus utility components.

Reusable HTML elements

Let's think about HTML elements for a moment, before jumping into React component implementation. Depending on the type of HTML element, it's either *feature centric* or *utility centric*. The utility-centric HTML elements are more reusable than feature centric HTML elements. For example, consider the `<section>` element. Yes, this is a generic element that can be used just about anywhere, but its primary purpose is to compose the structural aspects of a feature—the outer shell of the feature and the inner sections of the feature. This is where the `<section>` element is most useful.

On the other side of the fence we have things like `<p>` and `` and `<button>` elements. These elements provide a high level of utility because they're generic by design. We're supposed to use `<button>` elements whenever we have something that's clickable by the user, resulting in an action. This is a level lower than the concept of a feature.

While it's easy to talk about HTML elements that have a high level of utility versus those that are geared toward specific features, the discussion is more detailed when *data* is involved. HTML is static markup—React components combine static markup with data. The question is, how do we do this and make sure that we're creating the right feature centric and utility centric components?

The aim of the remainder of this chapter is to find out how to go from a monolithic React component that defines a feature, to a smaller feature-centric component combined with utility components.

The difficulty with monolithic components

If you could implement just one component for any given feature, things would be quite simple, wouldn't they? At the very least, there wouldn't be many components to maintain, and there wouldn't be many communication paths for data to flow through, because everything would be internal to the component.

However, this idea doesn't work for a number of reasons. Having monolithic feature components makes it difficult to coordinate any kind of team development effort. Something I've noticed with monolithic components is that the bigger they become, the more difficult they are to refactor into something better later on.

There's also the problem of feature overlap and feature communication. Overlap happens because of similarities between features—it's unlikely that an application will have a set of features that are completely unique to one another. That would make the application very difficult to learn and use. Component communication essentially means that the state of something in one feature will impact the state of something in another feature. State is difficult to deal with, and even more so when there is a lot of state packaged up into a monolithic component.

The best way to learn how to avoid monolithic components is to experience one first hand. We'll spend the remainder of this section implementing a monolithic component. In the following section, you'll see how this component can be refactored into something a little more sustainable.

The JSX markup

The monolithic component we're going to implement is a feature that lists articles. It's just for illustrative purposes, so we don't want to go overboard on the size of the component. It'll be simple, yet monolithic. The user can add new items to the list, toggle the summary of items in the list, and remove items from the list. Here is the render method of the component:

```
render() {
  const {
    articles,
    title,
    summary,
  } = this.data.toJS();

  return (
    <section>
      <header>
        <h1>Articles</h1>
        <input
          placeholder="Title"
          value={title}
          onChange={this.onChangeTitle}
        />
        <input
          placeholder="Summary"
          value={summary}
          onChange={this.onChangeSummary}
        />
        <button onClick={this.onClickAdd}>Add</button>
      </header>
      <article>
        <ul>
          {articles.map(i => (
            <li key={i.id}>
              <a
                href="#"
                onClick={this.onClickToggle.bind(null, i.id)}
              >
                {i.title}
              </a>
              &nbsp;
              <a
                href="#"
                onClick={this.onClickRemove.bind(null, i.id)}
              >
                &#10007;
              </a>
            </li>
          ))}
        </ul>
      </article>
    </section>
  );
}
```

```
        </a>
      <p style={{ display: i.display }}>
        {i.summary}
      </p>
    </li>
  )));
</ul>
</article>
</section>
);
}
```

So, not a ton of JSX, but definitely more than necessary in one place. We'll improve on this in the following section, but for now, let's implement the initial state for this component.



I strongly encourage you to download the companion code for this book from <https://github.com/PacktPublishing/React-and-React-Native>. I can break apart the component code so that I can explain it on these pages. However, it's an easier learning experience if you can see the code modules in their entirety, in addition to running them.

Initial state and state helpers

Now let's look at the initial state of this component:

```
// The state of this component consists of
// three properties: a collection of articles,
// a title, and a summary. The "fromJS()" call
// is used to build an "Immutable.js" Map. Also
// note that this isn't set directly as the component
// state - it's in a "data" property of the state -
// otherwise, state updates won't work as expected.
state = {
  data: fromJS({
    articles: [
      {
        id: cuid(),
        title: 'Article 1',
        summary: 'Article 1 Summary',
        display: 'none',
      },
      {
        id: cuid(),
        title: 'Article 2',
        summary: 'Article 2 Summary',
        display: 'none',
      }
    ]
  })
}
```

```
        summary: 'Article 2 Summary',
        display: 'none',
    },
{
    id: cuid(),
    title: 'Article 3',
    summary: 'Article 3 Summary',
    display: 'none',
},
{
    id: cuid(),
    title: 'Article 4',
    summary: 'Article 4 Summary',
    display: 'none',
},
],
title: '',
summary: '',
}),
}
```

There's nothing extraordinary about the state itself; it's just a collection of objects. There are two interesting functions used to initialize the state. The first is `cuid()` from the `cuid` package—a useful tool for generating unique IDs. The second is `fromJS()` from the `immutable` package. Here are the imports that pull in these two dependencies:

```
// Utility for constructing unique IDs...
import cuid from 'cuid';

// For building immutable component states...
import { fromJS } from 'immutable';
```

As the name suggests, the `fromJS()` function is used to construct an immutable data structure. `Immutable.js` has very useful functionality for manipulating the state of React components. We'll be using `Immutable.js` throughout the remainder of the book, and you'll learn more of the specifics as you go, starting with this example.

You may remember from the previous chapter that the `setState()` method only works with plain objects. Well, `Immutable.js` objects aren't plain objects. If we want to use immutable data, we need to wrap them somehow in a plain object. Let's implement a helper getter and setter for this:

```
// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}
```

```
// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}
```

Now, we can use our immutable component state inside of our event handlers.

Event handler implementation

At this point, we have the initial state, state helper properties, and the JSX of the component. Now it's time to implement the event handlers themselves:

```
// When the title of a new article changes, update the state
// of the component with the new title value, by using "set()"
// to create a new map.
onChangeTitle = (e) => {
  this.data = this.data.set(
    'title',
    e.target.value,
  );
}

// When the summary of a new article changes, update the state
// of the component with the new summary value, by using "set()"
// to create a new map.
onChangeSummary = (e) => {
  this.data = this.data.set(
    'summary',
    e.target.value
  );
}

// Creates a new article and empties the title
// and summary inputs. The "push()" method creates a new
// list and "update()" is used to update the list by
// creating a new map.
onClickAdd = () => {
  this.data = this.data
    .update(
      'articles',
      a => a.push(fromJS({
        id: cuid(),
        title: this.data.get('title'),
        summary: this.data.get('summary'),
        display: 'none',
      }))
}
```

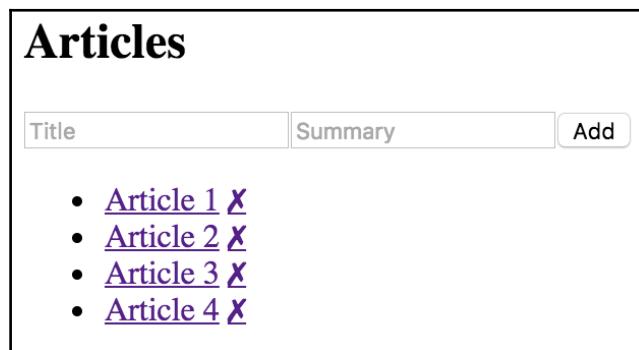
```
)  
.set('title', '')  
.set('summary', ''));  
}  
  
// Removes an article from the list. Calling "delete()"  
// creates a new list, and this is set in the new component  
// state.  
onClickRemove = (id) => {  
  const index = this.data  
  .get('articles')  
  .findIndex(  
    a => a.get('id') === id  
  );  
  
  this.data = this.data  
  .update(  
    'articles',  
    a => a.delete(index)  
  );  
}  
  
// Toggles the visibility of the article summary by  
// setting the "display" state of the article. This  
// state is dependent on the current state.  
onClickToggle = (id) => {  
  const index = this.data  
  .get('articles')  
  .findIndex(  
    a => a.get('id') === id  
  );  
  
  this.data = this.data  
  .update(  
    'articles',  
    articles => articles.update(  
      index,  
      a => a.set(  
        'display',  
        a.get('display') ? '' : 'none'  
      )  
    )  
  );  
}
```

Yikes! That's a lot of `Immutable.js` code! Not to worry, it's actually quite straightforward, especially compared to trying to implement these transformations using plain JavaScript. Here are some pointers to help you understand this code:

- `setState()` is always called with a plain object as its argument. This is why we've introduced the data setter. When you assign a new value to `this.data`, it will call `setState()` with a plain object. You only need to worry about `Immutable.js` data. Likewise, the data getter returns the `Immutable.js` object instead of the whole state.
- Immutable methods always return a new instance. When you see something like `article.set(...)`, it doesn't actually change `article`, it creates a new one.
- In the `render()` method, the immutable data structures are converted back to plain JavaScript arrays and objects for use in the JSX markup.

If necessary, take all the time you need to understand what is happening here. As you progress through the book, you'll see ways that immutable state can be exploited by React components. Something else worth pointing out here is that these event handlers can only change the state of this component. That is, they can't accidentally change the state of other components. As you'll see in the following section, these handlers are actually in pretty good shape as they are.

Here's a screenshot of the rendered output:



Refactoring component structures

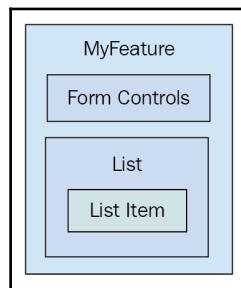
We have a monolithic feature component—now what? Let's make it better.

In this section, you'll learn how to take the feature component that we just implemented in the preceding section and split it into more maintainable components. We'll start with the JSX, as this is probably the best refactor starting point. Then, we'll implement new components for the feature.

Finally, we'll make these new components functional, instead of class-based.

Start with the JSX

The JSX of any monolithic component is the best starting point for figuring out how to refactor it into smaller components. Let's visualize the structure of the component that we're currently refactoring:



The top part of the JSX is form controls, so this could easily become its own component:

```
<header>
  <h1>Articles</h1>
  <input
    placeholder="Title"
    value={title}
    onChange={this.onChangeTitle}
  />
  <input
    placeholder="Summary"
    value={summary}
    onChange={this.onChangeSummary}
  />
  <button onClick={this.onClickAdd}>Add</button>
</header>
```

Next, we have the list of articles:

```
<ul>
  {articles.map(i => (
    <li key={i.id}>
      <a
        href="#"
        onClick={
          this.onClickToggle.bind(null, i.id)
        }
      >
        {i.title}
      </a>
      &nbsp;
      <a
        href="#"
        onClick={this.onClickRemove.bind(null, i.id)}
      >
        &#10007;
      </a>
      <p style={{ display: i.display }}>
        {i.summary}
      </p>
    </li>
  ))}
</ul>
```

Within this list, we have the potential for an article item, which would be everything in the `` tag.

As you can see, the JSX alone paints a picture of how the UI structure can be decomposed into smaller React components. This refactoring exercise would be difficult without declarative JSX markup.

Implementing an article list component

Here's what the article list component implementation looks like:

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

export default class ArticleList extends Component {
  render() {
    // The properties include things that are passed in
    // from the feature component. This includes the list
    // of articles to render, and the two event handlers
    // that change state of the feature component.
  }
}
```

```
const {
  articles,
  onClickToggle,
  onClickRemove,
} = this.props;

return (
  <ul>
    {articles.map(i => (
      <li key={i.id}>
        { /* The "onClickToggle()" callback changes
           the state of the "MyFeature" component. */ }
        <a
          href="#"
          onClick={onClickToggle.bind(null, i.id)}>
          {i.title}
        </a>
        &nbsp;
        { /* The "onClickRemove()" callback changes
           the state of the "MyFeature" component. */ }
        <a
          href="#"
          onClick={onClickRemove.bind(null, i.id)}>
          &#10007;
        </a>
        <p style={{ display: i.display }}>
          {i.summary}
        </p>
      </li>
    ))}
  </ul>
);
}
```

As you can see, we're just taking the relevant JSX out of the monolithic component and putting it here. Now let's see what the feature component JSX looks like:

```
render() {
  const {
    articles,
    title,
    summary,
  } = this.data.toJS();
```

```
return (
  <section>
    <header>
      <h1>Articles</h1>
      <input
        placeholder="Title"
        value={title}
        onChange={this.onChangeTitle}
      />
      <input
        placeholder="Summary"
        value={summary}
        onChange={this.onChangeSummary}
      />
      <button onClick={this.onClickAdd}>Add</button>
    </header>

    { /* Now the list of articles is rendered by the
       "ArticleList" component. This component can
       now be used in several other components. */ }
    <ArticleList
      articles={articles}
      onClickToggle={this.onClickToggle}
      onClickRemove={this.onClickRemove}
    />
  </section>
);
}
```

The list of articles is now rendered by the `<ArticleList>` component. The list of articles to render is passed to this component as a property as well as two of the event handlers.



Wait, why are we passing event handlers to a child component? The reason is simple; it is so that the `ArticleList` component doesn't have to worry about state or how the state changes. All it cares about is rendering content, and making sure the appropriate event callbacks are hooked up to the appropriate DOM elements. This is a *container component* concept that I'll expand upon later in this chapter.

Implementing an article item component

After implementing the article list component, you might decide that it's a good idea to break this component down further, because the item might be rendered in another list on another page. Perhaps, the most important aspect of implementing the article list item as its own component is that we don't know how the markup will change in the future.

Another way to look at it is this—if it turns out that we don't actually need the item as its own component, this new component doesn't introduce much indirection or complexity. Without further ado, here's the article item component:

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

export default class ArticleItem extends Component {
  render() {
    // The "article" is mapped from the "ArticleList"
    // component. The "onClickToggle()" and
    // "onClickRemove()" event handlers are passed
    // all the way down from the "MyFeature" component.
    const {
      article,
      onClickToggle,
      onClickRemove,
    } = this.props;

    return (
      <li>
        { /* The "onClickToggle()" callback changes
            the state of the "MyFeature" component. */ }
        <a
          href="#"
          onClick={onClickToggle.bind(null, article.id)}>
          {article.title}
        </a>
        &nbsp;

        { /* The "onClickRemove()" callback changes
            the state of the "MyFeature" component. */ }
        <a
          href="#"
          onClick={onClickRemove.bind(null, article.id)}>
          &#10007;
        </a>
        <p style={{ display: article.display }}>
          {article.summary}
        </p>
      </li>
    );
  }
}
```

Here's the new ArticleItem component being rendered by the ArticleList component:

```
import React, { Component } from 'react';
import ArticleItem from './ArticleItem';

export default class ArticleList extends Component {
  render() {
    // The properties include things that are passed in
    // from the feature component. This includes the list
    // of articles to render, and the two event handlers
    // that change state of the feature component. These,
    // in turn, are passed to the "ArticleItem" component.
    const {
      articles,
      onClickToggle,
      onClickRemove,
    } = this.props;

    // Now this component maps to an "<ArticleItem>" collection.
    return (
      <ul>
        {articles.map(i => (
          <ArticleItem
            key={i.id}
            article={i}
            onClickToggle={onClickToggle}
            onClickRemove={onClickRemove}
          />
        ))}
      </ul>
    );
  }
}
```

Do you see how this list just maps the list of articles? What if we wanted to implement another article list that does some filtering too? It's beneficial to have a reusable ArticleItem component.

Implementing an add article component

Now that we're done with the article list, it's time to think about the form controls used to add a new article. Let's implement a component for this aspect of the feature:

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

```
export default class AddArticle extends Component{
  render() {
    const {
      name,
      title,
      summary,
      onChangeTitle,
      onChangeSummary,
      onClickAdd
    } = this.props;

    return (
      <section>
        <h1>{name}</h1>
        <input
          placeholder="Title"
          value={title}
          onChange={onChangeTitle}
        />
        <input
          placeholder="Summary"
          value={summary}
          onChange={onChangeSummary}
        />
        <button onClick={onClickAdd}>Add</button>
      </section>
    );
  }
}
```

Now, we have the final version of the feature component JSX:

```
render() {
  const {
    articles,
    title,
    summary,
  } = this.state.data.toJS();

  return (
    <section>
      { /* Now the add article form is rendered by the
         "AddArticle" component. This component can
         now be used in several other components. */ }
      <AddArticle
        name="Articles"
        title={title}
        summary={summary}
```

```
        onChangeTitle={this.onChangeTitle}
        onChangeSummary={this.onChangeSummary}
        onClickAdd={this.onClickAdd}
    />

    { /* Now the list of articles is rendered by the
        "ArticleList" component. This component can
        now be used in several other components. */
    <ArticleList
        articles={articles}
        onClickToggle={this.onClickToggle}
        onClickRemove={this.onClickRemove}
    />
    </section>
);
}
```

As you can see, the focus of this component is on the feature data while it defers to other components for rendering UI elements. Let's make one final tweak to the new components we've implemented for this feature.

Making components functional

While implementing these new components for the feature, you might have noticed that they don't have any responsibilities other than rendering JSX using property values. These components are good candidates for *pure function components*. Whenever you come across components that only use property values, it's a good idea to make them functional. For one thing, it makes it explicit that the component doesn't rely on any state or lifecycle methods. It's also more efficient, because React doesn't perform as much work when it detects that a component is a function.

Here is the functional version of the article list component:

```
import React from 'react';
import ArticleItem from './ArticleItem';

export default ({
    articles,
    onClickToggle,
    onClickRemove,
}) => (
    <ul>
        {articles.map(i => (
            <ArticleItem
                key={i.id}>
```

```
        article={i}
        onClickToggle={onClickToggle}
        onClickRemove={onClickRemove}
    />
)) }
</ul>
);
```

Here is the functional version of the article item component:

```
import React from 'react';

export default ({
  article,
  onClickToggle,
  onClickRemove,
}) => (
  <li>
    { /* The "onClickToggle()" callback changes
        the state of the "MyFeature" component. */ }
    <a
      href="#"
      onClick={onClickToggle.bind(null, article.id)}>
      {article.title}
    </a>
    &nbsp;

    { /* The "onClickRemove()" callback changes
        the state of the "MyFeature" component. */ }
    <a
      href="#"
      onClick={onClickRemove.bind(null, article.id)}>
      &#10007;
    </a>
    <p style={{ display: article.display }}>
      {article.summary}
    </p>
  </li>
);
```

Here is the functional version of the add article component:

```
import React from 'react';

export default ({
  name,
```

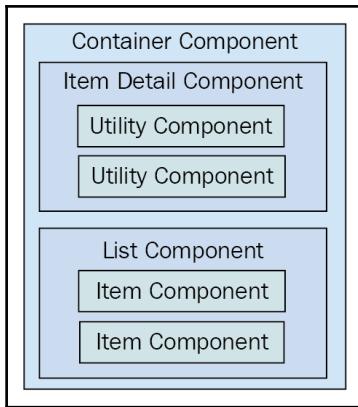
```
title,  
summary,  
onChangeTitle,  
onChangeSummary,  
onClickAdd,  
) => (  
  <section>  
    <h1>{name}</h1>  
    <input  
      placeholder="Title"  
      value={title}  
      onChange={onChangeTitle}  
    />  
    <input  
      placeholder="Summary"  
      value={summary}  
      onChange={onChangeSummary}  
    />  
    <button onClick={onClickAdd}>Add</button>  
  </section>  
) ;
```

Another added benefit of making components functional is that there's less opportunity to introduce unnecessary methods or other data, because it's not a class where it's easier to add more stuff.

Rendering component trees

In the previous section, we refactored a large monolithic component into several smaller and more focused on components. Let's take a moment and reflect on what we've accomplished. The feature component that was once monolithic, ended up focusing almost entirely on the *state data*. It handled the initial state and handled transforming the state, and it would handle network requests that fetch state, if there were any. This is a typical *container component* in a React application, and it's the starting point for data.

The new components that we implemented, to better compose the feature, were the recipients of this data. The difference between these components and their container is that they only care about the properties that are passed into them at the time they're rendered. In other words, they only care about *data snapshots* at a particular point in time. From here, these components might pass the property data into their own child components as properties. The generic pattern to composing React components is as follows:



The container component will typically contain one direct child. In this diagram, you can see that the container has either an item detail component or a list component. Of course, there will be variations on these two categories, as every application is different. This generic pattern has three levels of component composition. Data flows in one direction from the container all the way down to the utility components.

Once you add more than three layers, the application architecture becomes difficult to comprehend. There will be the odd case where you'll need to add four layers of React components, but as a rule-of-thumb, you should avoid this.

Feature components and utility components

As you saw with the monolithic component example, we started off with a single component that was entirely focused on a feature. This means that the component has very little utility elsewhere in the application.

The reason for this is because top-level components deal with application state. **Stateful components** are difficult to use in any other context. As you refactored the monolithic feature component, you created new components that moved further away from the data. The general rule is that the further your components move from stateful data, the more utility they have, because their property values could be passed in from anywhere in the application.

Summary

This chapter was about avoiding monolithic component design. However, monoliths are often a necessary starting point in the design of any React component.

We began by talking about HTML and how the different elements have varying degrees of utility. Next, we discussed the issues with monolithic React components and walked through the implementation of a monolithic component.

Then, you spent several sections learning how to refactor the monolithic component into a more sustainable design. From this exercise, you learned that container components should only have to think in terms of handling state, while smaller components have more utility because their property values can be passed from anywhere.

In the next chapter, you'll learn about the React component lifecycle. This is an especially relevant topic for implementing container components.

6

The React Component Lifecycle

The goal of this chapter is for you to learn about the **lifecycle** of React components and how to write code that responds to lifecycle events. We'll kick things off with a brief discussion on why components need a lifecycle in the first place. Then, you'll implement several example components that initialize their properties and state using these methods.

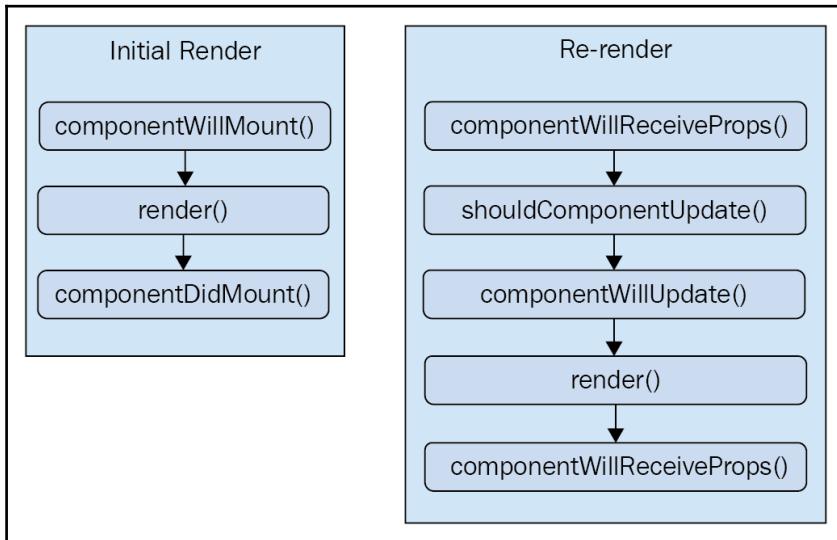
Next, you'll learn about how to optimize the rendering efficiency of your components by avoiding rendering when it isn't necessary. Finally, you'll see how to encapsulate imperative code in React components and how to clean up when components are unmounted.

Why components need a lifecycle

React components go through a lifecycle, whether our code knows about it or not. In fact, the `render()` method that you've implemented in your components so far in this book, is actually a lifecycle method. Rendering is just one lifecycle event in a React component.

For example, there's lifecycle events for when the component is about to be mounted into the DOM, for after the component has been mounted to the DOM, when the component is updated, and so on. Lifecycle events are yet another moving part, so you'll want to keep them to a minimum. As you'll learn in this chapter, some components do need to respond to lifecycle events to perform initialization, render heuristics, or clean up after the component when it's unmounted from the DOM.

The following diagram gives you an idea of how a component flows through its lifecycle, calling the corresponding methods in turn:



These are the two main lifecycle flows of a React component. The first happens when the component is initially rendered. The second happens whenever the component is re-rendered. However, the `componentWillReceiveProps()` method is only called when the component's properties are updated. This means that if the component is re-rendered because of a call to `setState()`, this lifecycle method isn't called, and the flow starts with `shouldComponentUpdate()` instead.

The other lifecycle method that isn't included in this diagram is `componentWillUnmount()`. This is the only lifecycle method that's called when a component is about to be removed. We'll see an example of how to use this method at the end of the chapter. On that note, let's get coding.

Initializing properties and state

In this section, you'll see how to implement initialization code in React components. This involves using lifecycle methods that are called when the component is first created. First, we'll walk through a basic example that sets the component up with data from the API. Then, you'll see how state can be initialized from properties, and also how state can be updated as properties change.

Fetching component data

One of the first things you'll want to do when your components are initialized is populate their state or properties. Otherwise, the component won't have anything to render other than its skeleton markup. For instance, let's say you want to render the following user list component:

```
import React from 'react';
import { Map as ImmutableMap } from 'immutable';

// This component displays the passed-in "error"
// property as bold text. If it's null, then
// nothing is rendered.
const ErrorMessage = ({ error }) =>
  ImmutableMap()
    .set(null, null)
    .get(
      error,
      (<strong>{error}</strong>)
    );

// This component displays the passed-in "loading"
// property as italic text. If it's null, then
// nothing is rendered.
const LoadingMessage = ({ loading }) =>
  ImmutableMap()
    .set(null, null)
    .get(
      loading,
      (<em>{loading}</em>)
    );

export default ({
  error,
  loading,
  users,
}) => (
  <section>
    { /* Displays any error messages... */ }
    <ErrorMessage error={error} />

    { /* Displays any loading messages, while
         waiting for the API... */ }
    <LoadingMessage loading={loading} />

    { /* Renders the user list... */ }
    <ul>
```

```
{users.map(i => (
  <li key={i.id}>{i.name}</li>
))
</ul>
</section>
);
```

There are three pieces of data that this JSX relies on:

- `loading`: This message is displayed while fetching API data
- `error`: This message is displayed if something goes wrong
- `users`: Data fetched from the API

There's also two helper components used here: `ErrorMessage` and `LoadingMessage`. They're used to format the error and the loading state, respectively. However, if `error` or `loading` are null, neither do we want to render anything nor do we want to introduce imperative logic into these simple functional components. This is why we're using a cool little trick with `Immutable.js` maps.

First, we create a map that has a single **key-value pair**. The key is null, and the value is null. Second, we call `get()` with either an `error` or a `loading` property. If the `error` or `loading` property is null, then the key is found and nothing is rendered. The trick is that `get()` accepts a second parameter that's returned if no key is found. This is where we pass in our *truthy* value and avoid imperative logic all together. This specific component is simple, but the technique is especially powerful when there are more than two possibilities.

How should we go about making the API call and using the response to populate the `users` collection? The answer is to use a container component, introduced in the preceding chapter that makes the API call and then renders the `UserList` component:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import { users } from './api';
import UserList from './UserList';

export default class UserListContainer extends Component {

  state = {
    data: fromJS({
      error: null,
      loading: 'loading...',
      users: [],
    }),
  }
}
```

```
// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

// When component has been rendered, "componentDidMount()" is called. This is where we should perform asynchronous behavior that will change the state of the component.
// In this case, we're fetching a list of users from the mock API.
componentDidMount() {
    users().then(
        (result) => {
            // Populate the "users" state, but also make sure the "error" and "loading" states are cleared.
            this.data = this.data
                .set('loading', null)
                .set('error', null)
                .set('users', fromJS(result.users));
        },
        (error) => {
            // When an error occurs, we want to clear the "loading" state and set the "error" state.
            this.data = this.data
                .set('loading', null)
                .set('error', error);
        }
    );
}

render() {
    return (
        <UserList {...this.data.toJS()} />
    );
}
```

Let's take a look at the `render()` method. It's sole job is to render the `<UserList>` component, passing in `this.state` as its properties. The actual API call happens in the `componentDidMount()` method. This method is called after the component is mounted into the DOM. This means that `<UserList>` will have rendered once, before any data from the API arrives. But this is fine, because we've set up the `UserListContainer` state to have a default loading message, and `UserList` will display this message while waiting for API data.

Once the API call returns with data, the `users` collection is populated, causing the `UserList` to re-render itself, only this time, it has the data it needs. So, why would we want to make this API call in `componentDidMount()` instead of in the component constructor, for example? The rule-of-thumb here is actually very simple to follow. Whenever there's asynchronous behavior that changes the state of a React component, it should be called from a lifecycle method. This way, it's easy to reason about how and when a component changes state.

Let's take a look at the `users()` mock API function call used here:

```
// Returns a promise that's resolved after 2
// seconds. By default, it will resolve an array
// of user data. If the "fail" argument is true,
// the promise is rejected.
export function users(fail) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (fail) {
        reject('epic fail');
      } else {
        resolve({
          users: [
            { id: 0, name: 'First' },
            { id: 1, name: 'Second' },
            { id: 2, name: 'Third' },
          ],
        });
      }
    }, 2000);
  });
}
```

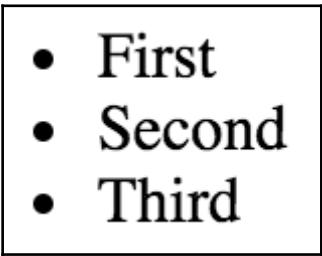
It simply returns a promise that's resolved with an array after 2 seconds. Promises are a good tool for mocking things like API calls because this enables you to use more than simple HTTP calls as a data source in your React components. For example, you might be reading from a local file or using some library that returns promises that resolve data from unknown sources.

Here's what the `UserList` component renders when the `loading` state is a string, and the `users` state is an empty array:



loading...

Here's what it renders when `loading` is `null` and `users` is non-empty:

- 
- First
 - Second
 - Third

I can't promise that this is the last time I'm going to make this point in the book, but I'll try to keep it to a minimum. I want to hammer home the separation of responsibilities between the `UserListContainer` and the `UserList` components. Because the container component handles the lifecycle management and the actual API communication, this enables us to create a very generic user list component. In fact, it's a functional component that doesn't require any state, which means this is easy to reuse throughout our application.

Initializing state with properties

The preceding example showed you how to initialize the state of a container component by making an API call in the `componentDidMount()` lifecycle method. However, the only populated part of the component state is the `users` collection. You might want to populate other pieces of state that don't come from API endpoints.

For example, the `error` and `loading` state messages have default values set when the state is initialized. This is great, but what if the code that is rendering `UserListContainer` wants to use a different loading message? You can achieve this by allowing properties to override the default state. Let's build on the `UserListContainer` component:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import { users } from './api';
import UserList from './UserList';

class UserListContainer extends Component {
  state = {
    data: fromJS({
      error: null,
      loading: null,
      users: []
    })
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // Called before the component is mounted into the DOM
  // for the first time.
  componentWillMount() {
    // Since the component hasn't been mounted yet, it's
    // safe to change the state by calling "setState()"
    // without causing the component to re-render.
    this.data = this.data
      .set('loading', this.props.loading);
  }

  // When component has been rendered, "componentDidMount()"
  // is called. This is where we should perform asynchronous
  // behavior that will change the state of the component.
  // In this case, we're fetching a list of users from
  // the mock API.
  componentDidMount() {
    users().then(  
  
```

```
(result) => {
  // Populate the "users" state, but also
  // make sure the "error" and "loading"
  // states are cleared.
  this.data = this.data
    .set('loading', null)
    .set('error', null)
    .set('users', fromJS(result.users));
},
(error) => {
  // When an error occurs, we want to clear
  // the "loading" state and set the "error"
  // state.
  this.data = this.data
    .set('loading', null)
    .set('error', error);
}
);
}

render() {
  return (
    <UserList {...this.data.toJS()} />
  );
}
}

UserListContainer.defaultProps = {
  loading: 'loading...',
};

export default UserListContainer;
```

You can see that `loading` no longer has a default string value. Instead, we've introduced `defaultProps`, which provide default values for properties that aren't passed in through JSX markup. The new lifecycle method we've added is `componentWillMount()`, and it uses the `loading` property to initialize the state. Since the `loading` property has a default value, it's safe to just change the state. However, calling `setState()` (via `this.data`) here doesn't cause the component to re-render itself. The method is called before the component mounts, so the initial render hasn't happened yet.

Let's see how we can pass state data to `UserListContainer` now:

```
import React from 'react';
import { render } from 'react-dom';

import UserListContainer from './UserListContainer';
```

```
// Renders the component with a "loading" property.  
// This value ultimately ends up in the component state.  
render()  
  <UserListContainer  
    loading="playing the waiting game..."  
  />  
,  
  document.getElementById('app')  
);
```

Pretty cool, right? Just because the component has state, doesn't mean that we can't be flexible and allow for customization of this state. We'll look at one more variation on this theme—updating component state through properties.

Here's what the initial loading message looks like when `UserList` is first rendered:

playing the waiting game...

Updating state with properties

You've seen how the `componentWillMount()` and `componentDidMount()` lifecycle methods help get your component the data it needs. There's one more scenario that we should consider here—re-rendering the component container.

Let's take a look at a simple `button` component that tracks the number of times it's been clicked:

```
import React from 'react';  
  
export default ({  
  clicks,  
  disabled,  
  text,  
  onClick,  
}) => (  
  <section>  
    { /* Renders the number of button clicks,  
       using the "clicks" property. */ }  
    <p>{clicks} clicks</p>  
  
    { /* Renders the button. It's disabled state
```

```
        is based on the "disabled" property, and
        the "onClick()" handler comes from the
        container component. */}
    <button
      disabled={disabled}
      onClick={onClick}
    >
      {text}
    </button>
  </section>
);
}
```

Now, let's implement a container component for this feature:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import MyButton from './MyButton';

class MyFeature extends Component {

  state = {
    data: fromJS({
      clicks: 0,
      disabled: false,
      text: '',
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // Sets the "text" state before the initial render.
  // If a "text" property was provided to the component,
  // then it overrides the initial "text" state.
  componentWillMount() {
    this.data = this.data
      .set('text', this.props.text);
  }

  // If the component is re-rendered with new
```

```
// property values, this method is called with the
// new property values. If the "disabled" property
// is provided, we use it to update the "disabled"
// state. Calling "setState()" here will not
// cause a re-render, because the component is already
// in the middle of a re-render.
componentWillReceiveProps({ disabled }) {
  this.data = this.data
    .set('disabled', disabled);
}

// Click event handler, increments the "click" count.
onClick = () => {
  this.data = this.data
    .update('clicks', c => c + 1);
}

// Renders the "<MyButton>" component, passing it the
// "onClick()" handler, and the state as properties.
render() {
  return (
    <MyButton
      onClick={this.onClick}
      {...this.data.toJS()}
    />
  );
}

MyFeature.defaultProps = {
  text: 'A Button',
};

export default MyFeature;
```

The same approach as the preceding example is taken here. Before the component is mounted, set the value of the text state to the value of the text property. However, we also set the text state in the `componentWillReceiveProps()` method. This method is called when property values change, or in other words, when the component is re-rendered. Let's see how we can re-render this component and whether or not the state behaves as we'd expect it to:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';

import MyFeature from './MyFeature';
```

```
// Determines the state of the button
// element in "MyFeature".
let disabled = true;

function render() {
  // Toggle the state of the "disabled" property.
  disabled = !disabled;

  renderJSX(
    (<MyFeature {...{ disabled }} />),
    document.getElementById('app')
  );
}

// Re-render the "<MyFeature>" component every
// 3 seconds, toggling the "disabled" button
// property.
setInterval(render, 3000);

render();
```

Sure enough, everything goes as planned. Whenever the button is clicked, the click counter is updated. But as you can see, `<MyFeature>` is re-rendered every 3 seconds, toggling the disabled state of the button. When the button is re-enabled and clicking resumes, the counter continues from where it left off.

Here is what the `MyButton` component looks like when first rendered:



Here's what it looks like after it has been clicked a few times and the button has moved into a disabled state:



Optimize rendering efficiency

The next lifecycle method you're going to learn about is used to implement heuristics that improve component rendering performance. You'll see that if the state of a component hasn't changed, then there's no need to render. Then, you'll implement a component that uses specific metadata from the API to determine whether or not the component needs to be re-rendered.

To render or not to render

The `shouldComponentUpdate()` lifecycle method is used to determine whether or not the component will render itself when asked to. For example, if this method were implemented, and returned false, the entire lifecycle of the component is short-circuited, and no render happens. This can be an important check to have in place if the component is rendering a lot of data and is re-rendered frequently. The trick is knowing whether or not the component state has changed.

This is the beauty of immutable data—we can easily check if it has changed. This is especially true if we're using a library such as `Immutable.js` to control the state of the component. Let's take a look at a simple list component:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

export default class MyList extends Component {
  state = {
    data: fromJS({
      items: new Array(5000)
```

```
.fill(null)
    .map((v, i) => i),
},
};

// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

// If this method returns false, the component
// will not render. Since we're using an Immutable.js
// data structure, we simply need to check for equality.
// If "state.data" is the same, then there's no need to
// render because nothing has changed since the last
// render.
shouldComponentUpdate(props, state) {
    return this.data !== state.data;
}

// Renders the complete list of items, even if it's huge.
render() {
    const items = this.data.get('items');

    return (
        <ul>
            {items.map(i => (
                <li key={i}>{i}</li>
            )));
        </ul>
    );
}
}
```

The `items` state is initialized to an `Immutable.js` `List` with 5000 items in it. This is a fairly large collection, so we don't want the virtual DOM inside React to constantly compute differences. The virtual DOM is efficient at what it does, but not nearly as efficient as code that can perform a simple `should` or `shouldn't` render check. The `shouldComponentRender()` method we've implemented here does exactly that. It compares the new state with the current state; if they're the same object, we completely sidestep the virtual DOM.

Now, let's put this component to work, and see what kind of efficiency gains we get:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';

import MyList from './MyList';

// Renders the "<MyList>" component. Then, it sets
// the state of the component by changing the value
// of the first "items" element. However, the value
// didn't actually change, so the same Immutable.js
// structure is reused. This means that
// "shouldComponentUpdate()" will return false.
function render() {
  const myList = renderJSX(
    (<MyList />),
    document.getElementById('app')
  );

  // Not actually changing the value of the first
  // "items" element. So, Immutable.js recognizes
  // that nothing changed, and instead of
  // returning a new object, it returns the same
  // "myList.data" reference.
  myList.data = myList.data
    .setIn(['items', 0], 0);
}

// Instead of performing 500,000 DOM operations,
// "shouldComponentUpdate()" turns this into
// 5000 DOM operations.
for (let i = 0; i < 100; i++) {
  render();
}
```

As you can see, we're just rendering `<MyList>`, over and over, in a loop. Each iteration has 5,000 list items to render. Since the state doesn't change, the call to `shouldComponentUpdate()` returns `false` on every one of these iterations. This is important for performance reasons, because there are a lot of them. Obviously, we're not going to have code that re-renders a component in a tight loop, in a real application. This code is meant to stress the rendering capabilities of React. If you were to comment out the `shouldComponentUpdate()` method, you'd see what I mean.



You may notice that we're actually changing state inside our `render()` function using `setIn()` on the `Immutable.js` map. This should result in a state change, right? This will actually return the same `Immutable.js` instance for the simple reason that the value we've set is the same as the current value: 0. When no change happens, `Immutable.js` methods return the same object, since it didn't mutate. Cool!

Using metadata to optimize rendering

In this section, we'll look at using metadata that's part of the API response to determine whether or not the component should re-render itself. Here's a simple user details component:

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

export default class MyUser extends Component {
  state = {
    modified: new Date(),
    first: 'First',
    last: 'Last',
  };

  // The "modified" property is used to determine
  // whether or not the component should render.
  shouldComponentUpdate(props, state) {
    return +state.modified > +this.state.modified;
  }

  render() {
    const {
      modified,
      first,
      last,
    } = this.state;

    return (
      <section>
        <p>{modified.toLocaleString()}</p>
        <p>{first}</p>
        <p>{last}</p>
      </section>
    );
  }
}
```

If you take a look at the `shouldComponentUpdate()` method, you can see that it's comparing the new modified state to the old modified state. This code makes the assumption that the modified value is a date that reflects when the data returned from the API was actually modified. The main downside to this approach is that the `shouldComponentUpdate()` method is now tightly coupled with the API data. The advantage is that we get a performance boost in the same way that we would with immutable data.

Here's how this heuristic looks in action:

```
import React from 'react';
import { render } from 'react-dom';

import MyUser from './MyUser';

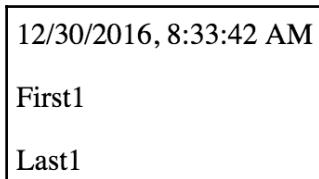
// Performs the initial rendering of "<MyUser>".
const myUser = render(
  <MyUser />,
  document.getElementById('app')
);

// Sets the state, with a new "modified" value.
// Since the modified state has changed, the
// component will re-render.
myUser.setState({
  modified: new Date(),
  first: 'First1',
  last: 'Last1',
});

// The "first" and "last" states have changed,
// but the "modified" state has not. This means
// that the "First2" and "Last2" values will
// not be rendered.
myUser.setState({
  first: 'First2',
  last: 'Last2',
});
```

As you can see, the component is now entirely dependent on the `modified` state. If it's not greater than the previous modified value, no render happens.

Here's what the component looks like after it's been rendered twice:



In this example, I didn't use immutable state data. Throughout this book, I'll use plain JavaScript objects as state for simple examples. `Immutable.js` is a great tool for this job, so I'll be using it a lot. At the same time, I want to make it clear that `Immutable.js` doesn't need to be used in every situation.

Rendering imperative components

Everything you've rendered so far in this book has been straightforward declarative HTML. As you know, life is never so simple: sometimes our React components need to implement some imperative code under the covers.

This is the key—hiding the imperative operations so that the code that renders your component doesn't have to touch it. In this section, you'll implement a simple jQuery UI button React component so that you can see how the relevant lifecycle methods help us encapsulate imperative code.

Rendering jQuery UI widgets

The jQuery UI widget library implements several widgets on top of standard HTML. It uses a progressive enhancement technique whereby the basic HTML is enhanced in browsers that support newer features. To make these widgets work, you first need to render HTML into the DOM somehow; then, make imperative function calls to create and interact with the widgets.

In this example, we'll create a React button component that acts as a wrapper around the jQuery UI widget. Anyone using the React component shouldn't need to know that behind the scenes, it's making imperative calls to control the widget. Let's see what the button component looks like:

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

// Import all the jQuery UI widget stuff...
import $ from 'jquery';
import 'jquery-ui/ui/widgets/button';
import 'jquery-ui/themes/base/all.css';

export default class MyButton extends Component {
  // When the component is mounted, we need to
  // call "button()" to initialize the widget.
  componentDidMount() {
    $(this.button).button(this.props);
  }

  // After the component updates, we need to use
  // "this.props" to update the options of the
  // jQuery UI button widget.
  componentDidUpdate() {
    $(this.button).button('option', this.props);
  }

  // Renders the "<button>" HTML element. The "onClick()"
  // handler will always be assigned, even if it's a
  // noop function. The "ref" property is used to assign
  // "this.button". This is the DOM element itself, and
  // it's needed by the "componentDidMount()" and
  // "componentDidUpdate()" methods.
  render() {
    return (
      <button
        onClick={this.props.onClick}
        ref={(button) => { this.button = button; }}
      />
    );
  }
}
```

The jQuery UI button widget expects a `<button>` element, so this is what's rendered by the component. An `onClick()` handler is assigned as well, and this function is expected to be found in the properties. There's also a `ref` property used here, which assigns the `button` argument to `this.button`. The reason this is done is so that the component has direct access to the underlying DOM element of the component. Generally, components don't need access to any DOM elements, but here, we need to issue imperative commands to the element.

For example, in the `componentDidMount()` method, we call the `button()` function and pass it properties from the component. We do something similar in the `componentDidUpdate()` method, which is called when property values change. Now, let's take a look at the button container component:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import MyButton from './MyButton';

class MyButtonContainer extends Component {
    // The initial state is an empty Immutable map, because
    // by default, we won't pass anything to the jQuery UI
    // button widget.
    state = {
        data: fromJS({}),
    }

    // Getter for "Immutable.js" state data...
    get data() {
        return this.state.data;
    }

    // Setter for "Immutable.js" state data...
    set data(data) {
        this.setState({ data });
    }

    // Before the component is mounted for the first time,
    // we have to bind the "onClick()" handler to "this"
    // so that the handler can set the state.
    componentWillMount() {
        this.data = this.data
            .merge(this.props, {
                onClick: this.props.onClick.bind(this),
            });
    }
}
```

```
// Renders the "<MyButton>" component with this
// component's state as properties.
render() {
  return (
    <MyButton {...this.state.data.toJS()} />
  );
}
}

// By default, the "onClick()" handler is a noop.
// This makes it easier because we can always assign
// the event handler to the "<button>".
MyButtonContainer.defaultProps = {
  onClick: () => {},
};

export default MyButtonContainer;
```

Once again, we have a container component that controls the state, which is then passed to `<MyButton>` as properties. The component has a default `onClick()` handler function. But, as you can see in the `componentWillMount()` method, we can pass a different click handler in as a property. Additionally, it's automatically bound to the component context, which is useful if the handler needs to change the button state. Let's look at an example of this:

```
import React from 'react';
import { render } from 'react-dom';

import MyButtonContainer from './MyButtonContainer';

// Simple button event handler that changes the
// "disabled" state when clicked.
function onClick() {
  this.data = this.data
    .set('disabled', true);
}

render(
  <section>
    { /* A simple button with a simple label. */ }
    <MyButtonContainer label="Text" />

    { /* A button with an icon, and a hidden label. */ }
    <MyButtonContainer
      label="My Button"
      icon="ui-icon-person"
      showLabel={false}
```

```
/>

{ /* A button with a click event handler. */
<MyButtonContainer
  label="Disable Me"
  onClick={onClick}
/>
</section>
),
document.getElementById('app')
);
```

Here, we have three jQuery UI button widgets, each controlled by a React component with no imperative code in sight. Here's how the buttons look:



Cleaning up after components

In this final section of the chapter, we'll think about cleaning up after components. You don't have to explicitly unmount components from the DOM—React handles that for us. There are some things that React doesn't know about and therefore cannot clean up for us once the component is removed.

It's for these types of circumstance that the `componentWillUnmount()` lifecycle method exists. The main use case for cleaning up after React components is asynchronous code.

For example, imagine a component that issues an API call to fetch some data when the component is first mounted. Now, imagine that this component is removed from the DOM before the API response arrives.

Cleaning up asynchronous calls

If your asynchronous code tries to set the state of a component that has been unmounted, nothing will happen. A warning will be logged, and the state isn't set. It's actually very important that this warning is logged; otherwise, we would have a hard time trying to figure subtle race condition bugs.

The correct approach is to create cancellable asynchronous actions. Here's a modified version of the `users()` API function we implemented earlier in the chapter:

```
// Adapted from:  
// https://facebook.github.io/react/blog/2015/12/16/  
// ismounted-antipattern.html  
function cancellable(promise) {  
  let cancelled = false;  
  
  // Creates a wrapper promise to return. This wrapper is  
  // resolved or rejected based on the wrapped promise, and  
  // on the "cancelled" value.  
  const promiseWrapper = new Promise((resolve, reject) => {  
    promise.then((val) => {  
      return cancelled ?  
        reject({ cancelled: true }) : resolve(val);  
    }, (error) => {  
      return cancelled ?  
        reject({ cancelled: true }) : reject(error);  
    });  
  );  
  
  // Adds a "cancel()" method to the promise, for  
  // use by the React component in "componentWillUnmount()".  
  promiseWrapper.cancel = function cancel() {  
    cancelled = true;  
  };  
  
  return promiseWrapper;  
}  
  
export function users(fail) {  
  // Make sure that the returned promise is "cancellable",  
  // by wrapping it with "cancellable()".  
  return cancellable(new Promise((resolve, reject) => {  
    setTimeout(() => {  
      if (fail) {  
        reject(fail);  
      } else {  
        resolve({  
          users: [  
            { id: 0, name: 'First' },  
            { id: 1, name: 'Second' },  
            { id: 2, name: 'Third' },  
          ],  
        });  
      }  
    }, 4000);  
  });  
}
```

```
});  
}
```

The trick is the `cancellable()` function, which wraps a promise with a new promise. The new promise has a `cancel()` method, which rejects the promise if called. It doesn't alter the actual asynchronous behavior that the promise is synchronizing. However, it does provide a generic and consistent interface for use within React components.

Now let's take a look at a container component that has the ability to cancel asynchronous behavior:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';
import { render } from 'react-dom';

import { users } from './api';
import UserList from './UserList';

// When the "cancel" link is clicked, we want to render
// a new element in "#app". This will unmount the
// "<UserListContainer>" component.
const onClickCancel = (e) => {
  e.preventDefault();

  render(
    (<p>Cancelled</p>),
    document.getElementById('app')
  );
};

export default class UserListContainer extends Component {
  state = {
    data: fromJS({
      error: null,
      loading: 'loading...',
      users: []
    })
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }
}
```

```
}

componentDidMount() {
  // We have to store a reference to any async promises,
  // so that we can cancel them later when the component
  // is unmounted.
  this.job = users();

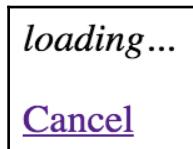
  this.job.then(
    (result) => {
      this.data = this.data
        .set('loading', null)
        .set('error', null)
        .set('users', fromJS(result.users));
    },
    // The "job" promise is rejected when it's cancelled.
    // This means that we need to check for the
    // "cancelled" property, because if it's true,
    // this is normal behavior.
    (error) => {
      if (!error.cancelled) {
        this.data = this.data
          .set('loading', null)
          .set('error', error);
      }
    },
  );
}

// This method is called right before the component
// is unmounted. It is here, that we want to make sure
// that any asynchronous behavior is cleaned up so that
// it doesn't try to interact with an unmounted component.
componentWillUnmount() {
  this.job.cancel();
}

render() {
  return (
    <UserList
      onClickCancel={onClickCancel}
      {...this.data.toJS()}
    />
  );
}
```

The `onClickCancel()` handler actually replaces the user list. This calls the `componentWillUnmount()` method, where we can cancel `this.job`. It's also worth noting that when the API call is made in `componentWillMount()`, a reference to the promise is stored in the component. This is necessary; otherwise, we'd have no way to cancel the async call.

Here's what the component looks like when rendered during a pending API call:



Summary

In this chapter, you learned a lot about the lifecycle of React components. We started things off with a discussion on why React components need a lifecycle in the first place. It turns out that React can't do everything automatically for us, so we need to write some code that's run at the appropriate time during the components' lifecycles.

Next, you implemented several components that were able to fetch their initial data and initialize their state from JSX properties. Then, you learned how to implement more efficient React components by providing a `shouldComponentRender()` method.

Lastly, you learned how to hide the imperative code that some components need to implement and how to clean up after asynchronous behavior. In the following chapter, you'll learn techniques that help ensure your components are being passed the right properties.

7

Validating Component Properties

In this chapter, you'll learn about property validation in React components. This might seem simple at first glance, but it's an important topic because it leads to bug-free components. We'll kick this off with a discussion about **predictable outcomes** and how this leads to components that are portable throughout the application.

Next, we'll walk through examples of some of the simple type-checking property validators that come with React. Then, we'll walk through some more complex property-validation scenarios. Finally, we'll wrap the chapter up with an example of how to implement your own custom validators.

Knowing what to expect

Property validation in React components is like field validation in HTML forms. The basic premise of validating form fields is that the user knows that they've provided a value that's not acceptable. Ideally, the validation error message is clear enough that the user can easily fix the situation. With React component property validation, we're trying to do the same thing—make it easy to fix a situation where an unexpected value was provided. Property validation enhances the developer experience, rather than the user experience.

The key aspect of property validation is knowing what's passed into the component as a property value. For example, if we're expecting an array and a Boolean is passed instead, something will probably go wrong. If you validate the property values using the built-in React validation mechanism, then you know that something unexpected was passed. If the component is expecting an array so that it can call the `map()` method, it'll fail if a Boolean value is passed because it has no `map()` method. However, before this failure happens, you'll see the property validation warning.

The idea isn't to **fail fast** with property validation. It's to provide information to the developer. When property validation fails, you know that something was provided as a component property that shouldn't have been. So, it's a simple matter of finding where the value is passed in the code and fixing it.



Fail fast is an architectural property of software in which the system will crash completely rather than continue running in an inconsistent state.

Promoting portable components

When we know what to expect from our component properties, the context in which the component is used becomes less important. This means that as long as the component is able to validate its property values, it really shouldn't matter where the component is used; it could easily be used by any feature.

If you want a generic component that's portable across application features, you can either write component validation code or you can write **defensive code** that runs at render time. The challenge with programming defensively is that it dilutes the value of declarative React components. Using React-style property validation, you can avoid writing defensive code. Instead, the property validation mechanism emits a warning when something doesn't pass, informing you that you need to go fix something.



Defensive code is code that needs to account for a number of edge cases during runtime, in a production environment. Coding defensively is necessary when potential problems cannot be detected during development, like with React component property validation.

Simple property validators

In this section, you'll learn how to use the simple property type validators available in the `PropTypes` object. Then, you'll learn how to accept any property value as well as make a property **required** instead of **optional**.

Basic type validation

Let's take a look at validators that handle the most primitive types of JavaScript values. You will use these validators frequently, as you'll want to know that a property is a string or that it's a function. This example will also introduce you to the mechanisms involved with setting up validation on a component. So, here's the component itself; it just renders some properties using basic markup:

```
import React, { PropTypes } from 'react';

const MyComponent = ({
  myString,
  myNumber,
  myBool,
  myFunc,
  myArray,
  myObject,
}) => (
  <section>
    { /* Strings and numbers can be rendered
         just about anywhere. */ }
    <p>{myString}</p>
    <p>{myNumber}</p>

    { /* Booleans are typically used as property
         values. */ }
    <p><input type="checkbox" defaultChecked={myBool} /></p>

    { /* Functions can return values, or be assigned as
         event handler property values. */ }
    <p>{myFunc()}</p>
```

```
{ /* Arrays are typically mapped to produce new
       JSX elements. */ }
<ul>
  {myArray.map(i => (
    <li key={i}>{i}</li>
  )))
</ul>

{ /* Objects typically use their properties in some
       way. */ }
<p>{myObject.myProp}</p>
</section>
);

// The "propTypes" specification for this component.
MyComponent.propTypes = {
  myString: PropTypes.string,
  myNumber: PropTypes.number,
  myBool: PropTypes.bool,
  myFunc: PropTypes.func,
  myArray: PropTypes.array,
  myObject: PropTypes.object,
};

export default MyComponent;
```

There are two key pieces to the property validation mechanism. First, you have the static `propTypes` property. This is a class-level property, not an instance property. When React finds `propTypes`, it uses this object as the property specification of the component. Second, you have the `PropTypes` tool, which has several built-in validator functions.

In this example, `MyComponent` has six properties, each with their own type. When you look at the `propTypes` specification, it should be clear what type of values this component will accept. Let's take a look at this now and render this component with some property values:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';

import MyComponent from './MyComponent';

// The properties that we'll pass to the component.
// Each property is a different type, and corresponds
// to the "propTypes" spec of the component.
const validProps = {
  myString: 'My String',
  myNumber: 100,
  myBool: true,
```

```
myFunc: () => 'My Return Value',
myArray: ['One', 'Two', 'Three'],
myObject: { myProp: 'My Prop' },
};

// These properties don't correspond to the "<MyComponent>" spec, and will cause warnings to be logged.
const invalidProps = {
  myString: 100,
  myNumber: 'My String',
  myBool: () => 'My Reaturn Value',
  myFunc: true,
  myArray: { myProp: 'My Prop' },
  myObject: ['One', 'Two', 'Three'],
};

// Renders "<MyComponent>" with the given "props".
function render(props) {
  renderJSX(
    (<MyComponent {...props} />),
    document.getElementById('app')
  );
}

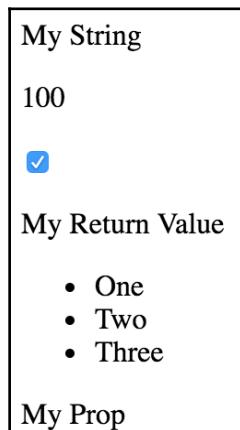
render(validProps);
render(invalidProps);
```

The first time `<MyComponent>` is rendered, it uses the `validProps` properties. These values all meet the component property specification, so no warnings are logged in the console. The second time around, the `invalidProps` properties are used, and this fails the property validation, because the wrong type is used in every property. The console output should look something like the following:

```
Invalid prop `myString` of type `number` supplied to `MyComponent`,
expected `string`
Invalid prop `myNumber` of type `string` supplied to `MyComponent`,
expected `number`
Invalid prop `myBool` of type `function` supplied to `MyComponent`,
expected `boolean`
Invalid prop `myFunc` of type `boolean` supplied to `MyComponent`, expected
`function`
Invalid prop `myArray` of type `object` supplied to `MyComponent`, expected
`array`
Invalid prop `myObject` of type `array` supplied to `MyComponent`, expected
`object`
TypeError: myFunc is not a function
```

This last error is interesting. You can clearly see that the property validation is complaining about the invalid property types. This includes the invalid function that was passed to `myFunc`. So, despite the type checking that happens on the property, the JSX will still try to call the value if it were a function.

Here's what the rendered output looks like:



Once again, the aim of property validation in React components is to help you discover bugs during development. When React is in production mode, property validation is turned off completely. This means that you don't have to concern yourself with writing expensive property validation code; it'll never run in production. However, the error will still occur, so fix it.

Requiring values

Let's make some adjustments to the preceding example. The component property specification required specific types for values, but these are only checked if the property is passed to the component as a JSX attribute. For example, you could have completely omitted the `myFunc` property and it would have validated. Thankfully, the `PropTypes` functions have a tool that lets you specify that a property must be provided and it must have a specific value. Here's the modified component:

```
const MyComponent = ({  
  myString,  
  myNumber,  
  myBool,  
  myFunc,
```

```
myArray,  
myObject,  
) => (  
  <section>  
    <p>{myString}</p>  
    <p>{myNumber}</p>  
    <p><input type="checkbox" defaultChecked={myBool} /></p>  
    <p>{myFunc()}</p>  
    <ul>  
      {myArray.map(i => (  
        <li key={i}>{i}</li>  
      ))}  
    </ul>  
    <p>{myObject.myProp}</p>  
  </section>  
)  
  
// The "propTypes" specification for this component. Every  
// property is required, because they each have the  
// "isRequired" property.  
MyComponent.propTypes = {  
  myString: PropTypes.string.isRequired,  
  myNumber: PropTypes.number.isRequired,  
  myBool: PropTypes.bool.isRequired,  
  myFunc: PropTypes.func.isRequired,  
  myArray: PropTypes.array.isRequired,  
  myObject: PropTypes.object.isRequired,  
};  
  
export default MyComponent;
```

As you can see, not much has changed between this component and the one we implemented in the preceding section. The main difference is with the specs in `propTypes`. You'll notice that the `isRequired` value is appended to each of the type validators used. So, for instance, `string.isRequired` means that the property value must be a string, and the property cannot be missing. Let's put this component to the test now:

```
import React from 'react';  
import { render as renderJSX } from 'react-dom';  
  
import MyComponent from './MyComponent';  
  
const validProps = {  
  myString: 'My String',  
  myNumber: 100,  
  myBool: true,  
  myFunc: () => 'My Return Value',
```

```
myArray: ['One', 'Two', 'Three'],
myObject: { myProp: 'My Prop' },
};

// The same as "validProps", except it's missing
// the "myObject" property. This will trigger a
// warning.
const missingProp = {
  myString: 'My String',
  myNumber: 100,
  myBool: true,
  myFunc: () => 'My Return Value',
  myArray: ['One', 'Two', 'Three'],
};

// Renders "<MyComponent>" with the given "props".
function render(props) {
  renderJSX(
    (<MyComponent {...props} />),
    document.getElementById('app')
  );
}

render(validProps);
render(missingProp);
```

The first time around, the component is rendered with all the correct property types. The second time around, the component is rendered without the `myObject` property. The console errors should look as follows:

```
Required prop `myObject` was not specified in `MyComponent`.
Cannot read property 'myProp' of undefined
```

You can see that, thanks to the property specification for `myObject`, it's obvious that an object value needs to be provided to the `myObject` property. The last error is because the component assumes that there is an object with `myProp` as a property. But that's fine, because thanks to the property validation, the problem is easy to find and fix.



Ideally, we would validate for the `myProp` object property in this example since it's directly used in the JSX. The specific properties that are used in the JSX markup for the shape of an object, and shape can be validated as you'll see later in the chapter.

Any property value

The final topic of this section is the `any` property validator. That is, it doesn't actually care what value it gets—anything is valid, including not passing a value at all. In fact, the `isRequired` validator can be combined with the `any` validator. For example, if you're working on a component and you just want to make sure that something is passed, but not sure exactly which type you're going to need yet, you could do something like `myProp: PropTypes.any.isRequired`.

Another reason to have the `any` property validator for the sake of consistency. Every component should have property specifications. The `any` validator is useful in the beginning, when we're not exactly sure what the property type will be. We can at least begin the property spec, then refine it later as things unfold.

Let's take a look at some code now:

```
import React, { PropTypes } from 'react';

// Renders a component with a header and a simple
// progress bar, using the provided property
// values.
const MyComponent = ({  
  label,  
  value,  
  max,  
}) => (  
  <section>  
    <h5>{label}</h5>  
    <progress {...{ max, value }} />  
  </section>  
)  
  
// These property values can be anything, as denoted by
// the "PropTypes.any" prop type.
MyComponent.propTypes = {
  label: PropTypes.any,
  value: PropTypes.any,
  max: PropTypes.any,
};
```

This component doesn't actually validate anything because the three properties in its property spec will accept anything. However, it's a good starting point, because at a glance, I can see the names of the three properties that this component uses. So later on, when I decide exactly which types these properties should have, the change is simple. Let's see this component in action now:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

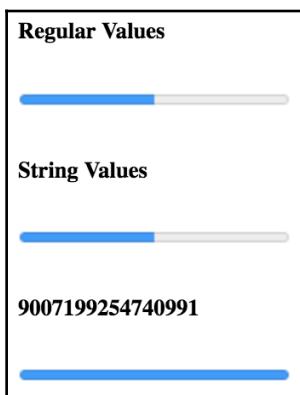
render((
  <section>
    { /* Passes a string and two numbers to
        "<MyComponent>". Everything works as
        expected. */ }
    <MyComponent
      label="Regular Values"
      max={20}
      value={10}
    />

    { /* Passes strings instead of numbers to the
        progress bar, but they're correctly
        interpreted as numbers. */ }
    <MyComponent
      label="String Values"
      max="20"
      value="10"
    />

    { /* The "label" has no issue displaying
        "MAX_SAFE_INTEGER", but the date that's
        passed to "max" causes the progress bar
        to break. */ }
    <MyComponent
      label={Number.MAX_SAFE_INTEGER}
      max={new Date()}
      value="10"
    />
  </section>
),
document.getElementById('app')
);
```

As you can see, strings and numbers are interchangeable in several places. So, restricting to just one or the other seems overly restrictive. As you'll see in the next section, React has other property validators that allow you to further restrict property values allowed into your component.

