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

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

Reviewer
August Marcello III

Commissioning Editor
Edward Gordon

Acquisition Editor
Nitin Dasan

Content Development Editor
Onkar Wani

Technical Editor
Prashant Mishra

Copy Editor
Charlotte Carneiro

Project Coordinator
Sheejal Shah

Proofreader
Safis Editing

Indexer
Aishwarya Gangawane

Graphics
Jason Monteiro

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 Marcello 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

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 a*lerts 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_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?

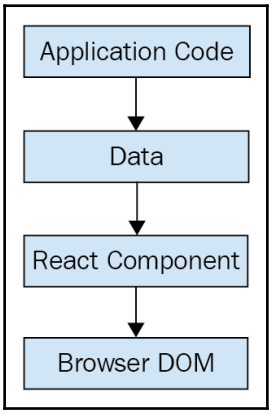
I think the one-line description of React on its homepage (`https://facebook.github.io/react`) is brilliant:

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:



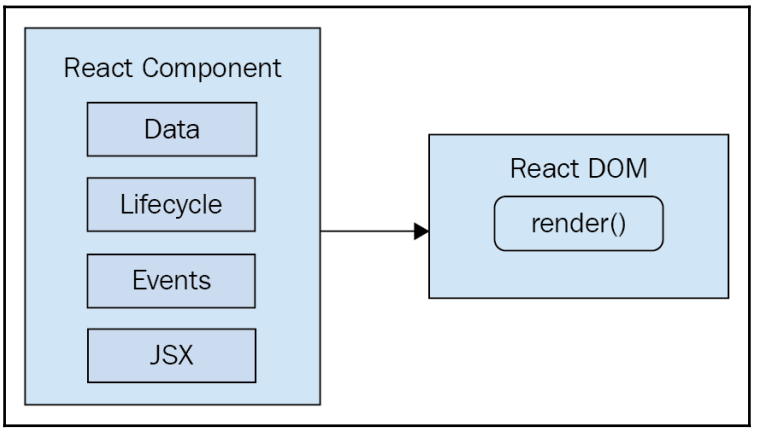
This is literally all there is to React—the core concept. Of course there will be subtle variations to this theme as we make our way through the book, but the flow is more or less the same. We have some application logic that generates some data. We want to render this data to the UI, so we pass it to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

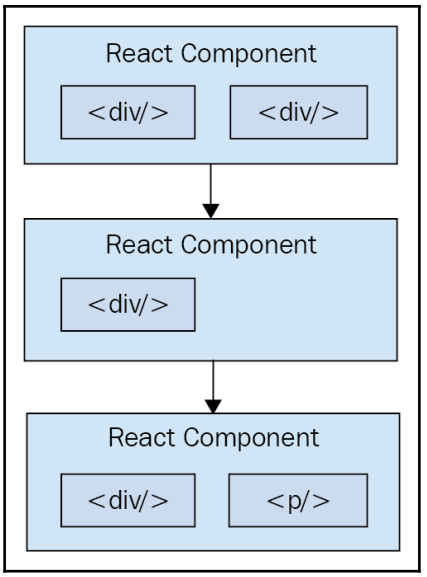
The syntax used by React components is called **JSX** (**JavaScript XML**). The idea is actually quite simple. A component renders content by returning some JSX. The JSX itself is usually HTML markup, mixed with custom tags for the React components. The specifics don't matter at this point; we'll get 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:

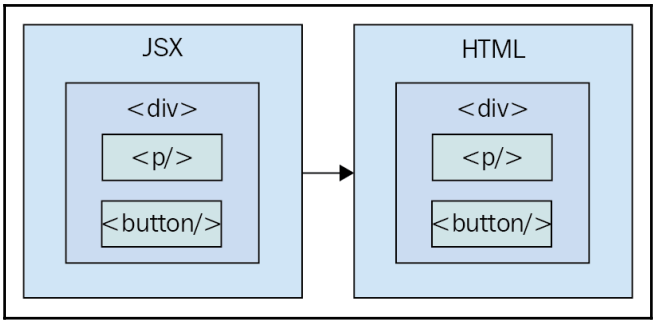


Each time the React component is rendered, it's like taking a snapshot of the JSX at that exact moment in time. As our application moves forward through time, we have an ordered collection of rendered user interface components. In addition to declaratively describing what a UI should be, re-rendering 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**.

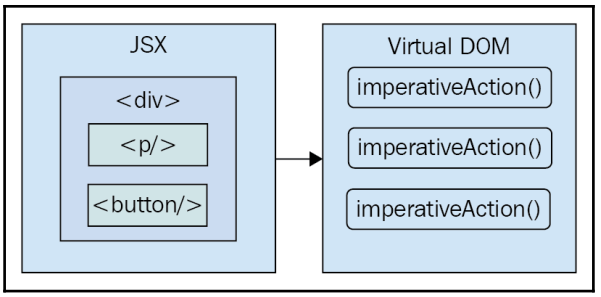
For example, having a declarative UI structure is fine for the initial rendering, because there's nothing on the page yet. So, the React renderer can look at the structure declared in JSX, and render it into the browser DOM. This is illustrated in the following diagram:



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.

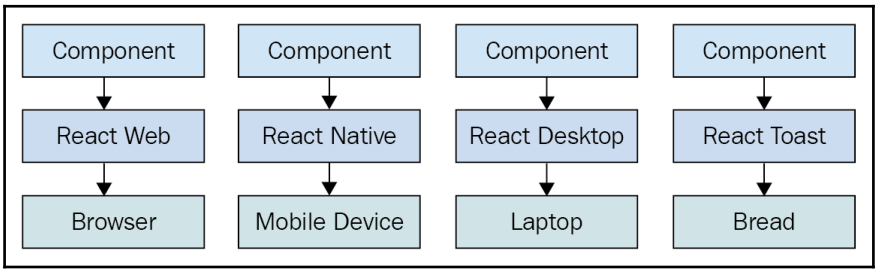
The right level of abstraction

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 transpiles 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.

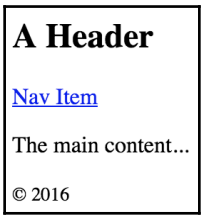
Describing UI structures

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.

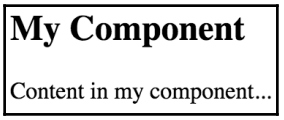
Encapsulating HTML

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 `<li>` tag is only useful as the child of a `<ul>` or `<ol>` 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 it's 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.

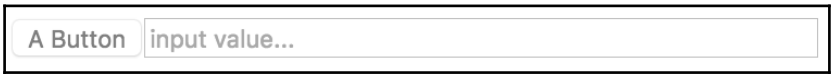
Dynamic property values and text

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:



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',
```

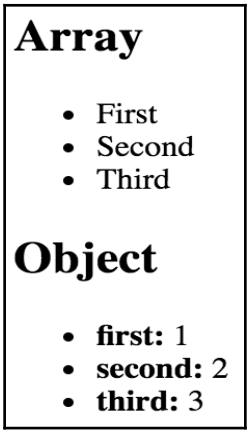
```
];

// 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 (`<li>`), 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 new-found 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 `<ul>` that maps a JavaScript collection to `<li>` 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.