Here's what our component looks like when rendered:



Type and value validators

In this section, we'll look at the more advanced validator functionality available in the React `PropTypes` facility. First, you'll learn about the element and node validators that check for values that can be rendered inside HTML markup. Then, you'll see how to check for specific types, beyond the primitive type checking you saw in the previous section. Finally, we'll implement validation that looks for specific values.

Things that can be rendered

Sometimes, you just want to make sure that a property value is something that can be rendered by JSX markup. For example, if a property value is an array, this can't be rendered by putting it in `{ }`. You have to map the array items to JSX elements.

This sort of checking is especially useful if your component passes property values to other elements as children. Let's look at an example of what this looks like:

```
import React, { PropTypes } from 'react';

const MyComponent = ({
  myHeader,
  myContent,
}) => (
  <section>
    <header>{myHeader}</header>
    <main>{myContent}</main>
  </section>
);

// The "myHeader" property requires a React
// element. The "myContent" property requires
// a node that can be rendered. This includes
// React elements, but also strings.
MyComponent.propTypes = {
  myHeader: PropTypes.element.isRequired,
  myContent: PropTypes.node.isRequired,
};

export default MyComponent;
```

This component has two properties that require values that can be rendered. The `myHeader` property wants an `element`. This can be any JSX element. The `myContent` property wants a `node`. This can be any JSX element or any string value. Let's pass this component some values and render it:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

// Two React elements we'll use to pass to
// "<MyComponent>" as property values.
const myHeader = (<h1>My Header</h1>);
const myContent = (<p>My Content</p>);

render(
  <section>
    { /* Renders as expected, both properties are passed
        React elements as values. */ }
    <MyComponent
      {...{ myHeader }}
```

```
{...{ myContent }}
```

/* Triggers a warning because "myHeader" is expecting
 a React element instead of a string. */

```
<MyComponent  
  myHeader="My Header"  
  {...{ myContent }}  
/>
```

```
/* Renders as expected. A string is a valid type for  
   the "myContent" property. */

```
<MyComponent
 {...{ myHeader }}
 myContent="My Content"
/>
```



```
/* Renders as expected. An array of React elements
 is a valid type for the "myContent" property. */

```
<MyComponent  
  {...{ myHeader }}  
  myContent={[myContent, myContent, myContent]}  
/>
```



```
</section>  
,
```



```
document.getElementById('app')
```



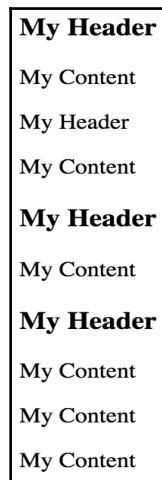
```
);
```


```


```

As you can see, the `myHeader` property is more restrictive about the values it will accept. The `myContent` property will accept a string, an element, or an array of elements. These two validators are important when passing in child data from properties, as this component does. For example, trying to pass a plain object or a function as a child will not work, and it's best if you check for this situation using a validator.

Here's what this component looks like when rendered:



Requiring specific types

Sometimes, you need a property validator that checks for a type defined by your application. For example, let's say you have the following user class:

```
import cuid from 'cuid';

// Simple class the exposes an API that the
// React component expects.
export default class MyUser {
  constructor(first, last) {
    this.id = cuid();
    this.first = first;
    this.last = last;
  }

  get name() {
    return `${this.first} ${this.last}`;
  }
}
```

Now, suppose that you have a component that wants to use an instance of this class as a property value. You would need a validator that checks that the property value is an instance of `MyUser`. Let's implement a component that does just that:

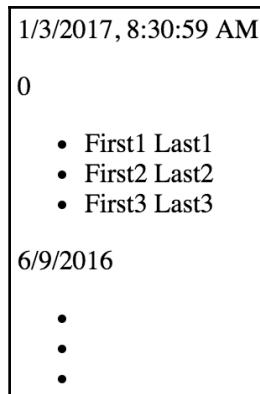
```
import React, { PropTypes } from 'react';
import MyUser from './MyUser';

const MyComponent = ({  
    myDate,  
    myCount,  
    myUsers,  
}) => (  
    <section>  
        { /* Requires a specific "Date" method. */ }  
        <p>{myDate.toLocaleString()}</p>  
  
        { /* Number or string works here. */ }  
        <p>{myCount}</p>  
        <ul>  
            { /* "myUsers" is expected to be an array of  
                "MyUser" instances. So we know that it's  
                safe to use the "id" and "name" property. */ }  
            {myUsers.map(i => (  
                <li key={i.id}>{i.name}</li>  
            ))}  
        </ul>  
    </section>  
);  
  
// The properties spec is looking for an instance of  
// "Date", a choice between a string or a number, and  
// an array filled with specific types.  
MyComponent.propTypes = {  
    myDate: PropTypes.instanceOf(Date),  
    myCount: PropTypes.oneOfType([  
        PropTypes.string,  
        PropTypes.number,  
    ]),  
    myUsers: PropTypes.arrayOf(PropTypes.instanceOf(MyUser)),  
};  
  
export default MyComponent;
```

This component has three properties that require specific types each that go beyond the basic type validators that you've seen so far in this chapter. Let's walk through these now:

- `myDate` requires an instance of `Date`. It uses the `instanceOf()` function to build a validator function that ensures the value is in fact a `Date` instance.
- `myCount` requires that the value either by a number or a string. This validator function is created by combining `oneOfType`, `PropTypes.number()`, and `PropTypes.string()`.
- `myUsers` requires an array of `MyUser` instances. This validator is built by combining `arrayOf()` and `instanceOf()`.

This example illustrates the number of scenarios that we can handle by combining the property validators provided by React. Here's what the rendered output looks like:



Requiring specific values

We've focused on validating the type of property values so far, but that's not always what we'll want to check for. Sometimes, specific values matter. Let's see how we can validate for specific property values:

```
import React, { PropTypes } from 'react';

// Any one of these is a valid "level"
// property value.
const levels = new Array(10)
  .fill(null)
  .map((v, i) => i + 1);
```

```
// This is the "shape" of the object we expect
// to find in the "user" property value.
const userShape = {
  name: PropTypes.string,
  age: PropTypes.number,
};

const MyComponent = ({  
  level,  
  user,  
}) => (  
  <section>  
    <p>{level}</p>  
    <p>{user.name}</p>  
    <p>{user.age}</p>  
  </section>  
);

// The property spec for this component uses
// "oneOf()" and "shape()" to define the required
// property values.
MyComponent.propTypes = {
  level: PropTypes.oneOf(levels),
  user: PropTypes.shape(userShape),
};

export default MyComponent;
```

The `level` property is expected to be a number from the `levels` array. This is easy to validate using the `oneOf()` function. The `user` property is expecting a specific `shape`. A `shape` is the expected properties and types of an object. The `userShape` defined in this example requires a `name` string and an `age` number. The key difference between `shape()` and `instanceOf()` is that we don't necessarily care about the type. We might only care about the values that are used in the JSX of the component.

Let's take a look at how this component is used:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

render((
  <section>
    { /* Works as expected. */ }
    <MyComponent
      level={10}
```

```
    user={{ name: 'Name', age: 32 }}
  />

  { /* Works as expected, the "online"
     property is ignored. */
<MyComponent
  user={{ name: 'Name', age: 32, online: false }}>
/>

  { /* Fails. The "level" value is out of range,
     and the "age" property is expecting a
     number, not a string. */
<MyComponent
  level={11}
  user={{ name: 'Name', age: '32' }}>
/>
</section>
),
document.getElementById('app')
);
```

Here's what the component looks like when it's rendered:

10
Name
32
Name
32
11
Name
32

Writing custom property validators

In this final section, you'll see how to build your own custom property validation functions and apply them in the property specification. Generally speaking, you should only implement your own property validator if you absolutely have to. The default validators available in `PropTypes` cover a wide range of scenarios.

However, sometimes, you need to make sure that very specific property values are passed to the component. Remember, these will not be run in production mode, so it's perfectly acceptable to iterate over collections. Let's implement a couple of custom validator functions now:

```
import React from 'react';

const MyComponent = ({  
  myArray,  
  myNumber,  
}) => (  
  <section>  
    <ul>  
      {myArray.map(i => (  
        <li key={i}>{i}</li>  
      ))}  
    </ul>  
    <p>{myNumber}</p>  
  </section>  
)  
  
MyComponent.propTypes = {  
  // Expects a property named "myArray" with a non-zero  
  // length. If this passes, we return null. Otherwise,  
  // we return a new error.  
  myArray: (props, name, component) =>  
    (Array.isArray(props[name]) &&  
     props[name].length) ? null : new Error(  
       `${component}.${name}: expecting non-empty array`  
    ),  
  
  // Expects a property named "myNumber" that's  
  // greater than 0 and less than 99. Otherwise,  
  // we return a new error.  
  myNumber: (props, name, component) =>  
    (Number.isFinite(props[name]) &&  
     props[name] > 0 &&  
     props[name] < 100) ? null : new Error(  
       `${component}.${name}: ` +  
       `expecting number between 1 and 99`  
    )  
};
```

```
)  
};  
  
export default MyComponent;
```

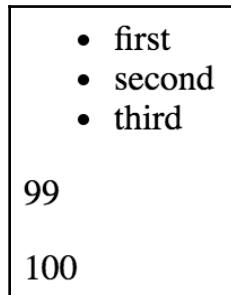
The `myArray` property expects a non-empty array, and the `myNumber` property expects a number that's greater than 0 and less than 100. Let's try passing these validators some data:

```
import React from 'react';  
import { render } from 'react-dom';  
  
import MyComponent from './MyComponent';  
  
render()  
  <section>  
    { /* Renders as expected... */ }  
    <MyComponent  
      myArray={['first', 'second', 'third']}  
      myNumber={99}  
    />  
  
    { /* Both custom validators fail... */ }  
    <MyComponent  
      myArray={[]}  
      myNumber={100}  
    />  
  </section>  
,  
document.getElementById('app')  
);
```

As you can see, the first element renders just fine, as both of the validators return null. However, the empty array and the number 100 cause both validators to return errors:

```
MyComponent.myArray: expecting non-empty array  
MyComponent.myNumber: expecting number between 1 and 99
```

Here's what the rendered output looks like:



Summary

The focus of this chapter has been React component property validation. When you implement property validation, you know what to expect; this promotes portability. The component doesn't care how the property values are passed to it, just as long as they're valid.

Then, we walked through several examples that used the basic React validators that check of primitive JavaScript types. You also learned that if a property is required, this must be made explicit. Then, you learned how to validate more complex property values by combining the built-in validators that come with React.

Finally, you implemented your own custom validator functions to perform validation that goes beyond what's possible with the built-in React validators. In the next chapter, you'll learn how to extend React components with new data and behavior.

8

Extending Components

In this chapter, you'll learn how to add new capabilities to existing components by extending them. There are two React mechanisms that you can use to extend a component, and we'll look at each in turn:

- Component inheritance
- Composition with higher-order components

We'll start by looking at basic component inheritance, just like the good old object-oriented class hierarchies that you're used to. Then we'll implement some higher-order components used as the ingredients in React component composition.

Component inheritance

Components are just classes. In fact, when you implement a component using **ES2015** class syntax, you extend the base `Component` class from React. You can keep on extending your classes like this to create your own base components.

In this section, you'll see how your components can inherit state, properties, and just about anything else, including JSX markup and event handlers.

Inheriting state

Sometimes, you have several React components that use the same initial state. You can implement a base component that sets this initial state. Then, any components that want to use this as their initial state can extend this component. Let's implement a base component that sets some basic state:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

export default class BaseComponent extends Component {
  state = {
    data: fromJS({
      name: 'Mark',
      enabled: false,
      placeholder: '',
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // The base component doesn't actually render anything,
  // but it still needs a render method.
  render() {
    return null;
  }
}
```

As you can see, the state is an immutable Map. This base component also implements the immutable data setter and getter methods. Let's implement a component that extends this one:

```
import React from 'react';
import BaseComponent from './BaseComponent';

// Extends "BaseComponent" to inherit the
// initial component state.
export default class MyComponent extends BaseComponent {
```

```
// This is our chance to build on the initial state.  
// We change the "placeholder" text and mark it as  
// "enabled".  
componentWillMount() {  
  this.data = this.data  
    .merge({  
      placeholder: 'Enter a name...',  
      enabled: true,  
    });  
}  
  
// Renders a simple input element, that uses the  
// state of this component as properties.  
render() {  
  const {  
    enabled,  
    name,  
    placeholder,  
  } = this.data.toJS();  
  
  return (  
    <label htmlFor="my-input">  
      Name:  
      <input  
        id="my-input"  
        disabled={!enabled}  
        defaultValue={name}  
        placeholder={placeholder}  
      />  
    </label>  
  );  
}  
}
```

This component doesn't actually have to set any initial state because it's already set by `BaseComponent`. Since the state is already an immutable `Map`, we can tweak the initial state in `componentWillMount()` using `merge()`. Here's what the rendered output looks like:



And if you delete the default text in the input element, you can see that the placeholder text added by `MyComponent` to the initial state is applied as expected:



Inheriting properties

Inheriting properties works exactly how you would expect. You define the default property values and the property types as static properties in a base class. Any classes that inherit this base also inherit the property values and the property specs. Let's take a look at a base class implementation:

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

export default class BaseComponent extends Component {
  // The specification for these base properties.
  static propTypes = {
    users: PropTypes.array.isRequired,
    groups: PropTypes.array.isRequired,
  }

  // The default values of these base properties.
  static defaultProps = {
    users: [],
    groups: [],
  }

  render() {
    return null;
  }
}
```

The class itself doesn't actually do anything. The only reason we're defining it is so that there's a place to declare the default property values and their type constraints. Respectively, these are the `defaultProps` and the `propTypes` static class attributes.

Now, let's take a look at a component that inherits these properties:

```
import React from 'react';
import { Map as ImmutableMap } from 'immutable';

import BaseComponent from './BaseComponent';

// Renders the given "text" as a header, unless
// the given "length" is 0.
const SectionHeader = ({ text, length }) =>
  ImmutableMap()
    .set(0, null)
    .get(length, (<h1>{text}</h1>));

export default class MyComponent extends BaseComponent {
  render() {
    const { users, groups } = this.props;

    // Renders the "users" and "groups" arrays. There
    // are not property validators or default values
    // in this component, since these are declared in
    // "BaseComponent".
    return (
      <section>
        <SectionHeader
          text="Users"
          length={users.length}
        />
        <ul>
          {users.map(i => (
            <li key={i}>{i}</li>
          )))
        </ul>

        <SectionHeader
          text="Groups"
          length={groups.length}
        />
        <ul>
          {groups.map(i => (
            <li key={i}>{i}</li>
          )))
        </ul>
      </section>
    );
  }
}
```

Let's try rendering `MyComponent` to make sure that the inherited properties are working as expected:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

const users = [
  'User 1',
  'User 2',
];

const groups = [
  'Group 1',
  'Group 2',
];

render(
  <section>
    { /* Renders as expected, using the defaults. */ }
    <MyComponent />

    { /* Renders as expected, using the "groups" default. */ }
    <MyComponent users={users} />
    <hr />

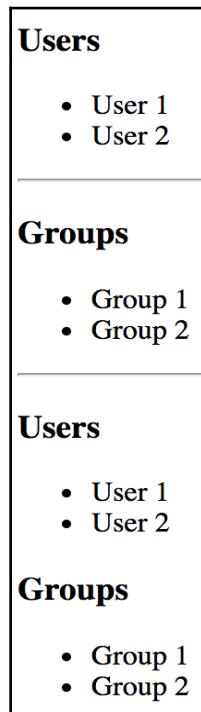
    { /* Renders as expected, using the "users" default. */ }
    <MyComponent groups={groups} />
    <hr />

    { /* Renders as expected, providing property values. */ }
    <MyComponent users={users} groups={groups} />

    { /* Fails to render, the property validators in the base
        component detect the invalid number type. */ }
    <MyComponent users={0} groups={0} />
  </section>
),
document.getElementById('app')
);
```

As you can see, despite the fact that `MyComponent` doesn't define any property defaults or types, we get the behavior we'd expect. When we try to pass numbers to the `users` and `groups` properties, we don't see anything rendered. That's because `MyComponent` is expecting a `map()` method on these property values, and there isn't one. However, before this exception happens, you can see the property validation failure warning, which explains exactly what happened. In this case, we passed an unexpected type.

If we were to remove this last element, everything else renders fine. Here's what the rendered content looks like:



Inheriting JSX and event handlers

The last area we'll touch upon with React component inheritance is JSX and event handlers. You might want to take this approach if you have a single UI component that has the same UI elements and event handling logic, but there are differences in what the initial state should be, depending on where the component is used.

For example, a base class would define the JSX and event handler methods while the more specific components define the initial state that's unique to the feature. Here's an example base class:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

export default class BaseComponent extends Component {
  state = {
    data: fromJS({
      items: [],
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // The click event handler for each item in the
  // list. The context is the lexically-bound to
  // this component.
  onClick = id => () => {
    this.data = this.data
      .update(
        'items',
        items => items
          .update(
            items.indexOf(items.find(i => i.get('id') === id)),
            item => item.update('done', d => !d)
          )
      );
  };

  // Renders a list of items based on the state
  // of the component. The style of the item
  // depends on the "done" property of the item.
  // Each item is assigned an event handler that
  // toggles the "done" state.
  render() {
    const { items } = this.data.toJS();

    return (

```

```
<ul>
  {items.map(i => (
    <li
      key={i.id}
      onClick={this.onClick(i.id)}
      style={{
        cursor: 'pointer',
        textDecoration: i.done ?
          'line-through' : 'none',
      }}
    >{i.name}</li>
  )));
</ul>
);
}
}
```

This base component renders a list of items that, when clicked, toggles the style of the item text. By default, the state of this component has an empty item list. This means that it is safe to render this component without setting the component state. However, that's not very useful, so let's give this list some items by inheriting the base component and setting the state:

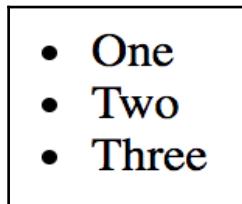
```
import React from 'react';
import { fromJS } from 'immutable';
import BaseComponent from './BaseComponent';

export default class MyComponent extends BaseComponent {

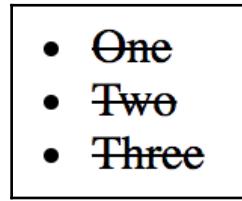
  // Initializes the component state, by using the
  // "data" getter method from "BaseComponent".
  componentWillMount() {
    this.data = this.data
      .merge({
        items: [
          { id: 1, name: 'One', done: false },
          { id: 2, name: 'Two', done: false },
          { id: 3, name: 'Three', done: false },
        ],
      });
  }
}
```

Remember, the `componentWillMount()` lifecycle method can safely set the state of the component. Here, the base component uses our `data` setter/getter to change the state of the component. Another thing that's handy about this approach is that if we want to override one of the event handlers of the base component, that's easy to do: simply define the method in `MyComponent`.

Here's what the list looks like when rendered:



And here's what the list looks like when all of the items have been clicked:



Composition with higher-order components

In this last section of the chapter, we'll cover **higher-order components**. If you're familiar with higher-order functions in functional programming, higher-order components work the same way. A **higher-order function** is a function that takes another function as input, and returns a new function as output. This returned function calls the original function in some way. The idea is to compose new behavior out of existing behavior.

With higher-order React components, you have a function that takes a component as input, and returns a new component as output. This is the preferred way to compose new behavior in React applications, and it seems that many of the popular React libraries are moving in this direction if they haven't already. There's simply more flexibility when composing functionality this way.

Conditional component rendering

One obvious use case for a higher-order component is conditional rendering. For example, depending on the outcome of some predicate, the component is rendered or nothing is rendered. The predicate could be anything that's specific to the application, such as permissions or something like that.

Implementing something like this in React is super easy. Let's say we have this super simple component:

```
import React from 'react';

// The world's simplest component...
export default () => (
  <p>My component...</p>
);
```

Now, to control the display of this component, we'd wrap it with another component. Wrapping is handled by the higher-order function.

If you hear the term wrapper in the context of React, it's probably referring to a higher-order component. Essentially, this is what it does, it wraps the component you pass to it.

Now let's see how easy it is to create a higher-order React component:

```
import React from 'react';

// A minimal higher-order function is all it
// takes to create a component repeater. Here, we're
// returning a function that calls "predicate()".
// If this returns true, then the rendered
// "<Component>" is returned.
export default (Component, predicate) =>
  props =>
    predicate() && (<Component {...props} />);
```

Only three lines? Are you kidding me? All thanks to the fact that we're returning a functional component. The two arguments to this function are `Component`, which is the component we're wrapping, and the `predicate` to call. As you can see, if the call to `predicate()` returns true, then `<Component>` is returned. Otherwise, nothing will be rendered.

Now, let's actually compose a new component using this function, and our super simple component that renders a paragraph of text:

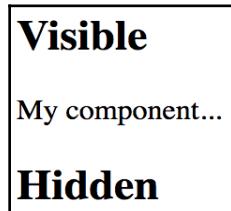
```
import React from 'react';
import { render } from 'react-dom';

import cond from './cond';
import MyComponent from './MyComponent';

// Two compositions of "MyComponent". The
// "ComposedVisible" version will render
// because the predicate returns true. The
// "ComposedHidden" version doesn't render.
const ComposedVisible = cond(MyComponent, () => true);
const ComposedHidden = cond(MyComponent, () => false);

render(
  <section>
    <h1>Visible</h1>
    <ComposedVisible />
    <h2>Hidden</h2>
    <ComposedHidden />
  </section>
),
document.getElementById('app')
);
```

We've just created two new components using `MyComponent`, `cond()`, and a predicate function. That's powerful, if you ask me. Here's the rendered output:



Providing data sources

Let's finish the chapter by looking at a more involved higher-order component example. You'll implement a data store function that wraps a given component with a data source. This type of pattern is handy to know, because it's used by React libraries such as **Redux**. Here's the `connect()` function that's used to wrap components:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

// The components that are connected to this store.
let components = fromJS([]);

// The state store itself, where application data is kept.
let store = fromJS({});

// Sets the state of the store, then sets the
// state of every connected component.
export function setState(state) {
  store = state;

  for (const component of components) {
    component.setState({
      data: store,
    });
  }
}

// Returns the state of the store.
export function getState() {
  return store;
}

// Returns a higher-order component that's connected
// to the "store".
export function connect(ComposedComponent) {
  return class ConnectedComponent extends Component {

    state = { data: store }

    // When the component is mounted, add it to
    // "components", so that it will receive updates
    // when the store state changes.
    componentWillMount() {
      components = components.push(this);
    }
  }
}
```

```
// Deletes this component from "components" when it is
// unmounted from the DOM.
componentWillUnmount() {
  const index = components.findIndex(this);
  components = components.delete(index);
}

// Renders "ComposedComponent", using the "store" state
// as properties.
render() {
  return (<ComposedComponent {...this.state.data.toJS()} />);
}
};

}
```

This module defines two internal immutable objects: `components` and `store`. The `components` list holds references to components that are listening to `store` changes. The `store` represents the application state as a whole.

The concept of a store stems from **Flux**, a set of architectural patterns used to build large-scale React applications. We'll touch on Flux ideas here and there throughout this book, but as a whole, Flux goes way beyond the scope of this book.

The important pieces of this module are the exported functions: `setState()`, `getState()`, and `connect()`. The `getState()` function simply returns a reference to the data store. The `setState()` function sets the state of the store, then notifies all components that the state of the application has changed. The `connect()` function is the higher-order function that wraps the given component with a new one. When the component is mounted, it registers itself with the store so that it will receive updates when the store changes state. It renders the composed component by passing the `store` as properties.

Now let's use this utility to build a simple filter and list. First, the list component:

```
import React, { PropTypes } from 'react';

// Renders an item list...
const MyList = ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    )))
  </ul>
);

MyList.propTypes = {
  items: PropTypes.array.isRequired,
```

```
};  
  
export default MyList;
```

Not too much happening here that you haven't already seen. Now let's look at the filter component:

```
import React, { PropTypes } from 'react';
import { fromJS } from 'immutable';
import { getState, setState } from './store';  
  
// When the filter input value changes.
function onChange(e) {
  // The state that we're working with...
  const state = getState();
  const items = state.get('items');
  const tempItems = state.get('tempItems');  
  
  // The new state that we're going to set on
  // the store.
  let newItems;
  let newTempItems;  
  
  // If the input value is empty, we need to restore the
  // items from "tempItems".
  if (e.target.value.length === 0) {
    newItems = tempItems;
    newTempItems = fromJS([]);
  } else {
    // If "tempItems" hasn't been set, make sure that
    // it gets the current items so that we can restore
    // them later.
    if (tempItems.isEmpty()) {
      newTempItems = items;
    } else {
      newTempItems = tempItems;
    }
  }
  // Filter and set "newItems".
  const filter = new RegExp(e.target.value, 'i');
  newItems = items.filter(i => filter.test(i));
}  
  
  // Updates the state of the store.
  setState(state.merge({
    items: newItems,
    tempItems: newTempItems,
  }));
}
```

```
}

// Renders a simple input element to filter a list.
const MyInput = ({ value, placeholder }) => (
  <input
    autoFocus
    value={value}
    placeholder={placeholder}
    onChange={onChange}
  />
);

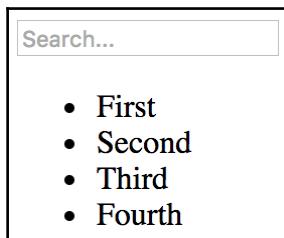
MyInput.propTypes = {
  value: PropTypes.string,
  placeholder: PropTypes.string,
};

export default MyInput;
```

The `MyInput` component itself is quite simple; it's just an `<input>` element. It's the `onChange` handler that needs some explanation, so let's spend some time here. The goal of this handler is to filter the user list so that only items that contain the current input text are displayed. Since the `MyList` component doesn't actually filter anything that's passed to it, this handler needs to alter what's passed to it. This is the essence of having a centralized store that holds the application state—it's how different components communicate.

The way we handle filtering the user list involves making a copy of it before anything else happens. This is because we need to actually modify the `items` state. Since we have the original, this is easy to restore later. Once the filtering has been performed, we use `setState()` to let the other components know that the application has changed state.

Here's what the rendered filter input and item list looks like:



Summary

In this chapter, you learned about the different ways to extend existing components. The first mechanism you learned about was inheritance. This is done using ES2015 class syntax and is useful for implementing common methods or JSX markup.

Then, you learned about higher-order components, where you use a function to wrap one component with another to provide it with new capabilities. This is the direction that new React applications are moving in, instead of inheritance and mixins.

In the next chapter, you'll learn how to render components based on the current URL.

9

Handling Navigation with Routes

Almost every web application requires **routing**. It is the process of responding to a given URL, based on a set of route configurations. A simple mapping, in other words, from the URL to rendered content. However, this simple task is more involved than it seems at first. This is why we're going to leverage the `react-router` package in this chapter, the *de facto* routing tool for React.

First, you'll learn the basics of declaring routes using JSX syntax. Then, we'll dig into the dynamic aspects of routing, such as dynamic path segments and query parameters. Next, you'll implement links using components from `react-router`. We'll close the chapter with an example that shows you how to lazy-load your components as the URL changes.

Declaring routes

If you've ever worked with routing outside of React, you probably already know that it can get messy quickly. With `react-router`, it's much easier to collocate routes with the content that they render. In this section, you'll see that this is due in large part to the declarative JSX syntax used to define routes.

We'll create a basic hello world example route so that you can get a basic handle on what routes look like in React applications. Then, we'll look at how you can organize your route declarations by feature instead of in a monolithic module. Finally, you'll implement a common parent-child routing pattern.

Hello route

Let's create a simple route that renders a simple component. First, we have the simplest possible React component that we want to render when the route is activated:

```
import React from 'react';

export default () => (
  <p>Hello Route!</p>
);
```

Next, let's look at the route definition itself:

```
import React from 'react';
import {
  Router,
  Route,
  browserHistory,
} from 'react-router';

import MyComponent from './MyComponent';

// Exports a "<Router>" component to be rendered.
export default (
  <Router history={browserHistory}>
    <Route path="/" component={MyComponent} />
  </Router>
);
```

As you can see, this module exports a JSX `<Router>` element. This is actually the top-level component of the application, as you'll see in a moment. But first, let's break down what's happening here. The `browserHistory` object is used to tell the Router that it should be using the native browser history API to keep track of navigation. Don't worry about this property too much; the majority of the time, you'll use this option.

Within the router, we have the actual routes declared as `<Route>` elements. There are two key properties of any route: `path` and `component`. These are pretty self-explanatory, when the `path` is matched against the active URL, the `component` is rendered. But where is it rendered, exactly?

To help answer this, let's take a look at the main module of the application:

```
import React from 'react';
import { render } from 'react-dom';

// Imports the "<Router>", and all the "<Route>"
```

```
// elements within.  
import routes from './routes';  
  
// The "<Router>" is the root element of the app.  
render(  
  routes,  
  document.getElementById('app')  
) ;
```

As I mentioned earlier, the router is the top-level component. The router doesn't actually render anything itself, it's merely responsible for managing how other components are rendered based on the current URL. Sure enough, when you look at this example in a browser, <MyComponent> is rendered as expected:

Hello Route!

That's the gist of how routing works in React. Let's look at declaring routes from the perspective of features now.

Decoupling route declarations

As the preceding section illustrated, declaring routes is nice and easy with `react-router`. The difficulty comes when your application grows and dozens of routes are all declared within a single module, because it's more difficult to mentally map a route to a feature.

To help with this, each top-level feature of the application can define its own `routes` module. This way, it's clear which routes belong to which feature. So, let's start with the root router module and see how this looks when it pulls in feature-specific routes:

```
import React from 'react';  
import {  
  Router,  
  browserHistory,  
} from 'react-router';  
  
// Import the routes from our features.  
import oneRoutes from './one/routes';  
import twoRoutes from './two/routes';  
  
// The feature routes are rendered as children of  
// the main router.  
export default (  
  <Router history={browserHistory}>  
    {oneRoutes}
```

```
{twoRoutes}  
</Router>  
);
```

In this example, the application has two phenomenally named features: one and two. The `oneRoutes` and `twoRoutes` values are pulled in from their respective feature directories and rendered as children in the `<Router>` element. Now this module will only get as big as the number of application features, instead of the number of routes, which could be substantially larger. Let's take a look at one of the feature routers now:

```
import React from 'react';  
import { Route, IndexRedirect } from 'react-router';  
  
// The pages that make up feature "one".  
import First from './First';  
import Second from './Second';  
  
// The routes of our feature. The "<IndexRedirect>"  
// handles "/one" requests by redirecting to "/one/1".  
export default (  
  <Route path="/one">  
    <IndexRedirect to="1" />  
    <Route path="1" component={First} />  
    <Route path="2" component={Second} />  
  </Route>  
) ;
```

This module exports a single `<Route>` element. Every route that this feature uses is a child `<Route>`. For example, if you run this example and point your browser to `/one/1`, you'll see the rendered content of `First`.

Feature 1, page 1

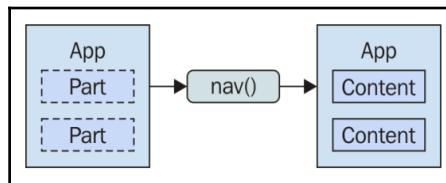
The `<IndexRedirect>` element is necessary to redirect `/one` to `/one/1`.

Toward the end of the chapter, we'll elaborate some more on how to structure routes and application features. But first, we need to think about parent and child routes in more depth.

Parent and child routes

The `App` component in the preceding example was the main component of the application. This is because it defined the root URL: `/`. However, once the user navigated to a specific feature URL, the `App` component was no longer relevant.

The challenge with routing is that sometimes you need a parent component, such as `App`, so that you don't have to keep re-rendering the page skeleton. By skeleton, I mean the content that's common to every page of the application. In this scenario, we always want `App` to render. When the URL changes, only certain parts of the `App` component render new content. Here's a diagram that illustrates the idea:



As the URL changes, the parent component is populated with the appropriate child components. Starting with the `App` component, let's figure out how to make this setup work:

```
import React, { PropTypes } from 'react';

// The "header" and "main" properties are the rendered
// components specified in the route. They're placed
// in the JSX of this component - "App".
const App = ({ header, main }) => (
  <section>
    <header>
      {header}
    </header>
    <main>
      {main}
    </main>
  </section>
);

// The "header" and "main" properties should be
// a React element.
App.propTypes = {
  header: PropTypes.element,
  main: PropTypes.element,
};
```

Now we have the page skeleton defined. It's now up to the router configuration to correctly render App. Let's look at this configuration now:

```
import React from 'react';
import {
  Router,
  Route,
  browserHistory,
} from 'react-router';

// The "App" component is rendered with every route.
import App from './App';

// The "User" components rendered with the "/users"
// route.
import UsersHeader from './users/UsersHeader';
import UsersMain from './users/UsersMain';

// The "Groups" components rendered with the "/groups"
// route.
import GroupsHeader from './groups/GroupsHeader';
import GroupsMain from './groups/GroupsMain';

// Configures the "/users" route. It has a path,
// and named components that are placed in "App".
const users = {
  path: 'users',
  components: {
    header: UsersHeader,
    main: UsersMain,
  },
};

// Configures the "/groups" route. It has a path,
// and named components that are placed in "App".
const groups = {
  path: 'groups',
  components: {
    header: GroupsHeader,
    main: GroupsMain,
  },
};

// Setup the router, using the "users" and "groups"
// route configurations.
export default (
  <Router history={browserHistory}>
    <Route path="/" component={App}>
```

```
<Route {...users} />
  <Route {...groups} />
</Route>
</Router>
);
```

This application has two pages/features—`users` and `groups`. Each of them has its own `App` components defined. For example, `UsersHeader` fills in the header value and `UsersMain` fills in the main value. Also note that each of the routes has a `components` property, where you can pass both the `header` and `main` values. This is how `App` is able to find the content that it needs based on the current route. If you run this example and navigate to `/users`, here's what you can expect to see:



A common mistake for newcomers to `react-router` is to think that the components themselves are passed to the parent. In reality, it's the rendered JSX element that's passed to the parent. The router actually renders the content and then passes the content to the parent.

Handling route parameters

The URLs that we've looked at so far in this chapter have all been static. Most applications will use both static and **dynamic routes**. In this section, you'll learn how to pass dynamic URL segments into your components, how to make these segments optional, and how to get query string parameters.

Resource IDs in routes

A common use case is to make the ID of a backend resource part of the URL. This makes it easy for our code to grab the ID, then make an API call that fetches the relevant resource data. Let's implement a route that renders a user detail page. This will require a route that includes the user ID, which then needs to somehow be passed to the component so that it can fetch the user.

We'll start with the route:

```
import React from 'react';
import {
  Router,
  Route,
  browserHistory,
} from 'react-router';

import UserContainer from './UserContainer';

export default (
  <Router history={browserHistory}>
    { /* Note the ":" before "id". This denotes
       a dynamic URL segment and the value will
       be available in the "params" property of
       the rendered component. */ }
    <Route path="/users/:id" component={UserContainer} />
  </Router>
);
```

The `:` syntax marks the beginning of a URL variable. So in this case, the `id` variable will be passed to the `UserContainer` component, which we'll now implement:

```
import React, { Component, PropTypes } from 'react';
import { fromJS } from 'immutable';

import User from './User';
import { fetchUser } from './api';

export default class UserContainer extends Component {
  state = {
    data: fromJS({
      error: null,
      first: null,
      last: null,
      age: null,
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }
}
```

```
}

componentWillMount() {
  // The dynamic URL segment we're interested in, "id",
  // is stored in the "params" property.
  const { params: { id } } = this.props;

  // Fetches a user based on the "id". Note that it's
  // converted to a number first.
  fetchUser(+id).then(
    // If the user was successfully fetched, then
    // merge the user properties into the state. Also,
    // make sure that "error" is cleared.
    (user) => {
      this.data = this.data
        .merge(user, { error: null });
    },
    // If the user fetch failed, set the "error" state
    // to the resolved error value. Also, make sure the
    // other user properties are restored to their defaults
    // since the component is now in an error state.
    (error) => {
      this.data = this.data
        .merge({
          error,
          first: null,
          last: null,
          age: null,
        });
    }
  );
}

render() {
  return (
    <User {...this.data.toJS()} />
  );
}
}

// Params should always be there...
UserContainer.propTypes = {
  params: PropTypes.object.isRequired,
};
```

The `params` property contains any dynamic parts of the URL. In this case, we're interested in the `id` parameter. Then, we pass the integer version of this value to the `fetchUser()` API call. If the URL is missing the segment completely, then this code won't run at all; the router will just complain about not being able to find a matching route. However, there's no type-checking done at the route level, which means it's up to you to design the failure mode.

In this example, the type cast will simply fail, if the user passes the string "one" for example, and a 500 error will happen. You could write a function that type-checks parameters and, instead of failing with an exception, responds with a 404 error. In any case, it's up to the application, not the `react-router` library, to provide a meaningful failure mode.

Now let's take a look at the API function we're using to respond with the user data:

```
// Mock data...
const users = [
  { first: 'First 1', last: 'Last 1', age: 1 },
  { first: 'First 2', last: 'Last 2', age: 2 },
];

// Returns a promise that resolves to a
// user from the "users" array, using the
// given "id" index. If nothing is found,
// the promise is rejected.
export function fetchUser(id) {
  return new Promise((resolve, reject) => {
    const user = users[id];

    if (user === undefined) {
      reject(`User ${id} not found`);
    } else {
      resolve(user);
    }
  });
}
```

Either the user is found in the `users` array of mock data or the promise is rejected. If rejected, the error-handling behavior of the `UsersContainer` component is invoked. Last but not least, we have the component responsible for actually rendering the user details:

```
import React, { PropTypes } from 'react';
import { Map as ImmutableMap } from 'immutable';

// Renders "error" text, unless "error" is
// null - then nothing is rendered.
const Error = ({ error }) =>
```

```
ImmutableMap()
  .set(null, null)
  .get(error, (<p><strong>{error}</strong></p>));

// Renders "children" text, unless "children"
// is null - then nothing is rendered.
const Text = ({ children }) =>
  ImmutableMap()
    .set(null, null)
    .get(children, (<p>{children}</p>));

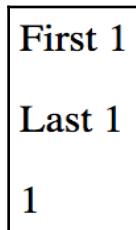
const User = ({
  error,
  first,
  last,
  age,
}) => (
  <section>
    { /* If there's an API error, display it. */ }
    <Error error={error} />

    { /* If there's a first, last, or age value,
       display it. */
      <Text>{first}</Text>
      <Text>{last}</Text>
      <Text>{age}</Text>
    }
  </section>
);

// Every property is optional, since we might
// have have to render them.
User.propTypes = {
  error: PropTypes.string,
  first: PropTypes.string,
  last: PropTypes.string,
  age: PropTypes.number,
};

export default User;
```

Now, if you run the example, and navigate to `/users/0`, this is what the rendered page looks like:



And if you navigate to a user that doesn't exist, `/users/2`, here's what you'll see:



Optional parameters

Sometimes, we need to specify that certain aspects of a feature be enabled. The best way to specify this in many cases is to encode the option as part of the URL or provide it as a query string. URLs work best for simple options, and query strings work best if there are many optional values the component can use.

Let's implement a user list component that renders a list of users. Optionally, we want to be able to sort the list in descending order. Let's make this an optional path segment in the route definition for this page:

```
import React from 'react';
import {
  Router,
  Route,
  browserHistory,
} from 'react-router';

import UsersContainer from './UsersContainer';

export default (
  <Router history={browserHistory}>
    { /* The ":desc" parameter is optional, and so
        is the "/" that precedes it. The "()" syntax
        marks anything that's optional. */ }
    <Route
```

```
    path="/users(/:desc)"
    component={UsersContainer}
  />
</Router>
);
```

It's the () syntax that marks anything in between as optional. In this case, it's the / and the—desc parameter. This means that the user can provide anything they want after /users/. This also means that the component needs to make sure that the string desc is provided, and that everything else is ignored.

It's also up to the component to handle any query strings provided to it. So while the route declaration doesn't provide any mechanism to define accepted query strings, the router will still process them and hand them to the component in a predictable way. Let's take a look at the user list container component now:

```
import React, { Component, PropTypes } from 'react';
import { fromJS } from 'immutable';

import Users from './Users';
import { fetchUsers } from './api';

export default class UsersContainer extends Component {
  // The "users" state is an empty immutable list
  // by default.
  state = {
    data: fromJS({
      users: []
    })
  }

  componentWillMount() {
    // The URL and query string data we need...
    const {
      props: {
        params,
        location: {
          query,
        },
      },
    } = this;

    // If the "params.desc" value is "desc", it means that
    // "desc" is a URL segment. If "query.desc" is true, it
    // means "desc" was provided as a query parameter.
    const desc = params.desc === 'desc' || !(query.desc);
```

```
// Tell the "fetchUsers()" API to sort in descending
// order if the "desc" value is true.
fetchUsers(desc).then((users) => {
  this.data = this.data
    .set('users', users);
});
}

render() {
  return (
    <Users {...this.data.toJS()} />
  );
}
}

UsersContainer.propTypes = {
  params: PropTypes.object.isRequired,
  location: PropTypes.object.isRequired,
};
```

The key aspect of this container is that it looks for either `params.desc` or `query.desc`. It uses this as an argument to the `fetchUsers()` API, to determine the sort order. So, here's what's rendered when you navigate to `/users`:

- User 1
- User 2
- User 3

And if we include the descending parameter by navigating to `/users/desc`, here's what we get:

- User 3
- User 2
- User 1

Using link components

In this section, we'll look at creating links. You might be tempted to use the standard `<a>` elements to link to pages controlled by `react-router`. The problem with this approach is that these links will try to locate the page on the backend by sending a GET request. This isn't what we want to happen, because the route configuration is already in the browser.

We'll start with a very simple example that illustrates how `<Link>` elements are just like `<a>` elements in most ways. Then, you'll see how to build links that use URL parameters and query parameters.

Basic linking

The idea of a link is pretty simple. We use it to point to routes, which subsequently render new content. However, the `Link` component also takes care of the browser history and looking up routes. Here's an application component that renders two links:

```
import React, { PropTypes } from 'react';
import { Link } from 'react-router';

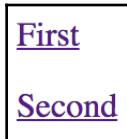
const App = ({ content }) => (
  <section>
    {content}
  </section>
);

App.propTypes = {
  content: PropTypes.node.isRequired,
};

// The "content" property has the default content
// for the "App" component. The "<Link>" elements
// handle dealing with the history API. Regular
// "<a>" links result in requests being sent to
// the server.
App.defaultProps = {
  content: (
    <section>
      <p><Link to="first">First</Link></p>
      <p><Link to="second">Second</Link></p>
    </section>
  ),
};
```

```
export default App;
```

The `to` property specifies the route to activate when clicked. In this case, the application has two routes—`/first` and `/second`. So here's what the rendered links look like:



And if you click the first link, the page content changes to look like this:



URL and query parameters

Constructing the dynamic segments of a path that's passed to `<Link>` is simple string manipulation. Everything that's part of the path goes in the `to` property. This means that we have to write more code ourselves to construct the string, but it also means less behind-the-scenes magic on behalf of the router.

On the other hand, the `query` property of `<Link>` accepts a map of query parameters and will construct the query string for you. Let's create a simple component that will echo back whatever is passed to the echo URL segment or the echo query parameter:

```
import React from 'react';

// Simple component that expects either an "echo"
// URL segment parameter, or an "echo" query parameter.
export default ({
  params,
  location: {
    query,
  },
}) => (
  <h1>{params.echo || query.echo}</h1>
);
```

Now let's take a look at the `App` component that renders two links. The first will build a string that uses a dynamic value as a URL parameter. The second will pass an object to the `query` property to build the query string:

```
import React, { PropTypes } from 'react';
import { Link } from 'react-router';

const App = ({ content }) => (
  <section>
    {content}
  </section>
);

App.propTypes = {
  content: PropTypes.node.isRequired,
};

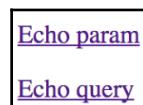
// Link parameter and query data...
const param = 'From Param';
const query = { echo: 'From Query' };

App.defaultProps = {
  content: (
    <section>
      { /* This "<Link>" uses a parameter as part of
          the "to" property. */ }
      <p><Link to={`echo/${param}`}>Echo param</Link></p>

      { /* This "<Link>" uses the "query" property
          to add query parameters to the link URL. */ }
      <p><Link to="echo" query={query}>Echo query</Link></p>
    </section>
  ),
};

export default App;
```

Here's what the two links look like when they're rendered:



The param link takes you to `/echo/From Param`, which looks like this:

From Param

The query link takes you to `/echo?echo=From+Query`, which looks like this:

From Query

Lazy routing

Earlier in the chapter, you learned how to split your route declarations by feature. This is done to avoid having a monolithic routes module, which can not only be difficult to decipher, but could also cause performance issues. Think about a large application that has dozens of features and hundreds of routes. Having to load all of these routes and components upfront would translate to a bad user experience.

In this final section of the chapter, we'll look at loading routes lazily. Part of this involves leveraging a loader such as Webpack, the loader used in this example. First, let's take a look at the main `App` component:

```
import React, { PropTypes } from 'react';
import { Link } from 'react-router';

// The "App" component is divided into
// "header" and "content" sections, and will
// simply render these properties.
const App = ({ header, content }) => (
  <section>
    <header>
      {header}
    </header>
    <main>
      {content}
    </main>
  </section>
);

// The "header" and "content" properties need to be
// renderable values.
App.propTypes = {
  header: PropTypes.node.isRequired,
  content: PropTypes.node.isRequired,
```

```
};

// The default content for our "App" component.
App.defaultProps = {
  header: (<h1>App</h1>),
  content: (
    <ul>
      <li><Link to="users">Users</Link></li>
      <li><Link to="groups">Groups</Link></li>
    </ul>
  ),
};

export default App;
```

This example application has two features: the brilliantly named `users` and `groups`. From the main page, there's a link to each feature. We know that this application will one day be successful and will grow to be huge. Therefore, we don't want to load all our routes and components up front, only when the user clicks one of these links.

Let's turn our attention to the main routes module now:

```
import React from 'react';
import {
  Router,
  browserHistory,
} from 'react-router';

import App from './App';

// Defines the main application route as
// a plain object, instead of a "<Route>"
// component.
const routes = {
  // The "path" and "component" are specified,
  // just like any other route.
  path: '/',
  component: App,

  // This allows for lazy route configuration loading.
  // The "require.ensure()" call allows for tools like
  // Webpack to split the code bundles into separate
  // modules that are loaded on demand. The actual
  // "require()" calls get the route configurations.
  // In this case, it's routes for the "users" feature
  // and the "groups" feature.
  getChildRoutes(partialNextState, callback) {
    require.ensure([], (require) => {
```

```
const { users } = require('./users/routes');
const { groups } = require('./groups/routes');

callback(null, [
  users,
  groups,
]);
});

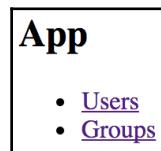
},
};

// The "routes" object is passed to the "routes"
// property. There are no nested "<Route>" elements
// needed for this router.
export default (
  <Router history={browserHistory} routes={routes} />
);
```

The key aspect of the App route is the `getChildRoutes()` method. This is called when a route lookup needs to happen. This is perfect, because this allows the user to interact with the application without having to load these routes, until they click on one of the links. This is called **lazy loading**, because this code loads two other route modules.

The `require.ensure()` function is what enables **code-splitting** in code bundlers such as Webpack. Instead of creating one big bundle, the `require.ensure()` function tells the bundler that the `require()` calls inside this callback can go into a separate bundle that's loaded on demand. I recommend running this example within the code you've downloaded for this book and looking at the network developer tools as you navigate around the application.

Here's what the main page looks like when it's rendered:



Now, let's take a look at one of the feature routes module:

```
import React from 'react';

import UsersHeader from './UsersHeader';
import UsersContent from './UsersContent';

// The route configuration for our "users" feature.
```

```
export const users = {

    // The components rendered by "App" are specified
    // here as "UsersHeader" and "UsersContent".
    path: 'users',
    components: {
        header: UsersHeader,
        content: UsersContent,
    },
    childRoutes: [

        // The "users/active" route lazily-loads the
        // "UsersActiveHeader" and "UsersActiveContent"
        // components when the route is activated.
        {
            path: 'active',
            getComponents(next, cb) {
                require.ensure([], (require) => {
                    const {
                        UsersActiveHeader,
                    } = require('./UsersActiveHeader');

                    const {
                        UsersActiveContent,
                    } = require('./UsersActiveContent');

                    cb(null, {
                        header: UsersActiveHeader,
                        content: UsersActiveContent,
                    });
                });
            },
        },
        // The "users/inactive" route lazily-loads the
        // "UsersInactiveHeader" and "UsersInactiveContent"
        // components when the route is activated.
        {
            path: 'inactive',
            getComponents(next, cb) {
                require.ensure([], (require) => {
                    const {
                        UsersInactiveHeader,
                    } = require('./UsersInactiveHeader');

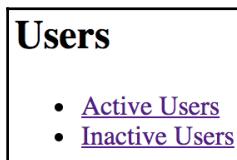
                    const {
                        UsersInactiveContent,
                    } = require('./UsersInactiveContent');
                });
            },
        },
    ],
}
```

```
        cb(null, {
          header: UsersInactiveHeader,
          content: UsersInactiveContent,
        });
      },
    ],
  );
}

export default users;
```

It follows the same lazy loading principle as the main `routes` module. This time, it's the `getComponents()` method that is called when a matched route is found. The component code bundle is then loaded.

Here's what the `Users` page looks like:



And here's what the `Active Users` page looks like:



This technique is absolutely necessary for large applications. However, refactoring existing features to use this lazy loading approach to routing isn't too difficult. So, don't worry too much about lazy loading in the early days of your application.

Summary

In this chapter, you learned about routing in React applications. The job of a router is to render content that corresponds to a URL. The `react-router` package is the standard tool for the job.

You learned how routes are JSX elements, just like the components they render. Sometimes you need to split routes into feature-based `routes` modules. A common pattern for structuring page content is to have a parent component that renders the dynamic parts as the URL changes.

You learned how to handle the dynamic parts of URL segments and query strings. You also learned how to build links throughout your application using the `<Link>` element. Finally, you saw how large applications are able to scale by lazily loading their route configurations and their components.

In the next chapter, you'll learn how to render React components in Node.js.

10

Server-Side React Components

Everything that you've learned so far in this book has been React code that runs in web browsers. However, React isn't confined to the browser for rendering, and in this chapter, you'll learn about rendering components from a Node.js server.

The first section of this chapter briefly touches upon the high-level server rendering concepts. The next four sections go into depth, teaching you how to implement the most crucial aspects of server-side rendering with React.

Let's do this.

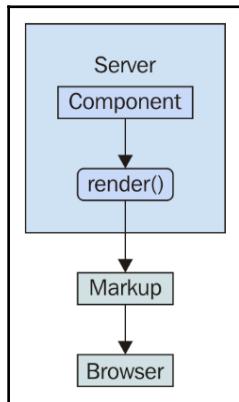
What is isomorphic JavaScript?

Another term for **server-side rendering** is **isomorphic JavaScript**. This is a fancy way of saying JavaScript code that can run in the browser and in Node.js without modification. In this section, we'll go over the basic concepts of isomorphic JavaScript before diving into code.

The server is a render target

The beauty of React is that it's a small abstraction layer that sits on top of a rendering target. So far, the target has been the browser, but it can also be the server. The render target can be anything, just as long as the correct translation calls are implemented behind the scenes.

In the case of rendering on the server, we're simply rendering our components to strings. The server can't actually display rendered HTML; all it can do is send the markup to the browser. The idea is illustrated in the following diagram:



We've established that it's possible to render a React component on the server and send the rendered output to the browser. The question is, why would you want to do this on the server instead of the browser?

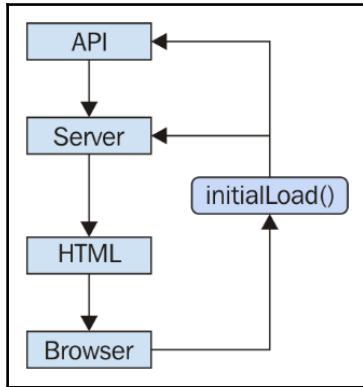
Initial load performance

The main motivation behind server-side rendering, for me personally, is improved performance. In particular, the initial rendering just feels faster for the user and this translates to an overall better user experience. It doesn't matter how fast your application is once it's loaded and ready to go; it's the initial load time that leaves a lasting impression on your users.

There are three reasons that this approach means better performance for the initial load:

- The rendering that takes place on the server is generating a string; there's no need to compute a diff or to interact with the DOM in any way. Producing a string of rendered markup is inherently faster than rendering components in the browser.
- The rendered HTML is displayed as soon as it arrives. Any JavaScript code that needs to run on the initial load is run after the user is already looking at the content.
- There are fewer network requests to fetch data from the API, because these have already happened in the backend.

The following diagram illustrates these performance ideas:



Sharing code between the backend and frontend

There's a good chance that your application will need to talk to API endpoints that are out of your control. For example, an application that is composed from many different microservice endpoints. It's rare that we can use data from these services without modification. Rather, we have to write code that transforms data so that it's usable by our React components.

If we're rendering our components in a Node.js server, then this data transformation code will be used by both the client and the server, because on the initial load, the server will need to talk to the API, and later on, the component in the browser will need to talk to the API.

It's not just about transforming the data that's returned from these services either. For example, we have to think about providing input to them as well, like when creating or modifying resources.

The fundamental adjustment that we need to make as React programmers is to assume that any given component we implement will need to be rendered on the server. This may seem like a minor adjustment, but the devil is in the details. Speaking of which, let's jump into some code examples now.

Rendering to strings

The aim with rendering components in Node.js is to render to strings, instead of trying to figure out the best way to insert them into the DOM. The string content is then returned to the browser, which displays this to the user immediately. Let's get our feet wet with a basic example. First, we have a simple component that we want to render:

```
import React, { PropTypes } from 'react';

const App = ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    )))
  </ul>
);

App.propTypes = {
  items: PropTypes
    .arrayOf(PropTypes.string)
    .isRequired,
};

export default App;
```

Next, let's implement the server that will render this component when the browser asks for it:

```
import React from 'react';

// The "renderToString()" function is like "render()",  

// except it returns a rendered HTML string instead of  

// manipulating the DOM.  

import { renderToString } from 'react-dom/server';  

import express from 'express';

// The component that we're going to render as a string.  

import App from './App';

// The "doc()" function takes the rendered "content"  

// of a React component and inserts it into an  

// HTML document skeleton.  

const doc = content =>

  <!doctype html>  

<html>  

  <head>
```

```
<title>Rendering to strings</title>
</head>
<body>
  <div id="app">${content}</div>
</body>
</html>
`;

const app = express();

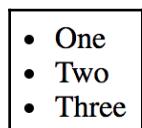
// The root URL of the APP, returns the rendered
// React component.
app.get('/', (req, res) => {
  // Some properties to render...
  const props = {
    items: ['One', 'Two', 'Three'],
  };

  // Render the "App" component using
  // "renderToString()"
  const rendered = renderToString(
    <App {...props} />
  );

  // Use the "doc()" function to build the final
  // HTML that is sent to the browser.
  res.send(doc(rendered));
});

app.listen(8080, () => {
  console.log('Listening on 127.0.0.1:8080');
});
```

Now if you visit `http://127.0.0.1:8080` in your browser, you'll see the rendered component content:



There are two things to pay attention to with this example. First, there's the `doc()` function. This creates the basic HTML document template with a placeholder for rendered React content. The second is the call to `renderToString()`, which is just like the `render()` call that you're used to. This is called in the server's request handler and the rendered string is sent to the browser.

Backend routing

In the preceding example, we implemented a single request handler in the server that responded to requests for the root URL (`/`). Obviously your application is going to need to handle more than a single route. You learned how to use the `react-router` package for routing in the previous chapter. Now, you're going to see how to use the router in Node.js.

First, let's take a look at the main app component:

```
import React, { PropTypes } from 'react';
import { Link } from 'react-router';

const App = ({ header, content }) => (
  <section>
    <header>
      {header}
    </header>
    <main>
      {content}
    </main>
  </section>
);

App.propTypes = {
  header: PropTypes.node.isRequired,
  content: PropTypes.node.isRequired,
};

App.defaultProps = {
  header: (<h1>App</h1>),
  content: (
    <ul>
      <li><Link to="first">First</Link></li>
      <li><Link to="second">Second</Link></li>
    </ul>
  ),
};
```

```
export default App;
```

It defines a simple structure where other components can fill in the blanks, so to speak. By default, it will render links to the two other pages of the application. Now let's take a look at the `routes` module:

```
import React from 'react';
import {
  Router,
  Route,
  browserHistory,
} from 'react-router';

import App from './App';
import FirstHeader from './first/FirstHeader';
import FirstContent from './first/FirstContent';
import SecondHeader from './second/SecondHeader';
import SecondContent from './second/SecondContent';

const first = {
  header: FirstHeader,
  content: FirstContent,
};

const second = {
  header: SecondHeader,
  content: SecondContent,
};

export default (
  <Router history={browserHistory}>
    <Route path="/" component={App}>
      <Route path="first" components={first} />
      <Route path="second" components={second} />
    </Route>
  </Router>
);
```

Once again, this should look familiar. We're importing components from feature directories and assigning them to child routes of the application. This configuration will work fine on the client, but will it work on the server? Let's implement that now:

```
import React from 'react';

import { renderToString } from 'react-dom/server';
import { match, RouterContext } from 'react-router';
import express from 'express';
```

```
// We need the main "routes" module in the
// server...
import routes from './routes';

const doc = content =>
`  

<!doctype html>
<html>
  <head>
    <title>Backend Routing</title>
  </head>
  <body>
    <div id="app">${content}</div>
  </body>
</html>
`;  

const app = express();  

// Servers all paths, because the URL pattern matching
// is handled by react-router instead of Express.
app.get('/*', (req, res) => {
  // The "match()" function from react-router
  // will generate properties that we can pass
  // to "<RouterContext>". We can then use
  // "renderToString()" to generate our static
  // markup.
  match({
    routes,
    location: req.url,
  }, (err, redirect, props) => {
    if (err) {
      res.status(500).send(err.message);
    } else if (redirect) {
      res.redirect(
        302,
        `${redirect.pathname}${redirect.search}`
      );
    } else if (props) {
      const rendered = renderToString((
        <RouterContext {...props} />
      ));
      res.send(doc(rendered));
    } else {
      res.status(404).send('Not Found');
    }
  });
});
```

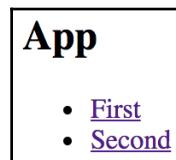
```
});  
  
app.listen(8080, () => {  
  console.log('Listening on 127.0.0.1:8080');  
});
```

Excellent, we now have both frontend and backend routing! How does this work exactly? Let's start with the request handler path. We've changed this so that it's now a wildcard (`/*`). Now this handler is called for every request.

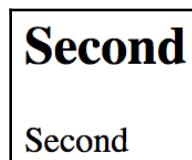
With every request, the handler calls the `match()` function from `react-router`. This is a low-level way to test the router configuration against the current URL. The callback that's passed to this function is called with `error`, `redirect`, and `property` values. It's up to you to handle these accordingly.

For example, if there's an error, we don't want to render the component. Instead, we simply respond to the client with a `500` status and an error message. The properties are passed to a `RouterContext` component, which basically renders the correct component for us based on the route.

Now our application is starting to look like a real end-to-end React rendering solution. This is what the server renders if you hit the root URL `/`:



If you hit the `/second` URL, the Node.js server will render the correct component:



If you navigate from the main page to the first page, the request goes back to the server. We need to figure out how to get the frontend code to the browser so that it can take over after the initial render.

Frontend reconciliation

The only thing that was missing from the last example was the client JavaScript code. No big deal, right? The user actually wants to use the application and the server needs to deliver the client code bundle. How would this work? We want routing to work in both the frontend and the backend, without modification to the routes themselves. In other words, the server handles routing in the initial request, then the browser takes over as the user starts clicking things and moving around in the application.

This is pretty easy to do. Let's create a main module (it probably looks familiar from examples in the previous chapter):

```
import React from 'react';
import { render } from 'react-dom';

import routes from './routes';

// Nothing special here. React sees the checksum on the
// root element, and determines that there's no need
// to render data yet.
render(
  routes,
  document.getElementById('app')
);
```

That's it on the client end. Instead of calling `match()` and `renderToString()` like we do in the Express request handler, we just render the routes using `render()`. React knows about backend rendering and will look for content that has already been rendered by React components. There's a **checksum** that's placed on the root element. This is compared against the checksum for actually rendering the component. If they don't match, the server content is replaced by re-rendering the component.

In other words, when a component renders content differently on the client than it would on the server, there's a problem, because there is nothing to gain; React is forced to re-render the component.

For example, trying to render JSX like this is never a good idea:

```
<strong>{typeof window}</strong>
```

This is guaranteed to have different HTML output on the client and on the server. Another trouble spot for rendering different content on the backend is when API data is involved. We'll address this issue next, but first we have one adjustment to make in our server code in order to make reconciliation work; we need to insert the link to the main **Webpack** bundle:

```
const doc = content =>
`<!doctype html>
<html>
  <head>
    <title>Frontend Reconciliation</title>
    <script src="/static/main-bundle.js" defer></script>
  </head>
  <body>
    <div id="app">${content}</div>
  </body>
</html>
`;
```

Fetching data

We're getting close to having a fully functional end-to-end rendering solution for our React application. The last remaining issue is state, more specifically, data that comes from some API endpoint. Our components need to be able to fetch this data on the server just as they would on the client so that they can generate the appropriate content. We also have to pass the initial state, along with the initial rendered content, to the browser. Otherwise, our code won't know when something has changed after the first render.

To implement this, I'm going to introduce the Flux concept for holding state. Flux is a huge topic that goes well beyond the scope of this book. Just know this: a store is something that holds application state and, when it's changed, React components are notified. Let's implement a basic store before we do anything else:

```
import EventEmitter from 'events';
import { fromJS } from 'immutable';

// A store is a simple state container that
// emits change events when the state is updated.
class Store extends EventEmitter {
  // If "window" is defined,
  // it means we're on the client and that we can
  // expect "INITIAL_STATE" to be there. Otherwise,
  // we're on the server and we need to set the initial
  // state that's sent to the client.
```

```
data = fromJS(
  typeof window !== 'undefined' ?
    window.INITIAL_STATE :
    { firstContent: { items: [] } }
)

// Getter for "Immutable.js" state data...
get state() {
  return this.data;
}

// Setter for "Immutable.js" state data...
set state(data) {
  this.data = data;
  this.emit('change', data);
}
}

export default new Store();
```

When the state changes, a `change` event is emitted. The initial state of the store is set, based on the environment we're in. If we're on the client, we're looking for an `INITIAL_STATE` object. This is the initial state that is sent from the server, so this store will be used in both the browser and in Node.js.

Now, let's take a look at one of the components that needs API data in order to render. It's going to use the store to coordinate its backend rendering with its frontend rendering:

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

import store from '../store';
import FirstContent from './FirstContent';

class FirstContentContainer extends Component {
  // Static method that fetches data from an API
  // endpoint for instances of this component.
  static fetchData = () =>
    new Promise(
      resolve =>
        setTimeout(() => {
          resolve(['One', 'Two', 'Three']);
        }, 1000)
    ).then((result) => {
      // We have to make sure that the data is set properly
      // in the store before returning the promise.
      store.state = store.state
        .updateIn(
```

```
        ['firstContent', 'items'],
        items => items
          .clear()
          .push(...result)
      );

      return result;
    });
}

// The default state of this component comes
// from the "store".
state = {
  data: store.state.get('firstContent'),
}

// Getter for "Immutable.js" state data...
get data() {
  return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}

componentDidMount() {
  // When the component mounts, we want to listen
  // changes in store state and re-render when
  // they happen.
  store.on('change', () => {
    this.data = store.state.get('firstContent');
  });
}

const items = this.data.get('items');

// If the state hasn't been fetched yet, fetch it.
if (items.isEmpty()) {
  FirstContentContainer.fetchData();
}
}

render() {
  return (
    <FirstContent {...this.data.toJS()} />
  );
}
}
```

```
export default FirstContentContainer;
```

As you can see, the initial state of the component comes from the store. The `FirstContent` component is then able to render its list, even though it's empty at first. When the component is mounted, it sets up a listener for the store. When the store changes state, it causes this component to re-render because it's calling `setState()`.

There's also a `fetchData()` static method defined on this component, and this declares the API dependencies this component has. Its job is to return a promise that's resolved when the API call returns and the store state has been updated. The `fetchData()` method is used by this component when it's mounted in the DOM, if there's no data yet. Otherwise, it means that the server used this method to fetch the state before it was rendered. Let's turn our attention to the server now to see how this is done.

First, we have a helper function that fetches the component data we need for a given request:

```
// Given a list of components returned from react-router
// "match()", find their data dependencies and return a
// promise that's resolved when all component data has
// been fetched.
const fetchComponentData = (components) =>
  Promise.all(
    components
      .reduce((result, i) => {
        // If the component is an object, it's
        // the "components" property of a route. In this
        // example, it's the "header" and "content"
        // components. So, we need to iterate over the
        // the object values to see if any of the components
        // has a "fetchData()" method.
        if (typeof i === 'object') {
          for (const k of Object.keys(i)) {
            if (i[k].hasOwnProperty('fetchData')) {
              result.push(i[k]);
            }
          }
        }
        // Otherwise, we assume that the item is a component,
        // and simply check if it has a "fetchData()" method.
      } else if (i && i.fetchData) {
        result.push(i);
      }
      return result;
    }, []
  )
  // Call "fetchData()" on all the components, mapping
  // the promises to "Promise.all()".
```

```
.map(i => i.fetchData())
);
```

The `components` argument comes from the `match()` call. These are all the components that need to be rendered, so this function iterates over them and checks if each one has a `fetchData()` method. If it does, then the promise that it returns is added to the result.

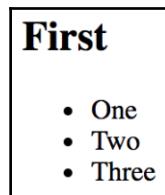
Now, let's take a look at the request handler that uses this function:

```
app.get('/*', (req, res) => {
  match({
    routes,
    location: req.url,
  }, (err, redirect, props) => {
    if (err) {
      res.status(500).send(err.message);
    } else if (redirect) {
      res.redirect(
        302,
        `${redirect.pathname}${redirect.search}`
      );
    } else if (props) {
      // If a route match is found, we pass
      // "props.components" to "fetchComponentData()".
      // Only when this resolves do we render the
      // components because we know the store has all
      // the necessary component data now.
      fetchComponentData(props.components).then(() => {
        const rendered = renderToString(
          <RouterContext {...props} />
        );

        res.send(doc(rendered, store.state.toJS()));
      });
    } else {
      res.status(404).send('Not Found');
    }
  });
});
```

This code is mostly the same as it has been throughout this chapter with one important change. It will now wait for `fetchComponentData()` to resolve before rendering anything. At this point, if there are any components with `fetchData()` methods, the store will be populated with their data.

For example, hitting the `/first` URL will cause Node.js to fetch data that the `FirstContentContainer` depends on, and set up the initial store state. Here's what this page looks like:



The only thing left to do is to make sure that this initial store state is serialized and passed to the browser somehow.

```
// In addition to the rendered component "content",
// this function now accepts the initial "state".
// This is set on "window.INITIAL_STATE" so that
// React can determine when the first change after
// the initial render happens.
const doc = (content, state) =>
`<!doctype html>
<html>
  <head>
    <title>Fetching Data</title>
    <script>
      window.INITIAL_STATE = ${JSON.stringify(state)};
    </script>
    <script src="/static/main-bundle.js" defer></script>
  </head>
  <body>
    <div id="app">${content}</div>
  </body>
</html>
`;
```

As you can see, the `window.INITIAL_STATE` value is passed a serialized version of the store state. Then, the client will rebuild this state. This is how we're able to avoid so many network calls, because we already have what we need in the store.

If you were to open the `/second` URL, you'll see something that looks like this:



Clicking this link, unsurprisingly, takes you to the first page. This will result in a new network call (mocked in this example) because the components on that page haven't been loaded yet.

Summary

In this chapter, you learned that React can be rendered on the server, in addition to the client. There are a number of reasons for doing this, like sharing common code between the frontend and the backend. The main advantage to server-side rendering is the performance boost that you get on the initial page load. This translates to a better user experience and therefore a better product.

You then progressively improved on a server-side React application, starting with a single page render. Then you were introduced to routing, client-side reconciliation, and component data fetching to produce a complete backend rendering solution.

In the following chapter, you'll learn how to implement React Bootstrap components to implement a mobile-first design.

11

Mobile-First React Components

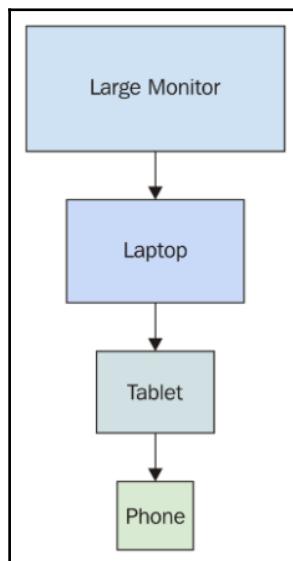
In this chapter, you'll be introduced to the `react-bootstrap` package. This tool provides mobile-first React components by leveraging the Bootstrap CSS framework. It's certainly not the only option available for doing mobile-first React, but it's a good choice, and it brings together two of the most popular technologies on the web.

We'll start things off with the motivation for adopting a mobile-first design strategy. We'll then spend the rest of this chapter implementing a few `react-bootstrap` components.

The rationale behind mobile-first design

Mobile-first design is a tactic that treats mobile devices as the primary target for user interfaces. Larger screens, such as laptops or big monitors, are secondary targets. This doesn't necessarily mean that the majority of users are accessing your app on their phones. It simply means that mobile is the starting point for scaling the user interface geometrically.

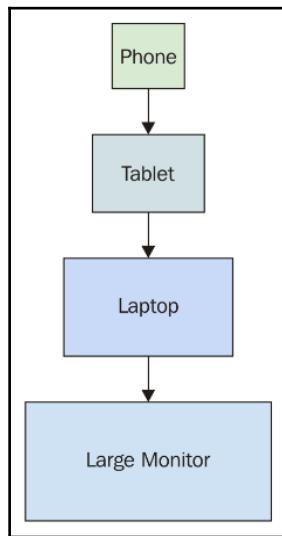
For example, when mobile browsers first came around it was customary to design the UI for normal desktop screens, and then to scale down to smaller screens when necessary. The approach is illustrated here:



The idea here is that you design the UI with larger screens in mind so that you can fit as much functionality onto the screen at once. When smaller devices are used, your code has to either use a different layout or different components on the fly.

This is very limiting for a number of reasons. First, it's very difficult to maintain code that has lots of special-case handling for different screen resolutions. Second, and the more compelling argument against this approach, is that it's next to impossible to provide a similar user experience across devices. If large screens have a ton of functionality displayed all at once, you simply cannot replicate this on smaller screens. Not only is there less real estate, but the processing power and network bandwidth of smaller devices are limiting factors as well.

The mobile-first approach to UI design tackles these issues by scaling the UI up, instead of trying to scale it down, as shown here:



This approach never used to make sense because you would be limiting the capabilities of your application; there weren't too many tablets or phones around. This is not the case today, where the expectation is that users can interact with applications on their mobile devices without issue. There are a lot more of them now, and mobile browsers are quite capable these days.

Once you've implemented your application functionality in a mobile context, scaling it up to larger screen sizes is a relatively easy problem to solve. Now let's take a look at how to approach mobile-first in React applications.

Using react-bootstrap components

While it's possible to implement mobile-first React user interfaces by rolling your own CSS, I would recommend against doing this. There are a number of CSS libraries that handle the seemingly endless edge cases for us. In this section, we'll introduce the `react-bootstrap` package—React components for Bootstrap.

Bootstrap is the most popular mobile-first library. However, using it directly means manually adding the right CSS classes to the right components. The `react-bootstrap` package exposes a number of components that serve as a thin abstraction layer between your application and Bootstrap HTML/CSS.

Let's implement some examples now. Another reason I'm showing you how to work with `react-bootstrap` components is that they're similar to `react-native` components, which you'll learn about starting in the next chapter.



The idea with the following examples isn't in-depth coverage of `react-bootstrap`, or Bootstrap itself for that matter. Rather, the idea is to give you a feel for what it's like to work with mobile-first components in React by passing them state from containers and so on. For now, take a look at the `react-bootstrap` documentation (<http://react-bootstrap.github.io/>) for specifics.

Implementing navigation

Perhaps the most important aspect of a mobile-first design is the navigation. It's especially difficult to get this right on mobile devices because there's barely enough room for feature content, let alone tools to move from feature to feature. Thankfully, Bootstrap handles much of the difficulty for us.

In this section, you'll learn how to implement two types of navigation. You'll start with toolbar navigation, and then you'll build a sidebar navigation section. This makes up part of the UI skeleton that you'll start with. What I find really useful about this approach is that, once the navigation mechanisms are in place, it's easy to add new pages and to move around in the app as I build it.

Let's start with the `Navbar`. This is a component found in most applications and is statically positioned at the top of the screen. Within this bar, we'll add some navigation links. Here's what the JSX for this looks like:

```
{ /* The "NavBar" is statically-placed across the
     top of every page. It contains things like the
     title of the application, and menu items. */ }
<Navbar className="navbar-top" fluid>
  <Navbar.Header>
    <Navbar.Brand>
      <Link to="/">Mobile-First React</Link>
    </Navbar.Brand>
  { /* The "<Navbar.Taggles>" component is used to
```

```
        replace any navigation links with a drop-down
        menu for smaller screens. */ }
<Navbar.Toggle />
</Navbar.Header>

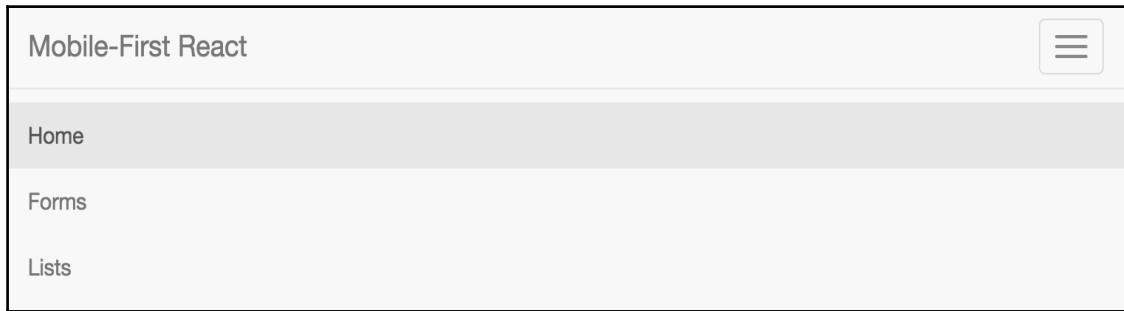
{ /* The actual menu with links to make. It's wrapped
   in the "<Navbar.Collapse>" component so that it
   works properly when the links have been
   collapsed. */
<Navbar.Collapse>
<Nav pullRight>
  <IndexLinkContainer to="/">
    <MenuItem>Home</MenuItem>
  </IndexLinkContainer>
  <LinkContainer to="forms">
    <MenuItem>Forms</MenuItem>
  </LinkContainer>
  <LinkContainer to="lists">
    <MenuItem>Lists</MenuItem>
  </LinkContainer>
</Nav>
</Navbar.Collapse>
</Navbar>
```

Here's what the navigation bar looks like:



The `<Navbar.Header>` component defines the title of the application and is placed to the left of the navigation bar. The links themselves are placed in the `<Nav>` element and the `pullRight` property aligns them to the right side of the navigation bar. You can see that, instead of using `<Link>` from the `react-router` package, we're using `<LinkContainer>` and `<IndexLinkContainer>`. These components come from the `react-router-bootstrap` package. They're necessary to make Bootstrap links work properly with the router.

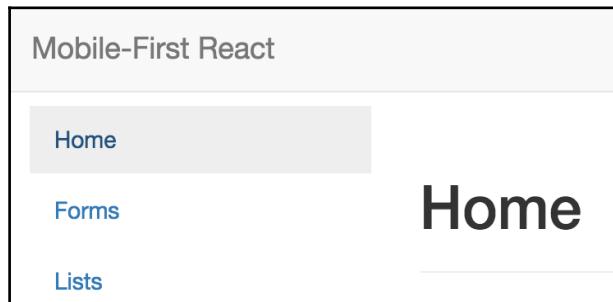
Something else worth noting is that the `<Nav>` element is wrapped in a `<Navbar.Collapse>` element and that the header contains a `<Navbar.Toggle>` button. These components are necessary to collapse the links into a drop-down menu for smaller screens. Since it's based on the browser width, you can just resize your browser window to see it in action:



The links that were displayed are now collapsed into a standard menu button. As you can see, when this button is clicked, the same links are displayed in a vertical fashion. This works much better on smaller devices. But with larger screens, having all navigation displayed in the top navigation bar might not be ideal. The standard approach is to implement a left-hand sidebar with navigation links stacked vertically. Let's implement this now:

```
{ /* This navigation menu has the same links
   as the top navbar. The difference is that
   this navigation is a sidebar. It's completely
   hidden on smaller screens. */
<Col sm={3} md={2} className="sidebar">
  <Nav stacked>
    <IndexLinkContainer to="/">
      <NavItem>Home</NavItem>
    </IndexLinkContainer>
    <LinkContainer to="forms">
      <NavItem>Forms</NavItem>
    </LinkContainer>
    <LinkContainer to="lists">
      <NavItem>Lists</NavItem>
    </LinkContainer>
  </Nav>
</Col>
```

Ignore the `<Col>` element for now. The only reason I'm including it here is that it's the container for the `<Nav>` and we've added our own class name to it. You'll see why in a moment. Inside the `<Nav>` element, things look exactly the same as they do in the navigation toolbar, with link containers and menu items. Here's what the sidebar looks like:



Now, the reason that we needed to add that custom `sidebar` class name to the containing element was so that we can hide it completely on smaller devices. Let's take a look at the simple CSS involved:

```
.sidebar {  
  display: none;  
}  
  
@media (min-width: 768px) {  
  .sidebar {  
    display: block;  
    position: fixed;  
    top: 60px;  
  }  
}
```

This CSS, along with the overall structure of this example, is adapted from this Bootstrap example: <http://getbootstrap.com/examples/dashboard/>. The idea behind this media query is simple. If the minimum browser width is 768px, then show the sidebar in a fixed position. Otherwise, hide it completely because we're on a smaller screen.

It's kind of cool, isn't it? Without much effort on our part, we have two navigation components collaborating with one another to change how they're displayed based on the screen resolution.

Lists

A common UI element in both mobile and desktop contexts is rendering lists of items. This is easy enough to do without the support of a CSS library, but libraries help keep the look and feel consistent. Let's implement a list that's controlled by a set of filters. First, we have the component that renders the react-bootstrap components:

```
import React, { PropTypes } from 'react';
import { Map as ImmutableMap } from 'immutable';
import {
  Button,
  ButtonGroup,
  ListGroupItem,
  ListGroup,
  Glyphicon,
} from 'react-bootstrap';

import './FilteredList.css';

// Utility function to get the bootstrap style
// for an item, based on the "done" value.
const itemStyle = done =>
  ImmutableMap()
    .set(true, { bsStyle: 'success' })
    .set(false, {})
    .get(done);

// Utility component for rendering a bootstrap
// icon based on the value of "done".
const ItemIcon = ({ done }) =>
  ImmutableMap()
    .set(true, (
      <Glyphicon
        glyph="ok"
        className="item-done"
      />
    ))
    .set(false, null)
    .get(done);

// Renders a list of items, and a set of filter
// controls to change what's displayed in the
// list.
const FilteredList = props => (
  <section>
    { /* Three buttons that control what's displayed
      in the list below. Clicking one of these
```

```
        buttons will toggle the state of the others. */ }
<ButtonGroup className="filters">
  <Button
    active={props.todoFilter}
    onClick={props.todoClick}
  >
    Todo
  </Button>
  <Button
    active={props.doneFilter}
    onClick={props.doneClick}
  >
    Done
  </Button>
  <Button
    active={props.allFilter}
    onClick={props.allClick}
  >
    All
  </Button>
</ButtonGroup>

{ /* Renders the list of items. It passes the
   "props.filter()" function to "items.filter()".
   When the buttons above are clicked, the "filter"
   function is changed. */
<ListGroup>
  {props.items.filter(props.filter).map(i => (
    <ListGroupItem
      key={i.name}
      onClick={props.itemClick(i)}
      href="#"
      {...itemStyle(i.done)}
    >
      {i.name}
      <ItemIcon done={i.done} />
    </ListGroupItem>
  ))}
</ListGroup>
</section>
);

FilteredList.propTypes = {
  todoFilter: PropTypes.bool.isRequired,
  doneFilter: PropTypes.bool.isRequired,
  allFilter: PropTypes.bool.isRequired,
  todoClick: PropTypes.func.isRequired,
  doneClick: PropTypes.func.isRequired,
```

```
    allClick: PropTypes.func.isRequired,
    itemClick: PropTypes.func.isRequired,
    filter: PropTypes.func.isRequired,
    items: PropTypes.array.isRequired,
};

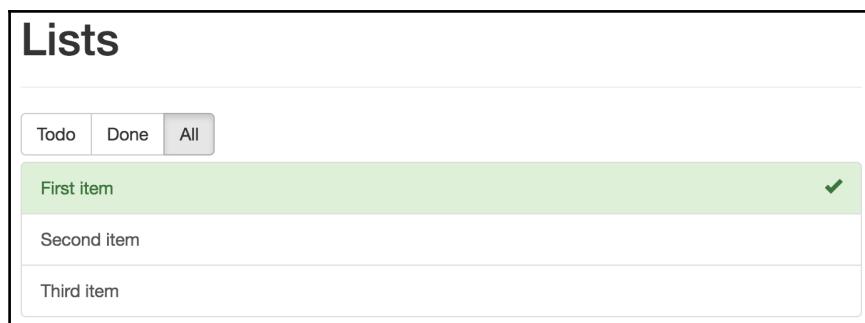
export default FilteredList;
```

First, we have the `<ButtonGroup>` and the `<Button>` elements within. These are the filters that the user can apply to the list. By default, only todo items are displayed. But, they can choose to filter by done items, or to show all items.

The list itself is a `<ListGroup>` element with `<ListGroupItem>` elements as children. The item renders differently, depending on the `done` state of the item. The end result looks like this:



You can toggle the done state of a list item simply by clicking on it. What's nice about the way this component works is that, if you're viewing todo items and mark one as done, it's taken off the list because it no longer meets the current filter criteria. The filter is re-evaluated because the component is re-rendered. Here's what an item that's marked as done looks like:



Now let's take a look at the container component that handles the state of the filter buttons and the item list:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import FilteredList from './FilteredList';

class FilteredListContainer extends Component {
  // Controls the state of the the filter buttons
  // as well as the state of the function that
  // filters the item list.
  state = {
    data: fromJS({
      // The items...
      items: [
        { name: 'First item', done: false },
        { name: 'Second item', done: false },
        { name: 'Third item', done: false },
      ],
      // The filter button states...
      todoFilter: true,
      doneFilter: false,
      allFilter: false,
      // The default filter...
      filter: i => !i.done,
      // The "todo" filter button was clicked.
      todoClick: () => {
        this.data = this.data.merge({
          todoFilter: true,
          doneFilter: false,
          allFilter: false,
          filter: i => !i.done,
        });
      },
      // The "done" filter button was clicked.
      doneClick: () => {
        this.data = this.data.merge({
          todoFilter: false,
          doneFilter: true,
          allFilter: false,
          filter: i => i.done,
        });
      },
    }),
  }
}
```

```
// The "all" filter button was clicked.
allClick: () => {
  this.data = this.data.merge({
    todoFilter: false,
    doneFilter: false,
    allFilter: true,
    filter: () => true,
  });
}

// When the item is clicked, toggle it's
// "done" state.
itemClick: item => (e) => {
  e.preventDefault();

  this.data = this.data.update(
    'items',
    items => items.update(
      items.findIndex(i =>
        i.get('name') === item.name),
      i => i.update('done', done => !done)
    )
  );
},
);

// Getter for "Immutable.js" state data...
get data() {
  return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}

render() {
  return (
    <FilteredList {...this.state.data.toJS()} />
  );
}
}

export default FilteredListContainer;
```

This code looks more complicated than it actually is. It's just four pieces of state and four event handler functions. Three pieces of state do nothing more than track which filter button is selected. The `filter` state is the callback function that's used by `<FilteredList>` to filter the items. The tactic used here is to pass a different filter function to the child view, based on the filter selection.

Forms

In this final section of the chapter, we'll implement a few form components from `react-bootstrap`. Just like the filter buttons you created in the preceding section, form components have state that needs to be passed down from a container component.

However, even simple form controls have many moving parts. We'll start by looking at text inputs. There's the input itself, but there's also the label, the placeholder, the error text, the validation function, and so on. To help glue all these pieces together, let's create a generic component that encapsulates all of the Bootstrap parts:

```
import React, { PropTypes } from 'react';
import {
  FormGroup,
  FormControl,
  ControlLabel,
  HelpBlock,
} from 'react-bootstrap';

// A generic input element that encapsulates several
// of the react-bootstrap components that are necessary
// for event simple scenarios.
const Input = ({
  type,
  label,
  value,
  placeholder,
  onChange,
  validationState,
  validationText,
}) => (
  <FormGroup validationState={validationState}>
    <ControlLabel>{label}</ControlLabel>
    <FormControl
      type={type}
      value={value}
      placeholder={placeholder}
      onChange={onChange}
    />
)
```

```
<FormControl.Feedback />
<HelpBlock>{validationText}</HelpBlock>
</FormGroup>
);

Input.propTypes = {
  type: PropTypes.string.isRequired,
  label: PropTypes.string,
  value: PropTypes.any,
  placeholder: PropTypes.string,
  onChange: PropTypes.func,
  validationState: PropTypes.oneOf([
    undefined,
    'success',
    'warning',
    'error',
  ]),
  validationText: PropTypes.string,
};

export default Input;
```

There are two key advantages to this approach. One is that, instead of having to use `<FormGroup>`, `<FormControl>`, `<HelpBlock>`, and so on, we just need our `<Input>` element. Another advantage is that only the `type` property is required, meaning that `<Input>` can be used for simple and complex controls.

Let's see this component in action now:

```
import React, { PropTypes } from 'react';
import { Panel } from 'react-bootstrap';

import Input from './Input';

const InputsForm = (props) => (
  <Panel header={(<h3>Inputs</h3>) }>
    <form>
      { /* Uses the <Input> element to render
          a simple name field. There's a lot of
          properties passed here, many of them
          come from the container component. */ }
      <Input
        type="text"
        label="Name"
        placeholder="First and last..."
        value={props.nameValue}
        onChange={props.nameChange}
        validationState={props.nameValidationState}
```

```
        validationText={props.nameValidationText}
      />

      { /* Uses the "<Input>" element to render a
         password input. */ }
      <Input
        type="password"
        label="Password"
        value={props.passwordValue}
        onChange={props.passwordChange}
      />
    </form>
  </Panel>
);

InputsForm.propTypes = {
  nameValue: PropTypes.any,
  nameChange: PropTypes.func,
  nameValidationState: PropTypes.oneOf([
    undefined,
    'success',
    'warning',
    'error',
  ]),
  nameValidationText: PropTypes.string,
  passwordValue: PropTypes.any,
  passwordChange: PropTypes.func,
};

export default InputsForm;
```

As you can see, there's only one component used to create all of the necessary Bootstrap pieces underneath. Everything is passed in through a property. Here's what this form looks like:



Now let's look at the container component that controls the state of these inputs:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import InputsForm from './InputsForm';

// Validates the given "name". It should have a space,
// and it should have more than 3 characters. There are
// many scenarios not accounted for here, but are easy
// to add.
function validateName(name) {
  if (name.search(/ /) === -1) {
    return 'First and last name, separated with a space';
  } else if (name.length < 4) {
    return 'Less than 4 characters? Srsly?';
  }

  return null;
}

class InputsFormContainer extends Component {
  state = {
    data: fromJS({
      // "Name" value and change handler.
      nameValue: '',
      // When the name changes, we use "validateName()"
      // to set "nameValidationState" and
      // "nameValidationText".
      nameChange: (e) => {
        this.data = this.data.merge({
          nameValue: e.target.value,
          nameValidationState:
            validateName(e.target.value) === null ?
              'success' : 'error',
          nameValidationText: validateName(e.target.value),
        });
      },
      // "Password" value and change handler.
      passwordValue: '',
      passwordChange: (e) => {
        this.data = this.data.set(
          'passwordValue', e.target.value
        );
      },
    }),
  }
}
```

```
// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

render() {
    return (
        <InputsForm {...this.state.toJS()} />
    );
}
}

export default InputsFormContainer;
```

The event handlers for the inputs are part of the state and get passed to `InputsForm` as properties. Now let's take a look at some checkboxes and radio buttons. We'll use the `<Radio>` and the `<Checkbox>` react-bootstrap components:

```
import React, { PropTypes } from 'react';
import {
    Panel,
    Radio,
    Checkbox,
    FormGroup,
} from 'react-bootstrap';

const RadioForm = (props) => (
    <Panel header={(<h3>Radios & Checkboxes</h3>)}>
        { /* Renders a group of related radio buttons. Note
            that each radio needs to have the same "name"
            property, otherwise, the user will be able to
            select multiple radios in the same group. The
            "checked", "disabled", and "onChange" properties
            all come from the container component. */}
        <FormGroup>
            <Radio
                name="radio"
                onChange={props.checkboxEnabledChange}
                checked={props.checkboxEnabled}
                disabled={!props.radiosEnabled}
            >
                Checkbox enabled
            </Radio>
        </FormGroup>
    </Panel>
)
```

```
<Radio
  name="radio"
  onChange={props.checkboxDisabledChange}
  checked={!props.checkboxEnabled}
  disabled={!props.radiosEnabled}
>
  Checkbox disabled
</Radio>
</FormGroup>

{ /* Renders a checkbox and uses the same approach
   as the radios above: setting it's properties from
   state that's passed in from the container. */}

<FormGroup>
  <Checkbox
    onChange={props.checkboxChange}
    checked={props.radiosEnabled}
    disabled={!props.checkboxEnabled}
  >
    Radios enabled
  </Checkbox>
</FormGroup>
</Panel>
);

RadioForm.propTypes = {
  checkboxEnabled: PropTypes.bool.isRequired,
  radiosEnabled: PropTypes.bool.isRequired,
  checkboxEnabledChange: PropTypes.func.isRequired,
  checkboxDisabledChange: PropTypes.func.isRequired,
  checkboxChange: PropTypes.func.isRequired,
};

export default RadioForm;
```

The idea is that the radio buttons toggle the `enabled` state of the checkbox and the checkbox toggles the `enabled` state of the radios. Note that, although the two `<Radio>` elements are in the same `<FormGroup>`, they need to have the same `name` property value. Otherwise, you'll be able to select both radios at the same time. Here's what this form looks like:

Radios & Checkboxes

- Checkbox enabled
- Checkbox disabled
- Radios enabled

Finally, let's look at the container component that handles the state of the radios and the checkbox:

```
import React, { Component } from 'react';
import { fromJS } from 'immutable';

import RadioForm from './RadioForm';

class RadioFormContainer extends Component {
    // Controls the enabled state of a group of
    // radio buttons and a checkbox. The radios
    // toggle the state of the checkbox while the
    // checkbox toggles the state of the radios.
    state = {
        data: fromJS({
            checkboxEnabled: false,
            radiosEnabled: true,
            checkboxEnabledChange: () => {
                this.data = this.data.set(
                    'checkboxEnabled', true
                );
            },
            checkboxDisabledChange: () => {
                this.data = this.data.set(
                    'checkboxEnabled', false
                );
            },
            checkboxChange: () => {
                this.data = this.data.update(
                    'radiosEnabled',
                    enabled => !enabled
                );
            },
        }),
    }
}

// Getter for "Immutable.js" state data...
```

```
get data() {
    return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

render() {
    return (
        <RadioForm {...this.data.toJS()} />
    );
}
}

export default RadioFormContainer;
```

Summary

This chapter introduced you to the concept of mobile-first design. We did a brief overview of why we might want to use the mobile-first strategy. It boils down to the fact that scaling mobile designs up to larger devices is much easier than scaling in the opposite direction.

Next, we discussed what this means in the context of a React application. In particular, we want to use a framework such as Bootstrap that handles the scaling details for us. We then implemented several examples using components from the `react-bootstrap` package.

This concludes the first part of this book. You're now ready to tackle React projects that live on the web, including mobile browsers! Mobile browsers are getting better, but they're no match for the native capabilities of mobile platforms. Part 2 of this book teaches you how to use React Native.

12

Why React Native?

Facebook created React Native to build its mobile applications. The motivation to do so came from the fact that React for the Web was so successful. If React is such a good tool for UI development, and you need a native application, then why fight it? Just make React work with native mobile OS UI elements!

In this chapter, you'll be introduced to React Native and the motivations for using it to build native mobile web applications.

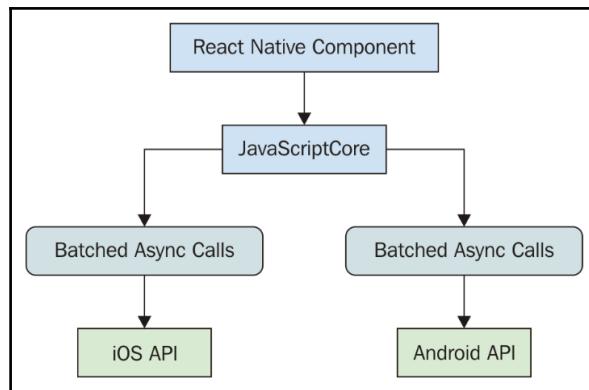
What is React Native?

Earlier in this book, I introduced the notion of a render target—the thing that React components render to. The render target is abstract as far as the React programmer is concerned. For example, in React, the render target can be a string or it could be the DOM. This is why your components never directly interface with the render target, because you can never assume that it is what you think it is.

A mobile platform has UI widget libraries that developers can leverage to build apps for that platform. On Android, developers implement Java apps while on iOS developers implement Swift apps. If you want a functional mobile app, you're going to have to pick one. However, you'll need to learn both languages, as supporting only one of two major platforms isn't realistic for success.

For React developers, this isn't a problem. The same React components that you build work all over the place, even on mobile browsers! Having to learn two more programming languages to build and ship a mobile application is cost and time prohibitive. The simple solution to this is to introduce a new React that supports a new render target—native mobile UI widgets.

React Native uses a technique that makes asynchronous calls to the underlying mobile OS, which calls the native widget APIs. There's a JavaScript engine, and the React API is mostly the same as React for the Web. The difference is mostly with the target; instead of a DOM, there are asynchronous API calls. The concept is visualized here:



This obviously oversimplifies everything that's happening under the covers, but the basic ideas are as follows:

- The same React library that's used on the Web is used by React Native and runs in JavaScriptCore
- Messages that are sent to native platform APIs are asynchronous and batched for performance purposes
- React Native ships with components implemented for mobile platforms, instead of components that are HTML elements



Much more on the history and the mechanics of React Native can be found here at <https://code.facebook.com/posts/1014532261909640>.

React and JSX are familiar

Implementing a new render target for React isn't straightforward. It's essentially the same thing as inventing a new DOM that runs on iOS and Android. So why go through all the trouble?

First, there's a huge demand for mobile apps in general. The reason is that the mobile web browser user experience isn't as good. I'll elaborate on this in the next section. Second, JSX is simply awesome for building user interfaces. Rather than having to learn a new technology, it's much easier to use what you know.

It's the latter point that's the most relevant to you, the reader. If you're reading this book, you're probably interested in using React for both web applications, and native mobile applications. I simply can't put into words how valuable React is from a development resource perspective. Instead of having a team that does web UIs, a team that does iOS, a team that does Android, and so on, there's just the UI team that understands React. Easy peasy!

The mobile browser experience

Mobile browsers lack many capabilities of mobile applications. This is due to the fact that browsers cannot replicate the same native platform widgets as HTML elements. We can try to do this, but it's often better to just use the native widget, rather than try to replicate it. Partly because this requires less maintenance effort on our part, and partly because using widgets that are native to the platform means that they're consistent with the rest of the platform. For example, if a datepicker in your application looks different from all the datepickers the user interacts with on their phone, this isn't a good thing. Familiarity is key and using native platform widgets makes familiarity possible.

User interactions on mobile devices are fundamentally different from the interactions that we typically design for on the Web. Web applications assume the presence of a mouse, for example, and that the click event on a button is just one phase. But, things become more complicated when the user uses their fingers to interact with the screen. Mobile platforms have what's called a gesture system to deal with these. React Native is a much better candidate for handling gestures than React for the web, because it handles these types of things that you don't have to think about much in a web app.

As the mobile platform is updated, you want the components of your app to stay updated too. This isn't a problem with React Native because they're using actual components from the platform. Once again, consistency and familiarity are important for a good user experience. So, when the buttons in your app look and behave in exactly same way as the buttons in every other app on the device, your app feels like part of the device.

Android and iOS, different yet the same

When I first heard about React Native, I automatically thought that it would be some cross-platform solution that lets you write a single React application that will run natively on any device. Do yourself a favor and get out of this mindset before you start working with React Native. iOS and Android are different on many fundamental levels. Even their user experience philosophies are different, so trying to write a single app that runs on both platforms is categorically misguided.

Besides, this is not the goal of React Native. The goal is *React components everywhere*, not write once run anywhere. In some cases, you'll want your app to take advantage of an iOS-specific widget or an Android-specific widget. This provides a better user experience for that particular platform and should trump the portability of a component library.



In later chapters, we'll look at different tactics for organizing platform-specific modules.



Having said that, there are several areas that overlap between iOS and Android where the differences are trivial. In other words, the two widgets aim to accomplish the same thing for the user, in roughly the same way. In these cases, React Native will handle the difference for you and provide a unified component.

At the time of writing, Windows support is rumored to be coming to React Native. If that happens between now and publication, I'm okay with it because iOS and Android still dominate the mobile market.

The case for mobile web apps

We spent the previous chapter implementing mobile-first React components. Should you simply forget all of that, now that you know about React Native? No. Not every one of your users is going to be willing to install an app, especially not if you don't yet have a high download count and rating. The barrier to entry is much lower with web applications—the user only needs a browser.

Despite not being able to replicate everything that native platform UIs have to offer, you can still implement awesome things in a mobile web UI. Maybe having a good web UI is the first step toward getting those download counts and ratings up for your mobile app.

Ideally, what you should aim for is the following:

- Standard web (laptop/desktop browsers)
- Mobile web (phone/tablet browsers)
- Mobile apps (phone/tablet native platform)

Putting an equal amount of effort into all three of these spaces probably doesn't make much sense, as your users probably favor one area over another. Once you know, for example, that there's a really high demand for your mobile app compared to the web versions, that's when you allocate more effort there.

Summary

In this chapter, you learned that React Native is an effort by Facebook to reuse React to create native mobile applications. React and JSX are really good at declaring UI components, and since there's now a huge demand for mobile applications, it makes sense to use what you already know for the Web.

The reason there's such a demand for mobile applications over mobile browsers is that they just feel better. Web applications lack the ability to handle mobile gestures the same way apps can, and they generally don't feel like part of the mobile experience from a look and feel perspective.

React Native isn't trying to implement a component library that lets you build a single React app that runs on any mobile platform. iOS and Android are fundamentally different in many important ways. Where there's overlap, React Native does try to implement common components. Will we do away with mobile web apps now that we can build natively using React? This will probably never happen, because the user can only install so many apps.

Now that you know what React Native is and why it's awesome, you'll learn how to get started with new React Native projects in the following chapter.

13

Kickstarting React Native Projects

In this chapter, you'll get up-and-running with React Native. Thankfully, much of the boilerplate stuff regarding the creation of a new project is handled for us by React Native command line tools. I'll explain what's actually created for you when you initialize an empty project. Then, I'll show you how to run the project on iOS and on Android simulators.

Using the React Native command-line tool

In this section, we're going to install the `react-native-cli` tool and use it to create an empty React Native project. Well, the created project isn't completely empty: it has a basic screen implemented so you can verify that everything is functioning correctly. Technically, you don't need `react-native-cli` to run a React Native project, but I would strongly urge you to use it.

So, without further ado, open up a command line terminal and run the following:

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

This will install the `react-native` command onto your system. This command is what's used to kickstart a new project. Change into a directory where you want to keep your React Native projects and run the following (the code bundle for this book contains the files for this project already, in case you feel like skipping this step):

```
react-native init MyProject
```

This will create a new `MyProject` directory with several files and directories within. Let's take a look at what gets added. First we'll walk through the files at the root of `MyProject`:

- `package.json`: The typical npm package definition. It lists `react` and `react-native` as dependencies
- `index.android.js`: The entry point used by `react-native` when building Android apps
- `index.ios.js`: The entry point used by `react-native` when building iOS apps

There are several other configuration files here, but we won't worry about these because they're related to the build and probably won't need to be modified by us. Now let's look at the directories created at the root of `MyProject`:

- `android`: The project files used to build Android apps
- `ios`: The project files used to build iOS apps
- `node_modules`: The `react-native` package and all its dependencies

The `node_modules` directory isn't very interesting because you've probably seen it several times already, in almost any other JavaScript project. However, the `android` and the `ios` directories form the core of your respective Android and iOS React Native apps.

If you dig around in either of these directories, you'll discover that they look like your typical Android or iOS project, only there isn't much there. The reason is that the majority of the code for these apps is implemented in React Components. The main job of this code is to set up a shell application in which it can embed the JavaScriptCore runtime and the mechanism used to script the native platform components.

For example, here's the main application file for Android:

```
package com.myproject;

import android.app.Application;
import android.util.Log;

import com.facebook.react.ReactApplication;
import com.facebook.react.ReactInstanceManager;
import com.facebook.react.ReactNativeHost;
import com.facebook.react.ReactPackage;
import com.facebook.react.shell.MainReactPackage;

import java.util.Arrays;
import java.util.List;

public class MainApplication extends Application implements
```

```
ReactApplication {

    private final ReactNativeHost mReactNativeHost = new
    ReactNativeHost(this) {
        @Override
        protected boolean getUseDeveloperSupport() {
            return BuildConfig.DEBUG;
        }

        @Override
        protected List<ReactPackage> getPackages() {
            return Arrays.<ReactPackage>asList(
                new MainReactPackage()
            );
        }
    };

    @Override
    public ReactNativeHost getReactNativeHost() {
        return mReactNativeHost;
    }
}
```

This code uses Java modules from Facebook to pull in our React Native components. As our components change, this source doesn't change at all. The React Native components that you implement pass messages to the platform to instrument the native components.

Now let's take a look at the contents of `index.android.js`:

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 * @flow
 */

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

class MyProject extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
```

```
        Welcome to React Native!
    </Text>
    <Text style={styles.instructions}>
        To get started, edit index.android.js
    </Text>
    <Text style={styles.instructions}>
        Shake or press menu button for dev menu
    </Text>
</View>
);
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});
AppRegistry.registerComponent(
  'MyProject',
  () => MyProject
);
```

There are several React Native components used here: don't concern yourself with the specifics of how they work just yet. This file was created for us as part of the project initialization process. It's created as a jumping off point so that you can run the project and make sure everything is set up correctly.

Now, if you open the `index.ios.js` file, it looks nearly identical to `index.android.js`. The only difference is with the text content; the actual structure of the app is exactly the same. This is because the components used by the app all work the same on both iOS and Android. When the app builds, React Native knows which one of these files to use based on the build target.



As I was implementing this example, I saw strange JavaScript errors. It was a clean project, and I hadn't even touched the generated example yet. I couldn't for the life of me figure out what was happening. After Googling for some time, I realized that the `.babelrc` file in a parent directory was being used and it had settings that are incompatible with React Native. So, I added the new `MyProject/.babelrc` file with settings that worked. The things that developers have to spend time on.

iOS and Android simulators

The majority of your time developing iOS and Android applications using React Native will be spent in a simulator. A simulator is a virtual machine that runs the mobile OS that you want to test your app in, on your desktop. If you had to validate every change on the actual hardware that you want to target, it would be both very time-consuming and very costly to implement. In this section, we'll get you set up with simulators for iOS and Android.

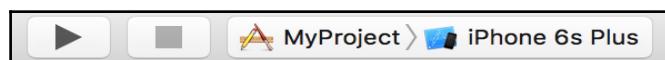
Xcode

If you don't already have Xcode installed, you can download and install it for free from the app store or <https://developer.apple.com/>. You'll need to do this before you're able to simulate any iOS devices that can run your React Native app.



You'll need a Mac and OSX if you're going to build iOS apps. This includes React Native. There's simply no way around this. Well played Apple.

Once you have Xcode installed, you can open the project by double-clicking on the `MyProject/ios/MyProject.xcodeproj` file. This will open Xcode with your project. The play button at the top of the screen will build and run the project with the selected simulator:



If you click on the name of the device, in this case the default is **iPhone 6s Plus**, you can see a list of devices that you can simulate in order to run your React Native app:



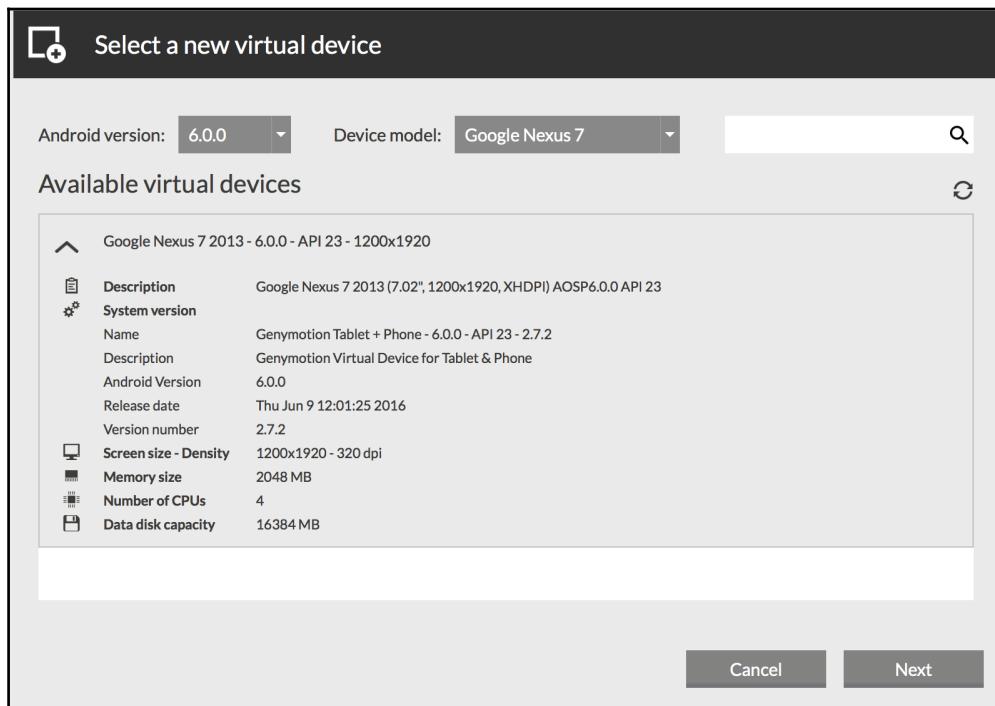
Genymotion

Genymotion is a development tool to help build Android apps. The reason I'd recommend using this tool over the standard Android development tools is that it just works much better. Using Genymotion actually makes React Native development feel a lot more like it does with Xcode on OSX.

Also, Genymotion uses VirtualBox to run the Android OS, which is partly why it's easy to use Genymotion for Android development on any platform. Although this is licensed software, you can try it out for free, and the pricing is very affordable once you have a production app to maintain.

The first step is to install it; instructions can be found here:

<https://www.genymotion.com/thank-you-freemium/>. Once Genymotion is installed, you should create an account, which will allow you to access their device repositories. Then, when you start up Genymotion, you can create a new virtual Android device, which looks something along these lines:



Once you have all of the Android devices that you want to develop against, the only thing left to do is deploy your React Native app to the device. We'll cover this now.

Running the project

At this point in the chapter, you know how to kickstart a new React Native project. You also have your iOS and Android device simulators ready to go. In this final section, we'll walk through the process of building and deploying your project to a virtual device in development mode.

Running iOS apps

In the previous section, you opened up the iOS project for your React Native using the Xcode application itself. But it turns out that, most of the time, you won't actually need to do this. The simulator is a separate process, so we can run it using the React Native tools, from the `MyProject/` directory:

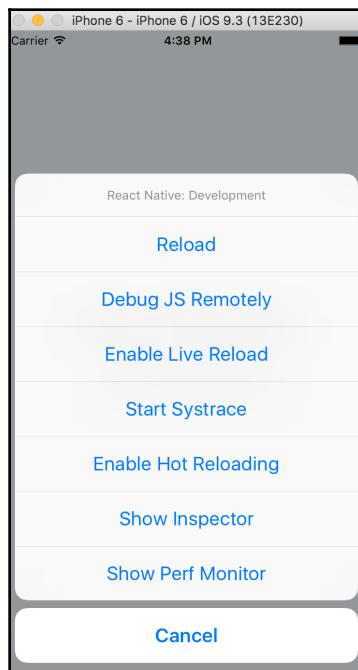
```
react-native run-ios
```

This will start up the React Native packager process. It builds the app and is used to start up the simulator and to communicate with it. Here's what the iOS simulator looks like with the default React Native project:



There you have it; you're up-and-running in a simulated iOS environment! But rather than restarting the simulator every time you want to test a code change, why not have the packager rebuild and update the app as changes are made? There are two ways to do this. We can enable live reloading or hot module reloading. Live reloading refreshes the entire application, while hot reloading keeps the application state while swapping in new component code.

If you're making frequent changes to a specific feature, then hot reloading is probably the way to go so that you don't have to keep navigating back to it. But hot reloading is more resource-intensive, so if it's a simple app you're working on, live reloading might be the better choice. Either of these options can be set in the developer menu. In the simulator UI, press Cmd+D:



Finally, if you want to run your iOS using a different device simulator, you can specify this as an argument to the `run-ios` command. First, you can issue the following command to get a list of available device names:

```
xcrun simctl list devices | grep unavailable -vi
```

We're using `grep` here to filter out unavailable device names. The output should look something like this:

```
== Devices ==
-- iOS 9.3 --
iPhone 4s (2E2C7E46-FB08-4464-BA9B-BA90CBDEE2D9) (Shutdown)
iPhone 5 (0028976C-7E4B-4764-AA9A-3E099E0A5DCF) (Shutdown)
iPhone 5s (221FABF3-0BDC-42E1-849E-0F4D8FECDCB1) (Shutdown)
iPhone 6 (14399036-4B2C-4BA0-90B6-90948BC1CA8D) (Shutdown)
iPhone 6 Plus (8BE16249-B9E1-4862-85E5-1AEAAADCF3008) (Shutdown)
iPhone 6s (DE4D9DA5-9FF9-48D2-A87E-58EC1BF73757) (Shutdown)
iPhone 6s Plus (0E37E6E8-4B00-404F-9E47-C9796963C138) (Shutdown)
iPad 2 (6F6FCDF37-AD12-430F-A06D-ACC34310A2D2) (Shutdown)
iPad Retina (855D92C5-4911-4392-B320-C99D3602C6CB) (Shutdown)
iPad Air (76D2B2FB-7D90-4A6E-8575-414B18600E96) (Shutdown)
iPad Air 2 (2C0CC11F-7F2F-4558-B5ED-124B84BFA6BB) (Shutdown)
iPad Pro (B2BFC081-0E89-46A0-92CA-B78E6BCD90F6) (Shutdown)
-- tvOS 9.2 --
Apple TV 1080p (59F20EC8-3FEF-42A7-916A-67665A5C8B24) (Shutdown)
-- watchOS 2.2 --
Apple Watch - 38mm (FCF7F7A2-6D12-4A2B-B4B5-FD08E01895F9)
(Shutdown)
Apple Watch - 42mm (6E7749AD-580C-431D-B920-27E44FF514D6)
(Shutdown)
```

Now, you can specify one of these device names as the simulator argument when running the React Native tool:

```
react-native run-ios --simulator='iPad 2'
```

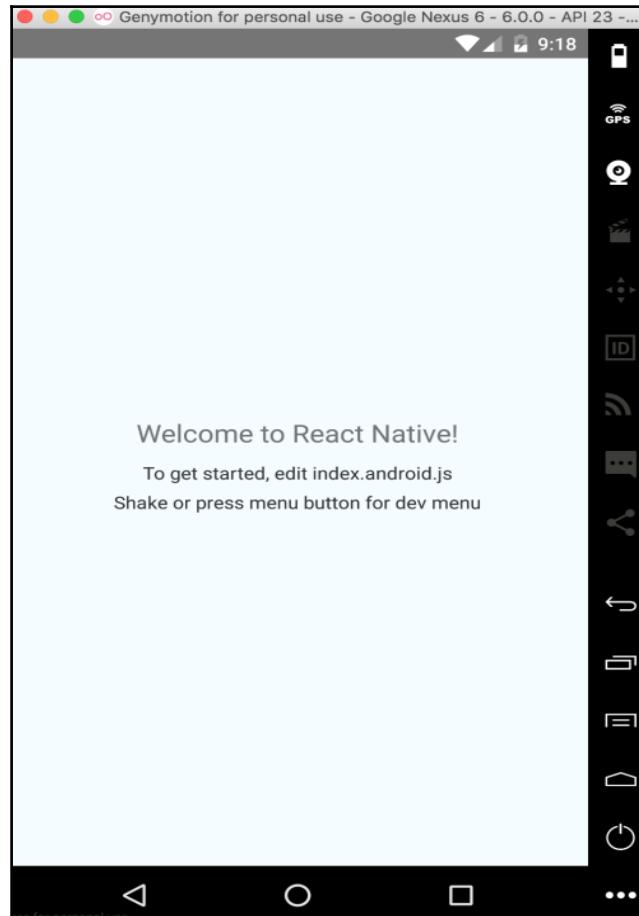
Running Android apps

Running your React Native app in a simulated Android environment works the same way as running it in a simulated iOS environment. The differences are in the tooling. Instead of an Xcode simulator, you use Genymotion to simulate a device. The main difference is that the simulated device needs to be up-and-running before you deploy your app via the React Native packager.

Otherwise, you'll see an error message that looks like this when you try running `react-native run-android`:

```
Make sure you have an Android emulator running or a device connected and
have set up your Android development environment
```

Easy enough; let's go ahead and fire up a simulated Android device from within Genymotion. In your list of devices, just select the device you want to use and click the start button. This will start the virtual Android environment. Now when you run `react-native run-android`, you should see the React Native application show up on the emulated screen:



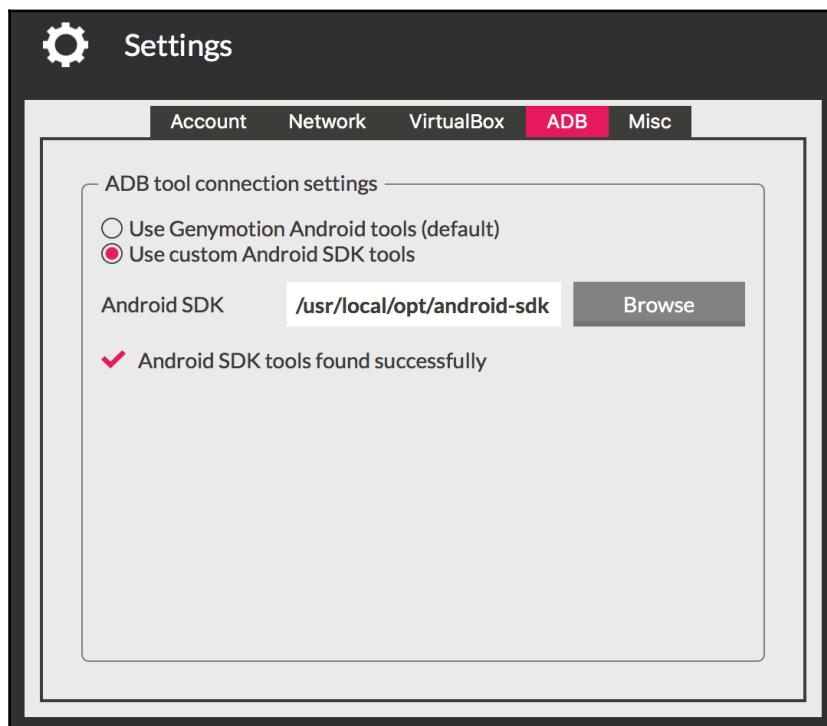
The same developer menu that we looked at in the preceding section is here as well; you just access it differently.

When implementing this example, I ran into some trouble trying to deploy my React Native app to my virtual device. I was seeing something like this:

```
E/adb: error: could not install *smartsocket* listener: Address already  
in use  
E/adb: ADB server didn't ACK  
E/adb: * failed to start daemon *  
E/ddms: '/usr/local/opt/android-sdk/platform-tools/adb,start-server'  
failed -- run manually if necessary  
E/adb: error: cannot connect to daemon
```

Genymotion uses its own ADB (Android Debug Bridge):

<https://developer.android.com/studio/command-line/adb.html>) to connect with the virtual Android environment. It turns out that I had already installed Android Studio, which meant I had two ADB instances running. This was easy enough to fix, I just had to tell Genymotion to use the existing ADB tool:



Summary

This chapter introduced you to the tools that ship with React Native to help you get started with new projects. We walked through creating an empty example project, and we walked through what's actually generated for us. Then, you learned about iOS and Android simulators, since they're where you'll spend the majority of your time developing React Native applications.

Lastly, you learned about the React Native tools that build and deploy your app into the simulator for you. You also learned that you can enable live reloading or hot reloading from within the app.

In the next chapter, you'll spend more time writing React Native application code by building flexible layouts.

14

Building Responsive Layouts with Flexbox

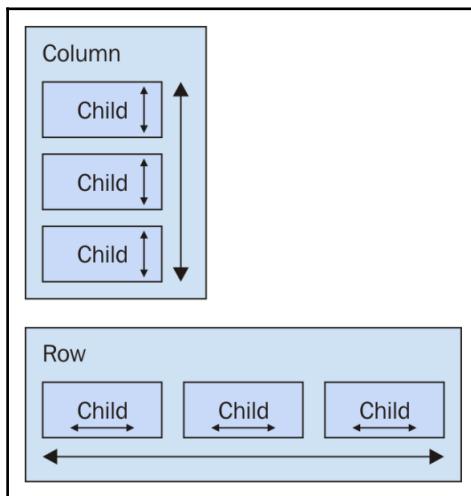
In this chapter, you'll get a feel for what it's like to lay out components on the screen of mobile devices. Thankfully, React Native polyfills many CSS properties that you've probably used in the past to implement page layouts in web applications. We'll use the flexbox model to layout our React Native screens.

Before we dive into implementing layouts, you'll get a brief primer on flexbox and using CSS style properties in React Native apps—it's not quite what you're used to with regular CSS stylesheets. With that out of the way, you'll implement several React Native layouts using flexbox.

Flexbox is the new layout standard

Before the flexible box layout model was introduced to CSS, the various approaches to building layouts felt hacky and were error prone. Flexbox fixes this by abstracting many of the properties that you would normally have to provide in order to make the layout work.

In essence, the flexbox is exactly what it sounds like—a box model that's flexible. That's the beauty of flexbox—it's simple. You have a box that acts as a container, and you have child elements within that box. Both the container and the child elements are flexible in how they're rendered on the screen, as illustrated here:



Flexbox containers have a direction, either column (up/down) or row (left/right). This actually confused me when I was first learning flexbox: my brain refused to believe that rows move from left to right. Rows stack on top of one another! The key thing to remember is that it's the direction that the box flexes, not the direction that boxes are placed on the screen.



For a more in-depth treatment of flexbox concepts, check out this page:
<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>.

Introducing React Native styles

It's time to implement your first React Native app, beyond the boilerplate that's generated by `react-native init`. I want to make sure that you feel comfortable using React Native stylesheets before you start implementing flexbox layouts in the next section. Here's what a React Native stylesheet looks like:

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

```
// Exports a "stylesheet" that can be used
// by React Native components. The structure is
// familiar for CSS authors.
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'ghostwhite',
  },
  box: {
    width: 100,
    height: 100,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: 'lightgray',
  },
  boxText: {
    color: 'darkslategray',
    fontWeight: 'bold',
  },
});
export default styles;
```

This is a JavaScript module, not a CSS module. If you want to declare React Native styles, you need to use plain objects. Then, you call `StyleSheet.create()` and export this from the style module.

As you can see, this stylesheet has three styles: `container`, `box`, and `boxText`. Let's see how these styles are imported and applied to React Native components:

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

// Imports the "styles" stylesheet from the
// "styles" module.
import styles from './styles';

// Renders a view with a square in the middle, and
// some text in the middle of that. The "style" property
// is passed a value from the "styles" stylesheet.
```

```
const Stylesheets = () => (
  <View style={styles.container}>
    <View style={styles.box}>
      <Text style={styles.boxText}>
        I'm in a box
      </Text>
    </View>
  </View>
);

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

As you can see, the styles are assigned to each component via the `style` property. We're trying to render a box with some text in the middle of the screen. Let's make sure that this looks as we expect:



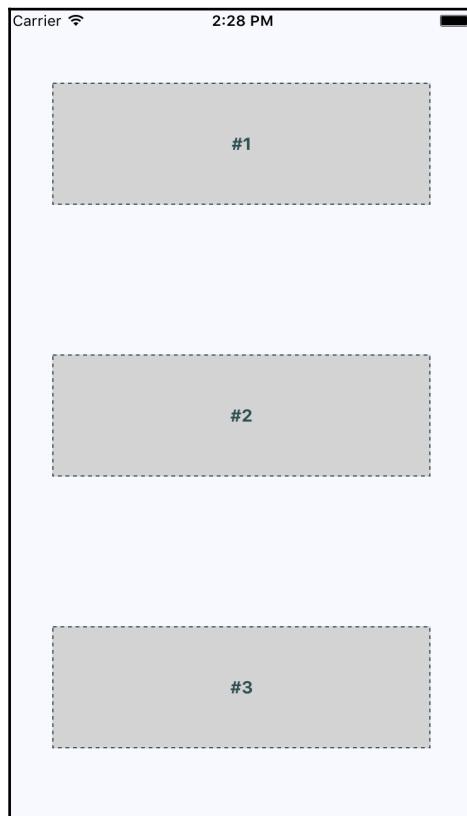
Perfect! Now that you have an idea of how to set styles on React Native elements, it's time to start creating some screen layouts.

Building flexbox layouts

In this section, we'll walk through several potential layouts that you can use in your React Native applications. I want to stay away from the idea that one layout is better than others. Instead, I'll show you how powerful the flexbox layout model is for mobile screens so that you can design the layout that best suits your application.

Simple three column layout

To start things off, let's implement a simple layout with three sections that flex in the direction of the column (top to bottom). Let's start by taking a look at the resulting screen:



The idea with this example is that we've styled and labeled the three screen sections so that they stand out. In other words, these components wouldn't necessarily have any styling in a real application since they're used to arrange other components on the screen.

With that said, let's take a look at the components used to create this screen layout:

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

import styles from './styles';

// Renders three "column" sections. The "container"
// view is styled so that it's children flow from
// the top of the screen, to the bottom of the screen.
const ThreeColumnLayout = () => (
  <View style={styles.container}>
    <View style={styles.box}>
      <Text style={styles.boxText}>
        #1
      </Text>
    </View>
    <View style={styles.box}>
      <Text style={styles.boxText}>
        #2
      </Text>
    </View>
    <View style={styles.box}>
      <Text style={styles.boxText}>
        #3
      </Text>
    </View>
  </View>
);

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

As you can see, the container view is the column and the child views are the rows. The `<Text>` component is used to label each row.



Maybe this example could have been called *three row layout*, since it has three rows. But at the same time, the three layout sections are flexing in the direction of the column that they're in. Use the naming convention that makes the most conceptual sense to you.

Now let's take a look at the styles used to create this layout:

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

// Exports a "stylesheet" that can be used
// by React Native components. The structure is
// familiar for CSS authors.
const styles = StyleSheet.create({

    // The "container" for the whole screen.
    container: {
        // Enables the flexbox layout model...
        flex: 1,
        // Tells the flexbox to render children from
        // top to bottom...
        flexDirection: 'column',
        // Aligns children to the center on the container...
        alignItems: 'center',
        // Defines the spacing relative to other children...
        justifyContent: 'space-around',
        backgroundColor: 'ghostwhite',
    },

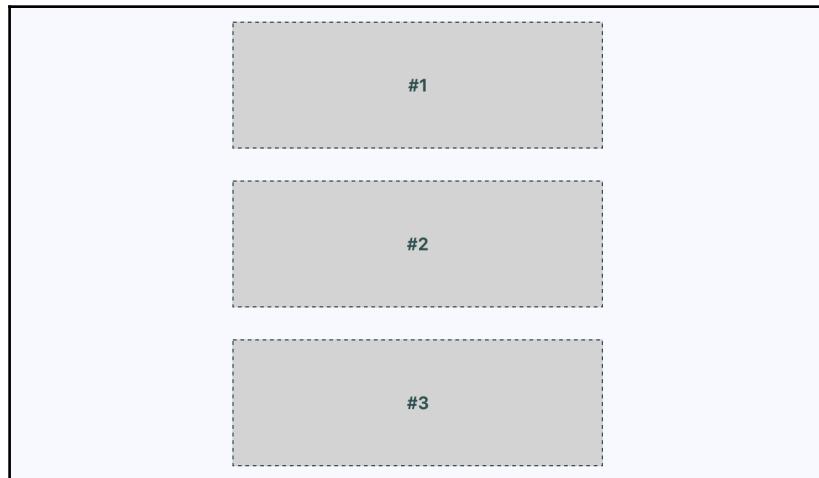
    box: {
        width: 300,
        height: 100,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: 'lightgray',
        borderWidth: 1,
        borderStyle: 'dashed',
        borderColor: 'darkslategray',
    },

    boxText: {
        color: 'darkslategray',
        fontWeight: 'bold',
    },
});

export default styles;
```

The `flex` and `flexDirection` properties of container are what enable the layout of the rows to flow from top to bottom. The `alignItems` and `justifyContent` properties align the child elements to the center of the container and add space around them, respectively.

Let's see how this layout looks when you rotate the device from a portrait orientation to a landscape orientation:



Wow, that's pretty handy—the flexbox automatically figured out how to preserve the layout for us. However, I think we can improve on this a little bit. For example, the landscape orientation has a lot of wasted space to the left and right now. We could also create our own abstraction for the boxes we're rendering.

Improved three column layout

There's a few things that I think we can improve upon from the last example. For starters, let's get rid of the duplicate code in the `index.ios.js` and `index.android.js` files. We'll create a new `index.js` file and put the code there. Since it's not specific to any one platform, it doesn't need a special name. Now the `index.ios.js` and `index.android.js` files both have one line in them:

```
import './index.js';
```

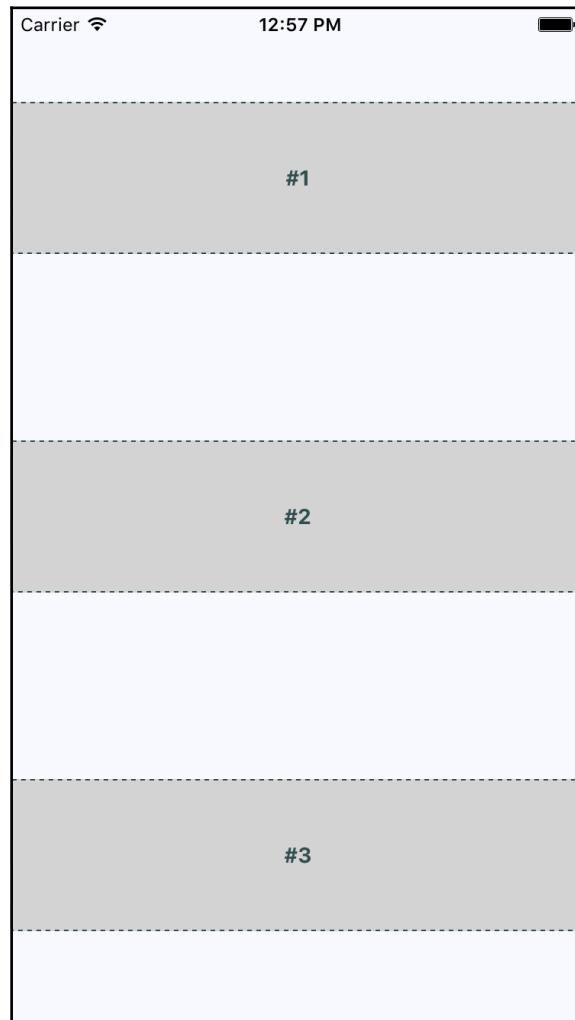
Next, let's fix the styles so that the children of the flexbox stretch to take advantage of the available space. Remember in the last example when you rotated the device from portrait to landscape orientation? There was a lot of wasted space. It would be nice to have the components automatically adjust themselves. Here's what the new styles module looks like:

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

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: 'ghostwhite',
    alignItems: 'center',
    justifyContent: 'space-around',
  },
  box: {
    height: 100,
    justifyContent: 'center',
    // Instead of given the flexbox child a width, we
    // tell it to "stretch" to fill all available space.
    alignSelf: 'stretch',
    alignItems: 'center',
    backgroundColor: 'lightgray',
    borderWidth: 1,
    borderStyle: 'dashed',
    borderColor: 'darkslategray',
  },
  boxText: {
    color: 'darkslategray',
    fontWeight: 'bold',
  },
});

export default styles;
```

The key change here is the `alignSelf` property. This tells elements with the `box` style to change their width or height (depending on the `flexDirection` of their container) to fill space. Also, note that the `box` style no longer defines a `width` property because this will be computed on the fly now. Here's what the sections look like in portrait mode:



Now each section takes the full width of the screen, which is exactly what we want to happen. The issue of wasted space was actually more prevalent in landscape orientation, so let's rotate the device and see what happens to these sections now:



Nice. Now your layout is utilizing the entire width of the screen, regardless of orientation. Lastly, let's implement a proper `Box` component that can be used by `index.js` instead of having repetitive style properties in place. Here's what the `Box` component looks like:

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

import styles from './styles';

// Exports a React Native component that
// renders a "<View>" with the "box" style
// and a "<Text>" component with the "boxText"
// style.
const Box = ({ children }) => (
  <View style={styles.box}>
    <Text style={styles.boxText}>
      {children}
    </Text>
  </View>
);

Box.propTypes = {
  children: PropTypes.node.isRequired,
};

export default Box;
```

We have the beginnings of a nice layout. Now it's time to focus on flexing in the other direction—left to right.

Flexible rows

In this section, you'll learn how to make screen layout sections stretch from top to bottom. To do this, you need a flexible row. Here's what the styles for this screen look like:

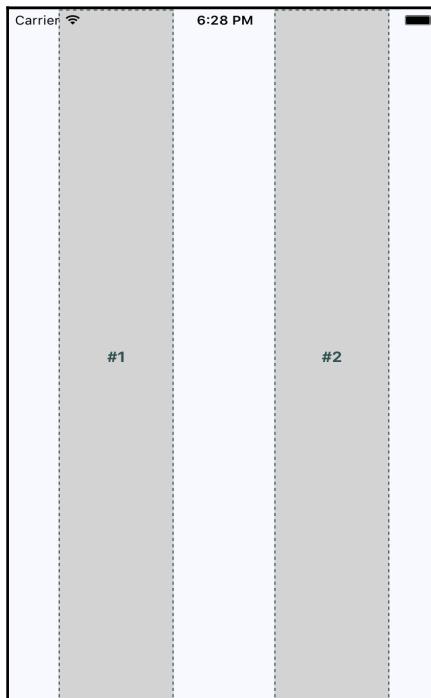
```
import { StyleSheet } from 'react-native';
const styles = StyleSheet.create({
  container: {
    flex: 1,
    // Tells the child elements to flex from left to
    // right...
    flexDirection: 'row',
    backgroundColor: 'ghostwhite',
    alignItems: 'center',
    justifyContent: 'space-around',
  },
  box: {
    // There's no height, so "box" elements will stretch
    // from top to bottom.
    width: 100,
    justifyContent: 'center',
    alignSelf: 'stretch',
    alignItems: 'center',
    backgroundColor: 'lightgray',
    borderWidth: 1,
    borderStyle: 'dashed',
    borderColor: 'darkslategray',
  },
  boxText: {
    color: 'darkslategray',
    fontWeight: 'bold',
  },
});
export default styles;
```

Here's the React Native component, using the same Box component that you implemented in the previous section:

```
import React from 'react';
import {
```

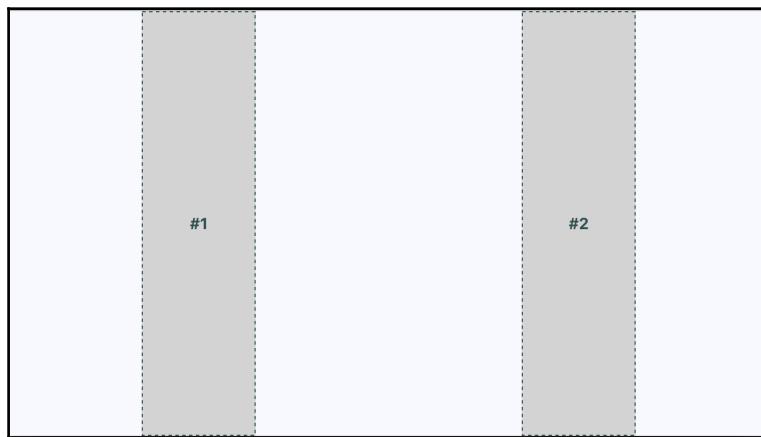
```
AppRegistry,  
View,  
} from 'react-native';  
  
import styles from './styles';  
import Box from './Box';  
  
// Renders a single row with two boxes that stretch  
// from top to bottom.  
const FlexibleRows = () => (  
  <View style={styles.container}>  
    <Box>#1</Box>  
    <Box>#2</Box>  
  </View>  
)  
  
AppRegistry.registerComponent(  
  'FlexibleRows',  
  () => FlexibleRows  
)
```

Here's what the resulting screen looks like in portrait mode:



The two columns stretch all the way from the top of the screen to the bottom of the screen because of the `alignSelf` property, which doesn't actually say which direction to stretch in. The two `Box` components stretch from top to bottom because they're displayed in a flex row. Note how the spacing between these two section goes from left to right? This is because of the container's `flexDirection` property, which has a value of `row`.

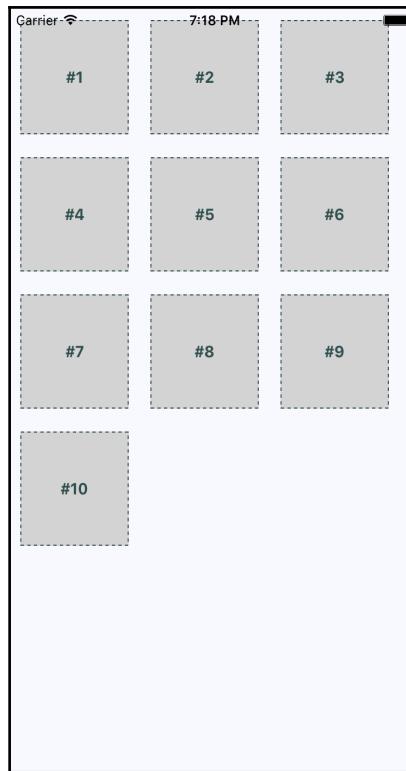
Now let's see how this flex direction impacts the layout when the screen is rotated into a landscape orientation:



Since the flexbox has a `justifyContent` style property value of `space-around`, space is proportionally added to the left, the right, and in between the sections.

Flexible grids

Sometimes you need a screen layout that flows like a grid. For example, what if you have several sections that are the same width and height, but you're not sure how many of these sections will be rendered? The flexbox makes it easy to build a row that flows from left to right until the end of the screen is reached. Then, it automatically continues rendering elements from left to right on the next row. Here's an example layout in portrait mode:



The beauty of this approach is that you don't need to know in advance how many columns are in a given row. The dimensions of each child determine what will fit in a given row. Let's take a look at the styles used to create this layout:

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

// Exports a "stylesheet" that can be used
// by React Native components. The structure is
// familiar for CSS authors.
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    // The child elements of this container will flow
    // from left to right. The "wrap" value here will
    // make the row jump to the next row, and start
    // flowing from left to right again.
    flexWrap: 'wrap',
    backgroundColor: 'ghostwhite',
    alignItems: 'center',
```

```
},
box: {
  // When there's an exact width and height for
  // a flexbox child, there's no need to "stretch" it.
  height: 100,
  width: 100,
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: 'lightgray',
  borderWidth: 1,
  borderStyle: 'dashed',
  borderColor: 'darkslategray',
  // Margins usually work better than "space-around"
  // for grids.
  margin: 10,
},
boxText: {
  color: 'darkslategray',
  fontWeight: 'bold',
},
});
export default styles;
```

Here's the React Native component that renders each section:

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

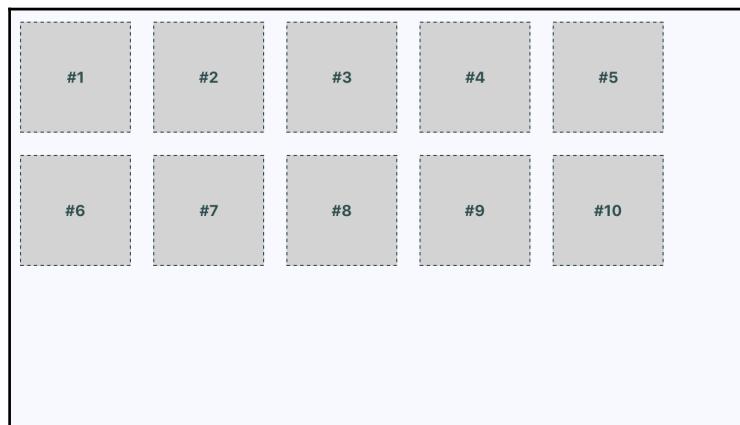
import styles from './styles';
import Box from './Box';

// An array of 10 numbers, representing the grid
// sections to render.
const boxes = new Array(10)
  .fill(null)
  .map((v, i) => i + 1);

const FlexibleGrids = () => (
  <View style={styles.container}>
    {/* Renders 10 "<Box>" sections */}
    {boxes.map(i => (
      <Box key={i}>#{i}</Box>
    ))}
  </View>
)
```

```
) } }  
</View>  
);  
  
AppRegistry.registerComponent(  
  'FlexibleGrids',  
  () => FlexibleGrids  
) ;
```

Lastly, let's make sure that the landscape orientation works with this layout:



You might have noticed that there's some superfluous space on the right side. Remember, these sections are only visible in this book because we want them to be visible. In a real app, they're just grouping other React Native components. However, if the space to the right of the screen becomes an issue, play around with the margin and the width of the child components.

Flexible rows and columns

In this final section of the chapter, we'll combine rows and columns to create a sophisticated layout for our screen. For example, sometimes you need the ability to nest columns within rows or rows within columns. Let's take a look at the main module of an application that nests columns within rows:

```
import React, { Component } from 'react';  
import {  
  AppRegistry,  
  View,  
}
```

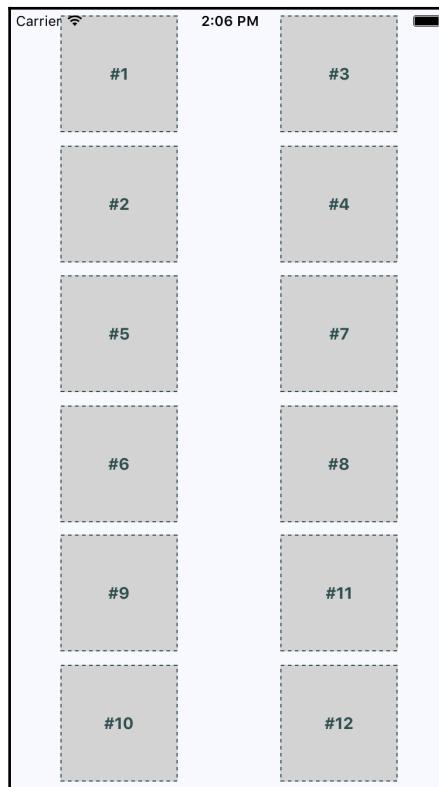
```
    } from 'react-native';

    import styles from './styles';
    import Row from './Row';
    import Column from './Column';
    import Box from './Box';

    class FlexibleRowsAndColumns extends Component {
        render() {
            return (
                <View style={styles.container}>
                    /* This row contains two columns. The first column
                     has boxes "#1" and "#2". They will be stacked on
                     top of one another. The next column has boxes
                     "#3" and "#4", which are also stacked on top
                     of one another */
                    <Row>
                        <Column>
                            <Box>#1</Box>
                            <Box>#2</Box>
                        </Column>
                        <Column>
                            <Box>#3</Box>
                            <Box>#4</Box>
                        </Column>
                    </Row>
                    <Row>
                        <Column>
                            <Box>#5</Box>
                            <Box>#6</Box>
                        </Column>
                        <Column>
                            <Box>#7</Box>
                            <Box>#8</Box>
                        </Column>
                    </Row>
                    <Row>
                        <Column>
                            <Box>#9</Box>
                            <Box>#10</Box>
                        </Column>
                        <Column>
                            <Box>#11</Box>
                            <Box>#12</Box>
                        </Column>
                    </Row>
                </View>
            );
        }
    }
}
```

```
        }  
    }  
  
AppRegistry.registerComponent(  
    'FlexibleRowsAndColumns',  
    () => FlexibleRowsAndColumns  
) ;
```

This is actually straightforward to parse and understand, because we've created abstractions for the layout pieces (`<Row>` and `<Column>`) and the content piece (`<Box>`). We'll check these parts out momentarily, but for now, let's see what this screen looks like:



This layout probably looks familiar, because we've done it already in this chapter. The key difference is in how these content sections are ordered. For example, #2 doesn't go to the left of #1, it goes below it. This is because we've placed #1 and #2 in a `<Column>`. Same with #3 and #4. These two columns are placed in a row. Then the next row begins, and so on.

This is just one of many possible layouts that you can achieve by nesting row flexboxes and column flexboxes. Let's take a look at the `Row` component now:

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

import styles from './styles';

// Renders a "View" with the "row" style applied to
// it. It's "children" will flow from left to right.
const Row = ({ children }) => (
  <View style={styles.row}>
    {children}
  </View>
);

Row.propTypes = {
  children: PropTypes.node.isRequired,
};

export default Row;
```

It doesn't do much, except apply the `row` style to the `<View>` component. Well, as you saw in the main module, this is actually a big deal, because it leads to cleaner JSX markup. Finally, let's look at the `Column` component:

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

import styles from './styles';

// Renders a "View" with the "column" style applied
// to it. It's children will flow from top-to-bottom.
const Column = ({ children }) => (
  <View style={styles.column}>
    {children}
  </View>
);

Column.propTypes = {
  children: PropTypes.node.isRequired,
};

export default Column;
```

Again, this looks just like the `Row` component, only with a different style applied to it. But it means easy layout in other modules.

Summary

This chapter introduced you to styles in React Native. Though you can use many of the same CSS style properties that you're used to, the CSS stylesheets used in web applications look very different. Namely, they're composed of plain JavaScript objects.

Then, you learned how to work with the main React Native layout mechanism—the flexbox. This is the preferred way to layout most web applications these days, and so it only makes sense to be able to reuse this approach in a native app. You created several different layouts, and you saw how they looked in portrait and in landscape orientation.

In the following chapter, you'll start implementing navigation for your app.

15

Navigating Between Screens

Routing is an essential part of any React web application. Without a router, the pages that correspond to URLs would be impossible to manage, making your application impossible to scale. React Native applications don't have URLs that map to pages, but you can implement navigation that's conceptually similar to routing URLs to pages.

We'll get started by thinking about how to divide a mobile application into screens and introduce the common terminology that's used with React Native navigation. Then we'll walk through several examples that demonstrate the navigational abilities of React Native.

Screen organization

Thinking in terms of screens is easy in web applications because you have a URL that points to each screen. Put differently, URLs make thinking about screen organization natural and easy. Take the URLs away and suddenly screen organization becomes more difficult.

Imagine trying to build screen transitions in a mobile application without the page abstraction that we take for granted in web interfaces. You would have to build your own abstraction that ensures that each component for each screen is rendered, and subsequently removed once the user moves away from the page. Not ideal.

But, you really want to work with some kind of page because that's what you're used to working with when building React applications for the web, right? In the following section, we'll cover the terminology used in React Native that helps make this happen. The real challenge is simply coming up with the correct screens. It's challenging because there aren't URLs in the traditional sense. However, just pretend that there are URLs; this will help you decompose your app content into the correct screens.

Navigators, scenes, routes, and stacks

The core mechanism you'll use to control navigation in your React Native app is the `Navigator` component. It's used to control route stacks and scenes. I'll quickly define these concepts here, and then we'll dive into some code.

- **Navigator:** The overarching component that's used to control how users navigate through your application
- **Scene:** A simple React component that represents what the user is currently looking at. Instead of pressing a link on a page that takes them to another page, the `Navigator` takes them to another scene
- **Route:** A JavaScript object containing information about a scene. The `Navigator` figures out how to render a scene based on information provided by a route
- **Stack:** A stack of routes held by the `Navigator`. These are all the routes that the user can navigate to in a React Native application

Confused? Don't worry! This will all start to make sense once I start speaking in code: right now.



There are actually two other `Navigator` components offered by React Native. I'm only covering the generic `Navigator` component because it works the same on both iOS and Android. But if you're interested, there's `NavigatorIOS` which is geared toward iOS devices, and `NavigatorExperimental`, which takes a different approach to handling routing.

Responding to routes

Let's implement a simple UI with three scenes in it (remember, scenes are synonymous with pages). Each scene will link to the other two scenes so that you can actually navigate the application. Let's start with the main application module:

```
import React from 'react';
import {
  AppRegistry,
  Navigator,
} from 'react-native';

import routes from './routes';

// Renders a scene. The "Scene" is a React component, and
```

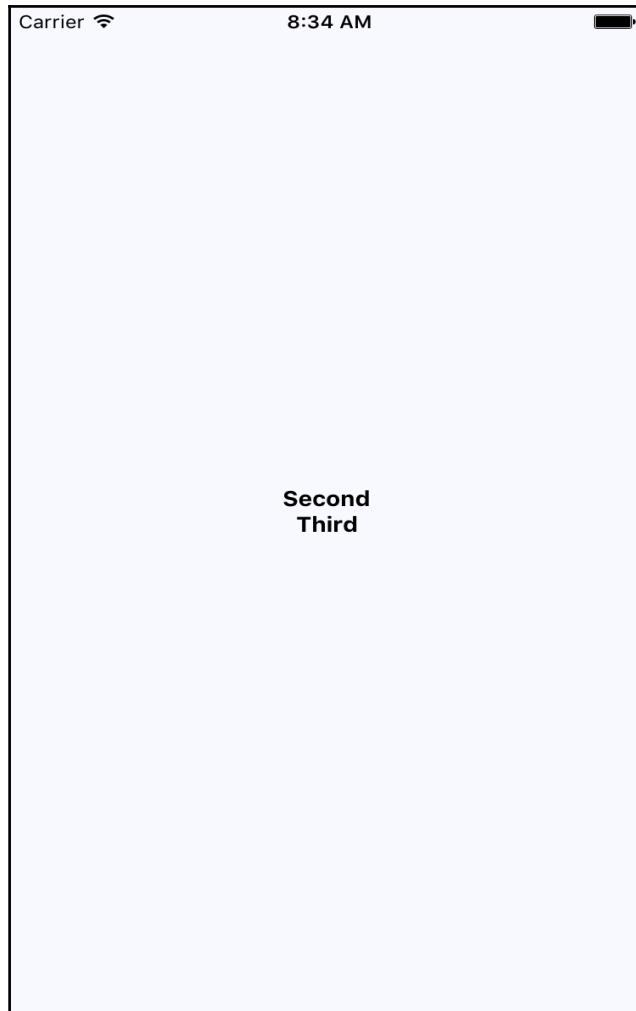
```
// is found as a property of the "route". The "navigator"
// instance is passed to the scene as a property.
const renderScene = (route, navigator) => (
  <route.Scene navigator={navigator} />
);

// Renders a "<Navigator>" component. This is the root
// component of the application because it manages
// "scenes" that represent actual content. It's given
// and "initialRoute" to render when the application
// starts, and a "initialRouteStack" for all possible
// routes.
const RespondingToRoutes = () => (
  <Navigator
    initialRoute={routes[0]}
    initialRouteStack={routes}
    renderScene={renderScene}
  />
);

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

This component doesn't actually render any application content, just the `<Navigator>`. It's the navigator that manages how scenes get rendered. For example, the `renderScene()` function we've just created takes the `Scene` component from a route and renders it. The `navigator` uses this function to render content as the current route changes.

As you can see, the navigator is passed an initial route and a route stack. Both of these come from a routes module but, before we look at that, let's see what this application looks like:



There's obviously not much to this UI and this is by design; I'm trying to keep you focused on navigation. You're looking at the first page of this app, which has two text links that bring the user to those pages. Before we look at how these links work, let's look at the routes module:

```
// Imports all the scenes.  
import first from './scenes/first';  
import second from './scenes/second';  
import third from './scenes/third';  
  
// Exports the route stack, an array of all  
// available scenes.  
export default [  
  first,  
  second,  
  third,  
];
```

Pretty simple; it builds an array of scenes that the navigator can use as the initial route stack. You might be wondering why this indirection is even necessary; can't I just build the array in the main module? Sometimes, as you'll see later, it's desirable to augment route data. When this happens, it's nice to have all the routes in one place, away from everything else.

Now let's check out one of the scene modules. We'll only look at the first one because all three scenes are nearly identical:

```
import React, { PropTypes } from 'react';  
import { View, Text } from 'react-native';  
  
import styles from '../styles';  
  
// Import the scenes that this scene can jump to.  
import second from './second';  
import third from './third';  
  
// Renders a view with two text links in it.  
const Scene = ({ navigator }) => (  
  <View style={styles.container}>  
    <Text  
      style={styles.item}  
      onPress={() => navigator.jumpTo(second)}  
    >  
      Second  
    </Text>  
    <Text  
      style={styles.item}
```

```
onPress={() => navigator.jumpTo(third)}  
>  
    Third  
</Text>  
</View>  
)  
  
Scene.propTypes = {  
  navigator: PropTypes.object.isRequired,  
};  
  
export default {  
  Scene,  
};
```

The two links that are rendered by the `Scene` component are rendered using `<Text>` components that respond to `Press` events. Instead of just pointing to a URL like you would in a web app, you have to explicitly tell the navigator where you want to go. This is why the main module passes the `navigator` instance to scene components as a property.

The `navigator` method used here is `jumpTo()`, which takes a `route` object as an argument. This route is looked up in the route stack and the `renderScene()` method of the `navigator` is called. You can't jump to a route that isn't already in the route stack because these components are already rendered and the navigator is simply managing their display.



You might have noticed that the `Scene` component is exported as a property of an object, rather than just exporting the component directly. Indeed, I would like to just use a component as a route, and this does work, but the `Navigator` property validation complains because it's expecting objects, not functions. I can't live with myself if I let validation warnings slide.

Navigation bar

The previous example rendered navigation links as the main content. However, it's often better to place the navigation links in their own section. The `Navigator` component lets you specify a navigation bar. Let's try this out now, starting with the main module:

```
import React from 'react';
import {
  AppRegistry,
  Navigator,
} from 'react-native';

import routes from './routes';
import styles from './styles';

// Renders the scene, the "navigator" property
// is no longer needed here since navigation is
// now handled separately.
const renderScene = route => (<route.Scene />);

// The composition of the navigation bar is three components:
// "Title", "LeftButton", and "RightButton". These
// components come from the "route".
const routeMapper = {
  Title: (route, navigator) => (
    <route.Title navigator={navigator} />
  ),
  LeftButton: (route, navigator) => (
    <route.LeftButton navigator={navigator} />
  ),
  RightButton: (route, navigator) => (
    <route.RightButton navigator={navigator} />
  ),
};

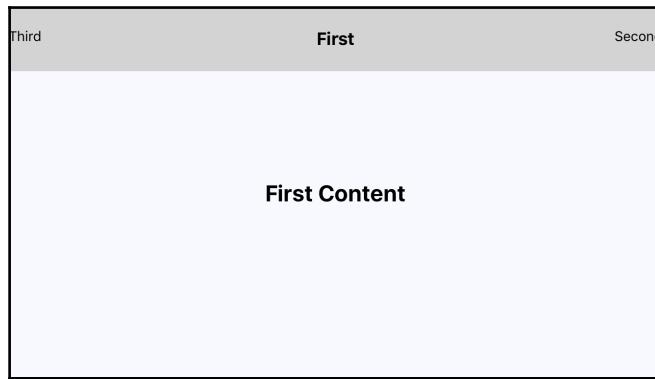
// Renders the "<NavigationBar>" component, using
// the "routeMapper" object to configure the
// navigation buttons. We can also style it however
// we want.
const navigationBar = (
  <Navigator.NavigationBar
    style={styles.nav}
    routeMapper={routeMapper}
  />
);

// Uses the "navigationBar" property to render the
```

```
// navbar independent of the scene content.
const NavigationBar = () => (
  <Navigator
    initialRoute={routes[0]}
    initialRouteStack={routes}
    renderScene={renderScene}
    navigationBar={navigationBar}
  />
);

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

This looks like the main module in the previous section, the main difference being that we're passing a navigation bar component to `<Navigator>`. There's some additional route data as well; the navigation bar needs components for the title, the left button, and the right button. Here's what the screen looks like now:



As you can see, the navigation is placed at the top of the screen, freeing up space for other content in the rest of the screen. The best part about this type of navigation bar is that it looks and functions the same in iOS and Android.

Now let's see what the first scene module looks like:

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

import styles from '../styles';
import second from './second';
import third from './third';
```

```
// The "Scene" component only has to worry about
// rendering content now. In this case, some simple
// text and styles.
const Scene = () => (
  <View style={styles.container}>
    <Text style={styles.content}>
      First Content
    </Text>
  </View>
);

// The title component for the navigation bar...
const Title = () => (
  <Text style={styles.title}>First</Text>
);

// The left button for the navigation bar...
const LeftButton = ({ navigator }) => (
  <Text
    onPress={() => navigator.jumpTo(third)}
  >
    Third
  </Text>
);

LeftButton.propTypes = {
  navigator: PropTypes.object.isRequired,
};

// The right button for the navigation bar...
const RightButton = ({ navigator }) => (
  <Text
    onPress={() => navigator.jumpTo(second)}
  >
    Second
  </Text>
);

RightButton.propTypes = {
  navigator: PropTypes.object.isRequired,
};

// The exported route now has components for the
// scene and for the navigation bar.
export default {
  Scene,
  Title,
  LeftButton,
```

```
    RightButton,  
};
```

There's a lot more going on here than there was in the preceding example. This is due to the fact that this module declares both the `Scene` component and the navigation components: four in total. However, it's best that you declare them all in the same module, because they're closely related, and they're small. If you take a look at the export, you can see that the route object actually contains more than just the `Scene` component; it's everything the route needs.

The only problem I have with this implementation is that it requires mounting three separate components that are essentially the same. There has to be a more efficient approach.

Dynamic scenes

Some apps, like the one we've been working on in this chapter, have very similar scenes. In fact, they're so similar that having three unique components for this purpose feels awkward and repetitive. It would make more sense to have a single scene and navigation bar and pass them the dynamic pieces of information through the route objects.

Let's make some changes to the application, starting with the main module so that we don't need duplicate components anymore:

```
import React from 'react';  
import {  
  AppRegistry,  
  Navigator,  
} from 'react-native';  
  
import routes from './routes';  
import styles from './styles';  
  
// The scene content now comes from the "route".  
const renderScene = route => (  
  <route.Scene  
    content={route.content}  
  />  
);  
  
// The "routeMapper" object now has to pass each navbar item  
// more properties since the same component is used now. For  
// example, the "LeftButton" component passes in "content"  
// and the "route" that's to be activated if the user presses
```

```
// the button.
const routeMapper = {
  Title: (route, navigator) => (
    <route.Title
      navigator={navigator}
      title={route.title}
    />
  ),
  LeftButton: (route, navigator) => (
    <route.LeftButton
      navigator={navigator}
      content={route.leftTitle}
      route={route.leftRoute}
    />
  ),
  RightButton: (route, navigator) => (
    <route.RightButton
      navigator={navigator}
      content={route.rightTitle}
      route={route.rightRoute}
    />
  ),
};

const navigationBar = (
  <Navigator.NavigationBar
    style={styles.nav}
    routeMapper={routeMapper}
  />
);

// The "Navigator" component no longer has an initial
// route stack passed to it. Instead, current routes
// are "replaced" by new routes.
const DynamicRouteData = () => (
  <Navigator
    initialRoute={routes[0]}
    renderScene={renderScene}
    navigationBar={navigationBar}
  />
);

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

You can see here that the code looks more or less as it did in the previous section, only now we're passing more properties to the components as they're rendered. For example, the main Scene component gets its content passed to it from a property. The left button gets its content and the route to follow passed in as property values.

Now, let's take a look at the scene module that's used for every screen in the application:

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

import styles from './styles';

// The content rendered by the scene now comes from
// a property, since this is the only scene component
// in the whole app.
const Scene = ({ content }) => (
  <View style={styles.container}>
    <Text style={styles.content}>
      {content}
    </Text>
  </View>
);

Scene.propTypes = {
  content: PropTypes.node.isRequired,
};

// The "title" value also comes from a prop.
const Title = ({ title }) => (
  <Text style={styles.title}>{title}</Text>
);

Title.propTypes = {
  title: PropTypes.node.isRequired,
};

// The left button label and the route that's activated
// on press are passed in as properties.
const LeftButton = ({ navigator, route, content }) => (
  <Text onPress={() => navigator.replace(route)}>
    {content}
  </Text>
);

LeftButton.propTypes = {
  navigator: PropTypes.object.isRequired,
  route: PropTypes.object.isRequired,
```

```
        content: PropTypes.node.isRequired,
    };

    // The right button label and the route that's activated
    // on press are passed in as properties.
    const RightButton = ({ navigator, route, content }) => (
        <Text onPress={() => navigator.replace(route)}>
            {content}
        </Text>
    );

    RightButton.propTypes = {
        navigator: PropTypes.object.isRequired,
        route: PropTypes.object.isRequired,
        content: PropTypes.node.isRequired,
    };

    export default {
        Scene,
        Title,
        LeftButton,
        RightButton,
    };
}
```

The key change between this module and the other scene modules in the preceding sections is that nothing is hardcoded here. The labels, and even the routes themselves, are passed in as component properties. Another change I've introduced here is the use of the `replace()` method to change the current route, instead of the `jumpTo()` method. The difference is that, since we don't have an active route stack, we can just unmount what's already rendered and re-render the new component. This should be really efficient because the component type is the same; it's just the property values that have changed.

Lastly, let's take a look at the `routes` module:

```
import mainRoute from './scene';

// Each route object has enough data to render
// the dynamic parts of the scene and the navbar.
const firstRoute = {
    index: 0,
    title: 'First',
    leftTitle: 'Third',
    rightTitle: 'Second',
    content: 'First Content',
};

const secondRoute = {
```

```
index: 1,
title: 'Second',
leftTitle: 'First',
rightTitle: 'Third',
content: 'Second Content',
};

const thirdRoute = {
  index: 2,
  title: 'Third',
  leftTitle: 'Second',
  rightTitle: 'First',
  content: 'Third Content',
};

// Each route is extended with the components from the
// scene. This means that the same component is reused,
// and new property values change the data as the
// user navigates the application.
export default [
  Object.assign(firstRoute, mainRoute, {
    leftRoute: thirdRoute,
    rightRoute: secondRoute,
  }),
  Object.assign(secondRoute, mainRoute, {
    leftRoute: firstRoute,
    rightRoute: thirdRoute,
  }),
  Object.assign(thirdRoute, mainRoute, {
    leftRoute: secondRoute,
    rightRoute: firstRoute,
  }),
];
```

The components that get rendered come from `mainRoute`. All we have to do is use `Object.assign()` to share the references to these components with each route.

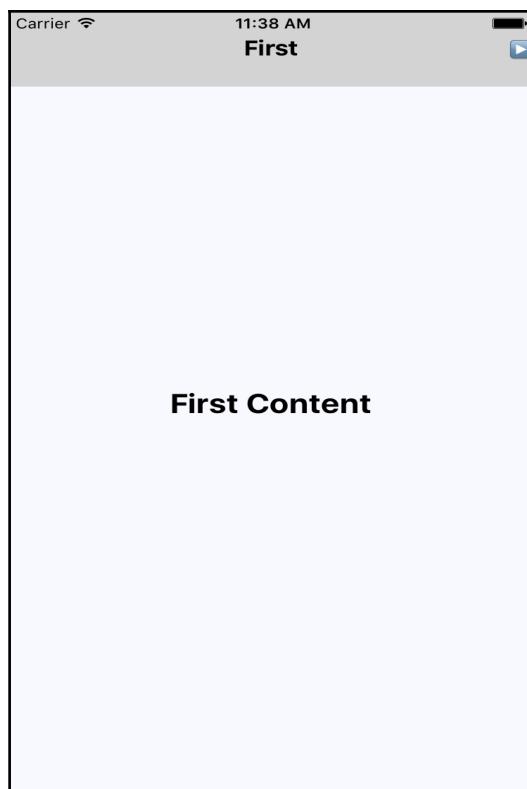


Technically, we don't have to export any array from this module because we're only passing the first route to the `navigator`. However, I'm not sure I like the assumption that you'll never use these routes as the initial route stack. When in doubt, export all your routes as an array.

Jumping back and forth

In the previous example, you told the left and right navigation buttons where they should link to by passing routes within routes. The challenge with this approach is that it takes a lot of work to keep this navigation data up-to-date, especially if all you need is simple back and forward behavior. Let's make some adjustments to the application so that the buttons automatically know which route to use.

First, let's take a look at the UI so that you can see what we're trying to achieve here:



If you take a look at the navigation bar, you'll notice that there are a title and a forward button. But there's no back button. This is because the user is on the first screen, so there's nowhere to navigate back to. When the user moves to the second screen, they'll see a forward and a back button. Now let's take a look at the main module:

```
import React from 'react';
import {
  AppRegistry,
  Navigator,
} from 'react-native';

import routes from './routes';
import styles from './styles';

const renderScene = (route, navigator) => (
  <route.Scene
    navigator={navigator}
    content={route.content}
  />
);

// The "LeftButton" and "RightButton" components
// are passed the "routes" array as a property so
// that they can do bounds-checking.
const routeMapper = {
  Title: (route, navigator) => (
    <route.Title
      navigator={navigator}
      title={route.title}
    />
  ),
  LeftButton: (route, navigator) => (
    <route.LeftButton
      navigator={navigator}
      route={route}
      routes={routes}
    />
  ),
  RightButton: (route, navigator) => (
    <route.RightButton
      navigator={navigator}
      route={route}
      routes={routes}
    />
  ),
};

const navigationBar = (
```

```
<Navigator.NavigationBar
  style={styles.nav}
  routeMapper={routeMapper}
/>
);

// Notice how we're passing in an initial route stack
// again? This is how we're able to use "jumpForward()"
// and "jumpBack()".
const JumpingBack = () => (
<Navigator
  initialRoute={routes[0]}
  initialRouteStack={routes}
  renderScene={renderScene}
  navigationBar={navigationBar}
/>
);
AppRegistry.registerComponent(
  'JumpingBack',
  () => JumpingBack
);
```

There are two things that we've changed from the previous example. First, you can see that we're passing the `initialRouteStack` property again. This means that all the routes in this array are rendered, but only `initialRoute` is displayed. We're doing this because in order to jump back and forth without prior knowledge of the next route, we need a route stack.

Another way to think about this navigation model is in terms of an array index. We're starting at 0, the first route in the stack. When the user presses forward, the index is incremented. When the user presses back, the index is decremented. Let's take a look at the simplified `routes` module now:

```
import mainRoute from './scene';

const firstRoute = {
  title: 'First',
  content: 'First Content',
};

const secondRoute = {
  title: 'Second',
  content: 'Second Content',
};

const thirdRoute = {
```

```
        title: 'Third',
        content: 'Third Content',
    };

// The exported routes no longer contain properties
// that point to other routes.
export default [
    Object.assign(firstRoute, mainRoute),
    Object.assign(secondRoute, mainRoute),
    Object.assign(thirdRoute, mainRoute),
];
```

This is much easier to digest. There are no more left and right button components. Finally, let's take a look at the `Scene` component that renders the screen:

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

import styles from './styles';

const Scene = ({ content }) => (
    <View style={styles.container}>
        <Text style={styles.content}>
            {content}
        </Text>
    </View>
);

Scene.propTypes = {
    content: PropTypes.node.isRequired,
};

const Title = ({ title }) => (
    <Text style={styles.title}>{title}</Text>
);

Title.propTypes = {
    title: PropTypes.node.isRequired,
};

// The left button is the "back" button. Notice that we
// don't require a specific route here - we just use
// the "jumpBack()" method.
const LeftButton = ({ navigator, route, routes }) =>
    routes.indexOf(route) === 0 ? null : (
        <Text onPress={() =>navigator.jumpBack()}>
            &#9654:
        </Text>
    );

```

```
) ;

LeftButton.propTypes = {
  navigator: PropTypes.object.isRequired,
  route: PropTypes.object.isRequired,
  routes: PropTypes.array.isRequired,
};

// The right button is the "forward" button. Notice that
// we don't require a specific route here - we just use
// the "jumpForward()" method.
const RightButton = ({ navigator, route, routes }) =>
  routes.indexOf(route) === (routes.length - 1) ? null : (
    <Text onPress={() => navigator.jumpForward()}>
      &#9664;
    </Text>
  );

RightButton.propTypes = {
  navigator: PropTypes.object.isRequired,
  route: PropTypes.object.isRequired,
  routes: PropTypes.array.isRequired,
};

export const route = {
  Scene,
  Title,
  LeftButton,
  RightButton,
};

export default route;
```

As you can see, the left and right buttons each render an arrow that uses the `jumpBack()` and `jumpForward()` methods, respectively. There's a simple bounds checking mechanism in place for each `Press` event that makes sure the button should actually render. For example, if the user is at the beginning of the route stack, then there's no need to render a back button. This little piece of additional logic is powerful, because this component doesn't need to rely on any routes being passed to it, only `Navigator` methods.

Summary

In this chapter, you learned about navigation in React Native applications. We compared the traditional mechanisms used to navigate web applications with what's used in native mobile apps. The key differentiator in React Native is that there's no URL. Instead, you have to rely on route objects.

Next, you implemented a basic example that rendered different screen content based on the link that was pressed. Links can be tricky because you're not passing a URL that's automatically handled by a web browser. Next, you learned about the navigation bar component that can be passed to the `navigator`, to provide consistent navigation between iOS and Android.

Then, you implemented dynamic scenes that passed content through the `route` object. Instead of providing an initial route stack, you only provided an initial route that was replaced whenever the user pressed a navigation button. Lastly, you learned a simple technique to implement simple back and forward navigation. In the next chapter, you'll learn how to render lists of data.

16

Rendering Item Lists

In this chapter, you'll learn how to work with item lists. Working with lists is a common development activity when building applications for the Web. It's also relatively straightforward to build lists using the `` and `` elements. Trying to do something similar on native mobile platforms is much more involved.

Thankfully, React Native provides a simple item list interface that hides all of the complexity. We'll kick things off by getting a feel for how item lists work by walking through an example. Then, we'll introduce some controls that change the data displayed in lists. Lastly, you'll see a couple of examples that fetch items from the network.

Rendering data collections

Let's start with a basic example. The React Native component you'll use to render lists is `ListView`, which works the same way on iOS and Android. List views take a data source property, which must be a `ListView.DataSource` instance. Don't worry; it's really just a wrapper around an array in most cases. The reason that the `ListView` component expects this type of data source is so that it can perform efficient rendering. Lists can be long and updating them frequently can cause performance issues.

So, let's implement a basic list now, shall we? Here's the code to render a basic 100-item list:

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

```
import styles from './styles';

// You always need a comparator function that's
// used to determine whether or not a row has
// changed. Even in simple cases like this, where
// strict inequality is used.
const rowHasChanged = (r1, r2) => r1 !== r2;

const source = new ListView
// A data source for the list. It's eventually passed
// to the "<ListView>" component, and it requires a
// "rowHasChanged()" comparator.
.DataSource({ rowHasChanged })

// Returns a clone of the data source with new data.
// The comparator function is used by the "<ListView>"
// component to determine what has changed.
.cloneWithRows(
  new Array(100)
    .fill(null)
    .map((v, i) => `Item ${i}`)
);

const RenderingDataCollections = () => (
  <View style={styles.container}>
    { /* Renders the list by providing a "dataSource"
       property and a "renderRow" function which
       renders each item in the list. */ }
    <ListView
      dataSource={source}
      renderRow={
        i => (<Text style={styles.item}>{i}</Text>)
      }
    />
  </View>
);

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

Let's walk through what's going on here, starting with the `source` constant. As you can see, this is created using the `ListView.DataSource()` constructor. Here, we're giving it a `rowHasChanged()` function. Data sources need to be told how to look for changes, even if it's a simple equality check. Then, we pass the actual data into the `cloneWithRows()` method. This actually results in a new instance of the data source, and is actually a confusing name because none of the data is actually cloned. All you're cloning are the options you give the data source, such as the `rowHasChanged()` function for example. `DataSource` instances are immutable, and we'll see how to actually update them in the following examples.

Next, we render the `<ListView>` component itself. It's within a `<View>` container because list views need a height in order to make scrolling work correctly. The `source` and the `renderRow` properties are passed to the `<ListView>`, which ultimately determines the rendered content.

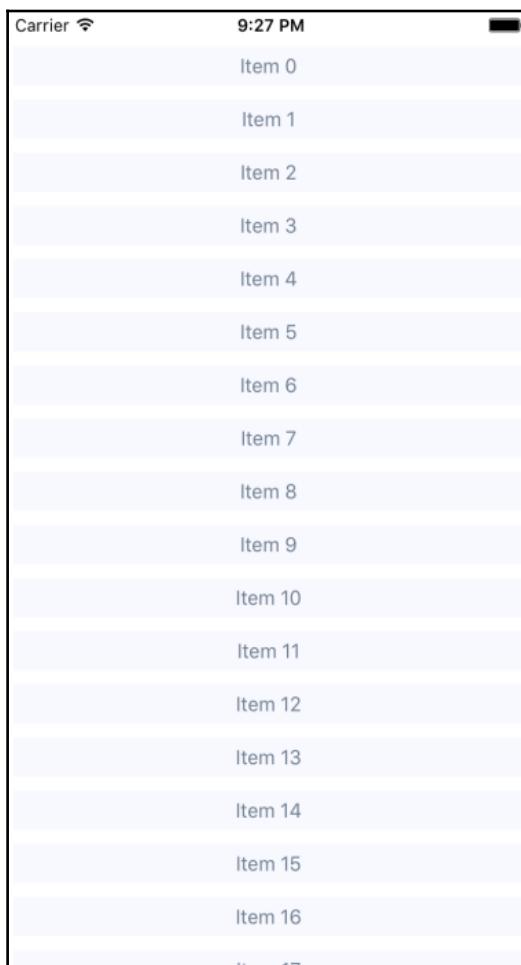
At first glance, it would seem that the `ListView` component doesn't do too much for us. We have to figure out how the items look? Well, yes, the `ListView` is supposed to be generic. It's supposed to excel at handling updates, and embeds scrolling capabilities into lists for us. Here are the styles that were used to render the list:

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

export default StyleSheet.create({
  container: {
    // Flexing from top to bottom gives the
    // container a height, which is necessary
    // to enable scrollable content.
    flex: 1,
    flexDirection: 'column',
    paddingTop: 20,
  },
  item: {
    margin: 5,
    padding: 5,
    color: 'slategrey',
    backgroundColor: 'ghostwhite',
    textAlign: 'center',
  },
});
```

Here, we're giving a basic style to each item in our list. Otherwise, each item would be text-only and would be difficult to differentiate between other list items. The container style gives the list a height by setting the flex direction to column. Without a height, you won't be able to scroll properly.

Let's see what this thing looks like now, shall we?

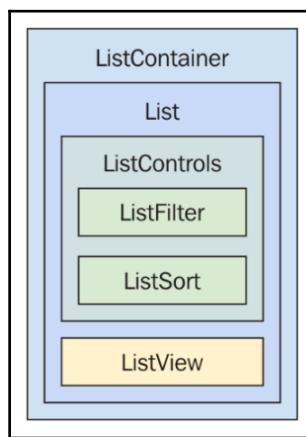


If you're running this example in a simulator, you can click and hold down the mouse button anywhere on the screen, like a finger, then scroll up or down through the items.

Sorting and filtering lists

Now that you have the basics of `ListView` components, and passing them `DataSource` instances, let's add some controls to the list you just implemented. The `ListView` component itself helps you render fixed-position content for list controls. You'll also see how to manipulate the data source itself, which ultimately drives what's rendered on the screen.

Before we jump into implementing list control components, it might be helpful if we go over the high-level structure of these components so that the code has more context. Here's an illustration of the component structure that we're going to implement:



Here's what each of these components is responsible for:

- `ListContainer`: The overall container for the list; it follows the familiar React container pattern
- `List`: A stateless component that passes the relevant pieces of state into the `ListControls` and the React Native `ListView` component
- `ListControls`: A component that holds the various controls that change the state of the list
- `ListFilter`: A control for filtering the item list
- `ListSort`: A control for changing the sort order of the list
- `ListView`: The actual React Native component that renders items

In some cases, splitting apart the implementation of a list like this is overkill. However, I think that if your list needs controls in the first place, you're probably implementing something that will stand to benefit from having a well thought out component architecture.

Now let's drill down into the implementation of this list, starting with the `ListContainer` component:

```
import React, { Component } from 'react';
import { ListView } from 'react-native';

import List from './List';

// The two comparator functions we need to pass
// to the data source. The "rowHasChanged()" function
// does simple strict inequality. So does
// "sectionHeaderHasChanged()".
const rowHasChanged = (r1, r2) => r1 !== r2;
const sectionHeaderHasChanged = rowHasChanged;

// Performs sorting and filtering on the given "data".
const filterAndSort = (data, text, asc) =>
  data.filter(
    i =>
      // Items that include the filter "text" are returned.
      // Unless the "text" argument is an empty string,
      // then everything is included.
      text.length === 0 ||
      i.includes(text)
  ).sort(
    // Sorts either ascending or descending based on "asc".
    asc ?
      (a, b) => b > a ? -1 : (a === b ? 0 : 1) :
      (a, b) => a > b ? -1 : (a === b ? 0 : 1)
  );
}

class ListContainer extends Component {
  constructor() {
    super();

    // The initial state. The "data" is what drives
    // the list, and it's initially filtered and sorted
    // here.
    this.state = {
      data: filterAndSort(
        new Array(100)
          .fill(null)
```

```
        .map((v, i) => `Item ${i}`)
        , '',
        asc: true,
        filter: '',
    );

// The "source" is also part of the component state,
// but it's based on "state.data", which is why it's
// set here. This is the data source that's ultimately
// used by the "<ListView>".
this.state.source = new ListView
    .DataSource({
        rowHasChanged,
        sectionHeaderHasChanged,
    })
    .cloneWithRows(this.state.data);
}

render() {
    return (
        <List
            source={this.state.source}
            asc={this.state.asc}
            onFilter={(text) => {
                // Updates the "filter" state, the actual filter
                // text, and the "source" of the list. The "data"
                // state is never actually touched -
                // "filterAndSort()" doesn't mutate anything.
                this.setState({
                    filter: text,
                    source: this.state.source.cloneWithRows(
                        filterAndSort(
                            this.state.data,
                            text,
                            this.state.asc
                        )
                    ),
                });
            }}
            onSort={() => {
                this.setState({
                    // Updates the "asc" state in order to change
                    // the order of the list. The same principles as
                    // used in the "onFilter()" handler are applied
                    // here, only with different arguments passed to
                    // "filterAndSort()"
                    asc: !this.state.asc,
                    source: this.state.source.cloneWithRows(

```

```
        filterAndSort(
          this.state.data,
          this.state.filter,
          !this.state.asc
        )
      ),
    );
  }
}

export default ListContainer;
```

If this seems like a bit much, it's because it is. This container component has a lot of state to handle. It also has some nontrivial behavior that it needs to make available to its children. So, if you look at it from the perspective of encapsulating state, this doesn't seem so complicated. Its job is to populate the list with state data and provide functions that operate on this state.

In an ideal world, child components of this container should be nice and simple since they don't have to directly interface with state. Let's take a look at the `List` component next:

```
import React, { PropTypes } from 'react';
import { Text, ListView } from 'react-native';

import styles from './styles';
import ListControls from './ListControls';

// Renders the actual "<ListView>" React Native
// component. The "renderSectionHeader" property
// is where our controls go. The "renderRow"
// property, as always, renders the actual item.
const List = ({
  Controls,
  source,
  onFilter,
  onSort,
  asc,
}) => (
  <ListView
    enableEmptySections
    dataSource={source}
    renderSectionHeader={() => (
      <Controls
        {...{ onFilter, onSort, asc }}}
```

```
        />
    ) }
renderRow={i => (
    <Text style={styles.item}>{i}</Text>
)
}
/>
);

List.propTypes = {
    Controls: PropTypes.func.isRequired,
    source: PropTypes.instanceOf(ListView.DataSource).isRequired,
    onFilter: PropTypes.func.isRequired,
    onSort: PropTypes.func.isRequired,
    asc: PropTypes.bool.isRequired,
};

// The "Controls" component is actually our own
// "ListControls" component by default. However,
// this can be overriden by anyone wanting to provide
// their own control components.
List.defaultProps = {
    Controls: ListControls,
};

export default List;
```

This component takes state from the `ListContainer` component as properties and renders a `ListView` component. The main difference here, relative to the previous example, is the `renderSectionHeader` property. This function renders the controls for our list. What's especially useful about this property is that it renders the controls outside the scrollable list content, ensuring the controls are always visible.



There's a `renderHeader` property as well, which does essentially the same thing as `renderSectionHeader`; however, the position isn't fixed.

Also, notice that we're specifying our own `ListControls` component as a default value for the `controls` property. This makes it easy for others to pass in their own list controls. Let's take a look at the `ListControls` component next:

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

import styles from './styles';
import ListFilter from './ListFilter';
```

```
import ListSort from './ListSort';

// Renders the "<ListFilter>" and "<ListSort>"
// components within a "<View>". The
// "styles.controls" style lays out the controls
// horizontally.
const ListControls = ({
  onFilter,
  onSort,
  asc,
}) => (
  <View style={styles.controls}>
    <ListFilter onFilter={onFilter} />
    <ListSort onSort={onSort} asc={asc} />
  </View>
);

ListControls.propTypes = {
  onFilter: PropTypes.func.isRequired,
  onSort: PropTypes.func.isRequired,
  asc: PropTypes.bool.isRequired,
};

export default ListControls;
```

This has probably been the simplest component so far in the example. It's bringing together the `ListFilter` and `ListSort` controls. So, if you were to add another list control, you would add it here. Let's take a look at the `ListFilter` implementation now:

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

import styles from './styles';

// Renders a "<TextInput>" component which allows the
// user to type in their filter text. This causes
// the "onFilter()" event handler to be called.
// This handler comes from "ListContainer" and changes
// the state of the list data source.
const ListFilter = ({ onFilter }) => (
  <View>
    <TextInput
      autoFocus
      placeholder="Search"
      style={styles.filter}
      onChangeText={onFilter}
    />
  </View>
);
```

```
) ;

ListFilter.propTypes = {
  onFilter: PropTypes.func.isRequired,
};

export default ListFilter;
```

The filter control is a simple text input that filters the list of items as user types. The `onChange` function that handles this comes all the way from the `ListContainer` component. The main thing to note about his component is how simple and obvious it is. There's no confusion about what it does; it calls some function when the user types in an input box.

The `ListSort` component has a similar simplicity to it:

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

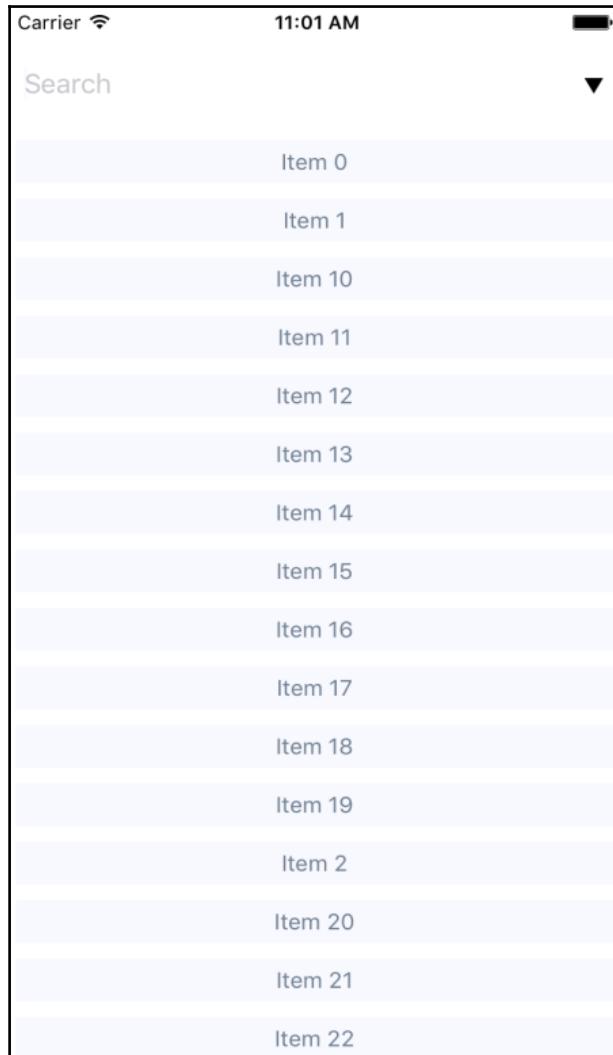
// The arrows to render based on the state of
// the "asc" property. Using a Map let's us
// stay declarative, rather than introducing
// logic into the JSX.
const arrows = new Map([
  [true, '▼'],
  [false, '▲'],
]);

// Renders the arrow text. When clicked, the
// "onSort()" function that's passed down from
// the container.
const ListSort = ({ onSort, asc }) => (
  <Text onPress={onSort}>{arrows.get(asc)}</Text>
);

ListSort.propTypes = {
  onSort: PropTypes.func.isRequired,
  asc: PropTypes.bool.isRequired,
};

export default ListSort;
```

Here's a look at the resulting list:



By default, the entire list is rendered in ascending order. You can see the placeholder text **Search** when the user hasn't provided anything yet. Let's see how this looks when we enter a filter and change the sort order:



This search includes items with 1 in it, and sorts the results in descending order. Note that you can either change the order first or enter the filter first. Both the filter and the sort order are part of the `ListContainer` state.

Fetching list data

Often, you'll fetch your list data from some API endpoint. In this section, you'll learn about making API requests from React Native components. The good news is that the `fetch()` API is polyfilled by React Native, so the networking code in your mobile applications should look and feel a lot like it does in your web applications.

To start things off, let's build a mock API for our list items using `fetch-mock`:

```
import fetchMock from 'fetch-mock';
import querystring from 'querystring';

// A mock item list...
const items = new Array(100)
  .fill(null)
  .map((v, i) => `Item ${i}`);

// The same filter and sort functionality
// as the previous example, only it's part of the
// API now, instead of part of the React component.
const filterAndSort = (data, text, asc) =>
  data.filter(
    i =>
      text.length === 0 ||
      i.includes(text)
  ).sort(
    asc ?
      (a, b) => b > a ? -1 : (a === b ? 0 : 1) :
      (a, b) => a > b ? -1 : (a === b ? 0 : 1)
  );

// Defines the mock handler for the "/items" URL.
fetchMock.mock(/\/items.*/, (url) => {
  // Gets the "filter" and "asc" parameters.
  const params = querystring.parse(url.split('?')[1]);

  // Performs the sorting and filtering before
  // responding.
  return ({
    items: filterAndSort(
      items,
      params.filter ? params.filter : '',
      !!params.asc
    ),
  });
});
```

With the mock API endpoint in place, let's make some changes to the list container component. Instead of using local data sources, you can now use the `fetch()` function to load data from the API mock:

```
import React, { Component } from 'react';
import { ListView } from 'react-native';

// Note that we're importing mock here to enable the API.
import './mock';
import List from './List';

const rowHasChanged = (r1, r2) => r1 !== r2;
const sectionHeaderHasChanged = rowHasChanged;

// Fetches items from the API using
// the given "filter" and "asc" arguments. The
// returned promise resolves a JavaScript object.
const fetchItems = (filter, asc) =>
  fetch(`/items?filter=${filter}&asc=${+asc}`)
    .then(resp => resp.json());

class ListContainer extends Component {
  constructor() {
    super();

    // The "source" state is empty because we need
    // to fetch the data from the API.
    this.state = {
      // data: [],
      asc: true,
      filter: '',
      source: new ListView
        .DataSource({
          rowHasChanged,
          sectionHeaderHasChanged,
        })
        .cloneWithRows([]),
    };
  }

  // When the component is first mounted, fetch the initial
  // items from the API, then
  componentDidMount() {
    fetchItems(this.state.filter, this.state.asc)
      .then(({ items }) => {
        this.setState({
          source: this.state.source.cloneWithRows(items),
        });
      });
  }
}
```

```
        });
    }

render() {
  return (
    <List
      source={this.state.source}
      asc={this.state.asc}
      onFilter={text => {
        // Makes an API call when the filter changes...
        fetchItems(text, this.state.asc)
          .then(({ items }) =>
            this.setState({
              filter: text,
              source: this.state.source.cloneWithRows(items),
            }));
      }}
      onSort={() => {
        // Makes an API call when the sort order
        // changes...
        fetchItems(this.state.filter, !this.state.asc)
          .then(({ items }) =>
            this.setState({
              asc: !this.state.asc,
              source: this.state.source.cloneWithRows(items),
            }));
      }}
    />
  );
}

export default ListContainer;
```

I think this looks a lot simpler now, despite the fact that it needs to reach out to the network in order to work. Any action that modifies the state of the list simply needs to call `fetchItems()`, and set the appropriate state once the promise resolves.

Lazy list loading

In this section, we'll implement a different kind of list, one that scrolls infinitely. Sometimes, users don't actually know what they're looking for, so filtering or sorting isn't going to help. Think about the Facebook news feed you see when you log into your account; it's the main feature of the application and rarely are you looking for something specific. You'll need to see what's going on by scrolling through the list.

To do this using a `ListView` component, you need to be able to fetch more API data when the user scrolls to the end of the list. To get an idea of how this works, we need a lot of API data to work with. Generators are great at this! So let's modify the mock we created in the previous example so that it just keeps responding with new data:

```
import fetchMock from 'fetch-mock';

// Items...keep'em coming!
function* genItems() {
  let cnt = 0;

  while (true) {
    yield `Item ${cnt++}`;
  }
}

const items = genItems();

// Grabs the next 20 items from the "items"
// generator, and responds with the result.
fetchMock.mock(/\/items.*/, () => {
  const result = [];

  for (let i = 0; i < 20; i++) {
    result.push(items.next().value);
  }

  return ({
    items: result,
  });
});
```

With this in place, you can now make an API request for new data every time the end of the list is reached. Well, eventually this will fail, but I'm just trying to show you in general terms the approach you can take to implement infinite scrolling in React Native. Here's what the `ListContainer` component looks like:

```
import React, { Component } from 'react';
import { ListView } from 'react-native';

import './mock';
import List from './List';

const rowHasChanged = (r1, r2) => r1 !== r2;
const sectionHeaderHasChanged = rowHasChanged;

class ListContainer extends Component {
  constructor() {
    super();

    this.state = {
      data: [],
      asc: true,
      filter: '',
      source: new ListView
        .DataSource({
          rowHasChanged,
          sectionHeaderHasChanged,
        })
        .cloneWithRows([]),
    };
  }

  // This function is passed to the "onEndReached"
  // property of the React Native "ListView" component.
  // Instead of replacing the "source", it concatenates
  // the new items with those that have already loaded.
  this.fetchItems = () =>
    fetch('/items')
      .then(resp => resp.json())
      .then(({ items }) =>
        this.setState({
          data: this.state.data.concat(items),
          source: this.state.source.cloneWithRows(
            this.state.data
          ),
        });
  };
}
```

```
// Fetches the first batch of items once the
// component is mounted.
componentDidMount() {
  this.fetchItems();
}

render() {
  return (
    <List
      source={this.state.source}
      fetchItems={this.fetchItems}
    />
  );
}

export default ListContainer;
```

Each time `fetchItems()` is called, the response is concatenated with the data array. This becomes the new list data source, instead of replacing it as you did in earlier examples. Now let's take a look at the `List` component to see how you respond to the end of the list being reached:

```
import React, { PropTypes } from 'react';
import { Text, ListView } from 'react-native';

import styles from './styles';

// Renders a "<ListView>" component, and
// calls "fetchItems()" and the user scrolls
// to the end of the list.
const List = ({
  source,
  fetchItems,
}) => (
  <ListView
    enableEmptySections
    dataSource={source}
    renderRow={i => (
      <Text style={styles.item}>{i}</Text>
    )}
    onEndReached={fetchItems}
  />
);

List.propTypes = {
  source: PropTypes.instanceOf(ListView.DataSource).isRequired,
  fetchItems: PropTypes.func.isRequired,
```

```
};  
  
export default List;
```

If you run this example, you'll see that, as you approach the bottom of the screen while scrolling, the list just keeps growing.

Summary

In this chapter, you learned about the `ListView` component in React Native. This component is general-purpose in that it doesn't impose any specific look for items that get rendered. Instead, the appearance of the list is up to you, while the `ListView` component helps with efficiently rendering a data source. The `ListView` component also provides a scrollable region for the items it renders.

You implemented an example that took advantage of section headers in list views. This is a good place to render static content such as list controls. You then learned about making network calls in React Native; it's just like using `fetch()` in any other web application. Finally, you implemented lazy lists that scroll infinitely, by only loading new items after they've scrolled to the bottom of what's already been rendered.

In the next chapter, you'll learn how to show the progress of things such as network calls.

17

Showing Progress

This chapter is all about communicating progress to the user. React Native has different components to handle the different types of progress that you want to communicate. We'll start with a short discussion on why we need to communicate progress like this in the first place. Then, we'll jump into implementing progress indicators and progress bars. After that, you'll see specific examples that show you how to use progress indicators with navigation while data loads, and using progress bars to communicate the current position in a series of steps.

Progress and usability

Imagine that you have a microwave oven that has no window and makes no sound. The only way to interact with it, is by pressing a button labeled cook. As absurd as this device sounds, it's what many software users are faced with—there's no indication of progress. Is the microwave cooking anything? If so, how do we know when it will be done?

Well, one way to improve the microwave situation is to add sound. This way, the user gets feedback after pressing the cook button. So, we've overcome one hurdle, but the user is still left guessing—where's my food? Before we go out of business, we had better add some sort of progress measurement display. A timer! Brilliant!

In all seriousness, it's not that UI programmers don't understand the basic principles of this usability concern; it's just that we have a stuff to get done and this sort of thing simply slips through the cracks in terms of priority. In React Native, there are components for giving the user indeterminate progress feedback, and for given precise progress measurements. It's always a good idea to make these things a top priority if you want a good user experience.

Indicating progress

In this section, you'll learn how to use the `<ActivityIndicator>` component. As the name suggests, you render this component when you need to indicate to the user that something is happening. The actual progress may be indeterminate, but at least you have a standardized means to show that something is happening, despite there being no results to display yet.

We'll create a super simple example just, so you can see what this component looks like. Here's the main module for the app:

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

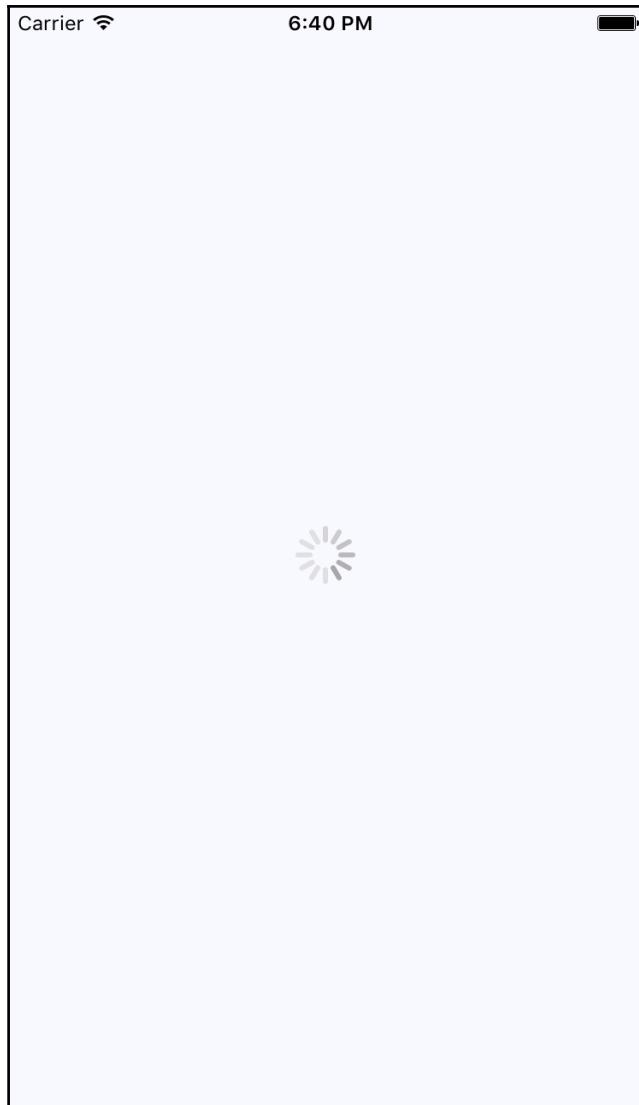
import styles from './styles';

// Renders an "<ActivityIndicator>" component in the
// middle of the screen. It will animate on it's own
// while displayed.
const IndicatingProgress = () => (
  <View style={styles.container}>
    <ActivityIndicator size="large" />
  </View>
);

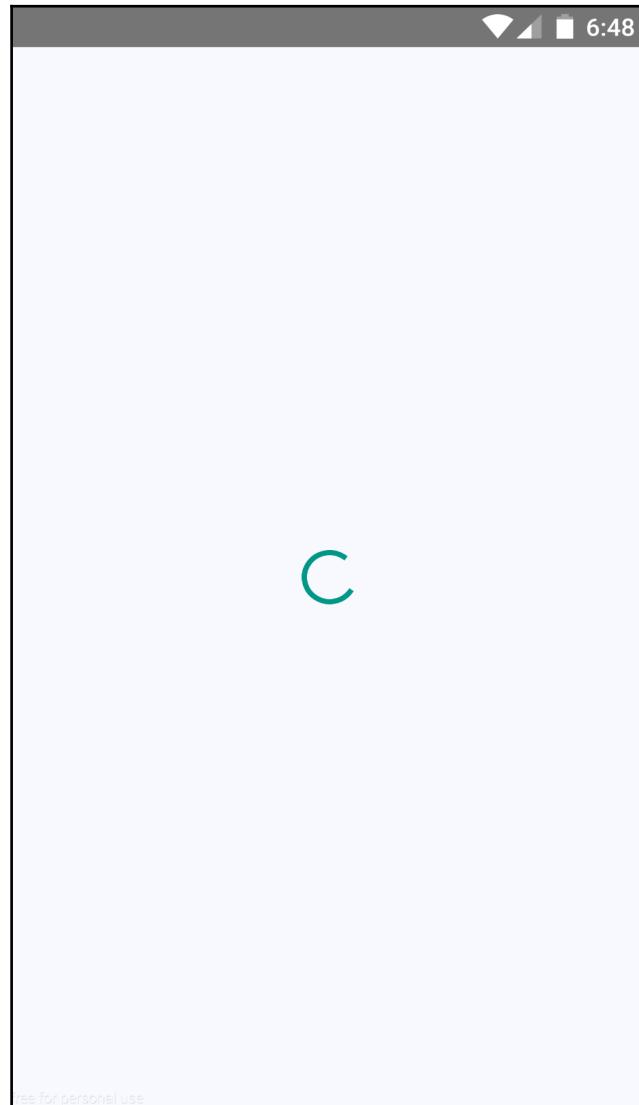
AppRegistry.registerComponent(
  'IndicatingProgress',
  () => IndicatingProgress
);
```

Showing Progress

The `<ActivityIndicator>` component is platform agnostic. Here's how it looks on iOS:



As you can see, this simply renders an animated spinner in the middle of the screen. This is the large spinner, as specified in the `size` property. The `ActivityIndicator` spinner can also be small, which makes more sense if you're rendering it inside another smaller element. Now let's take a look at how this looks on an Android device:



The spinner looks different, as it should, but our app conveys the same thing on both platforms—we're waiting for something.



This example just spins forever. Don't worry, there's a more realistic progress indicator example coming up that shows you how to work with navigation and loading API data.

Measuring progress

The downside of merely indicating that progress is being made is that there's no end in sight for the user. This leads to a feeling of unease, like when waiting for food in a microwave with no timer. When we know how much progress has been made, and how much is left to go, we feel better. This is why it's always better to use a deterministic progress bar whenever possible.

Unlike the `ActivityIndicator` component, there's no platform agnostic component in React Native for progress bars. So, we'll have to make one ourselves. We'll create a component that uses `<ProgressViewIOS>` on iOS and `<ProgressBarAndroid>` on Android.

Let's handle the cross-platform issues first. Remember, React Native knows to import the correct module based on its extension. So here's what our `ProgressBarComponent.ios.js` module looks like:

```
// Exports the "ProgressViewIOS" as the
// "ProgressBarComponent" component that
// our "ProgressBar" expects.
export {
  ProgressViewIOS as ProgressBarComponent,
} from 'react-native';

// There are no custom properties needed.
export const progressProps = {};
```

As you can see, we're directly exporting the `ProgressViewIOS` component from React Native. We're also exporting properties for the component that are specific to the platform. In this case, it's an empty object because there are no properties that are specific to `<ProgressViewIOS>`. Now, let's take a peek at the `ProgressBarComponent.android.js` module:

```
// Exports the "ProgressBarAndroid" component as
// "ProgressBarComponent" that our "ProgressBar"
// expects.
export {
  ProgressBarAndroid as ProgressBarComponent,
```

```
    } from 'react-native';

    // The "styleAttr" and "indeterminate" props are
    // necessary to make "ProgressBarAndroid" look like
    // "ProgressViewIOS".
    export const progressProps = {
      styleAttr: 'Horizontal',
      indeterminate: false,
    };
  
```

This module uses the exact same approach as the `ProgressBarComponent.ios.js` module. It exports the Android-specific component as well as Android-specific properties to pass to it. Now let's build the `ProgressBar` component that the application will use:

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

// Imports the "ProgressBarComponent" which is the
// actual react-native implementation. The actual
// component that's imported is platform-specific.
// The custom props in "progressProps" is also
// platform-specific.
import {
  ProgressBarComponent,
  progressProps,
} from './ProgressBarComponent'; // eslint-disable-line import/no-
unresolved

import styles from './styles';

// The "ProgressLabel" component determines what to
// render as a label, based on the boolean "label"
// prop. If true, then we render some text that shows
// the progress percentage. If false, we render nothing.
const ProgressLabel = ({ show, progress }) =>
  new Map([
    [true, (
      <Text style={styles.progressText}>
        {Math.round(progress * 100)}%
      </Text>
    )],
    [false, null],
  ])
  .get(show);
```

```
// Our generic progress bar component...
const ProgressBar = ({
  progress,
  label,
}) => (
  <View style={styles.progress}>
    <StatusLabel
      show={label}
      progress={progress}
    />
    { /* "<ProgressBarComponent>" is really a "<ProgressViewIOS>" or a "<ProgressBarAndroid>". */ }
    <ProgressBarComponent
      {...progressProps}
      style={styles.progress}
      progress={progress}
    />
  </View>
);
ProgressBar.propTypes = {
  progress: PropTypes.number.isRequired,
  label: PropTypes.bool.isRequired,
};
ProgressBar.defaultProps = {
  progress: 0,
  label: true,
};
export default ProgressBar;
```

We'll walk through what's going on in this module now, starting with the imports. The `ProgressBarComponent` and `progressProps` values are imported from our `ProgressBarComponent` module. React Native determines handles which module to import this from.

Next, we have the `StatusLabel` utility component. It figures out what label is rendered for the progress bar based on the `show` property. If false, nothing is rendered. If true, we render a `<Text>` component that displays the progress as a percentage.

Lastly, we have the `ProgressBar` component itself that our application will import and use. This simply renders the label and the appropriate progress bar component. It takes a `progress` property, which is a value between 0 and 1. Now let's put this component to use in the main application:

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

import styles from './styles';
import ProgressBar from './ProgressBar';

class MeasuringProgress extends Component {
  // Initially at 0% progress. Changing this state
  // updates the progress bar.
  state = {
    progress: 0,
  }

  componentDidMount() {
    // Continuously increments the "progress" state
    // every 300MS, until we're at 100%.
    const updateProgress = () => {
      this.setState({
        progress: this.state.progress + 0.01,
      });

      if (this.state.progress < 1) {
        setTimeout(updateProgress, 300);
      }
    };

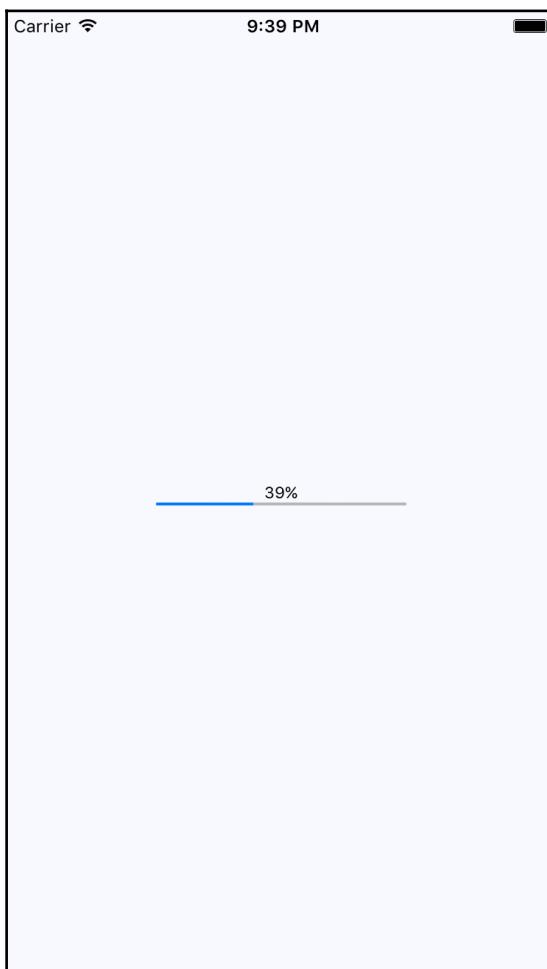
    updateProgress();
  }

  render() {
    return (
      <View style={styles.container}>
        { /* This is awesome. A simple generic
          "<ProgressBar>" component that works
          on Android and on iOS. */ }
        <ProgressBar
          progress={this.state.progress}
        />
      </View>
    );
  }
}
```

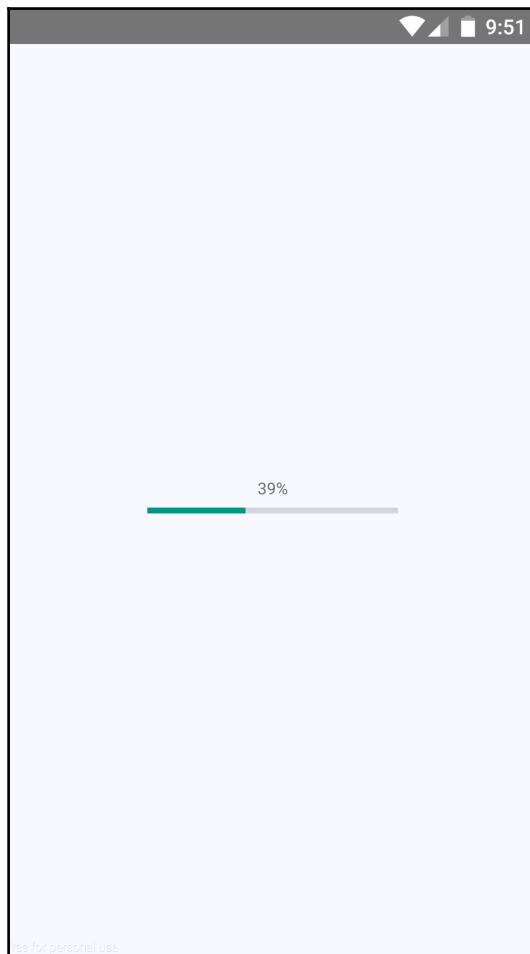
Showing Progress

```
) ;  
}  
}  
  
AppRegistry.registerComponent(  
  'MeasuringProgress',  
  () => MeasuringProgress  
) ;
```

Initially, the `<ProgressBar>` component is rendered at 0%. In the `componentDidMount()` method, we have an `updateProgress()` function that uses a timer to simulate a real process that we want to show progress for. Here's what the iOS screen looks like:



Here's what the same progress bar looks like on Android:



Navigation indicators

Earlier in the chapter, you were introduced to the `<ActivityIndicator>` component. In this section, you'll learn how it can be used when navigating an application that loads data. For example, the user navigates from page (scene) one to page two. However, page two needs to fetch data from the API to display for the user. So while this network call is happening, it makes more sense to display a progress indicator instead of a screen devoid of useful information.

Doing this is actually kind of tricky, because we have to make sure that the data required by the screen is fetched from the API each time the user navigates to the screen. So, we have two goals in mind here:

- Have the Navigator component automatically fetch API data for the scene that's about to be rendered.
- Use the promise that's returned by the API call as a means to display the spinner and hide it once the promise has resolved.

Since our scene components probably don't care about whether or not a spinner is displayed, let's implement this as a generic higher-order component:

```
import React, { Component, PropTypes } from 'react';
import {
  View,
  ActivityIndicator,
} from 'react-native';

import styles from './styles';

// Wraps the "Wrapped" component with a stateful component
// that renders an "<ActivityIndicator>" when the "loading"
// state is true.
const loading = Wrapped =>
  class LoadingWrapper extends Component {
    static propTypes = {
      promise: PropTypes.instanceOf(Promise),
    }

    state = {
      loading: true,
    }

    // Adds a callback to the "promise" that was
    // passed in. When the promise resolves, we set
    // the "loading" state to false.
    componentDidMount() {
      this.props.promise.then(
        () => this.setState({ loading: false }),
        () => this.setState({ loading: false })
      );
    }

    // If "loading" is true, render the "<ActivityIndicator>"
    // component. Otherwise, render the "<Wrapped>" component.
    render() {
      return new Map([

```

```
[true, (
  <View style={styles.container}>
    <ActivityIndicator size="large" />
  </View>
),
[false, (
  <Wrapped {...this.props} />
),
])
.get(this.state.loading);
};

export default loading;
```

This `loading()` function takes a component—the `Wrapped` argument and returns a `LoadingWrapper` component. The returned wrapper accepts a `promise` property, and when it's resolved, it changes the `loading` state to false. As you can see in the `render()` method, the `loading` state determines whether the spinner is rendered or the `Wrapped` component.

With the `loading()` higher-order function in place, let's take a look at one of our scene components to see how it's used:

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

import styles from '../styles';
import loading from '../loading';
import second from './second';
import third from './third';

// Renders links to other scenes...
const First = ({ navigator }) => (
  <View style={styles.container}>
    <Text
      style={styles.item}
      onPress={() => navigator.replace(second)}
    >
      Second
    </Text>
    <Text
      style={styles.item}
      onPress={() => navigator.replace(third)}
    >
      Third
    </Text>
  </View>
);
```

```
</View>
);

First.propTypes = {
  navigator: PropTypes.object.isRequired,
};

// Simulates a real "fetch()" call by returning a promise
// that's resolved after 1 second.
const fetchData = () => new Promise(
  resolve => setTimeout(resolve, 1000)
);

// The exported "Scene" component is composed with
// higher-order "loading()" function.
export default {
  Scene: loading(First),
  fetchData,
};
```

This module exports a `Scene` component and and a `fetchData()` function that talks to the API. The `loading()` function we created earlier is used here. It wraps the `First` component so that a spinner is displayed while the `fetchData()` promise is pending. The last step is getting that promise into the component whenever the user navigates to a given page. This happens in the `renderScene()` function in the main module:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Navigator,
} from 'react-native';

import first from './scenes/first';

// The "<route.Scene>" component gets a promise property
// passed to it, by calling "route.fetchData()". This
// promise is what controls the progress indicator display.
const renderScene = (route, navigator) => (
  <route.Scene
    promise={route.fetchData()}
    navigator={navigator}
  />
);

const NavigationIndicators = () => (
  <Navigator
    initialRoute={first}
```

```
    renderScene={renderScene}
  />
);

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

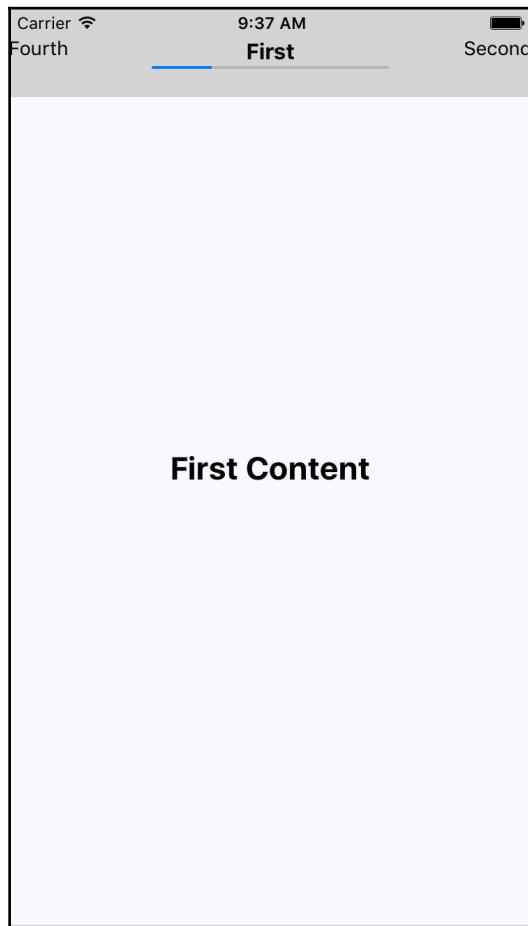
As you can see, the `fetchData()` function for any given route is called just before it's rendered, and this is how the `promise` property is set. Now when you navigate between screens, you'll see a spinner displayed in the middle of the screen that looks just like the first example in this chapter, until the promise resolves.

Step progress

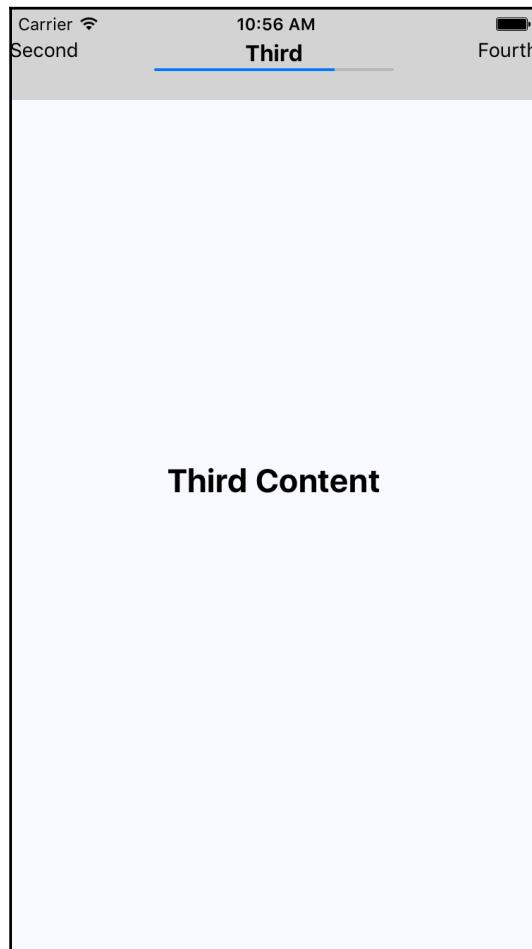
In this final example, we'll look at displaying the user's progress through a predefined number of steps. For example, it might make sense to split a form into several logical sections and organize them in such a way that as the user completes one section, they move to the next step. A progress bar would be helpful feedback for the user.

We're going to modify a navigation example from earlier in the book. We'll insert a progress bar into the navigation bar, just below the title so that the user knows how far they've gone and how far is left to go. We'll also reuse the `ProgressBar` component that you implemented earlier in this chapter!

Let's take a look at the result first. There are four screens in this app that the user can navigate to. Here's what the first page (scene) looks like:



The progress bar below the title reflects the fact that the user is 25% through the navigation. Let's see what the third screen looks like:



The progress is updated to reflect where the user is in the route stack. Let's take a look at the code required to make this happen:

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

```
import routes from './routes';
import styles from './styles';
import ProgressBar from './ProgressBar';

const renderScene = route => (<route.Scene />);

const routeMapper = {
  Title: (route, navigator) => (
    <View style={styles.progress}>
      <route.Title navigator={navigator} />
      { /* The "<ProgressBar>" component is rendered just
          below the title text. There's no progress label,
          just the bar itself. The "progress" itself is
          computed based on where the current route is
          in the route stack. */ }
      <ProgressBar
        label={false}
        progress={
          (routes.indexOf(route) + 1) / routes.length
        }
      />
    </View>
  ),
  LeftButton: (route, navigator) => (
    <route.LeftButton navigator={navigator} />
  ),
  RightButton: (route, navigator) => (
    <route.RightButton navigator={navigator} />
  ),
};

const navigationBar = (
  <Navigator.NavigationBar
    style={styles.nav}
    routeMapper={routeMapper}
  />
);

const StepProgress = () => (
  <Navigator
    initialRoute={routes[0]}
    initialRouteStack={routes}
    renderScene={renderScene}
    navigationBar={navigationBar}
  />
);

AppRegistry.registerComponent(
```

```
'StepProgress',
() => StepProgress
);
```

Take a look at the `Title` component in `routeMapper`. This is where the `<ProgressBar>` component is rendered. The actual progress value is based on where the current route is in the `routes` array. This determines the complete percentage of moving through the array.

Summary

In this chapter, you learned about how to show your users that something is happening behind the scenes. First, we discussed why showing progress is important for the usability of an application. Then, you implemented a basic screen that indicated progress was being made. You then implemented a `ProgressBar` component, used to measure specific progress amounts.

Indicators are good for indeterminate progress, and you implemented navigation that showed progress indicators while network calls were pending. In the final section, you implemented a progress bar that showed the user where they were in a predefined number of steps.

In the following chapter, you'll see React Native maps and geolocation data in action.

18

Geolocation and Maps

In this chapter, you'll learn about the geolocation and mapping capabilities of React Native. We'll start with using the geolocation API; then we'll move on to using the `MapView` component to plot points of interest and regions.

We'll rely on the `react-native-maps` package to implement maps (<https://github.com/airbnb/react-native-maps>). The goal of this chapter is to go over what's available in React Native for geolocation and React Native Maps for maps.

Where am I?

The geolocation API that web applications use to figure out where the user is located can also be used by React Native applications because the same API has been polyfilled. Outside of maps, this API is useful for getting precise coordinates from the GPS on mobile devices. We can then use this information to display meaningful location data to the user.

Unfortunately, the data that's returned by the geolocation API is of little use on its own; your code has to do the leg work to transform it into something useful. For example, latitude and longitude don't mean anything to the user, but we can use this data to lookup something that is of use to the user. This might be as simple as displaying where the user is currently located.

Let's implement an example that uses the geolocation API of React Native to look up coordinates and then use those coordinates to look up human-readable location information from the Google Maps API:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
```

```
    View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';

// For fetching human-readable address info.
const URL = 'https://maps.google.com/maps/api/geocode/json?latlng=';

class WhereAmI extends Component {
  // The "address" state is "loading..." initially because
  // it takes the longest to fetch.
  state = {
    data: fromJS({
      address: 'loading...',
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // We don't setup any geo data till the component
  // is ready to mount.
  componentWillMount() {
    const setPosition = (pos) => {
      // This component renders the "coords" data from
      // a geolocation response. This can simply be merged
      // into the state map.
      this.data = this.data.merge(pos.coords);

      // We need the "latitude" and the "longitude"
      // in order to lookup the "address" from the
      // Google Maps API.
      const {
        coords: {
          latitude,
          longitude,
        },
      } = pos;
    }
  }

  // Fetches data from the Google Maps API then sets
}
```

```
// the "address" state based on the response.
fetch(`${URL}${latitude},${longitude}`)
  .then(resp => resp.json(), e => console.error(e))
  .then(({{
    results: [
      { formatted_address },
    ],
  }) => {
  this.data = this.data
    .set('address', formatted_address);
});
};

// First, we try to lookup the current position
// data and update the component state.
navigator.geolocation.getCurrentPosition(setPosition);

// Then, we setup a high accuracy watcher, that
// issues a callback whenever the position changes.
this.watcher = navigator.geolocation.watchPosition(
  setPosition,
  err => console.error(err),
  { enableHighAccuracy: true }
);
}

// It's always a good idea to make sure that this
// "watcher" is cleared when the component is removed.
componentWillUnmount() {
  navigator.geolocation.clearWatch(this.watcher);
}

render() {
  // Since we want to iterate over the properties
  // in the state map, we need to convert the map
  // to pairs using "entries()". Then we need to
  // use the spread operator to make the map iterator
  // into a plain array. The "sort()" method simply
  // sorts the map based on it's keys.
  const state = [
    ...this.data
      .sortBy((v, k) => k)
      .entries(),
  ];
}

// Iterates over the state properties and renders them.
return (
  <View style={styles.container}>
```

```
        {state.map(([k, v]) => (
          <Text key={k} style={styles.label}>
            `${k[0].toUpperCase()}${k.slice(1)}`: {v}
          </Text>
        )));
      </View>
    );
}
}

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

The goal of this component is to render the properties returned by the geolocation API on the screen, as well as look up the user's specific location, and display it. If you take a look at the `componentWillMount()` method, you'll see that this is where most of the interesting code is. We've created a `setPosition()` function that's used as a callback in a couple of places. Its job is to set the state of our component.

First, it sets the `coords` properties. Normally, we wouldn't display this data directly, but this is an example that's showing the data that's available as part of the geolocation API. Second, it uses the `latitude` and `longitude` values to look up the name of where the user is currently, using the Google Maps API.

The `setPosition()` callback is used with `getCurrentPosition()`, which is only called once when the component is mounted. We're also using `setPosition()` with `watchPosition()`, which calls the callback any time the user's position changes.



The iOS emulator and Genymotion let you change locations via menu options. You don't have to install your app on a physical device every time you want to test changing locations.

Let's see what this screen looks like once the location data has loaded:



As you can see, the address information that we've fetched is probably more useful in an application than latitude and longitude data. Even better than physical address text is visualizing the user's physical location on a map; we'll cover this next.

What's around me?

The `MapView` component from `react-native-maps` is the main tool you'll use to render maps in your React Native applications.

Let's implement a basic `MapView` component to see what we get out of the box:

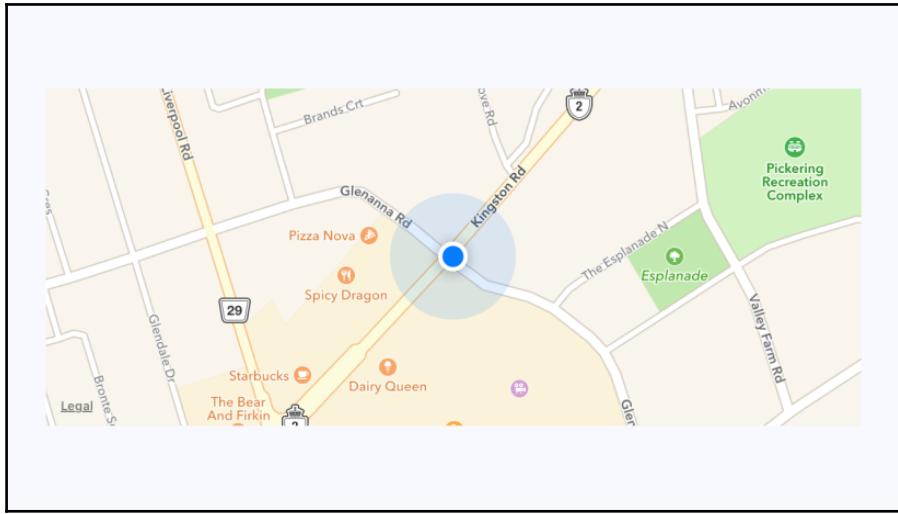
```
import React from 'react';
import {
  AppRegistry,
  View,
} from 'react-native';
import MapView from 'react-native-maps';

import styles from './styles';

const WhatsAroundMe = () => (
  <View style={styles.container}>
    <MapView
      style={styles.mapView}
      showsUserLocation
      followUserLocation
    />
  </View>
);

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

Not much to it. The two boolean properties that you've passed to `MapView` do a lot of work for you. The `showsUserLocation` property will activate the marker on the map that denotes the physical location of the device running this application. The `followUserLocation` property tells the map to update the location marker as the device moves around. Let's see the resulting map:



The current location of the device is clearly marked on the map. By default, points of interest are also rendered on the map. These are things in close proximity to the user so that they can see what's around them.

It's generally a good idea to use the `followUserLocation` property whenever using `showsUserLocation`. This makes the map zoom to region where the user is located.

Annotating points of interest

So far, you've seen how the `MapView` component can render the user's current location and points of interest around the user. The challenge here is that you probably want to show points of interest that are relevant to your application, instead of the points of interest that are rendered by default.

In this section, you'll learn how to render markers for specific locations on the map, as well as render regions on the map.

Plotting points

Let's plot some local breweries, shall we? Here's how you pass annotations to the `MapView` component:

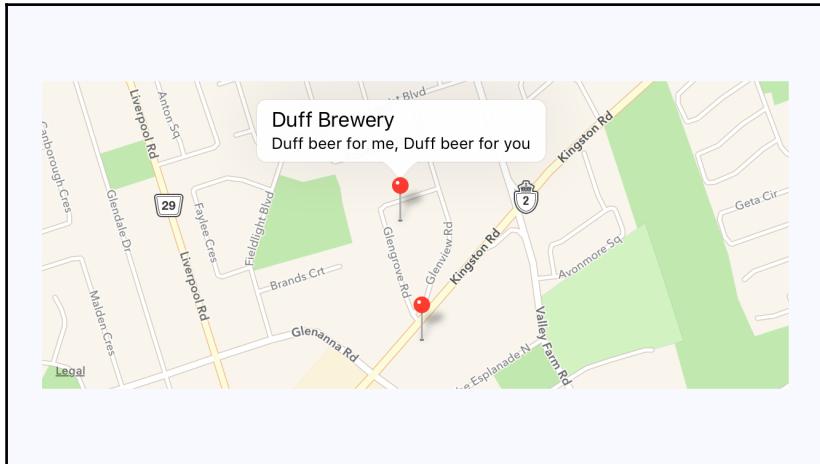
```
import React from 'react';
import {
  AppRegistry,
  View,
} from 'react-native';
import MapView from 'react-native-maps';

import styles from './styles';

const PlottingPoints = () => (
  <View style={styles.container}>
    <MapView
      style={styles.mapView}
      showsPointsOfInterest={false}
      showsUserLocation
      followUserLocation
    >
      <MapView.Marker
        description="Duff beer for me, Duff beer for you"
        coordinate={{
          latitude: 43.8418728,
          longitude: -79.086082,
        }}
      />
      <MapView.Marker
        description="New! Patriot Light!"
        coordinate={{
          latitude: -79.086082,
          longitude: -79.085407,
        }}
      />
    </MapView>
  </View>
);

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

Annotations are exactly what they sound like, additional information rendered on top of the basic map geography. In fact, you get annotations by default when you render `MapView` components because they will show points of interest. In this example, we've opted out of this capability by setting the `showsPointsOfInterest` property to false. The focus is now solely on the beer. Let's see where these breweries are located:



The callout is displayed when you press the marker that shows the location of the brewery on the map. The `title` and the `description` property values that you give to `<MapView.Marker>` are used to render this text.

Plotting overlays

In this last section of the chapter, you'll learn how to render region overlays. A point is a single latitude/longitude coordinate. Think of a region as a connect-the-dots drawing of several coordinates. Regions can serve many purposes, such as showing where we're more likely to find IPA drinkers versus stout drinkers. Here's what the code looks like:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
  Text,
} from 'react-native';
import MapView from 'react-native-maps';
import { fromJS } from 'immutable';

import styles from './styles';
```

```
// The "IPA" region coordinates and color...
const ipaRegion = {
  coordinates: [
    { latitude: 43.8486744, longitude: -79.0695283 },
    { latitude: 43.8537168, longitude: -79.0700046 },
    { latitude: 43.8518394, longitude: -79.0725697 },
    { latitude: 43.8481651, longitude: -79.0716377 },
    { latitude: 43.8486744, longitude: -79.0695283 },
  ],
  strokeColor: 'coral',
  strokeWidth: 4,
};

// The "stout" region coordinates and color...
const stoutRegion = {
  coordinates: [
    { latitude: 43.8486744, longitude: -79.0693283 },
    { latitude: 43.8517168, longitude: -79.0710046 },
    { latitude: 43.8518394, longitude: -79.0715697 },
    { latitude: 43.8491651, longitude: -79.0716377 },
    { latitude: 43.8486744, longitude: -79.0693283 },
  ],
  strokeColor: 'firebrick',
  strokeWidth: 4,
};

class PlottingOverlays extends Component {
  // The "IPA" region is rendered first. So the "ipaStyles"
  // list has "boldText" in it, to show it as selected. The
  // "overlays" list has the "ipaRegion" in it.
  state = {
    data: fromJS({
      ipaStyles: [styles.ipaText, styles.boldText],
      stoutStyles: [styles.stoutText],
      overlays: [ipaRegion],
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }
}
```

```
// The "IPA" text was clicked...
onClickIpa = () => {
  this.data = this.data
    // Makes the IPA text bold...
    .update('ipaStyles', i => i.push(styles.boldText))
    // Removes the bold from the stout text...
    .update('stoutStyles', i => i.pop())
    // Replaces the stout overlay with the IPA overlay...
    .update('overlays', i => i.set(0, ipaRegion));
}

// The "stout" text was clicked...
onClickStout = () => {
  this.data = this.data
    // Makes the stout text bold...
    .update('stoutStyles', i => i.push(styles.boldText))
    // Removes the bold from the IPA text...
    .update('ipaStyles', i => i.pop())
    // Replaces the IPA overlay with the stout overlay...
    .update('overlays', i => i.set(0, stoutRegion));
}

render() {
  const {
    ipaStyles,
    stoutStyles,
    overlays,
  } = this.data.toJS();

  return (
    <View style={styles.container}>
      <View>
        { /* Text that when clicked, renders the IPA
            map overlay. */ }
        <Text
          style={ipaStyles}
          onPress={this.onClickIpa}
        >
          IPA Fans
        </Text>

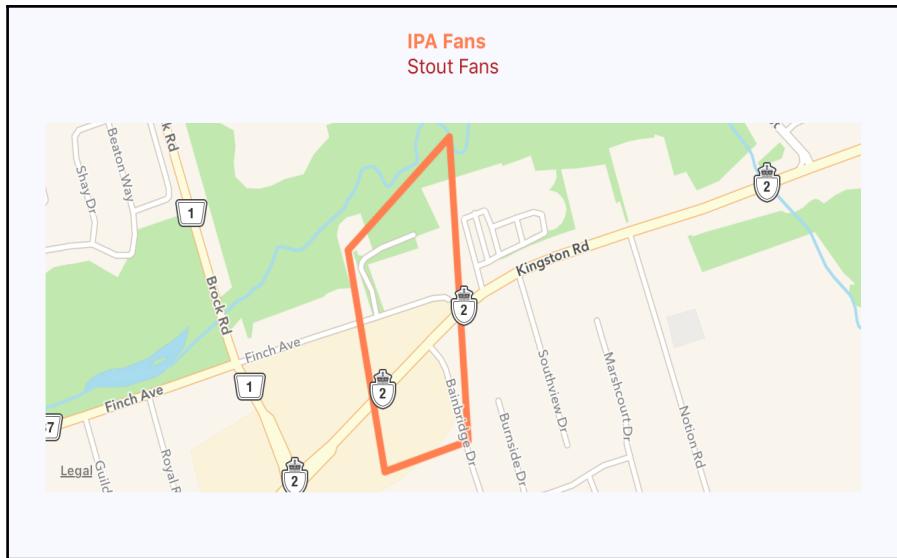
        { /* Text that when clicked, renders the stout
            map overlay. */ }
        <Text
          style={stoutStyles}
          onPress={this.onClickStout}
        >
          Stout Fans
        </Text>
      </View>
    </View>
  );
}
```

```
</Text>
</View>

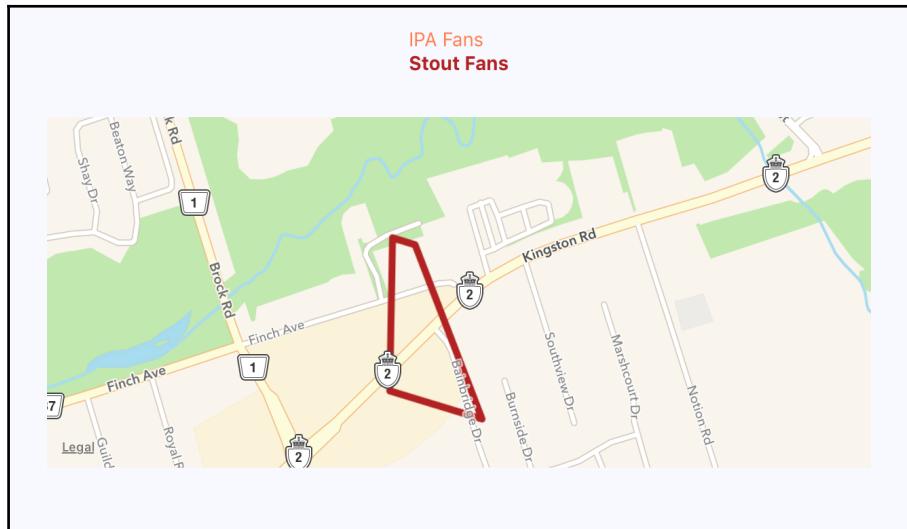
{ /* Renders the map with the "overlays" array.
   There will only ever be a single overlay
   in this array. */
<MapView
  style={styles.mapView}
  showsPointsOfInterest={false}
  showsUserLocation
  followUserLocation
>
  {overlays.map((v, i) => (
    <MapView.Polygon
      key={i}
      coordinates={v.coordinates}
      strokeColor={v.strokeColor}
      strokeWidth={v.strokeWidth}
    />
  )));
</MapView>
</View>
);
}

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

As you can see, the region data consists of several latitude/longitude coordinates that define the shape and location of the region. The rest of this code is mostly about handling state when the two text links are pressed. By default, the IPA region is rendered:



When the stout text is pressed, the IPA overlay is removed from the map and the stout region is added:



Summary

In this chapter, you learned about geolocation and mapping in React Native. The geolocation API works the same as its web counterpart. The only reliable way to use maps in React Native applications is to install the third-party `react-native-maps` package.

You saw the basic configuration `MapView` components, and how it can track the user's location, and show relevant points of interest. Then, you saw how to plot your own points of interest and regions of interest.

In the next chapter, you'll learn how to collect user input using React Native components that resemble HTML form controls.

19

Collecting User Input

In web applications, it's easy to collect user input because standard HTML form elements look and behave similar on all browsers. With native UI platforms, collecting user input is a little more nuanced.

In this chapter, you'll learn how to work with the various React Native components that are used to collect user input. These include text input, selecting from a list of options, checkboxes, and date/time selectors. You'll see the differences between iOS and Android, and how to implement the appropriate abstractions for your app.

Collecting text input

Collecting text input seems like something that should be super easy to implement, even in React Native. It's easy, but there's a lot to think about when implementing text inputs. For example, should it have placeholder text? Is this sensitive data that shouldn't be displayed on screen? Should we process text as it's entered, or when the user moves to another field? The list goes on, and I only have so much room in this book.

What's markedly different about mobile text input versus traditional web text input is that the former has its own built-in virtual keyboard that we can configure and respond to. So without further delay, let's get coding. We'll render several instances of the `<TextInput>` component:

```
import React, { Component, PropTypes } from 'react';
import {
  AppRegistry,
  Text,
  TextInput,
  View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';

// A Generic "<Input>" component that we can use in our app.
// It's job is to wrap the "<TextInput>" component in a "<View>"
// so that we can render a label, and to apply styles to the
// appropriate components.
const Input = props => (
  <View style={styles.textInputContainer}>
    <Text style={styles.textInputLabel}>
      {props.label}
    </Text>
    <TextInput
      style={styles.textInput}
      {...props}
    />
  </View>
);

Input.propTypes = {
  label: PropTypes.string,
};

class CollectingTextInput extends Component {
  // This state is only relevant for the "input events"
  // component. The "changedText" state is updated as
  // the user types while the "submittedText" state is
  // updated when they're done.
  state = {
    data: fromJS({
      changedText: '',
      submittedText: '',
    }),
  }
}
```

```
// Getter for "Immutable.js" state data...
get data() {
  return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}

render() {
  const {
    changedText,
    submittedText,
  } = this.data.toJS();

  return (
    <View style={styles.container}>
      {/* The simplest possible text input. */}
      <Input
        label="Basic Text Input:"
      />

      { /* The "secureTextEntry" property turns
          the text entry into a password input
          field. */}
      <Input
        label="Password Input:"
        secureTextEntry
      />

      { /* The "returnKeyType" property changes
          the return key that's displayed on the
          virtual keyboard. In this case, we want
          a "search" button. */
      <Input
        label="Return Key:"
        returnKeyType="search"
      />

      { /* The "placeholder" property works just
          like it does with web text inputs. */
      <Input
        label="Placeholder Text:"
        placeholder="Search"
      />

      { /* The "onChangeText" event is triggered as

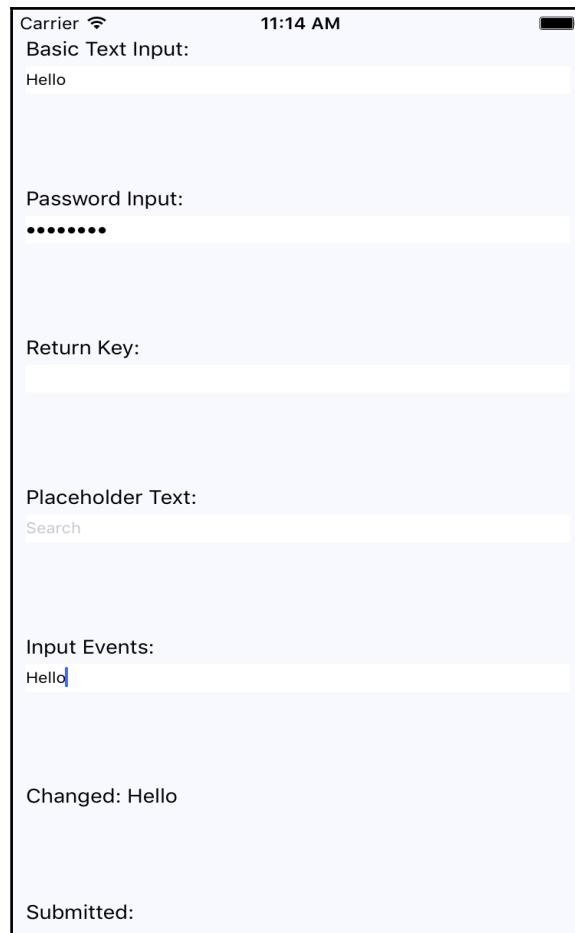
```

```
the user enters text. The "onSubmitEditing"
event is triggered when they click "search". */ }
<Input
  label="Input Events:"
  onChangeText={(e) => {
    this.data = this.data
      .set('changedText', e);
  }}
  onSubmitEditing={(e) => {
    this.data = this.data.set(
      'submittedText',
      e.nativeEvent.text
    );
  }}
  onFocus={() => {
    this.data = this.data
      .set('changedText', '')
      .set('submittedText', '');
  }}
/>

{ /* Displays the captured state from the
   "input events" text input component. */ }
<Text>Changed: {changedText}</Text>
<Text>Submitted: {submittedText}</Text>
</View>
);
}
}

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

I won't go into depth on what each of these `<TextInput>` components is doing—there are comments in the code. Let's see what these components look like on screen:



As you can see, the plain text input just shows the text that's been entered. The password field doesn't reveal any characters. The placeholder text is displayed when the input is empty. The changed text state is also displayed. You're not seeing the submitted text state, because I didn't press the submit button on the virtual keyboard before I took the screenshot.

Let's take a look at the actual virtual keyboard for the input element where we changed the return key:



When the keyboard return key reflects what's going to happen when they press it, the user feels more in tune with the application.

Selecting from a list of options

In web applications, you typically use the `<select>` element to let the user choose from a list of options. React Native comes with a `<Picker>` component that works on both iOS and Android. There is some trickery involved with styling this component, so let's hide all of this inside of a generic `Select` component:

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

// The "<Select>" component provides an
// abstraction around the "<Picker>" component.
// It actually has two outer views that are
// needed to get the styling right.
const Select = props => (
  <View style={styles.pickerHeight}>
    <View style={styles.pickerContainer}>
      {/* The label for the picker... */}
      <Text style={styles.pickerLabel}>
        {props.label}
```

```
</Text>
<Picker style={styles.picker} {...props}>
  { /* Maps each "items" value to a
       "<Picker.Item>" component. */
    props.items.map(i => (
      <Picker.Item key={i.label} {...i} />
    )));
  </Picker>
</View>
</View>
);

Select.propTypes = {
  items: PropTypes.array,
  label: PropTypes.string,
};

export default Select;
```

That's a lot of overhead for a simple `Select` component. Well, it turns out that it's actually quite hard to style the React Native `<Picker>` component. Here's what the styles look like:

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

export default StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'wrap',
    justifyContent: 'space-around',
    alignItems: 'center',
    backgroundColor: 'ghostwhite',
  },
  // The outermost container, needs a height.
  pickerHeight: {
    height: 175,
  },
  // The inner container lays out the picker
  // components and sets the background color.
  pickerContainer: {
    flex: 1,
    flexDirection: 'column',
    alignItems: 'center',
    marginTop: 40,
    backgroundColor: 'white',
    padding: 6,
```

```
        height: 240,  
    },  
  
    pickerLabel: {  
        fontSize: 14,  
        fontWeight: 'bold',  
    },  
  
    picker: {  
        width: 100,  
        backgroundColor: 'white',  
    },  
  
    selection: {  
        width: 200,  
        marginTop: 230,  
        textAlign: 'center',  
    },  
});
```

Now we can render our <Select> components:

```
import React, { Component } from 'react';  
import {  
    AppRegistry,  
    View,  
    Text,  
} from 'react-native';  
import { fromJS } from 'immutable';  
  
import styles from './styles';  
import Select from './Select';  
  
class SelectingOptions extends Component {  
    // The state is a collection of "sizes" and  
    // "garments". At any given time there can be  
    // selected size and garment.  
    state = {  
        data: fromJS({  
            sizes: [  
                { label: '', value: null },  
                { label: 'S', value: 'S' },  
                { label: 'M', value: 'M' },  
                { label: 'L', value: 'L' },  
                { label: 'XL', value: 'XL' },  
            ],  
            selectedSize: null,  
            garments: [  
            ]  
        })  
    };  
    // ...  
}
```

```
{ label: '', value: null, sizes: ['S', 'M', 'L', 'XL'] },
{ label: 'Socks', value: 1, sizes: ['S', 'L'] },
{ label: 'Shirt', value: 2, sizes: ['M', 'XL'] },
{ label: 'Pants', value: 3, sizes: ['S', 'L'] },
{ label: 'Hat', value: 4, sizes: ['M', 'XL'] },
],
availableGarments: [],
selectedGarment: null,
selection: '',
}),
}

// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

render() {
    const {
        sizes,
        selectedSize,
        availableGarments,
        selectedGarment,
        selection,
    } = this.data.toJS();

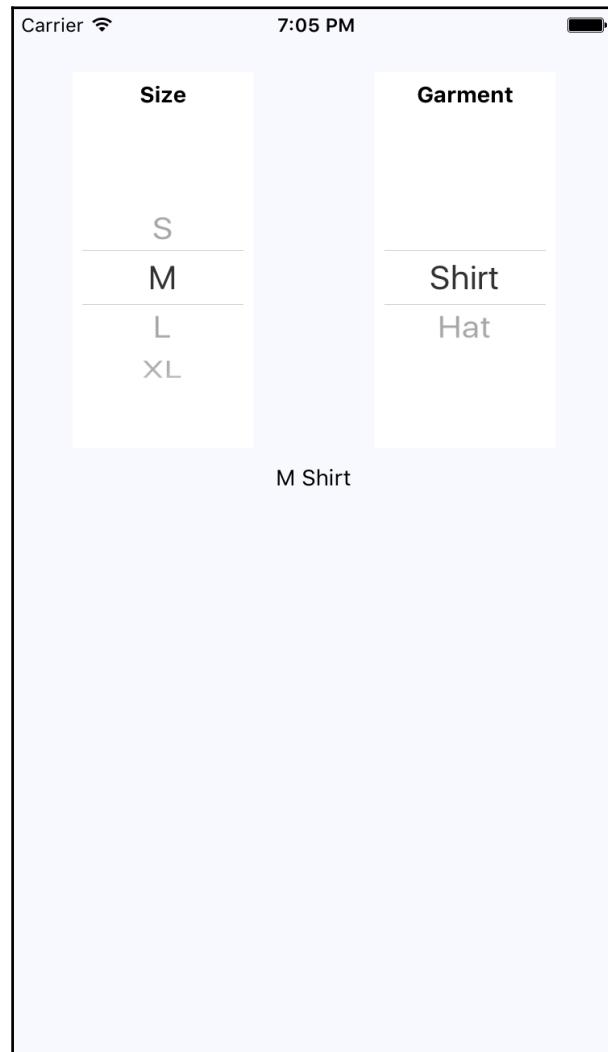
    // Renders two "<Select>" components. The first
    // one is a "size" selector, and this changes
    // the available garments to select from.
    // The second selector changes the "selection"
    // state to include the selected size
    // and garment.
    return (
        <View style={styles.container}>
            <Select
                label="Size"
                items={sizes}
                selectedValue={selectedSize}
                onValueChange={(size) => {
                    this.data = this.data
                        .set('selectedSize', size)
                        .set('selectedGarment', null)
                        .set('availableGarments',

```

```
        this.data.get('garments')
          .filter(
            i => i.get('sizes').includes(size)
          )
      );
    } }
/>
<Select
  label="Garment"
  items={availableGarments}
  selectedValue={selectedGarment}
  onValueChange={(garment) => {
    this.data = this.data
      .set('selectedGarment', garment)
      .set('selection',
        this.data.get('selectedSize') + ' ' +
        this.data.get('garments')
          .find(i => i.get('value') === garment)
          .get('label')
      );
  } }
/>
<Text style={styles.selection}>{selection}</Text>
</View>
);
}
}

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

The basic idea of this example is that the selected option in the first selector changes the available options in the second selector. When the second selector changes, the label shows the selected size and garment as a string. Here's how the screen looks:



Toggling between off and on

Another common element you'll see in web forms is checkboxes. React Native has a `Switch` component that works on both iOS and Android. Thankfully, this component is a little easier to style than the `Picker` component. Here's a look at a simple abstraction you can implement to provide labels for your switches:

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

import styles from './styles';

// A fairly straightforward wrapper component
// that adds a label to the React Native
// "<Switch>" component.
const CustomSwitch = props => (
  <View style={styles.customSwitch}>
    <Text>{props.label}</Text>
    <Switch {...props} />
  </View>
);

CustomSwitch.propTypes = {
  label: PropTypes.string,
};

export default CustomSwitch;
```

Now let's see how we can use a couple of switches to control application state:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import Switch from './Switch';

class TogglingOnAndOff extends Component {
  state = {
    data: fromJS({
```

```
        first: false,
        second: false,
    }),
}

// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

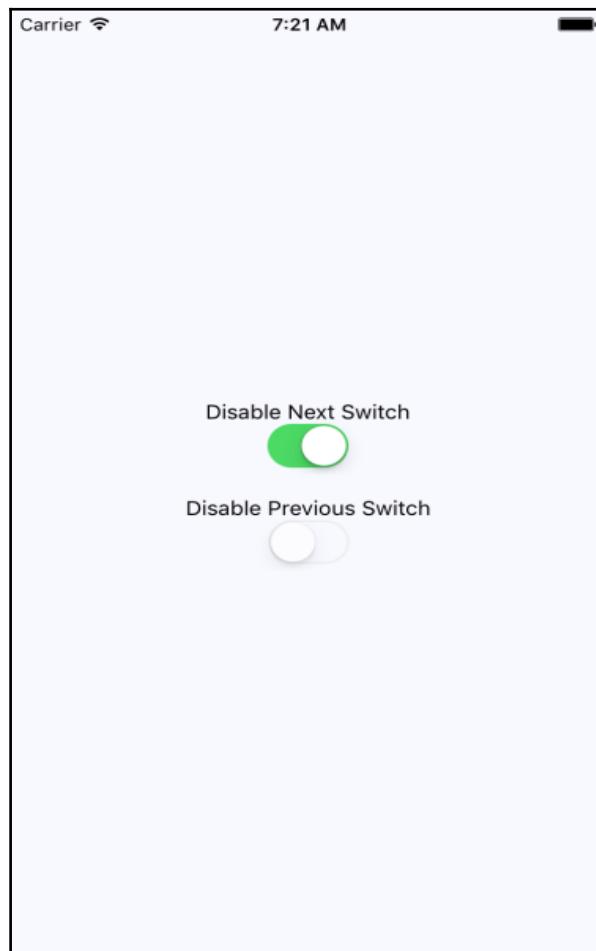
// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

render() {
    const {
        first,
        second,
    } = this.state.data.toJS();

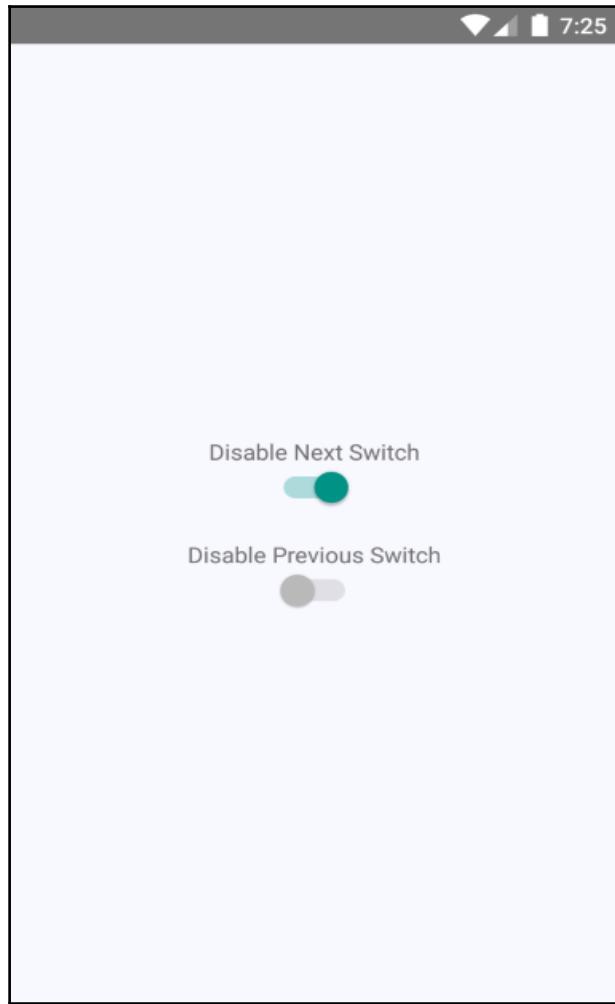
    return (
        <View style={styles.container}>
            { /* When this switch is turned on, the
                second switch is disabled. */ }
            <Switch
                label="Disable Next Switch"
                value={first}
                disabled={second}
                onValueChange={(v) => {
                    this.data = this.data.set('first', v);
                }}
            />
            { /* When this switch is turned on, the
                first switch is disabled. */ }
            <Switch
                label="Disable Previous Switch"
                value={second}
                disabled={first}
                onValueChange={(v) => {
                    this.data = this.data.set('second', v);
                }}
            />
        </View>
    );
}
}
```

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

These two switches simply toggle the `disabled` property of one another. Here's what the screen looks like in iOS:



Here's what the same screen looks like on Android:



Collecting date/time input

In this final section of the chapter, you'll learn how to implement date/time pickers. React Native has independent date/time picker components for iOS and Android, which means that it is up to you to handle the cross-platform differences between the components.

So, let's start with a date picker component for iOS:

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

import styles from './styles';

// A simple abstraction that adds a label to
// the "<DatePickerIOS>" component.
const DatePicker = (props) => (
  <View style={styles.datePickerContainer}>
    <Text style={styles.datePickerLabel}>
      {props.label}
    </Text>
    <DatePickerIOS mode="date" {...props} />
  </View>
);

DatePicker.propTypes = {
  label: PropTypes.string,
};

export default DatePicker;
```

There's not a lot to this component; it simply adds a label to the `DatePickerIOS` component. The Android version of our date picker needs a little more work. Let's take a look at the implementation:

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

import styles from './styles';
```

```
// Opens the "DatePickerAndroid" dialog and handles
// the response. The "onDateChange" function is
// a callback that's passed in from the container
// component and expects a "Date" instance.
const pickDate = (options, onDateChange) => {
  DatePickerAndroid.open(options)
    .then(date => onDateChange(new Date(
      date.year,
      date.month,
      date.day
    )));
};

// Renders a "label" and the "date" properties.
// When the date text is clicked, the "pickDate()"
// function is used to render the Android
// date picker dialog.
const DatePicker = ({
  label,
  date,
  onDateChange,
}) => (
  <View style={styles.datePickerContainer}>
    <Text style={styles.datePickerLabel}>
      {label}
    </Text>
    <Text
      onPress={() => pickDate(
        { date },
        onDateChange
      )}
    >
      {date.toLocaleDateString()}
    </Text>
  </View>
);

DatePicker.propTypes = {
  label: PropTypes.string,
  date: PropTypes.instanceOf(Date),
  onDateChange: PropTypes.func.isRequired,
};

export default DatePicker;
```

The key difference between the two date pickers is that the Android version doesn't use a React Native component, such as `DatePickerIOS`. Instead, we have to use the imperative `DatePickerAndroid.open()` API. This is triggered when the user presses the date text that our component renders, and opens a date picker dialog. The good news is that this component of ours hides this API behind a declarative component.



I've also implemented a time picker component that follows this exact pattern. So rather than listing that code here, I suggest that you download the code for this book from <https://github.com/PacktPublishing/React-and-React-Native>, so that you can see the subtle differences and run the example.

Now let's see how to use our date and time picker components:

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

import styles from './styles';

// Imports our own platform-independent "DatePicker"
// and "TimePicker" components.
import DatePicker from './DatePicker';
import TimePicker from './TimePicker';

class CollectingDateTimeInput extends Component {
  state = {
    date: new Date(),
    time: new Date(),
  }

  setDate = (date) => {
    this.setState({ date });
  }

  setTime = (time) => {
    this.setState({ time });
  }

  render() {
    const {
      setDate,
      setTime,
      state: {
        date,
        time,
      }
    } = this.state;
    return (
      <View>
        <Text>Date:</Text>
        <Text>{date}</Text>
        <Text>Time:</Text>
        <Text>{time}</Text>
        <Text>Set Date:</Text>
        <Text>Set Time:</Text>
      </View>
    );
  }
}
```

```
        time,
    },
} = this;

// Pretty self-explanatory - renders a "<DatePicker>"  

// and a "<TimePicker>". The date/time comes from the  

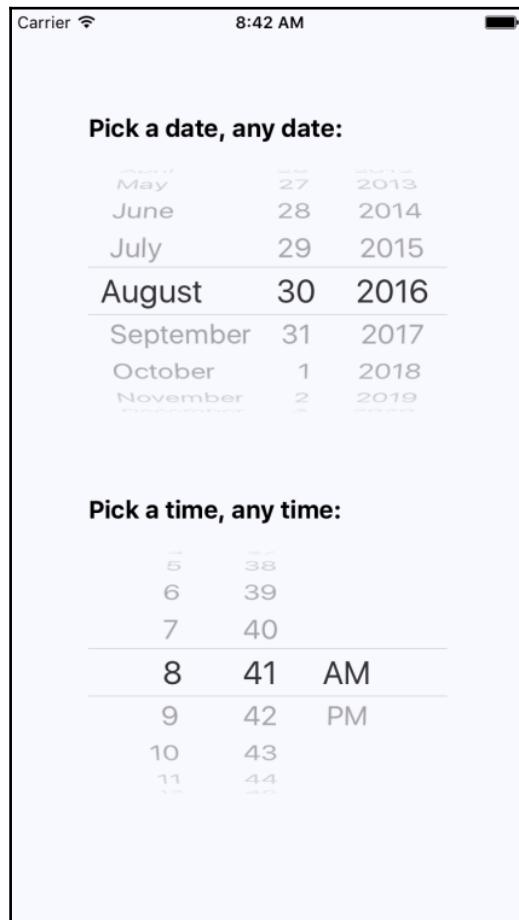
// state of this component and when the user makes a  

// selection, the "setDate()" or "setTime()" function  

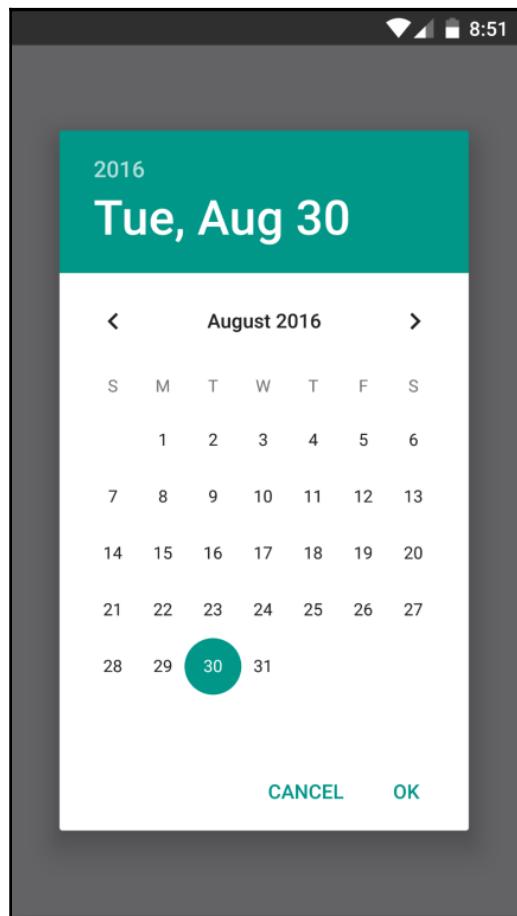
// is called.
return (
    <View style={styles.container}>
        <DatePicker
            label="Pick a date, any date:"
            date={date}
            onDateChange={setDate}
        />
        <TimePicker
            label="Pick a time, any time:"
            date={time}
            onTimeChange={setTime}
        />
    </View>
);
}

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

Awesome! Now we have two simple components that work on iOS and Android. Let's see how the pickers look on iOS:



As you can see, the iOS date and time pickers use the `Picker` component that you learned about earlier in this chapter. The Android picker looks a lot different—let's look at it now:



Summary

In this chapter, you learned about the various React Native components that resemble the form elements from the Web that you're used to. You started off by learning about text input, and how each text input has its own virtual keyboard to take into consideration. Next, you learned about `Picker` components that allow the user to select an item from a list of options. Then, you learned about `Switch` components that are kind of like checkboxes.

In the final section, you learned how to implement generic date/time pickers that work on both iOS and Android. In the next chapter, you'll learn about modal dialogs in React Native.

20

Alerts, Notifications, and Confirmation

The goal of this chapter is to show you how to present information to the user in unconventional ways. The conventional approach is to use a `View` component, and render this directly on the screen. There are times, however, when there's important information that the user needs to see, but we don't necessarily want to kick them off the current page.

We'll get started with a quick discussion on important information. Knowing what important information is and when to use it, you'll see how to get user acknowledgement—both for error and success scenarios. Then, we'll implement some passive notifications that show the user that something has happened. Finally, we'll implement modal views that show the user that something is happening in the background.

Important information

Before we dive into implementing alerts, notifications, and confirmations, let's take a few minutes and think about what each of these items mean. I think this is important, because if you end up passively notifying the user about an error it can easily get missed, and this is not something you want to happen. So, here are my definitions of these items:

- **Alert:** Something important just happened and we need to ensure that the user sees what's going on. Possibly, the user needs to acknowledge the alert.
- **Notification:** Something happened but it's not important enough to completely block what the user is doing. These typically go away on their own.

Confirmation is actually part of an alert. For example, if the user has just performed an action, and then wants to make sure that they know if it was successful before carrying on, they would have to confirm that they've seen the information in order to close the modal. A confirmation could also exist within an alert, warning the user about an action that they're about to perform.

The trick is to try to use notifications where the information is good to know, but not critical. Use confirmations only when the workflow of the feature cannot continue without the user acknowledging what's going on. In the following sections, you'll see examples of alerts and notifications used for different purposes.

Getting user confirmation

In this section, you'll learn how to show modal views in order to get confirmation from the user. We'll start with the successful scenario, where an action generates a successful outcome that we want the user to be aware of. Then we'll look at the error scenario, where something went wrong and you don't want the user to move forward without acknowledging the issue.

Success confirmation

Let's start by implementing a modal view that's displayed as the result of the user successfully performing an action. Here's the `Modal` component that's used to show the user a success confirmation:

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

import styles from './styles';

// Uses "<Modal>" to display the underlying view
// on top of the current view. Properties passed to
// this component are also passed to the modal.
const ConfirmationModal = props => (
  <Modal {...props}>
    { /* Slightly confusing, but we need an inner and
      an outer "<View>" to style the height of the
      modal correctly. */ }
  
```

```
<View style={styles.modalContainer}>
  <View style={styles.modalInner}>
    {/* The confirmation message... */}
    <Text style={styles.modalText}>Dude, srsly?</Text>

    { /* The confirmation and the cancel buttons. Each
        button triggers a different callback function
        that's passed in from the container
        component. */}
    <Text
      style={styles.modalButton}
      onPress={props.onPressConfirm}>
    >
      Yep
    </Text>
    <Text
      style={styles.modalButton}
      onPress={props.onPressCancel}>
    >
      Nope
    </Text>
  </View>
</View>
</Modal>
);

ConfirmationModal.propTypes = {
  visible: PropTypes.bool.isRequired,
  onPressConfirm: PropTypes.func.isRequired,
  onPressCancel: PropTypes.func.isRequired,
};

ConfirmationModal.defaultProps = {
  transparent: true,
  onRequestClose: () => {},
};

export default ConfirmationModal;
```

As you can see, the properties that are passed to `ConfirmationModal` are forwarded to the React Native `Modal` component. You'll see why in a moment. First, let's see what this confirmation modal looks like:



The modal that's displayed once the user completes an action has our own styling and confirmation message. It also has two actions, but it might only need one, depending on whether this confirmation is pre- or post-action. Here are the styles used for this modal:

```
modalContainer: {  
  flex: 1,  
  justifyContent: 'center',  
  alignItems: 'center',  
},  
  
modalInner: {
```

```
    backgroundColor: 'azure',
    padding: 20,
    borderWidth: 1,
    borderColor: 'lightsteelblue',
    borderRadius: 2,
    alignItems: 'center',
  },
  modalText: {
    fontSize: 16,
    margin: 5,
    color: 'slategrey',
  },
  modalButton: {
    fontWeight: 'bold',
    margin: 5,
    color: 'slategrey',
  },
},
```

With the React Native `Modal` component, it's pretty much up to you how you want your confirmation modal view to look. Think of them as regular views, with the only difference being that they're rendered on top of other views.

A lot of the time you might not care to style your own modal views. For example, in web browsers you can simply call the `alert()` function, which shows text in a window that's styled by the browser. React Native has something similar: `Alert.alert()`. The tricky part here is that this is an imperative API, and you don't necessarily want to expose it directly in your application.

Instead, let's implement an alert confirmation component that hides the details of this particular React Native API so that your app can just treat this like any other component:

```
import React, { Component, PropTypes } from 'react';
import {
  Alert,
} from 'react-native';

// The "actions" Map will map the "visible"
// property to the "Alert.alert()" function,
// or to a noop function.
const actions = new Map([
  [true, Alert.alert],
  [false, () => {}],
]);

class ConfirmationAlert extends Component {
```

```
// When the component is mounted, show the
// alert if "visible" is true. Otherwise,
// this is a noop.
componentDidMount() {
  actions.get(this.props.visible) (
    this.props.title,
    this.props.message,
    this.props.buttons,
  );
}

// When the "visible" property value changes,
// display the alert if the "visible"
// property is true.
componentWillReceiveProps(props) {
  actions.get(props.visible) (
    props.title,
    props.message,
    props.buttons,
  );
}

// Since an imperative API is used to display
// the alert, we don't actually render anything.
render() {
  return null;
}
}

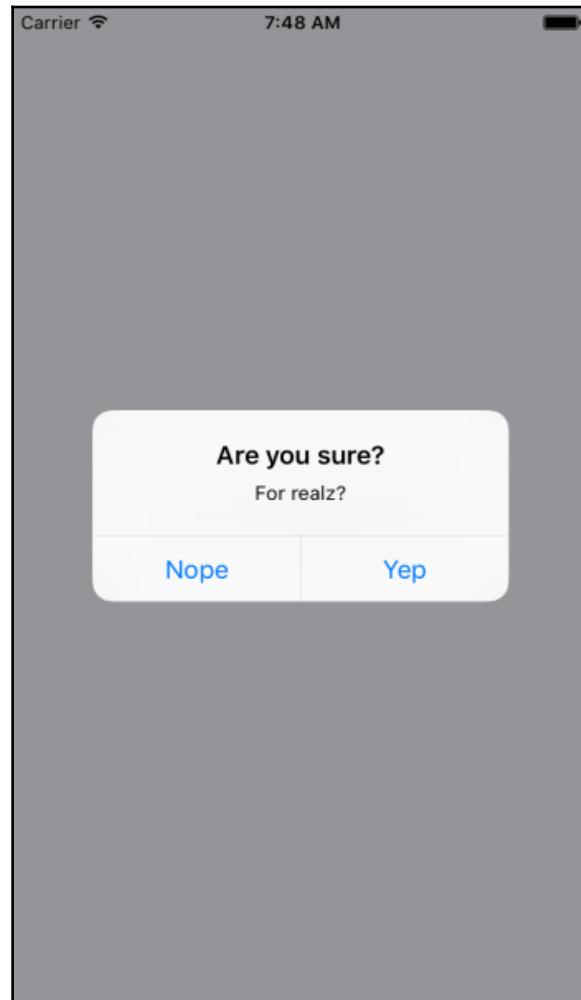
ConfirmationAlert.propTypes = {
  visible: PropTypes.bool.isRequired,
  title: PropTypes.string,
  message: PropTypes.string,
  buttons: PropTypes.array,
};

export default ConfirmationAlert;
```

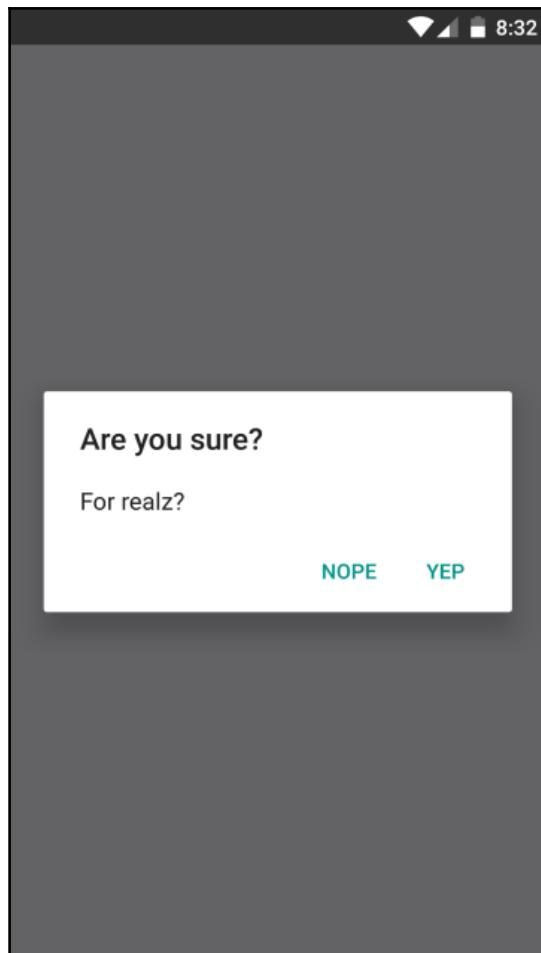
There are two important aspects to this component. First, take a look at the `actions` map. Its keys—`true` and `false`—correspond to the `visible` property value. The values correspond to the imperative `alert()` API and a noop function. This is the key to translating the declarative React component interface we know and love into something that's hidden from view.

Second, note that the `render()` method doesn't need to render anything since this component deals exclusively with imperative React Native calls. But, it feels like something is being rendered to the person that's using `ConfirmationAlert`. Cool, right?

Here's what the alert looks like on iOS:



In terms of functionality, there's nothing really different here. There's a title and text beneath it, but that's something that can easily be added to a modal view if you wanted. The real difference is that this modal looks like an iOS modal, instead of something that's styled by the app. Let's see how this alert looks on Android:



This modal looks like an Android modal, and we didn't have to style it. I think using alerts over modals is a better choice most of the time. It makes sense to have something styled to look like it's part of iOS or part of Android. However, there are times when you need more control over how the modal looks, such as when displaying error confirmations. We'll look at these next. But first, here's the code that's used to display both the modal and the alert confirmation dialogs:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
  Text,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import ConfirmationModal from './ConfirmationModal';
import ConfirmationAlert from './ConfirmationAlert';

class SuccessConfirmation extends Component {
  // The two pieces of state used to control
  // the display of the modal and the alert
  // views.
  state = {
    data: fromJS({
      modalVisible: false,
      alertVisible: false,
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // A "modal" button was pressed. So show
  // or hide the modal based on its current state.
  toggleModal = () => {
    this.data = this.data
      .update('modalVisible', v => !v);
  }
}
```

```
// A "alert" button was pressed. So show
// or hide the alert based on its current state.
toggleAlert = () => {
  this.data = this.data
    .update('alertVisible', v => !v);
}

render() {
  const {
    modalVisible,
    alertVisible,
  } = this.data.toJS();

  const {
    toggleModal,
    toggleAlert,
  } = this;

  return (
    <View style={styles.container}>
      { /* Renders the "<ConfirmationModal>" component,
          which is hidden by default and controlled
          by the "modalVisible" state. */ }
      <ConfirmationModal
        animationType="fade"
        visible={modalVisible}
        onPressConfirm={toggleModal}
        onPressCancel={toggleModal}
      />

      { /* Renders the "<ConfirmationAlert>" component,
          which doesn't actually render anything since
          it controls an imperative API under the hood.
          The "alertVisible" state controls this API. */ }
      <ConfirmationAlert
        message="For realz?"
        visible={alertVisible}
        buttons={[
          {
            text: 'Nope',
            onPress: toggleAlert,
          },
          {
            text: 'Yep',
            onPress: toggleAlert,
          },
        ]}
    />
  )
}
```

```
{ /* Shows the "<ConfirmationModal>" component
   by changing the "modalVisible" state. */ }
<Text
  style={styles.text}
  onPress={toggleModal}
>
  Show Confirmation Modal
</Text>

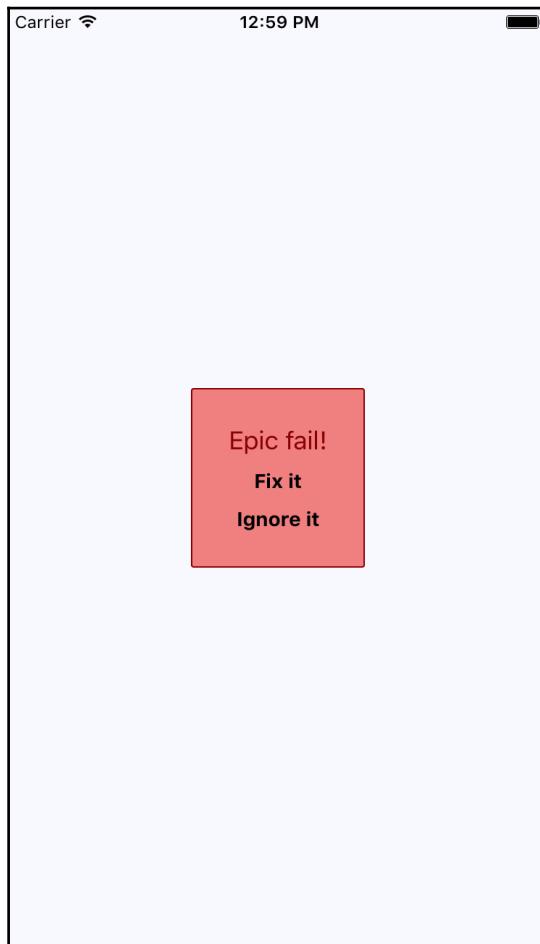
{ /* Shows the "<ConfirmationAlert>" component
   by changing the "alertVisible" state. */ }
<Text
  style={styles.text}
  onPress={toggleAlert}
>
  Show Confirmation Alert
</Text>
</View>
);
}

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

As you can see, the approach to rendering modals is different to the approach to rendering alerts. However, they're both still declarative components that change based on changing property values, which is key.

Error confirmation

All of the principles you learned about in the preceding section are applicable when you need the user to acknowledge an error. If you need more control of the display, use a modal. For example, you might want the modal to be red and scary looking:



Here are the styles used to create this look. Maybe you want something a little more subtle, but the point is that you can make this look however you want:

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

export default StyleSheet.create({
  container: {
```

```
flex: 1,
justifyContent: 'center',
alignItems: 'center',
backgroundColor: 'ghostwhite',
},

text: {
  color: 'slategrey',
},

modalContainer: {
  flex: 1,
  justifyContent: 'center',
  alignItems: 'center',
},
modalInner: {
  backgroundColor: 'azure',
  padding: 20,
  borderWidth: 1,
  borderColor: 'lightsteelblue',
  borderRadius: 2,
  alignItems: 'center',
},
modalInnerError: {
  backgroundColor: 'lightcoral',
  borderColor: 'darkred',
},
modalText: {
  fontSize: 16,
  margin: 5,
  color: 'slategrey',
},
modalTextError: {
  fontSize: 18,
  color: 'darkred',
},
modalButton: {
  fontWeight: 'bold',
  margin: 5,
  color: 'slategrey',
},
modalButtonError: {
```

```
        color: 'black',
    },
});
```

The same modal styles that you used for the success confirmations are still here. That's because the error confirmation modal needs many of the same styles. Here's how you apply both to the `Modal` component:

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

import styles from './styles';

// Declares styles for the error modal by
// combining regular modal styles with
// error styles.
const innerViewStyle = [
  styles.modalInner,
  styles.modalInnerError,
];

const textStyle = [
  styles.modalText,
  styles.modalTextError,
];

const buttonStyle = [
  styles.modalButton,
  styles.modalButtonError,
];

// Just like a success modal, accept for the addition of
// error styles.
const ErrorModal = props => (
  <Modal {...props}>
    <View style={styles.modalContainer}>
      <View style={innerViewStyle}>
        <Text style={textStyle}>
          Epic fail!
        </Text>
        <Text
          style={buttonStyle}
          onPress={props.onPressConfirm}
        >
```

```
        Fix it
      </Text>
      <Text
        style={buttonStyle}
        onPress={props.onPressCancel}
      >
        Ignore it
      </Text>
    </View>
  </View>
</Modal>
);

ErrorModal.propTypes = {
  visible: PropTypes.bool.isRequired,
  onPressConfirm: PropTypes.func.isRequired,
  onPressCancel: PropTypes.func.isRequired,
};

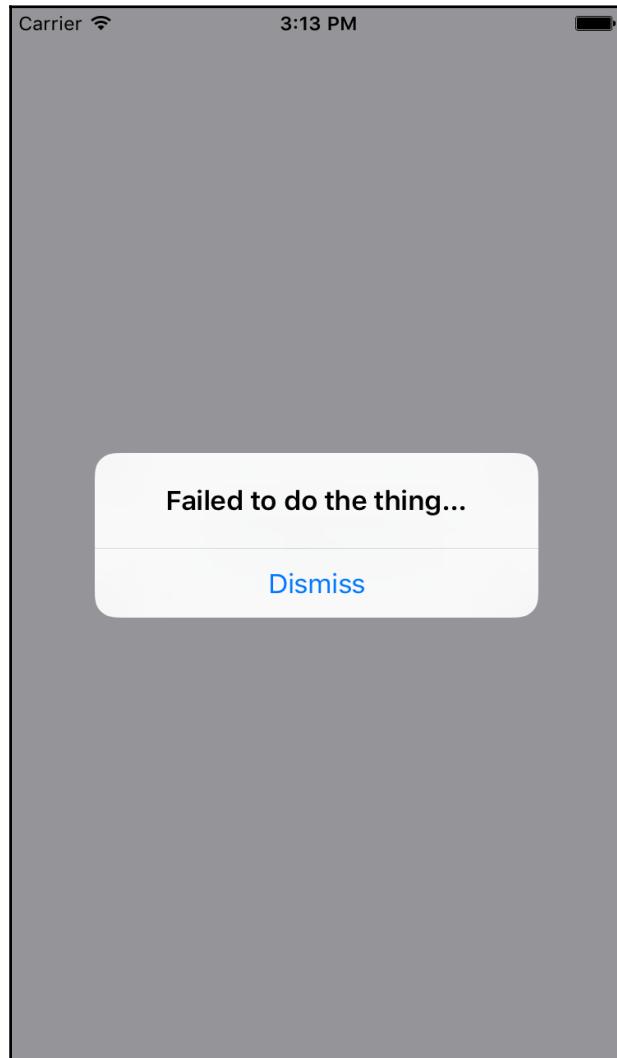
ErrorModal.defaultProps = {
  transparent: true,
  onRequestClose: () => {},
};

export default ErrorModal;
```

The thing to notice here is how the styles are combined as arrays before they're passed to the `style` property. The error styles always come last since conflicting style properties, such as `backgroundColor`, will be overridden by whatever comes later in the array.

In addition to styles in error confirmations, you can include whatever advanced controls you want. It really depends on how your application lets users cope with errors. For example, maybe there are several courses of action that can be taken.

However, the more common case is that something went wrong and there's nothing we can do about it, besides making sure that the user is aware of the situation. In these cases, you can probably get away with just displaying an alert:



Passive notifications

The notifications you've examined so far in this chapter have all required input from the user. This is by design because it's important information that we're forcing the user to look at. You don't want to over-do this, however. For notifications that are important but not life-altering if ignored, you can use passive notifications. These are displayed in a less obtrusive way than modals, and don't require any user action to dismiss.

In this section, we'll create a `Notification` component that uses the `Toast` API for Android, and creates a custom modal for iOS. It's called the `Toast` API because the information that's displayed looks like a piece of toast popping up. Here's what the Android component looks like:

```
import React, { PropTypes } from 'react';
import { ToastAndroid } from 'react-native';
import { Map as ImmutableMap } from 'immutable';

// Toast helper. Always returns "null" so that the
// output can be rendered as a React element.
const show = (message, duration) => {
  ToastAndroid.show(message, duration);
  return null;
};

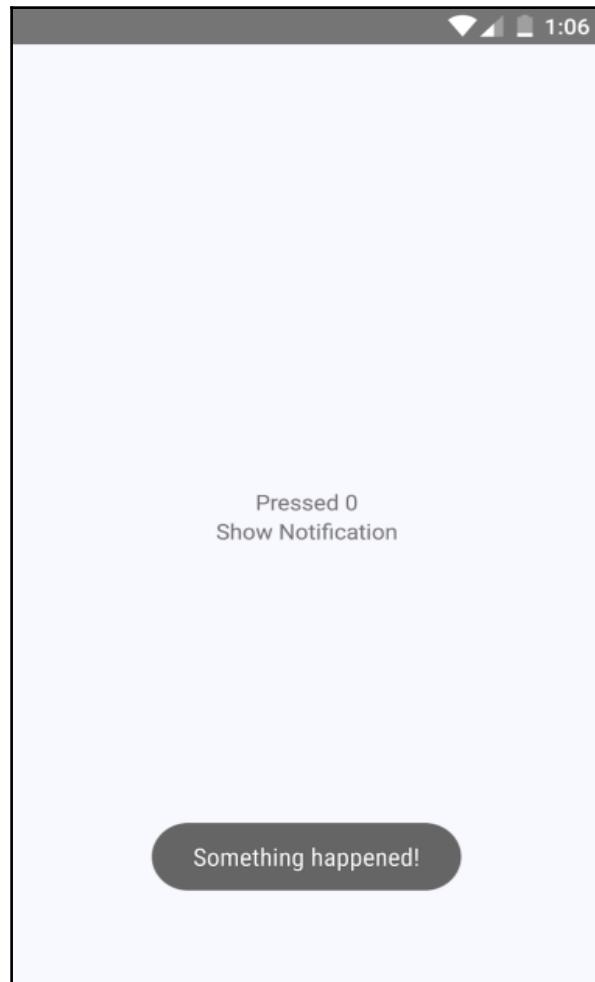
// This component will always return null,
// since it's using an imperative React Native
// interface to display popup text. If the
// "message" property was provided, then
// we display a message.
const Notification = ({ message, duration }) =>
  ImmutableMap()
    .set(null, null)
    .set(undefined, null)
    .get(message, show(message, duration));

Notification.propTypes = {
  message: PropTypes.string,
  duration: PropTypes.number.isRequired,
};

Notification.defaultProps = {
  duration: ToastAndroid.LONG,
};

export default Notification;
```

Once again, we're dealing with an imperative React Native API that you don't want to expose to the rest of your app. Instead, this component hides the imperative `ToastAndroid.show()` function behind a declarative React component. No matter what, this component returns null, because it doesn't actually render anything. Here's what the `ToastAndroid` notification looks like:



As you can see, the notification that something has happened is displayed at the bottom of the screen and is removed after a short delay. The key is that the notification is unobtrusive.

The iOS notification component is a little more involved, because it needs state and lifecycle events to make a modal view behave like a transient notification. Here's the code:

```
import React, { Component, PropTypes } from 'react';
import {
  View,
  Modal,
  Text,
} from 'react-native';
import { Map as ImmutableMap } from 'immutable';

import styles from './styles';

class Notification extends Component {

  static propTypes = {
    message: PropTypes.string,
    duration: PropTypes.number.isRequired,
  }

  static defaultProps = {
    duration: 1500,
  }

  // The modal component is either "visible", or not.
  // The "timer" is used to hide the notification
  // after some predetermined amount of time.
  state = { visible: false }
  timer = null

  showModal = (message, duration) => {
    // Update the "visible" state, based on whether
    // or not there's a "message" value.
    this.setState({
      visible: ImmutableMap()
        .set(null, false)
        .set(undefined, false)
        .get(message, true),
    });
    // Create a timer to hide the modal if a new
    // "message" is set.
    this.timer = ImmutableMap()
      .set(null, () => null)
      .set(undefined, () => null)
      .get(message, () => setTimeout(
        () => this.setState({ visible: false }),
        duration
      ));
  }
}
```

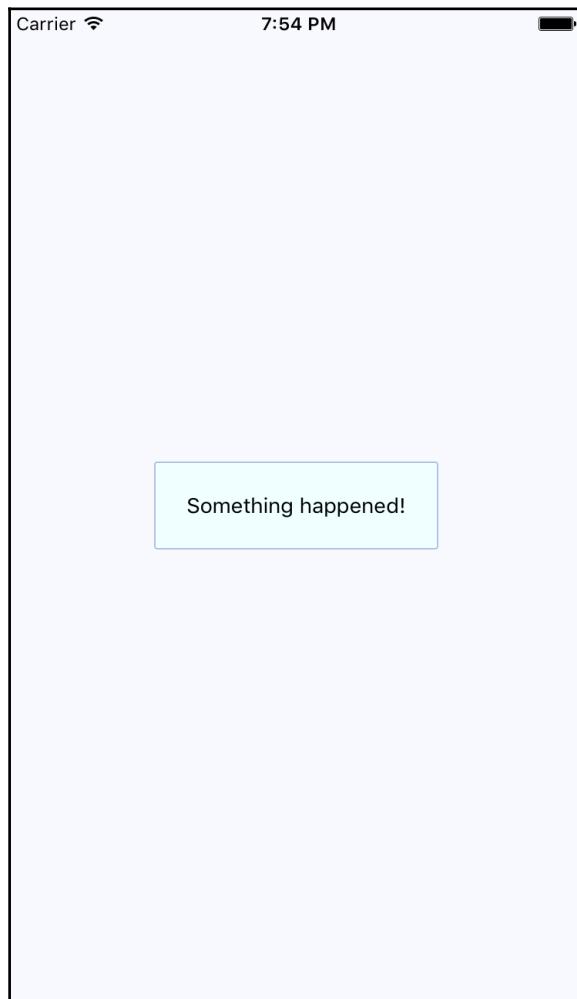
```
        ) ) () ;  
    }  
  
    // When the component is mounted, show the modal  
    // if a message was provided. Also set a timeout  
    // that automatically closes the modal.  
    componentWillMount() {  
        this.showModal(  
            this.props.message,  
            this.props.duration,  
        );  
    }  
  
    // Does the same thing as "componentWillMount()"  
    // except it's called when new properties are passed.  
    componentWillReceiveProps(props) {  
        this.showModal(  
            props.message,  
            props.duration,  
        );  
    }  
  
    componentWillUnmount() {  
        clearTimeout(this.timer);  
    }  
  
    render() {  
        const modalProps = {  
            animationType: 'fade',  
            transparent: true,  
            visible: this.state.visible,  
        };  
  
        return (  
            <Modal {...modalProps}>  
                <View style={styles.notificationContainer}>  
                    <View style={styles.notificationInner}>  
                        <Text>{this.props.message}</Text>  
                    </View>  
                </View>  
            </Modal>  
        );  
    }  
}  
  
Notification.propTypes = {  
    message: PropTypes.string,  
    duration: PropTypes.number.isRequired,
```

```
};

Notification.defaultProps = {
  duration: 1500,
};

export default Notification;
```

We have to style the modal to display the notification text, as well as the state that's used to hide the notification after a delay. Here's what the end result looks like for iOS:



Again, the same principle with the `ToastAndroid` API applies here. You might have noticed that there's another button in addition to the **Show Notification** button. This is a simple counter that re-renders the view. There's actually a reason for demonstrating this seemingly obtuse feature, as you'll see momentarily. Here's the code for the main application view:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import Notification from './Notification'; // eslint-disable-line
import/no-unresolved

class PassiveNotifications extends Component {
  // The initial state is the number of times
  // the counter button has been clicked, and
  // the notification message.
  state = {
    data: fromJS({
      count: 0,
      message: null,
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  render() {
    const {
      count,
      message,
    } = this.data.toJS();

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

```
{ /* The "Notification" component is
     only displayed if the "message" state
     has something in it. */
<Notification message={message} />

{ /* Updates the count. Also needs to make
     sure that the "message" state is null,
     even if the message has been hidden
     already. */
<Text
  onPress={() => {
    this.data = this.data
      .update('count', c => c + 1)
      .set('message', null);
  }}
>
  Pressed {count}
</Text>

{ /* Displays the notification by
     setting the "message" state. */
<Text
  onPress={() => {
    this.data = this.data
      .set('message', 'Something happened!');
  }}
>
  Show Notification
</Text>
</View>
);
}

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

The whole point of the press counter is to demonstrate that, even though the Notification component is declarative and accepts new property values when the state changes, you still have to set the message state to null when changing other state values. The reason for this is that if you re-render the component and the message state still has a string in it, it will display the same notification, over and over.

Activity modals

In this final section of the chapter, you'll implement a modal that shows a progress indicator. The idea is to display the modal, then hide it when the promise resolves. Here's the code for the generic `Activity` component that shows a modal with an activity indicator:

```
import React, { PropTypes } from 'react';
import {
  View,
  Modal,
  ActivityIndicator,
} from 'react-native';

import styles from './styles';

// The "Activity" component will only display
// if the "visible" property is true. The modal
// content is an "<ActivityIndicator>" component.
const Activity = props => (
  <Modal visible={props.visible} transparent>
    <View style={styles.modalContainer}>
      <ActivityIndicator size={props.size} />
    </View>
  </Modal>
);

Activity.propTypes = {
  visible: PropTypes.bool.isRequired,
  size: PropTypes.string.isRequired,
};

Activity.defaultProps = {
  visible: false,
  size: 'large',
};

export default Activity;
```

You might be tempted to pass the promise to the component so that it automatically hides itself when the promise resolves. I don't think this is a good idea, because then you would have to introduce the state into this component. Furthermore, it would depend on a promise in order to function. With the way we've implemented this component, you can show or hide the modal based on the `visible` property alone. Here's what the activity modal looks like on iOS:



There's a semi-transparent background on the modal that's placed over the main view with the **Fetch Stuff...** link. Here's how this effect is created:

```
modalContainer: {  
  flex: 1,  
  justifyContent: 'center',  
  alignItems: 'center',  
  backgroundColor: 'rgba(0, 0, 0, 0.2)',  
},
```

Instead of setting the actual Modal component to be transparent, we set the transparency in the backgroundColor, which gives the look of an overlay. Now, let's take a look at the code that controls this component:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import Activity from './Activity';

class ActivityModals extends Component {
  // The state is a "fetching" boolean value,
  // and a "promise" that is used to determine
  // when the fetching is done.
  state = {
    data: fromJS({
      fetching: false,
      promise: Promise.resolve(),
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // When the fetch button is pressed, the
  // promise that simulates async activity
  // is set, along with the "fetching" state.
  // When the promise resolves, the "fetching"
  // state goes back to false, hiding the modal.
  onPress = () => {
    this.data = this.data
      .merge({
        promise: new Promise(resolve =>
          setTimeout(resolve, 3000)
        ).then(() => {
          this.data = this.data
        })
      })
  }
}
```

```
        .set('fetching', false);
    }),
    fetching: true,
);
}

render() {
return (
<View style={styles.container}>
{ /* The "<Activity>" modal is only visible
   when the "fetching" state is true. */ }
<Activity
visible={this.data.get('fetching')}
/>
<Text onPress={this.onPress}>
  Fetch Stuff...
</Text>
</View>
);
}
}

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

When the fetch link is pressed, we create a new promise that simulates some async network activity. Then, when the promise resolves, we have to change the `fetching` state back to false so that the activity dialog is hidden.

Summary

In this chapter, you learned about the need to show mobile users important information. This sometimes involves explicit feedback from the user, even if that just means acknowledgement of the message. In other cases passive notifications work better, since they're less obtrusive than the confirmation modals.

There are two tools that you can use to display messages to users: modals and alerts. Modals are more flexible, because they're just like regular views. Alerts are good for displaying plain text and they take care of styling concerns for us. On Android, you have the additional `ToastAndroid` interface. You saw that it's also possible to do this on iOS, but it just requires more work.

In the next chapter, we'll dig deeper into the gesture response system inside React Native, which makes for a better mobile experience than browsers are able to provide.

21

Responding to User Gestures

All of the examples that you've implemented so far in this book have relied on user gestures. In traditional web applications, you generally worry about mouse events. However, touchscreens rely on the user manipulating elements with their fingers—fundamentally different from the mouse.

The goal of this chapter is to show you how the gesture response system inside of React Native works and some of the ways this system is exposed to us through components.

We'll begin with scrolling. This is probably the common gesture, besides touch. Then, we'll talk about giving the user the appropriate level of feedback when they interact with our components. Finally, we'll implement components that can be swiped.

Scrolling with our fingers

Scrolling in web applications is done by using the mouse pointer to drag the scrollbar back and forth or up and down, or by spinning the mousewheel. This obviously doesn't work in a mobile context because there's no mouse. Everything is controlled by gestures on the screen. For example, if you want to scroll down, you use your thumb or index finger to pull the content up by physically moving your finger on the screen.

This, by itself, is difficult to implement, but it gets more complicated. When you scroll on a mobile screen, the velocity of the dragging motion is taken into consideration. You drag the screen fast, then let go, and the screen will continue to scroll based on how fast you moved. You can also touch the screen while this is happening to stop it from scrolling.

Yikes! Thankfully, we don't have to handle most of this stuff. The `ScrollView` component handles much of the scrolling complexity for us. In fact, you've already used the `ScrollView` component, back in Chapter 16, *Rendering Item Lists*. The `ListView` component has `ScrollView` baked into it.



You can hack the low-level parts of user interactions by implementing gesture lifecycle methods. You'll probably never need to do this, but if you're interested, you can read about it at <http://facebook.github.io/react-native/releases/next/docs/gestur e-responder-system.html>.

You can use the `ScrollView` outside of the `ListView`. For example, if you're just rendering arbitrary content such as text and other widgets—not a list in other words—you can just wrap it in a `<ScrollView>`. Here's an example:

```
import React from 'react';
import {
  AppRegistry,
  Text,
  ScrollView,
  ActivityIndicator,
  Switch,
  View,
} from 'react-native';

import styles from './styles';

const FingerScrolling = () => (
  <View style={styles.container}>
    { /* The "<ScrollView>" can wrap any
        other component to make it scrollable.
        Here, we're repeating an arbitrary group
        of components to create some scrollable
        content */ }
    <ScrollView style={styles.scroll}>
      {new Array(6).fill(null).map((v, i) => (
        <View key={i}>
          { /* Arbitrary "<Text>" component... */ }
          <Text style={[styles.scrollItem, styles.text]}>
            Some text
          </Text>
      ))}
    </ScrollView>
  </View>
)
```

```
    {/* Arbitrary "<ActivityIndicator>"... */}
    <ActivityIndicator
      style={styles.scrollItem}
      size="large"
    />

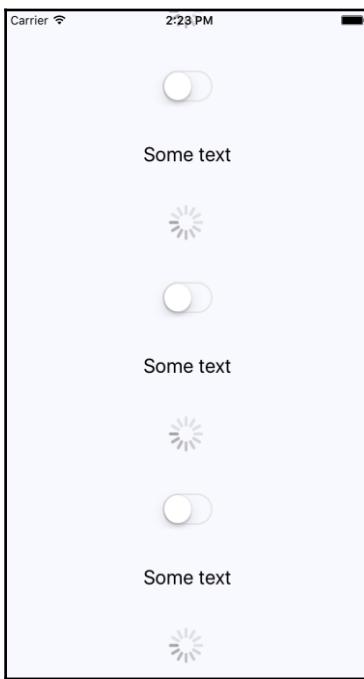
    {/* Arbitrary "<Switch>" component... */}
    <Switch style={styles.scrollItem} />
  </View>
)
)
</ScrollView>
</View>
);

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

The `ScrollView` component isn't of much use on its own—it's there to wrap other components. However, it does need a height in order to function correctly. Here's what the scroll style looks like:

```
scroll: {
  height: 1,
  alignSelf: 'stretch',
},
```

The height is set to 1, but the stretch value of alignSelf allows the items to display properly. Here's what the end result looks like:



As you can see, there's a vertical scrollbar on the right side of the screen as I drag the content down. If you run this example, you can play around with making the various gestures, like making content scroll on its own and then making it stop.

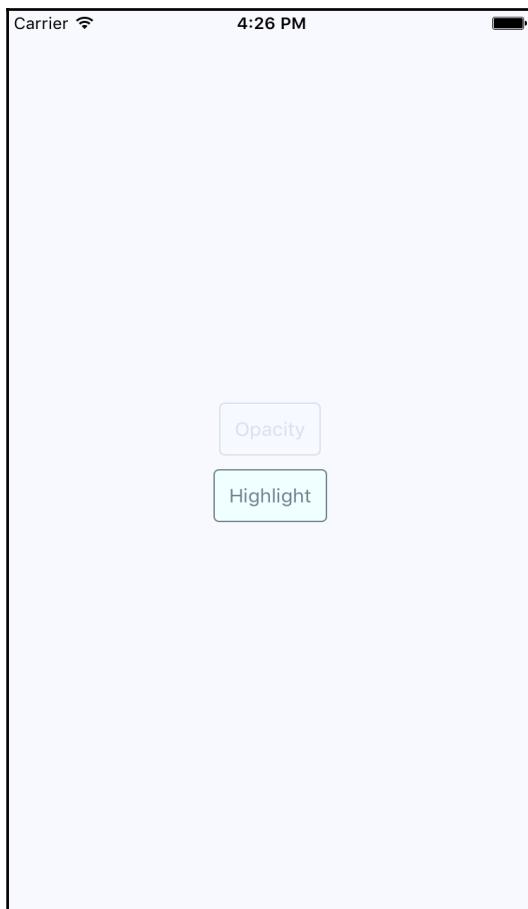
Giving touch feedback

The React Native examples you've worked with so far in this book have used plain text to act as buttons or links. In web applications, it's pretty easy to make text look like something that can be clicked—you just wrap it with the appropriate link. There's no such thing as mobile links, so you can style your text to look like a button.

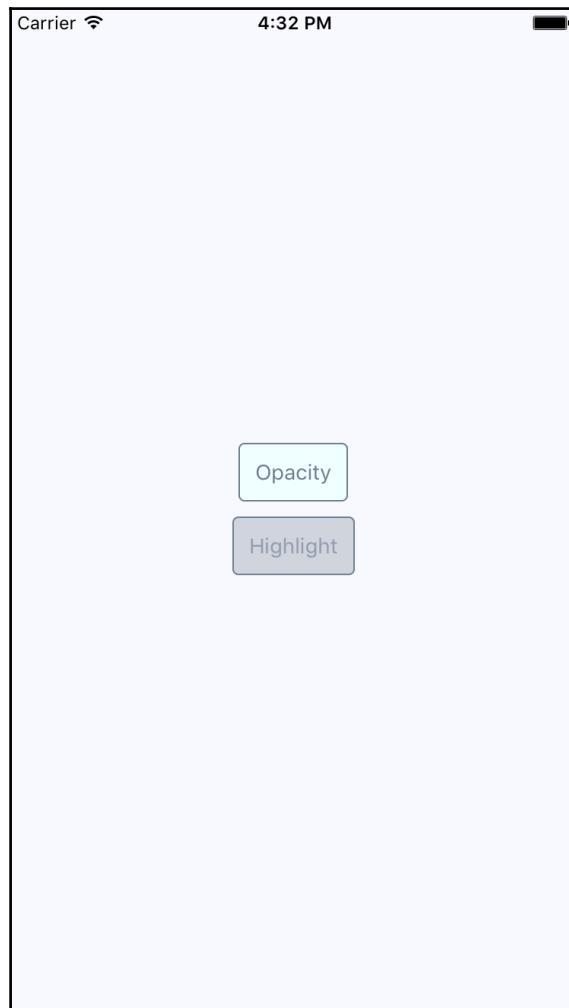


The problem with trying to style text as links on mobile devices is that they're too hard to press. Buttons provide a bigger target for my fat fingers, and they're easier to give apply touch feedback on.

So, let's style some text as a button. This is a great first step, making the text look touchable. But, we also want to give visual feedback to the user when they start interacting with the button. React Native provides two components to help with this: `TouchableOpacity` and `TouchableHighlight`. But before we dive into the code, let's take a look at what these components look like visually when users interact with them, starting with `TouchableOpacity`:



There's two buttons rendered here, and the top one labeled **Opacity** is currently being pressed by the user. The opacity of the button is dimmed when pressed, which provides important visual feedback for the user. Let's see what the **Highlight** button looks like when pressed:



Instead of changing the opacity when pressed, the `TouchableHighlight` component adds a highlight layer over the button. In this case, we're highlighting it using a more transparent version of the slate grey used in the font and border colors.

Which approach you use doesn't really matter. The important thing is that you provide the appropriate touch feedback for your users as they interact with your buttons. In fact, you might want to use the two approaches in the same app, but for different things. Let's create a `Button` component makes it easy to use either approach:

```
import React, { PropTypes } from 'react';
import {
  Text,
  TouchableOpacity,
  TouchableHighlight,
} from 'react-native';

import styles from './styles';

// The "touchables" map is used to get the right
// component to wrap around the button. The
// "undefined" key represents the default.
const touchables = new Map([
  ['opacity', TouchableOpacity],
  ['highlight', TouchableHighlight],
  [undefined, TouchableOpacity],
]);

const Button = ({
  label,
  onPress,
  touchable,
}) => {
  // Get's the "Touchable" component to use,
  // based on the "touchable" property value.
  const Touchable = touchables.get(touchable);

  // Properties to pass to the "Touchable"
  // component.
  const touchableProps = {
    style: styles.button,
    underlayColor: 'rgba(112,128,144,0.3)',
    onPress,
  };

  // Renders the "<Text>" component that's
  // styled to look like a button, and is
  // wrapped in a "<Touchable>" component
  // to properly handle user interactions.
  return (
    <Touchable {...touchableProps}>
      <Text>{label}</Text>
    </Touchable>
  );
}
```

```
<Text style={styles.buttonText}>
  {label}
</Text>
</Touchable>
);
};

Button.propTypes = {
  onPress: PropTypes.func.isRequired,
  label: PropTypes.string.isRequired,
  touchable: PropTypes.oneOf([
    'opacity',
    'highlight',
  ]),
};

export default Button;
```

As you can see, the `touchables` map is used to determine which React Native touchable component wraps the text, based on the `touchable` property value. Here's the styles used to create this button:

```
button: {
  padding: 10,
  margin: 5,
  backgroundColor: 'azure',
  borderWidth: 1,
  borderRadius: 4,
  borderColor: 'slategrey',
},
buttonText: {
  color: 'slategrey',
}
```

Here's how we put those buttons to use in the main app module:

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

import styles from './styles';
import Button from './Button';

const TouchFeedback = () => (
  <View style={styles.container}>
```

```
{ /* Renders a "<Button>" that uses
    "TouchableOpacity" to handle user
    gestures, since that is the default */ }
<Button
  onPress={() => {}}
  label="Opacity"
/>

{ /* Renders a "<Button>" that uses
    "TouchableHighlight" to handle
    user gestures. */ }
<Button
  onPress={() => {}}
  label="Highlight"
  touchable="highlight"
/>
</View>
);

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

Note that the `onPress` callbacks don't actually do anything, but they are called, and we're passing them because they're a required property.

Swipeable and cancellable

Part of what makes native mobile applications easier to use than mobile web applications is that they feel more intuitive. Using gestures, you can quickly get a handle on how things work. For example, swiping an element across the screen with your finger is a common gesture, but the gesture has to be discoverable.

Let's say that you're using an app, and you're not exactly sure what something on the screen does. So, you press down with your finger, and try dragging the element. It starts to move. Unsure of what will happen, you lift your finger up, and the element moves back into place. You've just discovered how part of this application works.

We'll use the `Scrollable` component to implement swipeable and cancelable behavior like this. We can create a somewhat generic component that allows the user to swipe text off the screen, and when that happens, call a callback function. But let's look at the code to render the swipeables before we look at the generic component itself:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import Swipeable from './Swipeable';

class SwipableAndCancellable extends Component {
  // The initial state is an immutable list of
  // 8 swipeable items.
  state = {
    data: fromJS(new Array(8)
      .fill(null)
      .map((v, id) => ({ id, name: 'Swipe Me' })))
  },
}

// Getter for "Immutable.js" state data...
get data() {
  return this.state.data;
}

// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}

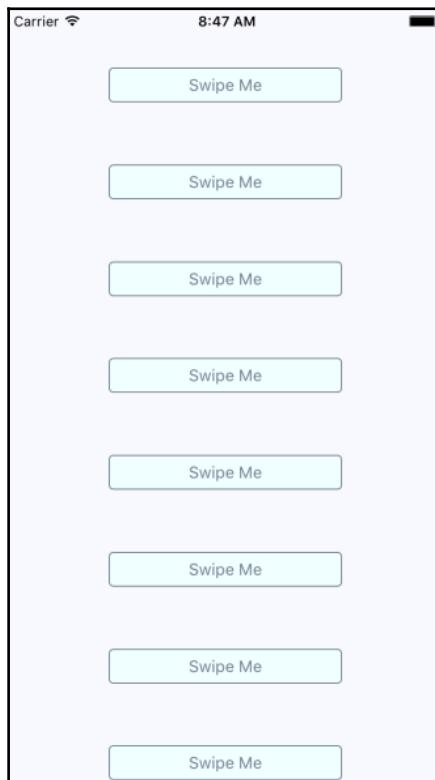
// The swipe handler passed to "<Swipeable>".
// The swiped item is removed from the state.
// This is a higher-order function that returns
// the real handler so that the "id" context
// can be set.
onSwipe = id => () => {
  this.data = this.data
    .filterNot(v => v.get('id') === id);
}

render() {
  return (
```

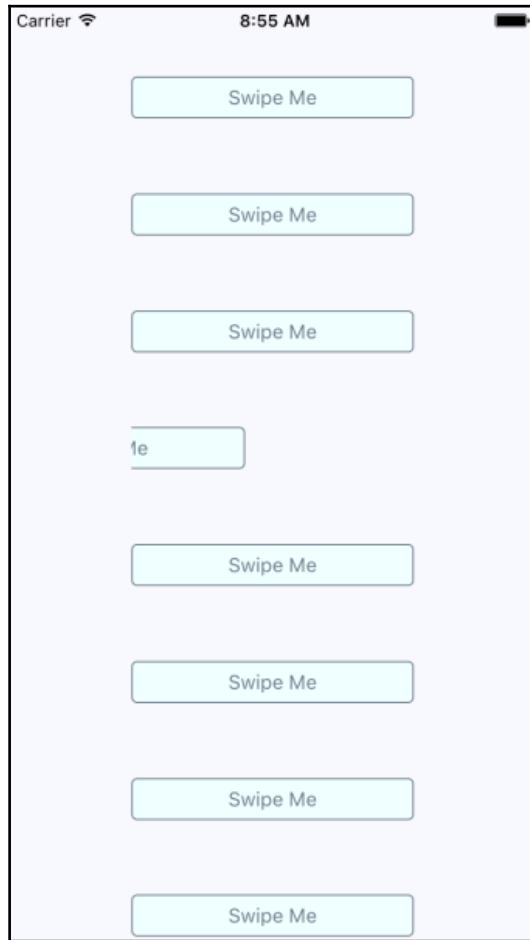
```
<View style={styles.container}>
  {this.data.toJS().map(i => (
    <Swipeable
      key={i.id}
      onSwipe={this.onSwipe(i.id)}
      name={i.name}
    />
  ))}
</View>
);
}
}

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

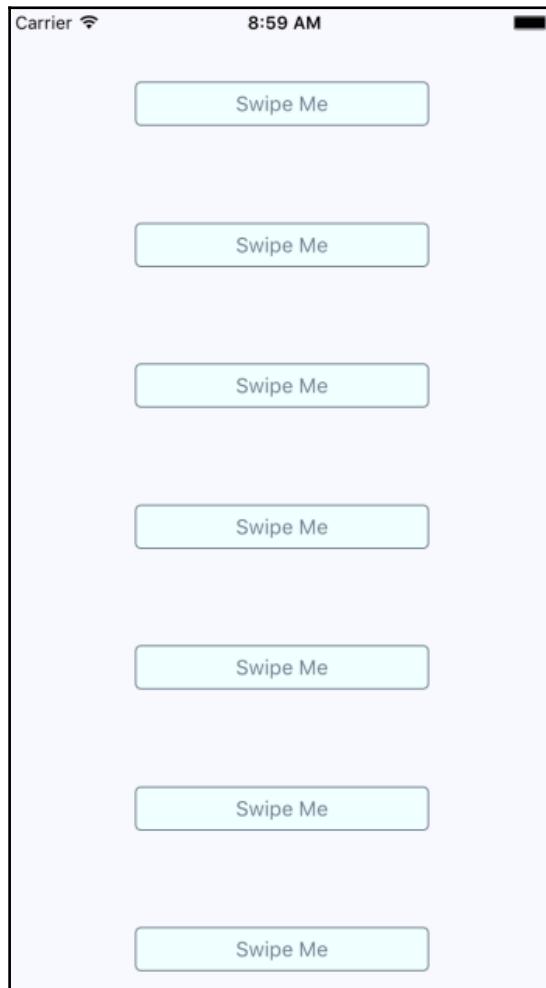
This will render eight `<Swipeable>` components on the screen. Let's see what this looks like:



Now, if you start to swipe one of these items to the left, it will move. Here's what it looks like:



If you don't swipe far enough, the gesture is cancelled and the item moves back into place as expected. If you swipe it all the way, the item is removed from the list completely and the items on the screen fill the empty space like this:



Now let's take a look at the `Swipeable` component itself:

```
import React, { PropTypes } from 'react';
import { Map as ImmutableMap } from 'immutable';
import {
  View,
  ScrollView,
  Text,
```

```
    TouchableOpacity,
} from 'react-native';

import styles from './styles';

// The "onScroll" handler. This is actually
// a higher-order function that retuns the
// actual handler. When the x offset is 200,
// when know that the component has been
// swiped and can call "onSwipe()".
const onScroll = onSwipe => e =>
  ImmutableMap()
    .set(200, onSwipe)
    .get(e.nativeEvent.contentOffset.x, () => {});

// The static properties used by the "<ScrollView>"
// component.
const scrollProps = {
  horizontal: true,
  pagingEnabled: true,
  showsHorizontalScrollIndicator: false,
  scrollEventThrottle: 10,
};

const Swipeable = ({
  onSwipe,
  name,
}) => (
  <View style={styles.swipeContainer}>
    { /* The "<View>" that wraps this "<ScrollView>" */
      is necessary to make scrolling work properly. */ }
    <ScrollView
      {...scrollProps}
      onScroll={onScroll(onSwipe)}
    >
      { /* Not strictly necessary, but "<TouchableOpacity>" */
        does provide the user with meaningful feedback
        when they initially press down on the text. */ }
      <TouchableOpacity>
        <View style={styles.swipeItem}>
          <Text style={styles.swipeItemText}>{name}</Text>
        </View>
      </TouchableOpacity>
      <View style={styles.swipeBlank} />
    </ScrollView>
  </View>
);

;
```

```
Swipeable.propTypes = {
  onSwipe: PropTypes.func.isRequired,
  name: PropTypes.string.isRequired,
};

export default Swipeable;
```

Note that the `<ScrollView>` component is set to be horizontal and that `pagingEnabled` is true. It's the paging behavior that snaps the components into place, and provides the cancellable behavior. This is why there's a blank component beside the component with text in it. Here's the styles used for this component:

```
swipeContainer: {
  flex: 1,
  flexDirection: 'row',
  width: 200,
  height: 30,
  marginTop: 50,
},

swipeItem: {
  width: 200,
  height: 30,
  backgroundColor: 'azure',
  justifyContent: 'center',
  borderWidth: 1,
  borderRadius: 4,
  borderColor: 'slategrey',
},

swipeItemText: {
  textAlign: 'center',
  color: 'slategrey',
},

swipeBlank: {
  width: 200,
  height: 30,
},
```

The `swipeBlank` style has the same dimensions as `swipeItem`, but nothing else. It's invisible.

Summary

In this chapter, you were introduced to the idea that gestures on native platforms are the big difference maker when compared to mobile web platforms. We started off by looking at the `ScrollView` component, and how it makes life much simpler for us by providing native scrolling behavior for wrapped components.

Next, we spent some time implementing buttons with touch feedback. This is another area that's tricky to get right on the mobile web. You learned how to use the `TouchableOpacity` and `TouchableHighlight` components.

Finally, you implemented a generic `Swipeable` component. Swiping is a common mobile pattern, and it allows for the user to discover how things work without feeling intimidated. In the next chapter, you'll learn how to control image display using React Native.

22

Controlling Image Display

So far, the examples in this book haven't rendered any images on mobile screens. This doesn't reflect the reality of mobile applications. Web applications display lots of images. If anything, native mobile applications rely on images even more than web applications because images are powerful tools when you have a limited amount of space.

You'll learn how to use the React Native `Image` component in this chapter, starting with loading images from different sources. Then, you'll see how you can use the `Image` component to resize images, and how you can set placeholders for lazy images. Finally, you'll learn how to implement icons using the `react-native-vector-icons` package.

Loading images

Let's get things started by figuring out how to actually load images. You can render the `<Image>` component and pass it properties just like any other React component. But this particular component needs image blob data to be of any use. Let's look at some code:

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

import styles from './styles';

// Renders two "<Image>" components, passing the
// properties of this component to the "source"
// property of each image.
const LoadingImages = ({
  reactSource,
```

```
        relaySource,
    }) => (
      <View style={styles.container}>
        <Image
          style={styles.image}
          source={reactSource}
        />
        <Image
          style={styles.image}
          source={relaySource}
        />
      </View>
    );
  }

  // The "source" property can be either
  // an object with a "uri" string, or a number
  // representing a local "require()" resource.
  const sourceProp = PropTypes.oneOfType([
    PropTypes.shape({
      uri: PropTypes.string.isRequired,
    }),
    PropTypes.number,
  ]).isRequired;

  LoadingImages.propTypes = {
    reactSource: sourceProp,
    relaySource: sourceProp,
  };

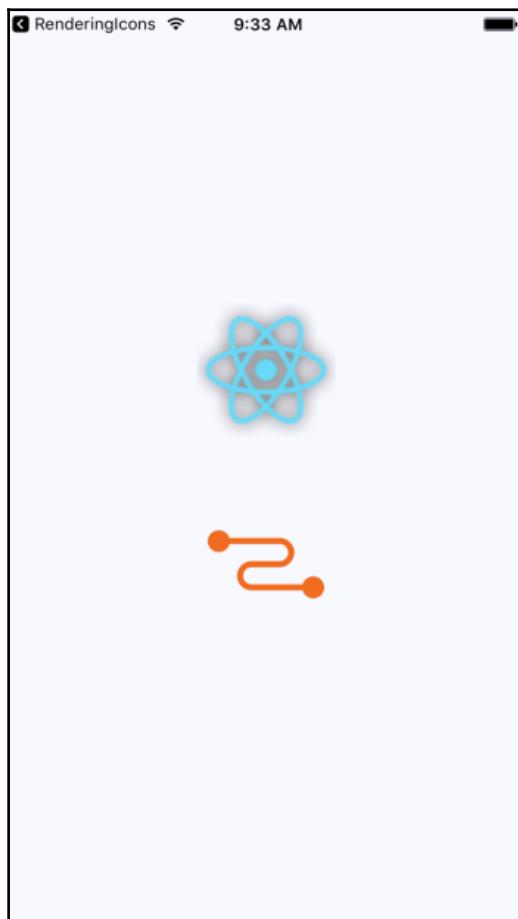
  LoadingImages.defaultProps = {
    // The "reactSource" image comes from a remote
    // location.
    reactSource: {
      uri: 'https://facebook.github.io/react/img/logo_small_2x.png',
    },
    // The "relaySource" image comes from a local
    // source.
    relaySource: require('./images/relay.png'),
  };

  AppRegistry.registerComponent(
    'LoadingImages',
    () => LoadingImages
  );
}
```

As you can see, there are two ways to load the blob data into an `<Image>` component. The first approach loads the image data from the network. This is done by passing an object with a `uri` property to `source`. The second `<Image>` component that we've rendered here is using a local image file, by calling `require()` and passing the result to `source`.

Take a look at the `sourceProp` property type validator that we've created. This gives you an idea of what can be passed to the `source` property. It's either an object with a `uri` string property or a number. It expects a number because `require()` returns a number.

Now, let's see what the rendered result looks like:



Here's the style that was used with these images:

```
image: {  
  width: 100,  
  height: 100,  
  margin: 20,  
},
```

Note that without `width` and `height` style properties, images will not render. In the next section, we'll look at how image resizing works when `width` and `height` values are set.

Resizing images

The `width` and `height` style properties of `Image` components determine the size of what's rendered on the screen. For example, you'll probably have to work with images at some point that have a larger resolution than you want displayed in your React Native application. Simply setting the `width` and `height` style properties on the `Image` is enough to properly scale the image.

Let's look at some code that lets you dynamically adjust the dimensions of an image using a control:

```
import React, { Component } from 'react';  
import {  
  AppRegistry,  
  View,  
  Text,  
  Image,  
  Slider,  
} from 'react-native';  
import { fromJS } from 'immutable';  
  
import styles from './styles';  
  
class ResizingImages extends Component {  
  // The initial state of this component includes  
  // a local image source, and the width/height  
  // image dimensions.  
  state = {  
    data: fromJS({  
      source: require('./images/flux.png'),  
      width: 100,  
      height: 100,  
    }),  
  };  
}
```

```
// Getter for "Immutable.js" state data...
get data() {
    return this.state.data;
}

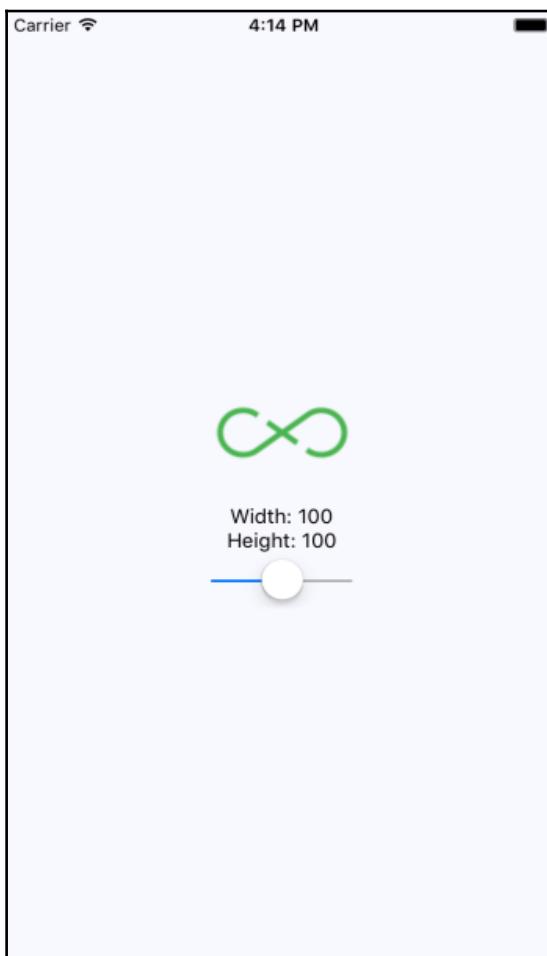
// Setter for "Immutable.js" state data...
set data(data) {
    this.setState({ data });
}

render() {
    // The state values we need...
    const {
        source,
        width,
        height,
    } = this.data.toJS();

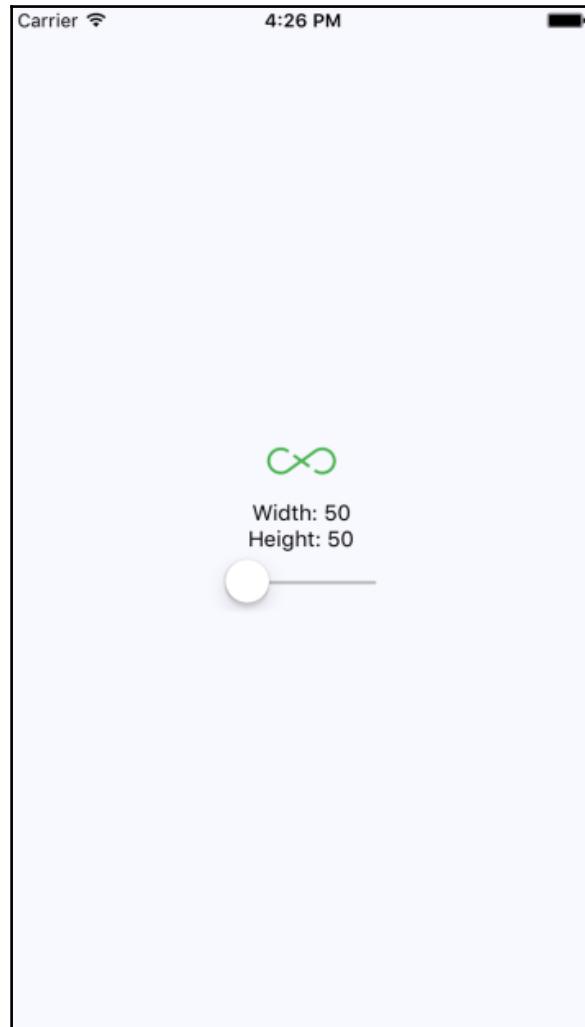
    return (
        <View style={styles.container}>
            { /* The image is rendered using the
                "source", "width", and "height"
                state values. */ }
            <Image
                source={source}
                style={{ width, height }}
            />
            { /* The current "width" and "height"
                values are displayed. */ }
            <Text>Width: {width}</Text>
            <Text>Height: {height}</Text>
            { /* This slider scales the image size
                up or down by changing the "width"
                and "height" states. */ }
            <Slider
                style={styles.slider}
                minimumValue={50}
                maximumValue={150}
                value={width}
                onValueChange={(v) => {
                    this.data = this.data
                        .merge({
                            width: v,
                            height: v,
                        });
                }}
            />
        </View>
    );
}
```

```
) ;  
}  
}  
  
AppRegistry.registerComponent(  
  'ResizingImages',  
  () => ResizingImages  
) ;
```

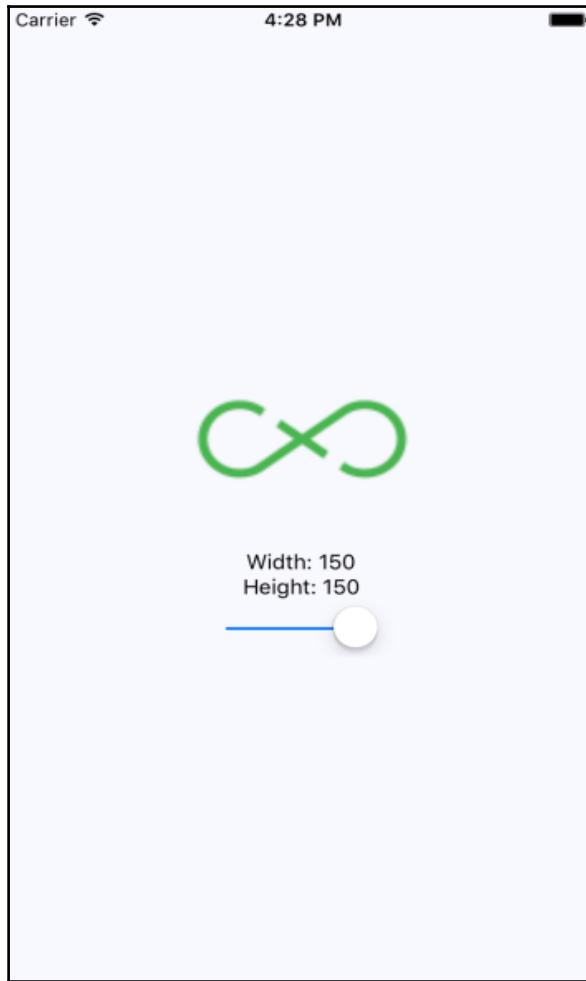
Here's what the image looks like if using the default 100×100 dimensions:



Here's a scaled-down version of the image:



Lastly, here's a scaled-up version of the image:



There's a `resizeMode` property that you can pass to `Image` components. This determines how the scaled image fits within the dimensions of the actual component. You'll see this property in action in the last section of this chapter.

Lazy image loading

Sometimes, you don't necessarily want an image to load at the exact moment that it's rendered. For example, you might be rendering something that's not yet visible on the screen. Most of the time, it's perfectly fine to fetch the image source from the network before it's actually visible. But if you're fine-tuning your application and discovering that loading lots of images over the network causes performance issues, you can lazily load the source.

I think the more common case in a mobile context is handling a scenario where you've rendered one or more images where they're visible, but the network is slow to respond. In this case, you will probably want to render a placeholder image so that the user sees something right away, rather than empty space.

To do this, we'll implement an abstraction that wraps the actual image that we want to show once it's loaded. Here's the code:

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

// The local placeholder image source.
const placeholder = require('./images/placeholder.png');

// The mapping to the "loaded" state that gets us
// the appropriate image component.
const Placeholder = props =>
  new Map([
    [true, null],
    [false, (
      <Image
        {...props}
        source={placeholder}
      />
    )],
  ]).get(props.loaded);

class LazyImage extends Component {
  // The "width" and "height" properties
  // are required. All other properties are
  // forwarded to the actual "<Image>" component.
  static propTypes = {
    style: PropTypes.shape({
      width: PropTypes.number.isRequired,
      height: PropTypes.number.isRequired,
    }),
  };
}
```

```
constructor() {
  super();

  // We assume that the source hasn't finished
  // loading yet.
  this.state = {
    loaded: false,
  };
}

render() {
  // The props and state this component
  // needs in order to render...
  const {
    props: {
      style: {
        width,
        height,
      },
    },
    state: {
      loaded,
    },
  } = this;

  return (
    <View style={{ width, height }}>
      { /* The placeholder image is just a standard
         "<Image>" component with a predefined
         source. It isn't rendered if "loaded" is
         true. */ }
      <Placeholder loaded={loaded} {...this.props} />
      { /* The actual image is forwarded props that
         are passed to "<LazyImage>". The "onLoad"
         handler ensures the "loaded" state is true,
         removing the placeholder image. */ }
      <Image
        {...this.props}
        onLoad={() => this.setState({
          loaded: true,
        })}
      />
    </View>
  );
}

export default LazyImage;
```

This component renders a `View` with two `Image` components inside. It also has a `loaded` state, which is initially false. When `loaded` is false, the placeholder image is rendered. The `loaded` state is set to true when the `onLoad()` handler is called. This means that the placeholder image is removed, and the main image is displayed.

Now let's use the `LazyImage` component that we've just implemented. We'll render the image without a source, and the placeholder image should be displayed. We'll add a button that gives the lazy image a source, and when it loads, the placeholder image should be replaced. Here's what the main app module looks like:

```
/* eslint-disable global-require */
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
} from 'react-native';

import styles from './styles';
import LazyImage from './LazyImage';
import Button from './Button';

// The remote image to load...
const remote = 'https://facebook.github.io/react/img/logo_small_2x.png';

class LazyLoading extends Component {
  state = {
    source: null,
  }

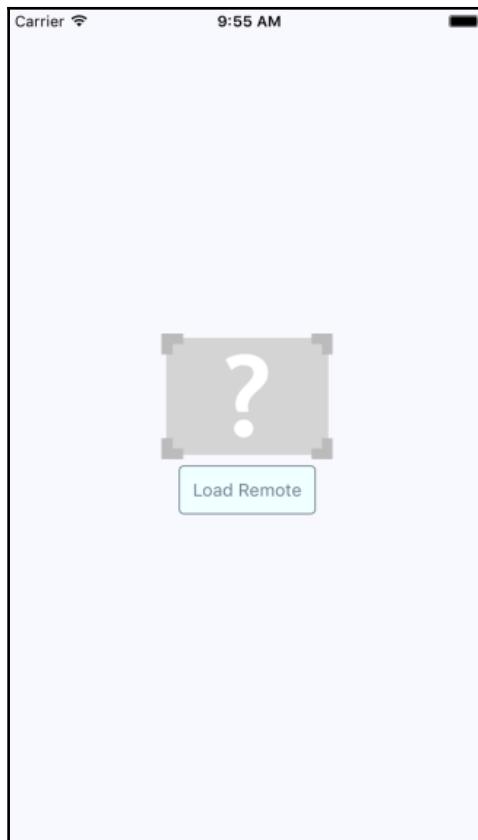
  render() {
    return (
      <View style={styles.container}>
        { /* Renders the lazy image. Since there's
            no "source" value initially, the placeholder
            image will be rendered. */ }
        <LazyImage
          style={{ width: 200, height: 100 }}
          resizeMode="contain"
          source={this.state.source}
        />
        { /* When pressed, this button changes the
            "source" of the lazy image. When the new
            source loads, the placeholder image is
            replaced. */ }
      <Button
        label="Load Remote"
        onPress={() => this.setState({
          source: remote
        })}
      />
    );
  }
}

AppRegistry.registerComponent('lazy-loading', () => LazyLoading);
```

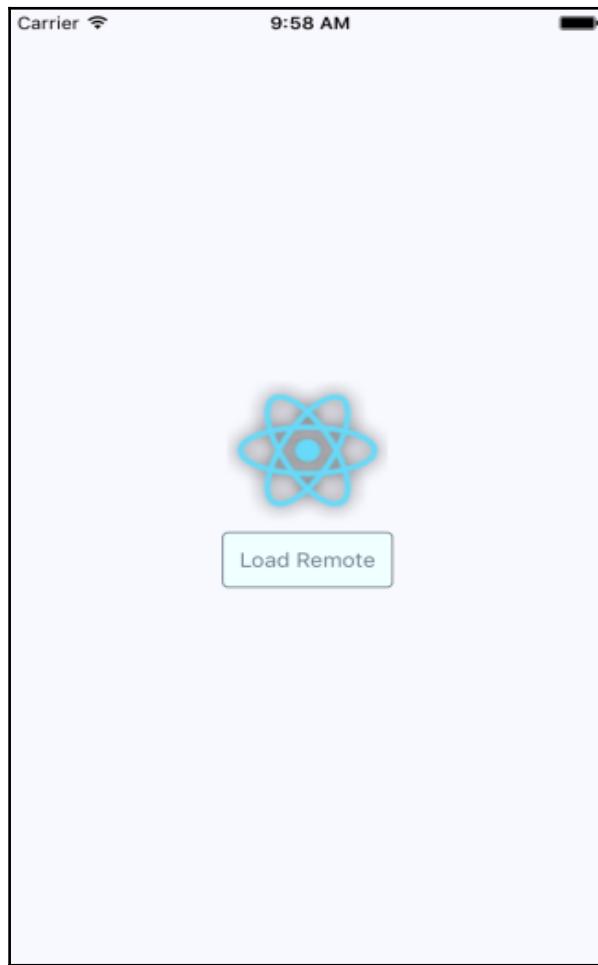
```
        source: { uri: remote },
    }) }
/>
</View>
);
}
}

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

This is what the screen looks like initially:



Then, if you click the **Load Remote** button, you'll eventually see the image that we actually want:



You might notice that depending on your network speed, the placeholder image remains visible even after you click the **Load Remote** button. This is by design, because you don't want to remove the placeholder image until you know for sure that the actual image is ready to be displayed.

Rendering icons

In the final section of this chapter, we'll look at rendering icons in React Native components. Using icons to indicate meaning makes web applications much more usable. So why should native mobile applications be any different?

You'll want to use the `react-native-vector-icons` package to pull in various vector font packages into your React Native project. This is very straightforward to install:

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

Now you can import the `Icon` component and render them. Let's implement an example that renders several `FontAwesome` icons based on a selected icon category:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  View,
  Picker,
  ListView,
  Text,
} from 'react-native';
import Icon from 'react-native-vector-icons/FontAwesome';
import { fromJS } from 'immutable';

import styles from './styles';
import iconName from './icon-names.json';

class RenderingIcons extends Component {
  // The initial state consists of the "selected"
  // category, the "icons" JSON object, and the
  // "listSource" used to render the list view.
  state = {
    data: fromJS({
      selected: 'Web Application Icons',
      icons: iconName,
      listSource: new ListView.DataSource({
        rowHasChanged: (r1, r2) => r1 !== r2,
      }),
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }
}
```

```
// Setter for "Immutable.js" state data...
set data(data) {
  this.setState({ data });
}

// Sets the "listSource" state based on the
// "selected" icon state. Also sets the "selected"
// state.
updateListSource = (selected) => {
  this.data = this.data
    .update('listSource', listSource =>
      listSource.cloneWithRows(
        this.data
          .getIn(['icons', selected])
          .toJS()
      )
    )
    .set('selected', selected);
}

// Make sure the "listSource" is populated
// before the first render.
componentWillMount() {
  this.updateListSource(this.data.get('selected'));
}

render() {
  const {
    updateListSource,
    } = this;

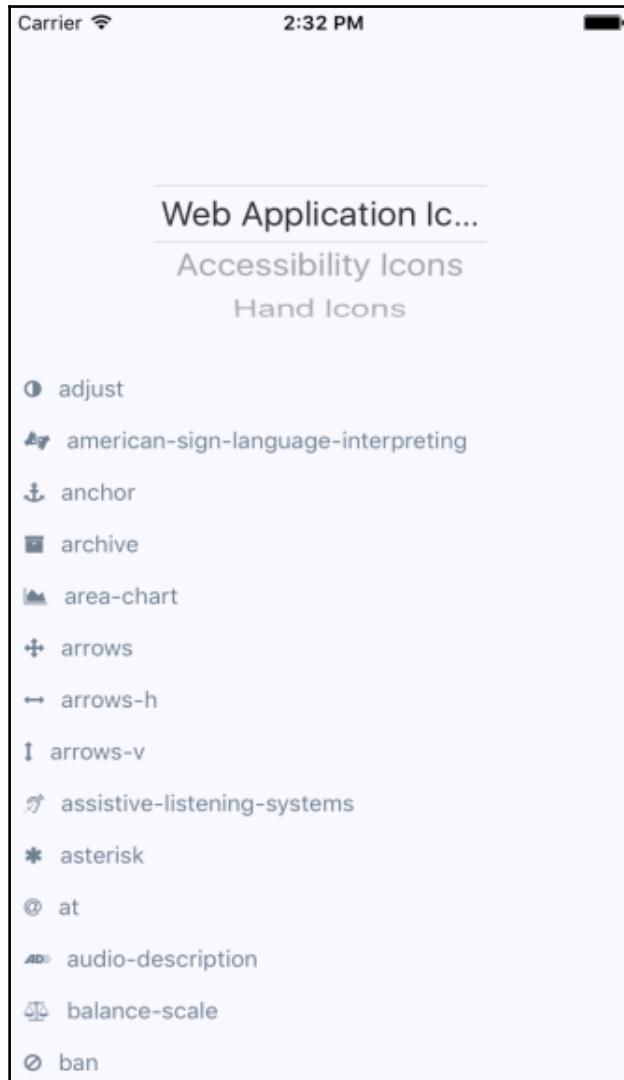
  // Get the state that we need to render the icon
  // category picker and the list view with icons.
  const selected = this.data
    .get('selected');
  const categories = this.data
    .get('icons')
    .keySeq()
    .toJS();
  const listSource = this.data
    .get('listSource');

  return (
    <View style={styles.container}>
      <View style={styles.picker}>
        { /* Lets the user select a FontAwesome icon
           category. When the selection is changed,
           the list view is changed. */ }
    
```

```
<Picker
    selectedValue={selected}
    onValueChange={updateListSource}
>
  {categories.map(c => (
    <Picker.Item
      key={c}
      label={c}
      value={c}
    />
  )))
</Picker>
</View>
<ListView
  style={styles.icons}
  dataSource={listSource}
  renderRow={icon => (
    <View style={styles.item}>
      { /* The "<Icon>" component is used
          to render the FontAwesome icon */ }
      <Icon
        name={icon}
        style={styles.itemIcon}
      />
      {/* Shows the icon class used */}
      <Text style={styles.itemText}>
        {icon}
      </Text>
    </View>
  ) }
  />
</View>
);
}
}

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

When you run the example, you should see something that looks like this:



As you can see, the color of the icon is specified the same way you would specify the color of text, via styles.

Summary

In this chapter, you learned about handling images in your React Native applications. Images in a native application are just as important in a native mobile context as they are in a web context—they improve the user experience.

You learned the different approaches to loading images, and then how to resize them. You also learned how to implement a lazy image that uses a placeholder image to display while the actual image is loading. Finally, you learned how to use icons in a React Native app.

In the next chapter, we'll take a look at local storage in React Native, which is handy in offline scenarios.

23

Going Offline

The expectation that applications will operate seamlessly with an unreliable network connection is the new norm. If your mobile application can't cope with transient network issues, then your users will simply use a different app. When there's no network, you have to persist data locally on the device. Or perhaps your app doesn't even require network access, in which case, you'll still need to store data locally.

In this chapter, you'll learn how to do three things with React Native. First, you'll learn how to detect the state of the network connection. Second, you'll learn how to store data locally. Lastly, you'll learn how to synchronize local data that's been stored due to network problems, once it comes back online.

Detecting the state of the network

If your code tries to make a request over the network while disconnected, using `fetch()` for example, an error will occur. You probably have error-handling code in place for these scenarios already, since the server could return some other type of error. However, in the case of connectivity trouble, you might want to detect this issue before the user attempts to make network requests.

There are two potential reasons for proactively detecting the network state. You might display a friendly message to the user stating that, since the network is disconnected, they can't do anything. You would then prevent the user from performing any network requests until you detect that it's back online. The other possible benefit of early network state detection is that you can prepare to perform actions offline and sync the app state when the network is connected again.

Let's look at some code that uses the `NetInfo` utility to handle changes in network state:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  View,
  NetInfo,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';

// Maps the state returned from "NetInfo" to
// a string that we want to display in the UI.
const connectedMap = {
  none: 'Disconnected',
  unknown: 'Disconnected',
  wifi: 'Connected',
  cell: 'Connected',
  mobile: 'Connected',
};

class NetworkState extends Component {
  // The "connected" state is a simple
  // string that stores the state of the
  // network.
  state = {
    data: fromJS({
      connected: '',
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // When the network state changes, use the
  // "connectedMap" to find the string to display.
  onNetworkChange = (info) => {
    this.data = this.data.set(
      'connected',
```

```
        connectedMap[info]
    );
}

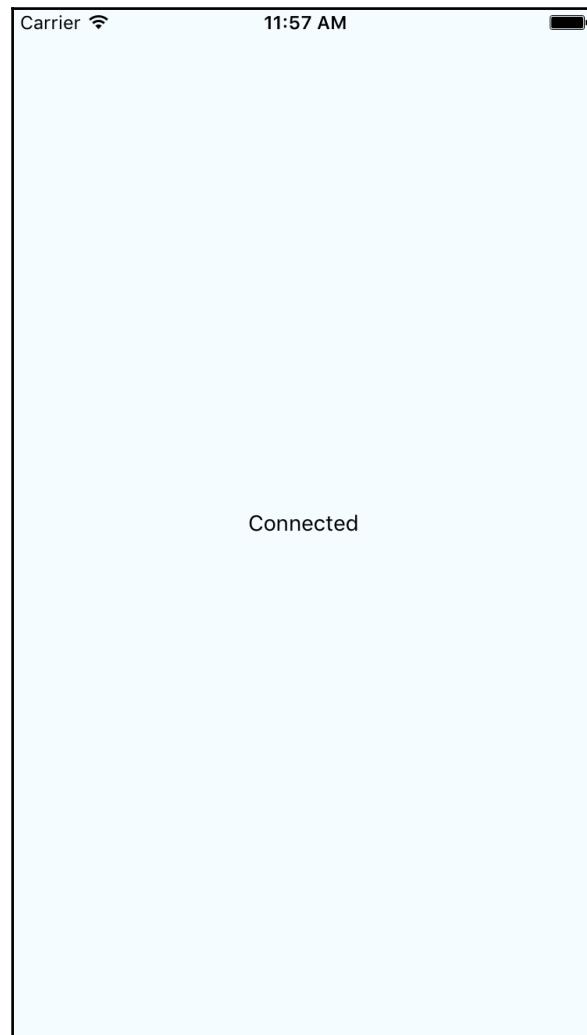
// When the component is mounted, we add a listener
// that changes the "connected" state when the
// network state changes.
componentWillMount() {
    NetInfo.addEventListener(
        'change',
        this.onNetworkChange
    );
}

// Make sure the listener is removed...
componentWillUnmount() {
    NetInfo.removeEventListener(
        'change',
        this.onNetworkChange
    );
}

// Simply renders the "connected" state as
// it changes.
render() {
    return (
        <View style={styles.container}>
            <Text>{this.data.get('connected')}</Text>
        </View>
    );
}
}

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

This component will render the state of the network, based on the string values in `connectedMap`. The change event of the `NetInfo` object will cause the `connected` state to change. For example, when you first run this app, the screen might look like this:



Then, if you turn off networking on your host machine, the network state will change on the emulated device as well, causing the state of our application to change:



While building this example, I had some trouble getting the change event to fire consistently when the state of the network state changed. This was an issue with both iOS and Android device emulators. So, if you're writing code that relies on network state detection, you might want to test it on a physical device if possible.

Storing application data

Now, let's shift our attention to storing data within a React Native application. The AsyncStorage API works the same on both iOS and Android platforms. You would use this API for applications that don't require any network connectivity in the first place, or to store data that's eventually synchronized using an API endpoint once a network becomes available.

Let's look at some code that allows the user to enter a key and a value, and then stores them:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  TextInput,
  View,
  ListView,
  AsyncStorage,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import Button from './Button';

class StoringData extends Component {

  // The initial state of this component
  // consists of the current "key" and "value"
  // that the user is entering. It also has
  // a "source" for the list view to display
  // everything that's been stored.
  state = {
    data: fromJS({
      key: null,
      value: null,
      source: new ListView.DataSource({
        rowHasChanged: (r1, r2) => r1 !== r2,
      }),
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
}
```

```
set data(data) {
  this.setState({ data });
}

// Uses "AsyncStorage.setItem()" to store
// the current "key" and "value" states.
// When this completes, we can delete
// "key" and "value" and reload the item list.
setItem = () =>
  AsyncStorage
    .setItem(
      this.data.get('key'),
      this.data.get('value')
    )
    .then(() => {
      this.data = this.data
        .delete('key')
        .delete('value');
    })
    .then(() => this.loadItems())

// Uses "AsyncStorage.clear()" to empty any stored
// values. Then, it loads the empty list of
// items to clear the item list on the screen.
clearItems = () =>
  AsyncStorage
    .clear()
    .then(() => this.loadItems())

// This method is async because awaits on the
// data store keys and values, which are two
// dependent async calls.
async loadItems() {
  const keys = await AsyncStorage.getAllKeys();
  const values = await AsyncStorage.multiGet(keys);

  this.data = this.data
    .update(
      'source',
      source => source.cloneWithRows(values)
    );
}

// Load any existing items that have
// already been stored when the app starts.
componentWillMount() {
  this.loadItems();
}
```

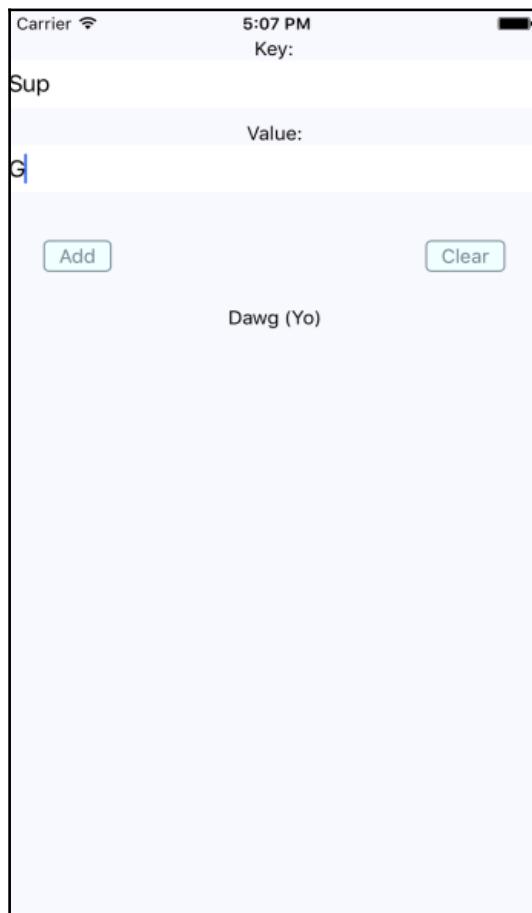
```
render() {
  // The methods that we need...
  const {
    setItem,
    clearItems,
  } = this;

  // The state that we need...
  const {
    source,
    key,
    value,
  } = this.data.toJS();

  return (
    <View style={styles.container}>
      <Text>Key:</Text>
      <TextInput
        style={styles.input}
        value={key}
        onChangeText={(v) => {
          this.data = this.data.set('key', v);
        }}
      />
      <Text>Value:</Text>
      <TextInput
        style={styles.input}
        value={value}
        onChangeText={(v) => {
          this.data = this.data.set('value', v);
        }}
      />
    <View style={styles.controls}>
      <Button
        label="Add"
        onPress={setItem}
      />
      <Button
        label="Clear"
        onPress={clearItems}
      />
    </View>
    <View style={styles.list}>
      <ListView
        enableEmptySections
        dataSource={source}
        renderRow={({[k, v]) => (
          <Text>{v} ({k})</Text>
        )}
      />
    </View>
  
```

```
) }  
    />  
  </View>  
  </View>  
);  
}  
  
AppRegistry.registerComponent(  
  'StoringData',  
  () => StoringData  
) ;
```

Before we walk through what this code is doing, let's first take a look at the screen, since it'll provide most of the explanation for me:



As you can see, there are two input fields and two buttons. The fields allow the user to enter a new key and value. The **Add** button allows the user to store this key-value pair locally on their device, while the **Clear** button clears any existing items that have been stored previously.

The `AsyncStorage` API works the same for both iOS and Android. Under the hood, `AsyncStorage` works very differently depending on which platform it's running on. The reason React Native is able to expose the same storage API on both platforms is due to its simplicity—it's just simple key-value pairs. Anything more complex than that is left up to the application developer.

The abstractions that we've created around `AsyncStorage` in this example are simple. The idea is to simply set and get items. However, even simple things like this warrant a little abstraction. For example, the `setItem()` method we've implemented here will make the asynchronous call to `AsyncStorage` and update the `items` state once that has completed. Loading items is even more complicated because we need to get the keys and values as two separate asynchronous operations.

So, you might be wondering, why the need for all the asynchronicity for simple storage calls? The main reason is to keep the UI responsive. If there are pending screen repaints that need to happen while data is being written to disk, preventing those from happening by blocking them would lead to a sub-optimal user experience.

Synchronizing application data

So far in this chapter, you've learned how to detect the state of a network connection, and how to store data locally in a React Native application. Now it's time to combine these two concepts and implement an app that can detect network outages and continue to function.

The basic idea is to only make network requests when we know for sure that the device is online. If we know that it isn't, we can store any changes in state locally. Then, when we're back online, we can synchronize those stored changes with the remote API.

Let's implement a simplified React Native app that does this. The first step is implementing an abstraction that sits between the React components and the network calls that store data. We'll call this module `store.js`:

```
import {
  NetInfo,
  AsyncStorage,
} from 'react-native';
import { Map as ImmutableMap } from 'immutable';
```

```
// Mock data that would otherwise come from a real
// networked API endpoint.
const fakeNetworkData = {
  first: false,
  second: false,
  third: false,
};

// We'll assume that the device isn't "connected"
// by default.
let connected = false;

// There's nothing to sync yet...
const unsynced = [];

// Sets the given "key" and "value". The idea
// is that application that uses this function
// shouldn't care if the network is connected
// or not.
export const set = (key, value) =>
  // The returned promise resolves to true
  // if the network is connected, false otherwise.
  new Promise((resolve, reject) => {
    if (connected) {
      // We're online - make the proper request (or fake
      // it in this case) and resolve the promise.
      fakeNetworkData[key] = value;
      resolve(true);
    } else {
      // We're offline - save the item using "AsyncStorage"
      // and add the key to "unsynced" so that we remember
      // to sync it when we're back online.
      AsyncStorage
        .setItem(key, value.toString())
        .then(
          () => {
            unsynced.push(key);
            resolve(false);
          },
          err => reject(err)
        );
    }
  });
}

// Gets the given key/value. The idea is that the application
// shouldn't care whether or not there is a network connection.
// If we're offline and the item hasn't been synced, read it
// from local storage.
```

```
export const get = key =>
  new Promise((resolve, reject) => {
    if (connected) {
      // We're online. Resolve the requested data.
      resolve(
        key ?
          fakeNetworkData[key] :
          fakeNetworkData
      );
    } else if (key) {
      // We've offline and they're asking for a specific key.
      // We need to look it up using "AsyncStorage".
      AsyncStorage
        .getItem(key)
        .then(
          item => resolve(item),
          err => reject(err)
        );
    } else {
      // We're offline and they're asking for all values.
      // So we grab all keys, then all values, then we
      // resolve a plain JS object.
      AsyncStorage
        .getAllKeys()
        .then(
          keys => AsyncStorage
            .multiGet(keys)
            .then(
              items => resolve(ImmutableMap(items).toJS()),
              err => reject(err)
            ),
            err => reject(err)
        );
    }
  });
}

// Check the network state when the module first
// loads so that we have an accurate value for "connected".
NetInfo.isConnected
  .fetch()
  .then(
    (isConnected) => { connected = isConnected; },
    () => { connected = false; }
  );

// Register a handler for when the state of the network changes.
NetInfo.addEventListener(
  'change',
```

```
(info) => {
  // Update the "connected" state...
  connected = [
    'wifi',
    'unknown',
  ].includes(info.toLowerCase());

  // If we're online and there's unsynced values,
  // load them from the store, and call "set()"
  // on each of them.
  if (connected && unsynced.length) {
    AsyncStorage
      .multiGet(unsynced)
      .then((items) => {
        items.forEach(([key, val]) => set(key, val));
        unsynced.length = 0;
      });
  }
};


```

This module exports two functions—`set()` and `get()`. Their job, unsurprisingly, is to set and get data, respectively. Since this is just a demonstration of how to sync between local storage and network endpoints, this module just mocks the actual network with the `fakeNetworkData` object.

Let's start by looking at the `set()` function. As you can see, it's an asynchronous function that will always return a promise that resolves to a Boolean value. If it's true, it means that we're online, and that the call over the network was successful. If it's false, it means that we're offline, and `AsyncStorage` was used to save the data.

The same approach is used with the `get()` function. It returns a promise that resolves a Boolean value that indicates the state of the network. If a key argument is provided, then the value for that key is looked up. Otherwise, all values are returned, either from the network or from `AsyncStorage`.

In addition to these two functions, this module does two other things. It uses `NetInfo.fetch()` to set the `connected` state. Then, it adds a listener for changes in the network state. This is how items that have been saved locally when we're offline become synced with the network when it's connected again.

Okay, now let's check out the main application that uses these functions:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  View,
  Switch,
  NetInfo,
} from 'react-native';
import { fromJS } from 'immutable';

import styles from './styles';
import { set, get } from './store';

// Used to provide consistent boolean values
// for actual booleans and their string representations.
const boolMap = {
  true: true,
  false: false,
};

class SynchronizingData extends Component {

  // The message state is used to indicate that
  // the user has gone offline. The other state
  // items are things that the user wants to change
  // and sync.
  state = {
    data: fromJS({
      message: null,
      first: false,
      second: false,
      third: false,
    }),
  }

  // Getter for "Immutable.js" state data...
  get data() {
    return this.state.data;
  }

  // Setter for "Immutable.js" state data...
  set data(data) {
    this.setState({ data });
  }

  // Generates a handler function bound to a given key.
```

```
save = key => (value) => {
  // Calls "set()" and depending on the resolved value,
  // sets the user message.
  set(key, value)
    .then(
      (connected) => {
        this.data = this.data
          .set(
            'message',
            connected ? null : 'Saved Offline'
          )
          .set(key, value);
      },
      (err) => {
        this.data = this.data.set('message', err);
      }
    );
}

componentWillMount() {
  // We have to call "NetInfo.fetch()" before
  // calling "get()" to ensure that the
  // connection state is accurate. This will
  // get the initial state of each item.
  NetInfo.fetch().then(() =>
    get()
      .then(
        (items) => {
          this.data = this.data.merge(items);
        },
        (err) => {
          this.data = this.data.set('message', err);
        }
      )
  );
}

render() {
  // Bound methods...
  const { save } = this;

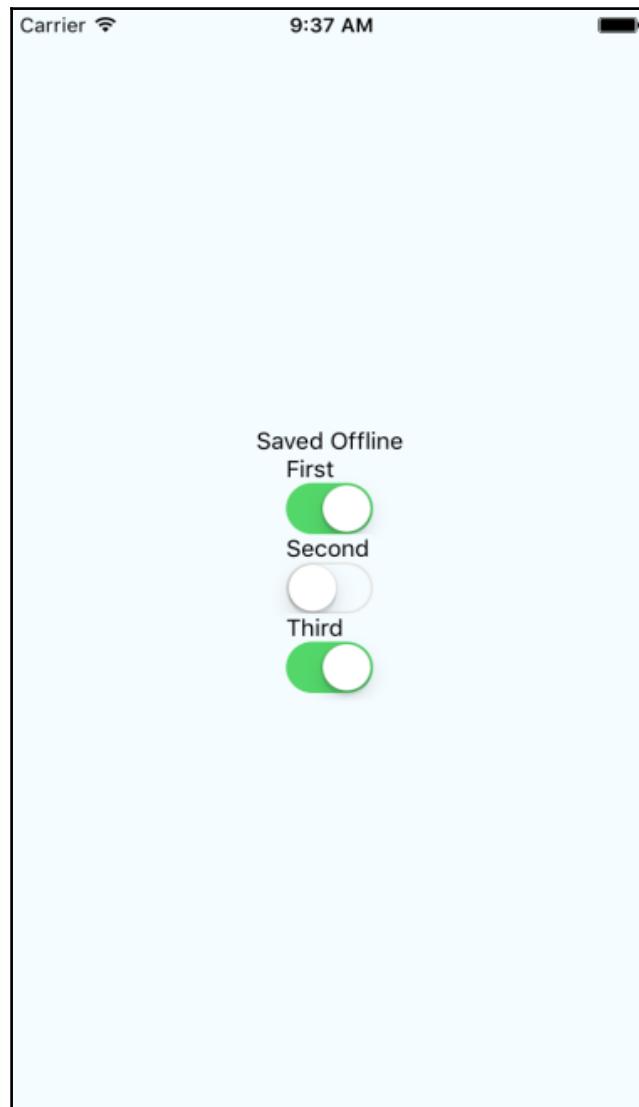
  // State...
  const {
    message,
    first,
    second,
    third,
  } = this.data.toJS();
```

```
return (
  <View style={styles.container}>
    <Text>{message}</Text>
    <View>
      <Text>First</Text>
      <Switch
        value={boolMap[first.toString()]}
        onValueChange={save('first')}
      />
    </View>
    <View>
      <Text>Second</Text>
      <Switch
        value={boolMap[second.toString()]}
        onValueChange={save('second')}
      />
    </View>
    <View>
      <Text>Third</Text>
      <Switch
        value={boolMap[third.toString()]}
        onValueChange={save('third')}
      />
    </View>
  </View>
);
}

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

As you can see, all we're doing here is saving the state of three checkboxes, which is easy enough, except for when you're providing the user with a seamless transition between online and offline modes. Thankfully, our `set()` and `get()` abstractions, implemented in another module, hide most of the details from the application functionality.

You will notice, however, that we need to check the state of the network in this module before we attempt to load any items. If we don't do this, then the `get()` function will assume that we're offline, even if the connection is fine. Here's what the app looks like:



Note that you won't actually see the **Saved Offline** message until you change something in the UI.

Summary

This chapter introduced you to storing data offline in React Native applications. The main reason you would want to store data locally is when the device goes offline and your app can't communicate with a remote API. However, not all applications require API calls and `AsyncStorage` can be used as a general purpose storage mechanism. You just need to implement the appropriate abstractions around it.

You learned how to detect changes in network state in React Native apps as well. It's important to know when the device has gone offline so that your storage layer doesn't make pointless attempts at network calls. Instead, you can let the user know that the device is offline, and then synchronize the application state when a connection is available.

That wraps up the second part of this book. You've seen how to build React components for the Web, and React components for mobile platforms. At the beginning of this book, I posited that the beauty of React lies in the notion of rendering targets. The declarative programming interface of React never has to change. The underlying mechanisms that translate JSX elements are completely replaceable—in theory, we can render React to anything.

In the final part of this book, I'll talk about state in React applications. State and the policies that govern how it flows through an application can make or break the React architecture.

24

Handling Application State

From early on in this book, you've been using state to control your React components. State is an important concept in any React application because it forms the essence of what the user interacts with. Without state, you just have a bunch of empty React components.

In this chapter, you'll learn about Flux and how it can serve as the basis of your information architecture. Then, we'll think about how to build an architecture that best serves web and mobile architectures. We'll also take a look at the Redux library before we talk about some of the limitations of React architectures and how you might overcome them.

Information architecture and Flux

It can be difficult to think of user interfaces as information architectures. More often, we get a rough idea of what the UI should look and behave like, and then we'll take a stab at implementing it. I do this all the time, and it's a great way to get the ball rolling, to discover issues with your approach early, and so on. But then I like to take a step back, and picture what's happening without any widgets. Inevitably, what I've built is flawed in terms of how state flows through the various components. This is fine; at least I have something to work with now. I just have to make sure that I address the information architecture before building too much.

Flux is a set of patterns created by Facebook that help developers think about their information architecture in a way that fits naturally in their apps. I'll go over the key concepts of Flux next, and then we can think about applying these ideas to a unified React architecture.

Unidirectionality

Earlier in this book, I introduced the container pattern for React components. It's pretty simple. The container component has state, but it doesn't actually render any UI elements. Instead, it renders other React components and passes in its state as properties. Whenever the container state changes, the child components are re-rendered with new property values. This is unidirectional data flow.

Flux takes this idea and applies it to something called a store. A store is an abstract concept that holds application state. As far as I'm concerned, a React container is a perfectly valid Flux store. I'll have more to say about stores in a moment. First, I want you to understand why unidirectional data flows are advantageous.

There's a good chance that you've implemented a UI component that changes state, but you're not always sure how it happens. Was it the result of some event in another component? Was it a side-effect from some network call completing? When that happens, you spend lots of time chasing down where the update came from. The effect is often a cascading game of whack-a-mole. When changes can only come from one direction, you can eliminate a number of other possibilities, thus, making the architecture as a whole more predictable.

Synchronous update rounds

When you change the state of a React container, it will re-render its children, who re-render their children, and so on. In Flux terminology, this is called an update round. From the time state changes to the time that the UI elements reflect this change, this is the boundary of the round. It's nice to be able to group the dynamic parts of application behavior into larger chunks like this because it's easier to reason about cause and effect.

A potential problem with React container components is that they can interweave with one another and render in a non-deterministic order. For example, what if some API call completes and causes a state update to happen before the rendering has completed in another update round? The side-effects of asynchronicity can accumulate and morph into unsustainable architectures if not taken seriously.

The solution in Flux architectures is to enforce synchronous update rounds, and to treat attempts to sidestep the update round order as an error. JavaScript is a single-threaded, run-to-completion environment, so we might as well embrace this fact by working with it rather than against it. Update the whole UI, and then update the whole UI again. It turns out that React is a really good tool for this job.

Predictable state transformations

So we know that, in Flux architecture, we have this thing called a store used to hold application state. We know that when state changes, it happens synchronously and unidirectionally, making the system as a whole more predictable and easy to reason about. However, there's still one more thing we can do to ensure that side-effects aren't introduced.

We're keeping all of our application state in a store, which is great, but we can still break things by mutating data in other places. These mutations might seem fairly innocent at first glance, but they're toxic to our architecture. For example, the callback function that handles a `fetch()` call might manipulate the data before passing it to the store. An event handler might generate some structure and pass it to the store. There are limitless possibilities.

The problem with performing these state transformations outside the store is that we don't necessarily know that they're happening. Think of mutating data as the butterfly effect: one small change has far-reaching consequences that aren't obvious at first. The solution is simple. Only mutate state in the store, without exception. It's predictable and easy to trace the cause and effect of your React architecture this way.

I've been using `Immutable.js` for state in most of the examples throughout the book. This will come in handy when you're thinking about state transformations in Flux architecture on a large scale. Controlling where state transformations take place is important, but so is state immutability. It helps to enforce the ideas of Flux architecture, and we'll explore these ideas in more depth momentarily when we look at Redux.

Unified information architecture

Let's take a moment to recap the ingredients of our application architecture so far:

- **React Web:** Applications that run in web browsers
- **React Native:** Applications that run natively on mobile platforms
- **Flux:** Patterns for scalable data in React applications

Remember, React is just an abstraction that sits on top of a render target. The two main render targets are browsers and mobile native. This list will likely grow, so it's up to us to design our architecture in a way that doesn't exclude future possibilities. The challenge is that you're not porting a web application to a native mobile application; they're different applications, but they serve the same purpose.

Having said that, is there a way that we can still have some kind of unified information architecture based on ideas from Flux that can be used by these different applications? The best answer I can come up with, unfortunately, is: sort of. You don't want to let the different web and mobile user experiences lead to drastically different approaches in handling state. If the goals of the applications are the same, then there has to be some common information that we can share, using the same Flux concepts.

The difficult part is the fact that web and native mobile are different experiences, because this means that the shape of our application state will be different. It has to be different; otherwise, we would just be porting from one platform to the other, which defeats the purpose of using React Native to leverage capabilities that don't exist in browsers.

So, let's see what we can implement.

Implementing Redux

We'll use a library called Redux to implement a basic application that demonstrates the Flux architecture. Redux doesn't strictly follow the patterns set forth by Flux. Instead, it borrows key ideas from Flux, and implements a small API to make it easy to implement Flux.

The application itself will be a newsreader, a specialized reader for hipsters that you probably haven't heard of. It's a simple app, but I want to highlight the architectural challenges as we walk through the implementation. Even simple apps get complex when you're actually paying attention to what's going on with the data.

We're going to implement three versions of this app. We'll start with the web version, and then we'll implement mobile native apps for iOS and Android. You'll see how you can share architectural concepts between your apps. This lowers the conceptual overhead when you need to implement the same application on several platforms. We're implementing three apps right now, but this will likely be more in the future as React expands its rendering capabilities.



Once again, I urge you to download the code samples for this book from <https://github.com/PacktPublishing/React-and-React-Native>. There are a lot of little details that I simply do not have room to cover in this book, especially for these example apps we're about to look at. I'll do my best to cover the important pieces, but I can't learn by tinkering for you.

Initial application state

Let's start by looking at the initial state of the Flux store. In Redux, the entire state of the application is represented by a single store. Here's what it looks like:

```
import { fromJS } from 'immutable';

// The state of the application is contained
// within an Immutable.js Map. Each key represents
// a "slice" of state.
export default fromJS({
    // The "App" state is the generic state that's
    // always visible. This state is not specific to
    // one particular feature, in other words. It has
    // the app title, and links to various article
    // sections.
    App: {
        title: 'Neckbeard News',
        links: [
            { name: 'All', url: '/' },
            { name: 'Local', url: 'local' },
            { name: 'Global', url: 'global' },
            { name: 'Tech', url: 'tech' },
            { name: 'Sports', url: 'sports' },
        ],
    },
    // The "Home" state is where lists of articles are
    // rendered. Initially, there are no articles, so
    // the "articles" list is empty until they're fetched
    // from the API.
    Home: {
        articles: [],
    },
    // The "Article" state represents the full article. The
    // assumption is that the user has navigated to a full
    // article page and we need the entire article text here.
    Article: {
        full: '',
    },
});
```

This module exports an `Immutable.js Map` instance. You'll see why we want to do this later on. But for now, let's look at the organization of this state. In Redux, you divide up application state by slices. In this case, it's a simple application, so the store only has three slices of state. Each slice of state is mapped to a major application feature.

For example, the `Home` key represents state that's used by the `Home` component of our app. It's important to initialize any state, even if it's an empty object or array, so that our components have initial properties. Now let's use some Redux functions to create an actual store that's used to get data to our React components.

Creating the store

The initial state is useful when the application first starts. This is enough to render components, but that's about it. Once the user starts interacting with the UI, we need a way to change the state of the store. In Redux, you assign a reducer function to each slice of state in your store. So, for example, our app would have a `Home` reducer, an `App` reducer, and an `Article` reducer.

The key concept of a reducer in Redux is that it's pure and side-effect free. This is where having `Immutable.js` structures as state comes in handy. Let's see how to tie our initial state to the reducer functions that will eventually change the state of our store:

```
import { createStore } from 'redux';
import { combineReducers } from 'redux-immutable';

// So build a Redux store, we need the "initialState"
// and all of our reducer functions that return
// new state.
import initialState from './initialState';
import App from './App';
import Home from './Home';
import Article from './Article';

// The "createStore()" and "combineReducers()" functions
// perform all of the heavy-lifting.
export default createStore(combineReducers({
  App,
  Home,
  Article,
}), initialState);
```

Pretty simple! The `App`, `Home`, and `Article` functions are named in exactly the same way as the slice of state that they manipulate. This makes it easier to add new states and reducer functions as the application grows.

We now have a Redux store that's ready to go (we'll look at reducer functions shortly). But we still haven't hooked it up to the React components that actually render state. Let's take a look at how to do this now.

Store provider and routes

Redux has a `Provider` component (technically, it's the `react-redux` package that provides it) that's used to wrap the top-level components of our application. This will ensure that Redux store data is available to every component in our application.

In the hipster news reader app we're developing, we'll wrap the `Router` component with a `Provider` component. Then, as we build our components, we know that store data will be available. Here's what this looks like:

```
import React from 'react';
import { Provider } from 'react-redux';
import {
  Router,
  Route,
  IndexRoute,
  browserHistory,
} from 'react-router';

// Components that render application state.
import App from './components/App';
import Home from './components/Home';
import Article from './components/Article';

// Our Redux/Flux store.
import store from './store';

// Higher order component for making the
// various article section components out of
// the "Home" component. The only difference
// is the "filter" property. Having unique JSX
// element names is easier to read than a bunch
// of different property values.
const articleList = filter => props => (
  <Home {...props} filter={filter} />
);
```

```
const Local = articleList('local');
const Global = articleList('global');
const Tech = articleList('tech');
const Sports = articleList('sports');

// Routes to the home page, the different
// article sections, and the article details page.
// The "<Provider>" element is how we pass Redux
// store data to each of our components.
export default (
  <Provider store={store}>
    <Router history={browserHistory}>
      <Route path="/" component={App}>
        <IndexRoute component={Home} />
        <Route path="local" component={Local} />
        <Route path="global" component={Global} />
        <Route path="tech" component={Tech} />
        <Route path="sports" component={Sports} />
        <Route path="articles/:id" component={Article} />
      </Route>
    </Router>
  </Provider>
);
```

The store that we created by taking initial state and combining it with reducer functions is passed to `<Provider>`. This means that, when our reducers cause the Redux store to change, the store data is automatically passed to each application component. We'll take a look at the `App` component next.

The App component

The `App` component is always visible and includes the page heading and a list of links to various article categories. When the user moves around the user interface, the `App` component is always rendered, but its child components change. Let's take a look at the component, and then we'll break down how it works:

```
import React, { PropTypes } from 'react';
import { IndexLink } from 'react-router';
import { connect } from 'react-redux';

const App = ({
  title,
  links,
  children,
}) => (
```

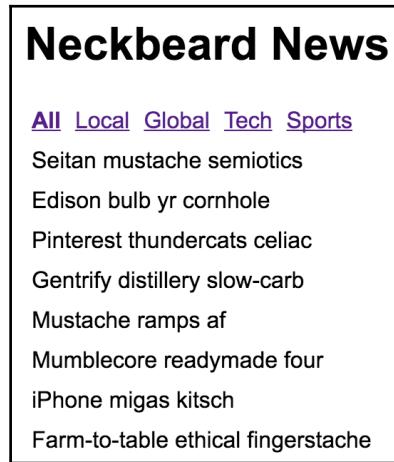
```
<main>
  <h1>{title}</h1>
  <ul style={categoryListStyle}>
    { /* Renders a link for each article category.
       The key thing to note is that the "links"
       value comes from a Redux store. */ }
    {links.map(l => (
      <li key={l.url} style={categoryItemStyle}>
        <IndexLink
          to={l.url}
          activeStyle={{ fontWeight: 'bold' }}>
          {l.name}
        </IndexLink>
      </li>
    )));
  </ul>
  <section>
    {children}
  </section>
</main>
);

App.propTypes = {
  title: PropTypes.string.isRequired,
  links: PropTypes
    .arrayOf(PropTypes.shape({
      name: PropTypes.string.isRequired,
      url: PropTypes.string.isRequired,
    })).isRequired,
  children: PropTypes.node,
};

export default connect(
  state => state.get('App').toJS()
)(App);
```

The JSX content itself is quite simple. It requires a `title` property and a `links` property. Both of these values are actually states that come from the Redux store. Note that we're exporting a higher-order component, created using the `connect()` function. This function accepts a callback function that transforms the store state into properties that the component needs.

In this example, we need the App state. Turning this map into a plain JavaScript object is easy enough with the `toJS()` method. This is how Redux state is passed to components. Here's what the rendered content of the App component looks like:



Ignore the amazing article titles for a moment; we'll return to these briefly. The title and the category links are rendered by the `App` component. The article titles are rendered by a child component of `App`.

Notice how the **All** category is bold? This is because it's the currently selected category. If the **Local** category is selected, the **All** text will go back to regular font, and the **Local** text will be emboldened. This is all controlled through Redux state. Let's take a look at the `App` reducer function now:

```
import { fromJS } from 'immutable';
import initialState from './initialState';

// The initial page heading.
const title = initialState.getIn(['App', 'title']);

// Links to display when an article is displayed.
const articleLinks = fromJS([
  {
    name: 'Home',
    url: '/',
  },
]);

// Links to display when we're on the home page.
const homeLinks = initialState.getIn(['App', 'links']);

// Maps the action type to a function
```

```
// that returns new state.
const typeMap = fromJS({
  // The article is being fetched, adjust
  // the "title" and "links" state.
  FETCHING_ARTICLE: state =>
    state
      .set('title', '...')
      .set('links', articleLinks),

  // The article has been fetched. Set the title
  // of the article.
  FETCH_ARTICLE: (state, payload) =>
    state.set('title', payload.title),

  // The list of articles are being fetched. Set
  // the "title" and the "links".
  FETCHING_ARTICLES: state =>
    state
      .set('title', title)
      .set('links', homeLinks),

  // The articles have been fetched, update the
  // "title" state.
  FETCH_ARTICLES: state =>
    state.set('title', title),
});

// This reducer relies on the "typeMap" and the
// "type" of action that was dispatched. If it's
// not found, then the state is simply returned.
export default (state, { type, payload }) =>
  typeMap.get(type, () => state)(state, payload);
```

There are two points I'd like to make about this reducer logic. First, you can now see how having immutable data structures in place makes this code concise and easy to follow. Second, a lot of state handling happens here in response to simple actions. Take the `FETCHING_ARTICLE` and `FETCHING_ARTICLES` actions for example. We want to change the UI before actually issuing a network request. I think this type of explicitness is the real value of Flux and Redux. We know exactly why something changes. It's explicit, but not verbose.

The Home component

The last major piece of the Redux architecture that's missing from this picture is the action creator functions. These are called by components in order to dispatch payloads to the Redux store. The end result of dispatching any action is a change in state. However, some actions need to go and fetch state before they can be dispatched to the store as a payload.

Let's look at the Home component of our Neckbeard News app. It'll show you how you can pass along action creator functions when wiring up components to the Redux store. Here's the code:

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { Link } from 'react-router';
import { Map } from 'immutable';

// What to render when the article list is empty
// (true/false). When it's empty, a single ellipses
// is displayed.
const emptyMap = Map()
  .set(true, (<li style={listItemStyle}>...</li>))
  .set(false, null);

class Home extends Component {
  static propTypes = {
    articles: PropTypes.arrayOf(
      PropTypes.object
    ).isRequired,
    fetchingArticles: PropTypes.func.isRequired,
    fetchArticles: PropTypes.func.isRequired,
    toggleArticle: PropTypes.func.isRequired,
    filter: PropTypes.string.isRequired,
  }

  static defaultProps = {
    filter: '',
  }

  // When the component is mounted, there's two actions
  // to dispatch. First, we want to tell the world that
  // we're fetching articles before they're actually
  // fetched. Then, we call "fetchArticles()" to perform
  // the API call.
  componentWillMount() {
    this.props.fetchingArticles();
    this.props.fetchArticles(this.props.filter);
  }
}
```

```
// When an article title is clicked, toggle the state of
// the article by dispatching the toggle article action.
onTitleClick = id => () =>
  this.props.toggleArticle(id);

render() {
  const { onTitleClick } = this;
  const { articles } = this.props;

  return (
    <ul style={listStyle}>
      {emptyMap.get(articles.length === 0)}
      {articles.map(a => (
        <li key={a.id} style={listItemStyle}>
          <button
            onClick={onTitleClick(a.id)}
            style={titleStyle}
          >
            {a.title}
          </button>
          { /* The summary of the article is displayed
              based on the "display" property. This state
              is toggled when the user clicks the title. */ }
          <p style={{ display: a.display }}>
            <small>
              <span>{a.summary} </span>
              <Link to={`articles/${a.id}`}>More...</Link>
            </small>
          </p>
        </li>
      ))}
    </ul>
  );
}

// The "connect()" function connects this component
// to the Redux store. It accepts two functions as
// arguments...
export default connect(
  // Maps the immutable "state" object to a JavaScript
  // object. The "ownProps" are plain JSX props that
  // are merged into Redux store data.
  (state, ownProps) => Object.assign(
    state.get('Home').toJS(),
    ownProps
  ),

```

```
// Sets the action creator functions as props. The
// "dispatch()" function is what actually invokes
// store reducer functions that change the state
// of the store, and cause new prop values to be passed
// to this component.
dispatch => ({
  fetchingArticles: () => dispatch({
    type: 'FETCHING_ARTICLES',
  }),

  fetchArticles: (filter) => {
    const headers = new Headers();
    headers.append('Accept', 'application/json');

    fetch(`/api/articles/${filter}`, { headers })
      .then(resp => resp.json())
      .then(json => dispatch({
        type: 'FETCH_ARTICLES',
        payload: json,
      }));
  },
  toggleArticle: payload =>
    dispatch({
      type: 'TOGGLE_ARTICLE',
      payload,
    }),
})
)(Home);
```

Let's focus on the `connect()` function, which is used to connect the `Home` component to the store. The first argument is a function that takes relevant state from the store and returns it as `props` for this component. It's using `ownProps` so that we can pass `props` directly to the component and override anything from the store. The `filter` property is why we need this capability.

The second argument is a function that returns action creator functions as `props`. The `dispatch()` function is how these action creator functions are able to deliver payloads to the store. For example, the `toggleArticle()` function is a simple call directly to `dispatch()`, and is called in response to the user clicking the article title. However, the `fetchingArticles()` call involves asynchronous behavior. This means that `dispatch()` isn't called until the `fetch()` promise resolves. It's up to us to make sure that nothing unexpected happens in between.

Let's wrap things up by looking at the reducer function that's used with the `Home` component:

```
import { fromJS } from 'immutable';

const typeMap = fromJS({

  // Clear any old articles right before
  // we fetch new articles.
  FETCHING_ARTICLES: state =>
    state.update('articles', a => a.clear()),

  // Articles have been fetched. Update the
  // "articles" state, and make sure that the
  // summary display is "none".
  FETCH_ARTICLES: (state, payload) =>
    state.set(
      'articles',
      fromJS(payload)
        .map(a => a.set('display', 'none'))
    ),

  // Toggles the state of the selected article
  // "id". First we have to find the index of
  // the article so that we can update it's
  // "display" state. If it's already hidden,
  // we show it, and vice-versa.
  TOGGLE_ARTICLE: (state, id) =>
    state.updateIn([
      'articles',
      state
        .get('articles')
        .findIndex(a => a.get('id') === id),
      'display',
    ], display =>
      display === 'none' ?
        'block' : 'none'
    ),
  });

export default (state, { type, payload }) =>
  typeMap.get(type, s => s)(state, payload);
```

The same technique of using a type map to change state based on the action type is used here. Once again, this code is easy to reason about, yet everything that can change in the system is explicit.

State in mobile apps

What about using Redux in React Native mobile apps? Of course you should, if you're developing the same application for the web and for native platforms. In fact, I've implemented Neckbeard News in React Native for both iOS and Android. I encourage you to download the code for this book and get this application running for both web and native mobile.

There really is no difference in how you actually use Redux in a mobile app. The only difference is in the shape of state that's used. In other words, don't think that you can use the exact same Redux store and reducer functions in the web and native versions of your app. Think about React Native components. There's no one-size-fits-all component for many things. You have some components that are optimized for the iOS platform while others are optimized for the Android platform. It's the same idea with Redux state. Here's what the initial state looks like for mobile Neckbeard News:

```
import { fromJS } from 'immutable';

export default fromJS({
  Main: {
    title: 'All',
    component: 'articles',
  },
  Categories: {
    items: [
      {
        title: 'All',
        filter: '',
        selected: true,
      },
      {
        title: 'Local',
        filter: 'local',
        selected: false,
      },
      {
        title: 'Global',
        filter: 'global',
        selected: false,
      },
    ],
  },
});
```

```
{  
  title: 'Tech',  
  filter: 'tech',  
  selected: false,  
},  
{  
  title: 'Sports',  
  filter: 'sports',  
  selected: false,  
},  
],  
,  
Articles: {  
  filter: '',  
  items: [],  
},  
Article: {  
  full: '',  
},  
});
```

As you can see, the same principles that apply in a Web context apply here in a mobile context. It's just the state itself that differs, in order to support the given components we're using and the unique ways that you're using them to implement your application.

Scaling the architecture

By now, you probably have a pretty good handle on Flux concepts, the mechanisms of Redux, and how they're used to implement sound information architectures for React applications. The question then becomes, How sustainable is this approach, and can it handle arbitrarily large and complex applications?

I think Redux is a great way to implement large-scale React applications. You can predict what's going to happen as the result of any given action because everything is explicit. It's declarative. It's unidirectional and without side-effects. But, it isn't without challenges.

The limiting factor with Redux is also its bread and butter; because everything is explicit, applications that need to scale up, in terms of feature count and complexity, ultimately end up with more moving parts. There's nothing wrong with this; it's just the nature of the game. The unavoidable consequence of scaling up is slowing down. You simply cannot grasp enough of the big picture in order to implement things quickly.

In the final two chapters of this book, we're going to look at a related but different approach to Flux; Relay/GraphQL. I think this technology can scale in ways that Redux cannot.

Summary

In this chapter, you learned about Flux, a set of architectural patterns that aid in building information architecture for your React application. The key ideas with Flux involve unidirectional data flow, synchronous update rounds, and predictable state transformations.

Next, we walked through a detailed implementation of a Redux/React application. Redux provides a simplified implementation of Flux ideas. The benefit is predictability everywhere.

Then, we discussed whether or not Redux has what it takes to build scalable architectures for our React applications. The answer is yes, for the most part. For the remainder of this book, however, we're going to explore Relay and GraphQL to see if these technologies can scale our applications to the next level.

25

Why Relay and GraphQL?

In the preceding chapter, we looked at the architectural principles of Flux. In particular, we used the Redux library to implement some concrete Flux concepts in a React application. Having a framework of patterns like Flux in place, to help you reason about how state changes and flows through your application, is definitely a good thing. At the end of the chapter, we thought about some potential limitations in terms of scale.

In this chapter, I'm going to walk you through yet another approach to handling state in a React application. Like Redux, Relay is used with both web and mobile React applications. Relay relies on a language called GraphQL used to fetch resources and to mutate those resources.

The premise of Relay is that it can scale in ways that Redux and other approaches to handling state are limiting. It does this by eliminating them, and keeping the focus on the data requirements of the component.

In the final chapter of this book, we'll dive into a React Native implementation of the ever popular **Todo MVC** application.

Yet another approach?

This was the exact question I had when I learned of Relay + GraphQL. Then I reminded myself that the beauty of React is that it's just the view abstraction of the UI; of course there's going to be many approaches to handling data. So the real question is, what makes Relay better or worse than something like Redux?

At a high level, you can think of Relay as an implementation of Flux architecture patterns, and you can think of GraphQL as the interface that describes how the Flux stores encapsulated within Relay work. At a more practical level, the value of Relay is ease of implementation. For example, with Redux, you have a lot of implementation work to do, just to populate the stores with data. This isn't difficult to do, but it does get verbose over time. It's this verbosity that makes Redux difficult to scale beyond a certain point.

It's not the individual data points that are difficult to scale. It's the aggregate effect of having 'lots of fetch requests that end up building very complicated stores. Relay changes this by allowing you to declare the data that a given component needs and letting Relay figure out the best way to fetch this data and synchronize it with the local store.

Is the Relay approach better than Redux and other approaches for handling data in React applications? In some respects, yes, it is. Is it perfect? Far from it. There is a learning curve involved, and not everyone is able to grok it. It's immutable, and parts of it are difficult to use. However, just knowing the premise of the Relay approach and seeing it in action is worth your while, even if you decide against it.

Now, let's pick apart some vocabulary.

Verbose vernacular

Before I start going into more depth on data dependencies and mutations, I think it makes sense for me to throw some general Relay + GraphQL terminology definitions out there.

- **Relay:** A library that manages application data fetching and data mutations and provides higher-order components that feed data into our application components
- **GraphQL:** A query language used to specify data requirements and data mutations
- **Data dependency:** An abstract concept that says a given React component depends on particular data
- **Query:** A query is the part of a data dependency, expressed in GraphQL syntax and executed by an encapsulated Relay mechanism
- **Fragment:** A part of a larger GraphQL query
- **Container:** A Relay React component that passes fetched data into the application React component

- **Mutation:** A special type of GraphQL query that changes the state of some remote resource, and Relay has to figure out how to reflect this change in the frontend once it completes

Confused yet? Good. Let's quickly talk about data dependencies and mutations so that we can look at some application code.

Declarative data dependencies

Relay uses the term colocation to describe declarative data dependencies that live beside the component that uses the data. This means that we don't have to go digging around for action creator functions that actually get the component data that is scattered across several modules. With colocation, we can see exactly what the component needs.

Let's get a taste of what this looks like. If you want to display the first and last name of a user, you need to tell Relay that your component needs this data. Then, you can rest assured that the data will always be there for your component. Here's an example:

```
const User = ({ first, last }) => (
  <section>
    <p>{first}</p>
    <p>{last}</p>
  </section>
);

const UserContainer = Relay.createContainer(User, {
  fragments: {
    user: () => Relay.QL`fragment on User {
      first,
      last,
    }`,
  },
});
```

We have two components here. First, there's the `User` component. This is the application component that actually renders the UI elements for the `first` and `last` name data. Note that this is just a plain old React component, rendering props that are passed to it. As you can see with the `UserContainer` component that we've created, Relay follows the container pattern that you learned about earlier in this book. It's in the `createContainer()` function that we specify the data dependencies that this component needs by passing a fragment of GraphQL syntax.

Once again, don't dwell on the Relay/GraphQL specifics just yet. The idea here is to simply illustrate that this is all the code that we need to write to get our component the data it needs. The rest is just bootstrapping the Relay query mechanism, which you'll see in the next chapter.

Mutating application state

Relay mutations are the actions that cause side effects in your systems, because they change the state of some resource that your UI cares about. What's interesting about Relay mutations is that they care about side effects that happen to your data as a result of a change in the state of something. For example, if you change the name of a user, this will certainly impact a screen that displays the user details. But, it could also impact a listing screen that shows several users.

The idea with Relay mutations is that you can tell them, ahead of time, where the application might be impacted in terms of side effects. Since other components may be affected by a mutation, we have already declared their data dependencies, Relay can handle this sort of thing! Awesome! Let's see what a mutation looks like:

```
class ChangeAgeMutation extends Relay.Mutation {
  static fragments = {
    user: () => Relay.QL`fragment on User {
      id,
    }
  `,
    viewer: () => Relay.QL`fragment on Viewer {
      id,
    }
  `,
  }

  getMutation() {
    return Relay.QL`mutation{changeAge}`;
  }

  getFatQuery() {
    return Relay.QL`fragment on ChangeAgePayload @relay(pattern: true) {
      user {
        age,
      },
      viewer {
        ...
      }
    }
  `}
}
```

```
        users,
    },
}
`;
}

getConfigs() {
    return [
        type: 'FIELDS_CHANGE',
        fieldIDs: {
            user: this.props.user.id,
            viewer: this.props.viewer.id,
        },
    ];
}

getVariables() {
    return {
        age: this.props.age,
        id: this.props.todo.id,
    };
}
}
```

Yikes, that's a lot of code for changing the name of a user. Well, it really isn't that much because this includes handling side effects. Let's do a quick overview of what makes up this mutation:

- `fragments`: These GraphQL snippets tell the mutation what data is used by the component, before the mutation actually happens.
- `getMutation()`: The actual mutation from the server that will change some backend resource.
- `getFatQuery()`: This is how Relay is able to determine what might be affected as a side effect of performing this mutation. For example, the `user` might change, but also the `viewer.users` collection.
- `getConfigs()`: This tells Relay about the type of mutation we are about to perform so that it can plan accordingly. In this case, we're just changing a property value, but a mutation can also add or remove items from a collection, and so on.
- `getVariables()`: The parameters of the mutation that are sent to the backend GraphQL server where the actual mutation is performed.

This is just the declaration of the mutation itself. In the next chapter, we'll walk through performing the actual mutation in response to some event, such as user interaction.

The GraphQL backend and microservices

Everything we've discussed so far about Relay is stuff that's in the browser. Relay needs to send its GraphQL queries somewhere. For this, we need a GraphQL backend. This is pretty easy to implement, using Node.js and a handful of GraphQL libraries. We create what's called a schema, describing all the datatypes, queries, and mutations that will be used.

In the browser, Relay helps you scale your applications by reducing data-flow complexity. You have a means to declare what data is needed, without worrying about how it is fetched. It's the schema in the backend that actually needs to resolve this data.

This is another scaling problem that GraphQL helps address. Modern web applications are composed out of microservices. These are smaller, self-contained API endpoints that serve some particular purpose that's smaller than an entire app (hence the term micro). It's the job of our application to stitch together these microservices and provide the frontend with meaningful data.

Again, we're faced with a scalability issue—how do we maintain a backend that's composed out of many microservices without introducing insurmountable complexity? This is something that GraphQL types excel at. In the following chapter, we'll begin the implementation of our Todo application with the backend GraphQL service.

Summary

The goal of this chapter was to quickly introduce you to the concepts of Relay and GraphQL prior to the final chapter of this book, where we're going to implement some Relay/GraphQL code.

Relay is yet another approach to the state management problem in React applications. It's different in the sense that it reduces the complexities associated with the data fetching code that we have to write with other approaches to Flux, such as Redux.

The two key aspects of Relay are declarative data dependencies and explicit mutation side effect handling. All of this is expressed through GraphQL syntax. In order to have a Relay application, you need a GraphQL backend where the data schema lives. Now, onto the final chapter, where we'll look at these Relay/GraphQL concepts in more detail.

26

Building a Relay React App

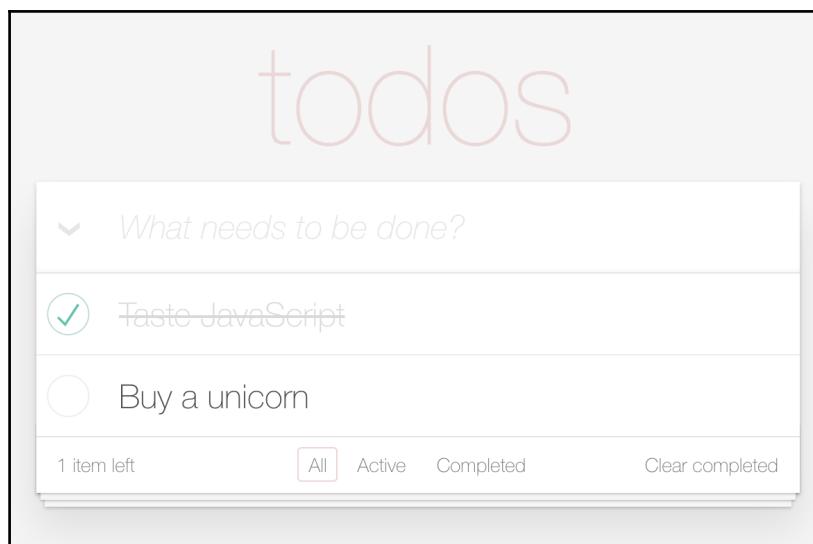
In the previous chapter, you got a ten thousand foot introduction to Relay/GraphQL, and learned why you should use the approach for your React application. With that out of the way, we can build our Todo React Native application and talk about the code as we go. By the end of this chapter, you should feel comfortable about how data moves around in a GraphQL centric architecture. Let's go.

TodoMVC and Relay

I had originally planned to extend the Neckbeard News app that we worked on earlier, in this chapter. Instead, I decided that the TodoMVC example for Relay (<https://github.com/relayjs/relay-examples>) is a robust yet concise example that I would have trouble beating.

So, the plan is this: I'm going to walk you through an example React Native implementation of a Todo app. The key is that it'll use the same GraphQL backend as the web UI. I think this is an absolute win for React developers that want to build both web and native versions of their apps; they can literally share the same schema!

I've included the web version of the TodoMVC app in the code that ships with this book, but I won't dwell on the details of how it works. If you've worked on web development in the past 5 years, you've probably come across a sample Todo app. Here's what the web version looks like:



The functionality is pretty self explanatory, so even if you haven't used any of the TodoMVC apps before, I would recommend playing with this one before trying to implement the native version, which is what we'll be doing for the remainder of the chapter.

The goal of the native version that we're about to implement isn't functional parity. In fact, we're shooting for a very minimal subset of todo functionality. The aim is to show you that Relay works mostly the same on native platforms as it does on web platforms and that the GraphQL backend can be shared between web and native apps.

The GraphQL schema

The schema is the vocabulary used by GraphQL backend server, and the Relay components in the frontend. The GraphQL type system enables the schema to describe the data that's available, and how to put it all together when a query request comes in. This is what makes the whole approach so scalable, the fact that the GraphQL runtime figures out how to put data together. All we need to supply are functions that tell GraphQL where the data is. For example, in a database or in some remote service endpoint.

Let's take a look at the types used in the GraphQL schema for the TodoMVC app:

```
import {
  GraphQLBoolean,
  GraphQLID,
  GraphQLInt,
  GraphQLList,
  GraphQLNonNull,
  GraphQLObjectType,
  GraphQLSchema,
  GraphQLString,
} from 'graphql';

import {
  connectionArgs,
  connectionDefinitions,
  connectionFromArray,
  cursorForObjectInConnection,
  fromGlobalId,
  globalIdField,
  mutationWithClientMutationId,
  nodeDefinitions,
  toGlobalId,
} from 'graphql-relay';

import {
  Todo,
  User,
  addTodo,
  changeTodoStatus,
  getTodo,
  getTodos,
  getUser,
  getViewer,
  markAllTodos,
  removeCompletedTodos,
  removeTodo,
  renameTodo,
} from './database';

const { nodeInterface, nodeField } = nodeDefinitions(
  (globalId) => {
    const { type, id } = fromGlobalId(globalId);

    if (type === 'Todo') {
      return getTodo(id);
    } else if (type === 'User') {
      return getUser(id);
    }
  }
);
```

```
}

    return null;
},
(obj) => {
  if (obj instanceof Todo) {
    return GraphQLTodo;
  } else if (obj instanceof User) {
    return GraphQLUser;
  }
}

return null;
}
);

const GraphQLTodo = new GraphQLObjectType({
  name: 'Todo',
  fields: {
    id: globalIdField('Todo'),
    text: {
      type: GraphQLString,
      resolve: ({ text }) => text,
    },
    complete: {
      type: GraphQLBoolean,
      resolve: ({ complete }) => complete,
    },
  },
  interfaces: [nodeInterface],
});

const {
  connectionType: TodosConnection,
  edgeType: GraphQLTodoEdge,
} = connectionDefinitions({
  name: 'Todo',
  nodeType: GraphQLTodo,
});

const GraphQLUser = new GraphQLObjectType({
  name: 'User',
  fields: {
    id: globalIdField('User'),
    todos: {
      type: TodosConnection,
      args: {
        status: {
          type: GraphQLString,
        }
      }
    }
  }
});
```

```
        defaultValue: 'any',
    },
    ...connectionArgs,
},
resolve: (obj, { status, ...args }) =>
  connectionFromArray(getTodos(status), args),
},
totalCount: {
  type: GraphQLInt,
  resolve: () => getTodos().length,
},
completedCount: {
  type: GraphQLInt,
  resolve: () => getTodos('completed').length,
},
},
interfaces: [nodeInterface],
});

const Root = new GraphQLObjectType({
  name: 'Root',
  fields: {
    viewer: {
      type: GraphQLUser,
      resolve: () => getViewer(),
    },
    node: nodeField,
  },
});
```

There are a lot of things being imported here, so we'll start with the imports. I wanted to include all of these imports because I think they're contextually relevant for this discussion. First, we have the primitive GraphQL types from the `graphql` library. Next, we have a bunch of helpers from the `graphql-relay` library that simplify defining a GraphQL schema. Lastly, we have imports from our own `database` module. This isn't necessarily a database, in fact in this case, it's just mock data. We could replace `database` with `api` for instance, if we needed to talk to remote API endpoints, or we could combine the two; it's all GraphQL as far as our React components are concerned.

Then, we define some of our own GraphQL types. For example, the `GraphQLTodo` type has two fields—`text` and `complete`. One is a Boolean and one is a string. The important thing to note about these fields is the `resolve()` function. This is how we tell the GraphQL runtime how to populate these fields when they're required. These two fields simply return property values.

Then, there's the `GraphQLUser` type. This field represents the user's entire universe within the UI, hence the name. The `todos` field, for example, is how we query for todo items from Relay components. It's resolved using the `connectionFromArray()` function, which is a shortcut that removes the need for more verbose field definitions. Then, there's the `Root` type. This has a single `viewer` field that's used as the root of all queries.

Now, let's take a peek at the add todo mutation. We're not going to go over every mutation that's used by the web version of this app, in the interest of space:

```
const GraphQLAddTodoMutation = mutationWithClientMutationId({
  name: 'AddTodo',
  inputFields: {
    text: { type: new GraphQLNonNull(GraphQLString) },
  },
  outputFields: {
    todoEdge: {
      type: GraphQLTodoEdge,
      resolve: ({ localTodoId }) => {
        const todo = getTodo(localTodoId);

        return {
          cursor: cursorForObjectInConnection(
            getTodos(),
            Todo
          ),
          node: todo,
        };
      },
    },
    viewer: {
      type: GraphQLUser,
      resolve: () => getViewer(),
    },
  },
  mutateAndGetPayload: ({ text }) => {
    const localTodoId = addTodo(text);
    return { localTodoId };
  },
});

const Mutation = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    addTodo: GraphQLAddTodoMutation,
```

```
  ...  
},  
});
```

All mutations have a `mutateAndGetPayload()` method, which is how the mutation actually makes a call to some external service to change the data. The returned payload can be the changed entity, but it can also include data that's changed as a side effect. This is where the `outputFields` come into play. This is the information that's handed back to Relay in the browser so that it has enough information to properly update components based on the side effects of the mutation. Don't worry, we'll see what this looks like from Relay's perspective in a little bit.

The mutation type that we've created here is used to hold all application mutations; obviously, we've omitted some of them here. Lastly, here's how the entire schema is put together and exported from the module:

```
export const schema = new GraphQLSchema({  
  query: Root,  
  mutation: Mutation,  
});
```

We won't worry about how this schema is fed into the GraphQL server for now. If you're interested in how this works, take a look at `server.js`.

Bootstrapping Relay

At this point, we have the GraphQL backend up and running. Now, we can focus on our React components in the frontend. In particular, we're going to look at Relay in a React Native context, which really only has minor differences. For example, in web apps, it's usually `react-router` that bootstraps Relay. In React Native, it's a little different. Let's look at the index file that serves as the entry point for our native app:

```
import React from 'react';  
import { AppRegistry } from 'react-native';  
import Relay, {  
  DefaultNetworkLayer,  
  RootContainer,  
} from 'react-relay';  
  
import viewerQueries from './queries/ViewerQueries';  
import TodoApp from './TodoApp';  
  
// Since this is a native app instead of a web  
// app, we have to tell Relay where to find
```

```
// the GraphQL backend.  
Relay.injectNetworkLayer(  
  new DefaultNetworkLayer('http://localhost:8080')  
)  
  
AppRegistry.registerComponent(  
  'TodoRelayMobile',  
  () => () => (  
    // The "<RootContainer>" component is the entry  
    // point for Relay in React Native. It takes the  
    // main component - "TodoApp" - and uses  
    // "viewerQueries" to kick-off communication  
    // with the backend.  
    <RootContainer  
      Component={TodoApp}  
      route={{  
        name: 'viewer',  
        params: {},  
        queries: viewerQueries,  
      }}  
    />  
  )  
)  
);
```

There's some extra things that need to happen here. The interesting bit is the `queries` prop that's passed to the `<RootContainer>` component. Let's take a look at the `ViewerQuery` module:

```
import Relay from 'react-relay';  
  
export default {  
  viewer: () => Relay.QL`query { viewer }`,  
};
```

Cool, so this means that if the `TodoApp` component needs data; it's parent component knows about the `viewer` query. Remember from the GraphQL schema that we created earlier—this is the universe of data available to the app. Anything a component may need, can be found within this query. More on this shortly.

Adding todo items

In the TodoApp component, we'll add a text input that allows the user to enter new todo items. When they're done entering the todo, Relay will need to send a mutation to the backend GraphQL server. Here's what the component code looks like:

```
import React, { Component, PropTypes } from 'react';
import {
  View,
  TextInput,
} from 'react-native';
import Relay from 'react-relay';

import styles from './styles';
import AddTodoMutation from './mutations/AddTodoMutation';
import TodoList from './TodoList';

export class TodoRelayMobile extends Component {
  static propTypes = {
    viewer: PropTypes.any.isRequired,
    relay: PropTypes.shape({
      commitUpdate: PropTypes.func.isRequired,
    }),
  }

  // We need to keep track of new todo item text
  // as the user enters them.
  state = {
    text: '',
  }

  // When the user creates the todo by pressing enter,
  // the "AddTodoMutation" is sent to the backend,
  // with the new "text" and the "viewer" as the
  // arguments.
  onSubmitEditing = ({ nativeEvent: { text } }) => {
    this.props.relay.commitUpdate(
      new AddTodoMutation({
        text,
        viewer: this.props.viewer,
      })
    );

    this.setState({ text: '' });
  }

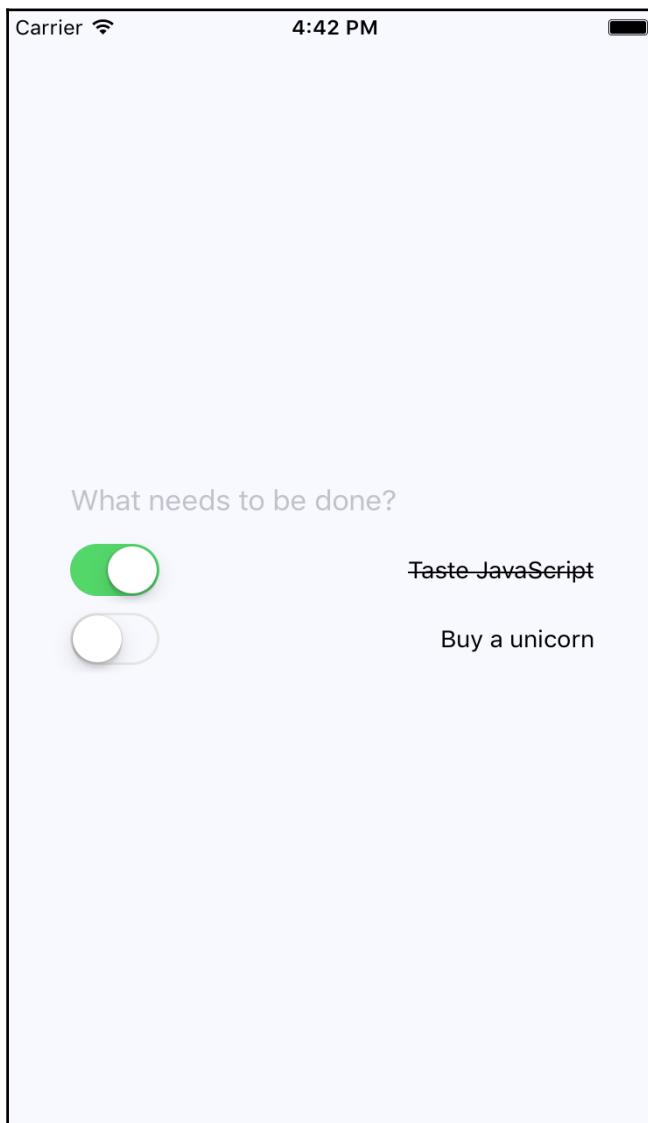
  onChangeText = text => this.setState({ text })
}
```

```
render() {
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.textInput}
        placeholder="What needs to be done?"
        onSubmitEditing={this.onSubmitEditing}
        onChangeText={this.onChangeText}
        value={this.state.text}
      />
      <TodoList viewer={this.props.viewer} />
    </View>
  );
}
}
```

It doesn't look all that different from your typical React Native component. The piece that stands out is the mutation—`AddTodoMutation`. This is how we tell the GraphQL backend that we want a new `todo` node created. At this point, the `TodoApp` component is still just a plain React component. This is how we create a Relay container and export it from the module:

```
// Turns the "TodoApp" component into a Relay
// container component. This is where the data
// dependency for "TodoApp" is declared. We tell
// the "queries" value that was passed to "RootContainer"
// that we want a fragment of fields from the "User" type.
export default Relay.createContainer(TodoRelayMobile, {
  fragments: {
    viewer: variables => Relay.QL`fragment on User {
      totalCount,
      ${AddTodoMutation.getFragment('viewer')},
      ${TodoList.getFragment('viewer', ...variables)},
    }
  },
});
```

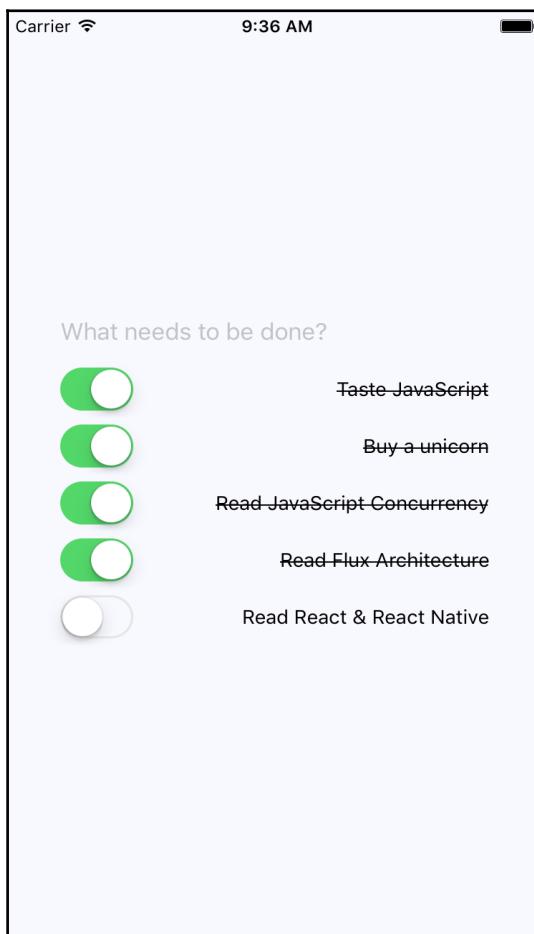
This is how we tell Relay about its data dependencies, including the `AddTodoMutation` that we might send if the user decides to add a new todo. The other thing to notice about this fragment is that it's passing the `viewer` fragment from `TodoList`. That's because even though `TodoApp` doesn't directly use the data that `TodoList` needs, it still needs to tell Relay about it so that when the `TodoList` component is rendered, it can get what it needs from its parent. Let's see what the application looks like so far:



The textbox for adding new todo items is just above the list of todo items. Now, we'll look at the `TodoList` component, which is responsible for rendering the todo item list.

Rendering todo items

It's the job of the `TodoList` component to render items todo list. When `AddTodoMutation` takes place, the `TodoList` component needs to be able to render this new item. Relay takes care of updating the internal data stores where all of our GraphQL data lives. Here's a look at the item list again, with several more todos added:



Here's the TodoList component itself:

```
import React, { PropTypes } from 'react';
import Relay from 'react-relay';
import { View } from 'react-native';

import Todo from './Todo';

// The list component itself is quite simple. Note the
// property that we're using to iterate over - there's
// "edges" and "nodes". This is reflective of a GraphQL
// collection.
const TodoList = ({ viewer }) => (
  <View>
    {viewer.todos.edges.map(edge => (
      <Todo
        key={edge.node.id}
        todo={edge.node}
        viewer={viewer}
      />
    ))}
  </View>
);

TodoList.propTypes = {
  viewer: PropTypes.any.isRequired,
};

export default Relay.createContainer(TodoList, {
  initialVariables: {
    status: null,
  },
  // Variables that are sent along with the query. These
  // can come from UI elements. In this case, we want every
  // item, so we're providing a static value.
  prepareVariables() {
    return {
      status: 'any',
    };
  },
  // The fragments used by this component. Notice the
  // arguments that are passed to the "todos" query -
  // "status" and "first". We're also traversing the
  // structure of the graph using "edges" and "node",
  // so that we can tell the backend exactly what
  // data this component needs.
});
```

```
fragments: {
  viewer: () => Relay.QL`  
    fragment on User {  
      todos(  
        status: $status,  
        first: 2147483647 # max GraphQLInt  
      ) {  
        edges {  
          node {  
            id,  
            ${Todo.getFragment('todo')},  
          },  
        },  
        ${Todo.getFragment('viewer')},  
      }  
    }  
  },
});
```

As you can see, the `fragments` property is where we write the relevant GraphQL to get the data we need. This is the declarative data dependency for the component. Also, the specific GraphQL fragment that's used for the todo item comes from `Todo.getFragment('todo')`. So, we're kind of sharing data dependencies between components. When we render the `<Todo>` component, we're passing it the `edge.todo` data. Now, let's see what the `Todo` component itself looks like.

Completing todo items

The last piece of this application is rendering each todo item and providing the ability to change the status of the todo. Let's take a look at this code:

```
import React, { Component, PropTypes } from 'react';
import Relay from 'react-relay';
import {
  Text,
  View,
  Switch,
} from 'react-native';

import styles from './styles';
import ChangeTodoStatusMutation from
  './mutations/ChangeTodoStatusMutation';

// How to style the todo text, based on the
```

```
// boolean value of the "completed" property.
const completeStyleMap = new Map([
  [true, { textDecorationLine: 'line-through' }],
  [false, {}],
]);

class Todo extends Component {
  static propTypes = {
    relay: PropTypes.any.isRequired,
    viewer: PropTypes.any.isRequired,
    todo: PropTypes.shape({
      text: PropTypes.string.isRequired,
      complete: PropTypes.bool.isRequired,
    }),
  }

  // Handles the "switch" button click. The "complete"
  // argument is the value of the switch UI control,
  // which is sent to the "ChangeTodoStatusMutation".
  onValueChange = complete =>
    this.props.relay.commitUpdate(
      new ChangeTodoStatusMutation({
        complete,
        todo: this.props.todo,
        viewer: this.props.viewer,
      })
    )

  render() {
    // The "todo" is passed in from the "TodoList"
    // component.
    const {
      props: {
        todo: {
          text,
          complete,
        },
      },
      onValueChange,
    } = this;

    // The actual todo is a "<Switch>" component,
    // and the todo item text, styled based on it's
    // "complete" value.
    return (
      <View style={styles.todoItem}>
        <Switch
          value={complete}
```

```
        onValueChange={onValueChange}
      />
      <Text style={completeStyleMap.get(complete)}>
        {text}
      </Text>
    </View>
  );
}
}

// The fragments defined here are actually used
// in the "TodoList" component when it runs the
// "todos" query. We also have to tell it about
// the fragments defined by "ChangeTodoStatusMutation".
export default Relay.createContainer(Todo, {
  fragments: {
    todo: () => Relay.QL`fragment on Todo {
      complete,
      id,
      text,
      ${ChangeTodoStatusMutation.getFragment('todo')},
    }`,
    viewer: () => Relay.QL`fragment on User {
      ${ChangeTodoStatusMutation.getFragment('viewer')},
    }`,
  },
});
```

The actual component that's rendered is quite simple—a switch control and the item text. When the user marks the todo as complete, the item text is styled as crossed off. The user can also uncheck items. The `ChangeTodoStatusMutation` mutation sends the request to the GraphQL backend to change the `todo` state. The GraphQL backend then talks to any microservices that needed to make this happen. Then, it responds with the fields that this component depends on.

The important part of this code that I want to point out is the fragments used in the Relay container. This container doesn't actually use them directly. Instead, they're used by the `todos` query in the `TodoList` component (`Todo.getFragment()`). This is useful because it means that we can use the `Todo` component in another context, with another query, and its data dependencies will always be satisfied.

Summary

In this chapter, we implemented some specific Relay and GraphQL ideas. Starting with the GraphQL schema, you learned how to declare the data that's used by the application and how these data types resolve to specific data sources, such as microservice endpoints. Then, we dove into bootstrapping GraphQL queries from Relay in our React Native app. Next, we walked through the specifics of adding, changing, and listing todo items. The application itself uses the same schema as the web version of the Todo application, which makes things much easier when you're developing web and native React applications.

Well, that's a wrap for this book. We've gone over a lot of material together, and I hope that you've learned from reading it as much as I have by writing it. If there was one theme from this book that you should walk away with, it's that React is simply a rendering abstraction. As new rendering targets emerge, new React libraries will emerge as well. As developers think of novel ways to deal with state at scale, you'll see new techniques and libraries released. My hope is that you're now well prepared to work in this rapidly evolving React ecosystem.

Index

A

abstraction 15
activity modals
 implementing 366, 367, 368
ADB (Android Debug Bridge)
 reference 226
add article component
 functional version 81
 implementing 78
alerts 343
Android apps
 executing 224, 225, 226
Android simulators 219
Android
 versus iOS 213
any property value 120, 121, 122
App component 430, 432
application data
 storing 410, 413, 414
 synchronizing 414, 417, 420
application state
 mutating 444
arrow function 59
article item component
 functional version 81
article list component
 functional version 80
 implementing 74, 76
asynchronous calls
 cleaning up 107, 108
attributes 34
auto bind 56

B

back button
 implementing 263, 264

backend routing 178

C

change in state 34
child route 154
code-splitting 169
component inheritance
 about 133
 event handlers, inheriting 139, 140, 142
 JSX, inheriting 139, 140, 142
 properties, inheriting 136, 137, 139
 state, inheriting 134, 136
component properties 34
component rendering
 efficiency, optimizing 98, 100
 metadata, used for optimization 101, 102, 103
component state
 about 33
 initial component state 35
 merging 39
 setting 35, 37
component structures, refactoring
 about 73
 add article component, implementation 78
 article item component, implementation 76
 article list component, implementation 74
 functional version of components 80
 JSX 73
component trees
 rendering 82
components
 asynchronous calls, cleaning up 107, 108
 cleaning up 107
conditional component
 rendering 143, 144
confirmation 344
container 442

container components 48
custom JSX elements
 HTML, encapsulating 23
 namespace components 26
 nested elements 25
custom property validators
 writing 130, 131

D

data collections
 rendering 269, 271, 272
data dependency 442
data sources
 used, for wrapping components 145
data
 fetching 183, 188, 189
date/time input
 collecting 336, 338, 340, 341
declarative approach
 about 51
 advantage 51
declarative data dependencies 443
declarative programming 12
default property values 41
defaults
 in functional components 47
defensive code 113
dynamic routes 156
dynamic scenes
 about 258
 creating 258, 260, 262

E

elements
 handlers, binding to 60
ES2015 class style declaration 56
ES2015 class syntax 133
event handler context
 auto-binding context 56
 binding 56
 component data, obtaining 57
event handlers
 declaring 51
 inheriting 139, 140, 142
event pooling 62

event property names, React
reference 52

F

fail fast 113
feature components 83
flexbox layouts
 building 232
 flexible grids 241
 flexible rows 239, 241
 flexible rows, combining with columns 244, 247, 248
 three column layout, implementing 232, 233, 235
 three column layout, improving 235, 236, 237, 238
flexbox
 about 228, 229
 URL 229
Flux
 about 146, 423
 predictable state transformations 425
 synchronous update rounds 424
 unidirectionality 424
forms 202, 203, 205, 207
forward button
 implementing 263, 264
fragment 442
frontend reconciliation 182, 183
functional components
 about 45
 defaults 47

G

generic handlers
 importing 53, 55
Genymotion
 about 220
 reference 221
geolocation API 307, 311, 312
gesture responder system
 URL 372
gestures
 cancelling 379, 380, 383
GraphQL 442

GraphQL backend 446
GraphQL schema 448, 451, 453

H

handler functions
 binding, to elements 60
 declaring 52
higher-order components
 about 142
 conditional component, rendering 143, 144
 data sources, using 145
higher-order function 142
Home component 434, 437, 438
HTML tags
 built-in tags 20
 conventions 21
 using 20

I

icons
 rendering 400, 403
images
 loading 387, 389, 390
 resizing 390, 393, 394
imperative components
 jQuery UI widgets, rendering 103, 104, 105
 rendering 103
imperative programming 12
information architecture
 about 423
 scaling 439
initial component state 35
inline event handlers 59
inline function 59
iOS apps
 executing 222, 223, 224
iOS simulator 219
iOS
 versus Android 213
isomorphic JavaScript
 about 173
 code sharing, between backend and frontend 175
initial load performance 174, 175
server, as render target 173, 174

J

JavaScript collections 30
JavaScript expressions
 collections, mapping to elements 30
 property values 29
 text values 29
 using 29
JavaScript XML (JSX)
 about 12, 18, 212
 application, creating 18
 declarative UI structure 19
 inheriting 139, 140, 142
jQuery UI widgets
 rendering 103, 104, 105

K

key-value pair 88

L

lazy image loading 395, 399
lazy list loading 285, 286
lazy loading 169
lazy routing 167, 168
link components
 linking 164
 query parameter 165, 167
 URL parameter 165, 167
 using 164
List component 273
list data
 fetching 282
list of options
 selecting from 326, 331
ListContainer component 273
ListControls component 273
ListFilter component 273
lists
 about 197, 200, 202
 filtering 273, 274, 280, 281
 sorting 273, 274, 280, 281
ListSort component 273
ListView component 269, 273

M

MapView component
 implementing 312, 313
 overlays, plotting 315, 318
 points of interest, annotating 313
 points, plotting 314, 315
microservices 446
mobile browser experience 212
mobile web apps
 case 213
mobile-first design 190, 191, 192
monolithic components, refactoring into
 substable
 about 66
 event handler implementation 70
 initial state 68
 JSX markup 67
 state helpers 68
monolithic components
 issues 66
multiple event handlers 52
mutation 443

N

named function 59
namespacing 27
navigation bar
 about 255
 specifying 255, 258
navigation indicators 298
navigation
 implementing 193, 195, 196
navigators 250
network state
 detecting 405, 409
notification 343

P

page abstraction 249
parent route 154
passive notifications
 about 359
 creating 359, 360, 361, 364
performance 14

portable components
 using 113
predictable outcomes 112
progress bar
 implementing 302, 304, 306
progress
 about 289
 indicating 290, 292
 measuring 293, 295, 297
properties
 about 34
 component data, fetching 87, 90, 91
 inheriting 136, 137
 initializing 86
 state, initializing 91, 92
 state, updating 94, 98
property validation 112, 113
property validators
 any property value 120, 121, 122
 type validation 114, 117
 using 114
 value, requisites 117
property values
 default property values 41
 passing 41
 rendering 122, 123, 125
 setting 42, 45
 validating 127
pure function 45

Q

query 442
query parameter 165, 166, 167

R

React component lifecycle 85, 86
React Native project
 executing 221
React Native styles 229
React Native
 about 210
 reference 211
react-bootstrap components
 forms 202, 203, 205, 207
 lists 197, 200, 202

navigation, implementing 193, 195, 196
reference link 193
using 192, 193

React

- about 9, 10, 11, 211
- data 11, 13
- events 11
- JSX 11
- lifecycle 11
- time 13

Reactive Native command-line tool

- using 215, 216, 219

Redux

- about 145
- App component 430, 432
- application state, viewing 427, 428, 429
- Home component 434, 437, 438
- implementing 426
- routes 429
- store provider 429
- store, creating 428
- using, in mobile apps 438

Relay

- about 442
- and TodoMVC 447, 448
- bootstrapping 453, 454

reusable HTML elements 65, 66

routes

- about 250
- child route 154
- creating 151
- declaration, decoupling 152, 153
- declaring 150
- optional parameters 161
- parameters, handling 156
- parent route 154
- resource IDs 156, 157, 159, 161
- responding 250, 252, 253
- routing 150

S

scenes 250

screen organization 249

scrolls

- implementing 371, 372, 374

server-side rendering 173

simulator 219

stacks 250

state

- about 33
- component data, fetching 87, 90, 91
- initializing 86
- initializing, with properties 91, 92
- updating, with properties 94, 98

stateful components 83

stateless components 45

strings

- rendering, to 176, 178

substitutability 48

swipe gesture

- implementing 379, 380, 383

Switch component

- using 332, 335

synchronous update rounds 424

synthetic event objects 61

synthetic instance pool 62

T

text input

- collecting 321, 322, 325, 326

todo items

- adding 455, 456, 458
- completing 460, 462
- rendering 458, 460

TodoMVC example, for Relay

- reference 447

TodoMVC

- and Relay 447, 448

touch feedback

- displaying 374, 375, 376, 377

type validation 114, 117

type validator

- about 122, 125, 126, 127
- property values, rendering 122, 123, 125

U

UI structures

- describing 22

unified information architecture 426

URL parameter 165, 166, 167

user confirmation

- error confirmation, acknowledging 354, 358
 - obtaining 344
 - success confirmation, displaying 344, 346, 349, 350, 351
- utility components** 83

V

value validator

- about 122
- property values, rendering 122, 123, 125

virtual DOM 15

X

Xcode

- installing 219