About React Explained

About the OSTraining Book Club

React Explained is part of the OSTraining Book Club.

The Book Club gives you access to all of the “Explained” books from OSTraining:

- These books are always up-to-date. Because we self-publish, we can release constant updates.
- These books are active. We don’t do long, boring explanations.
- You don’t need any experience. The books are suitable even for complete beginners.

Join the OSTraining Book Club today: https://ostraining.com/books.

Use the coupon **“reactexplained”** to save 35% on your membership.

We Often Update This Book

We aim to keep this book up-to-date, and so will regularly release new versions to keep up with changes in React.

Advantages and Disadvantages

We often release updates for this book. Most of the time, frequent updates are wonderful. If there is a change in React in the morning, we can have a new version of this book available in the afternoon. Most traditional publishers wait years and years before updating their books.

There are two disadvantages to be aware of:

- Page numbers do change. We often add and remove material from the book to reflect changes in React.
- There's no index at the back of this book. This is because page numbers do change, and also because our self-publishing platform doesn't have a way to create indexes yet. We hope to find a solution for that soon.

Hopefully, you think that the advantages outweigh the disadvantages. If you have any questions, we're always happy to chat: books@ostraining.com.

Thank You To Our Readers

If you find anything that is wrong or out-of-date, please email us at books@ostraining.com. We'll update the book, and to say thank you, we'll provide you with a new copy.

Are You an Author?

If you enjoy writing about the web, we'd love to talk with you.

Most publishing companies are slow, boring, inflexible and don't pay very well.

Here at OSTraining, we try to be different:

- **Fun**: We use modern publishing tools that make writing books as easy as blogging.
- **Fast**: We move quickly. Some books get written and published in less than a month.
- **Flexible**: It's easy to update your books. If technology changes in the morning, you can update your book by the afternoon.
- **Fair**: Profits from the books are shared 50/50 with the author.

Do you have a topic you'd love to write about? We publish books on almost all web-related topics.

Whether you want to write a short 100-page overview, or a comprehensive 500-page guide, we'd love to hear from you.

Contact us via email: books@ostraining.com.

Are You a Teacher?

We hope that many schools, colleges and organizations will adopt React Explained as a teaching guide to React.

This book is designed to be a step-by-step guide that students can follow at different speeds. The book can be used for a one-day class or a longer class over multiple weeks.

If you are interested in teaching React, we'd be delighted to help you with review copies, and all the advice you need.

Please email books@ostraining.com to talk with us.

Sponsor an OSTraining Book

Is your company interested in sponsoring an OSTraining book? Our books are some of the world's best-selling guides to the software they cover. People love to read our books and learn about new web design topics.

Why not reach those people? Partner with us to showcase your company to thousands of web developers. We have partnered with Acquia, Pantheon, Nexcess, GoDaddy, InMotion, GlowHost and Ecwid to provide sponsored training to millions of people.

If you want to learn more, visit https://ostraining.com/sponsor or email us at books@ostraining.com.

We Want to Hear From You!

Are you satisfied with your purchase of React Explained? Let us know and help us reach others who would benefit from this book.

We encourage you to share your experience. Here are two ways you can help:

- Leave your review on Amazon's product page of React Explained.
- Email your review to books@ostraining.com.

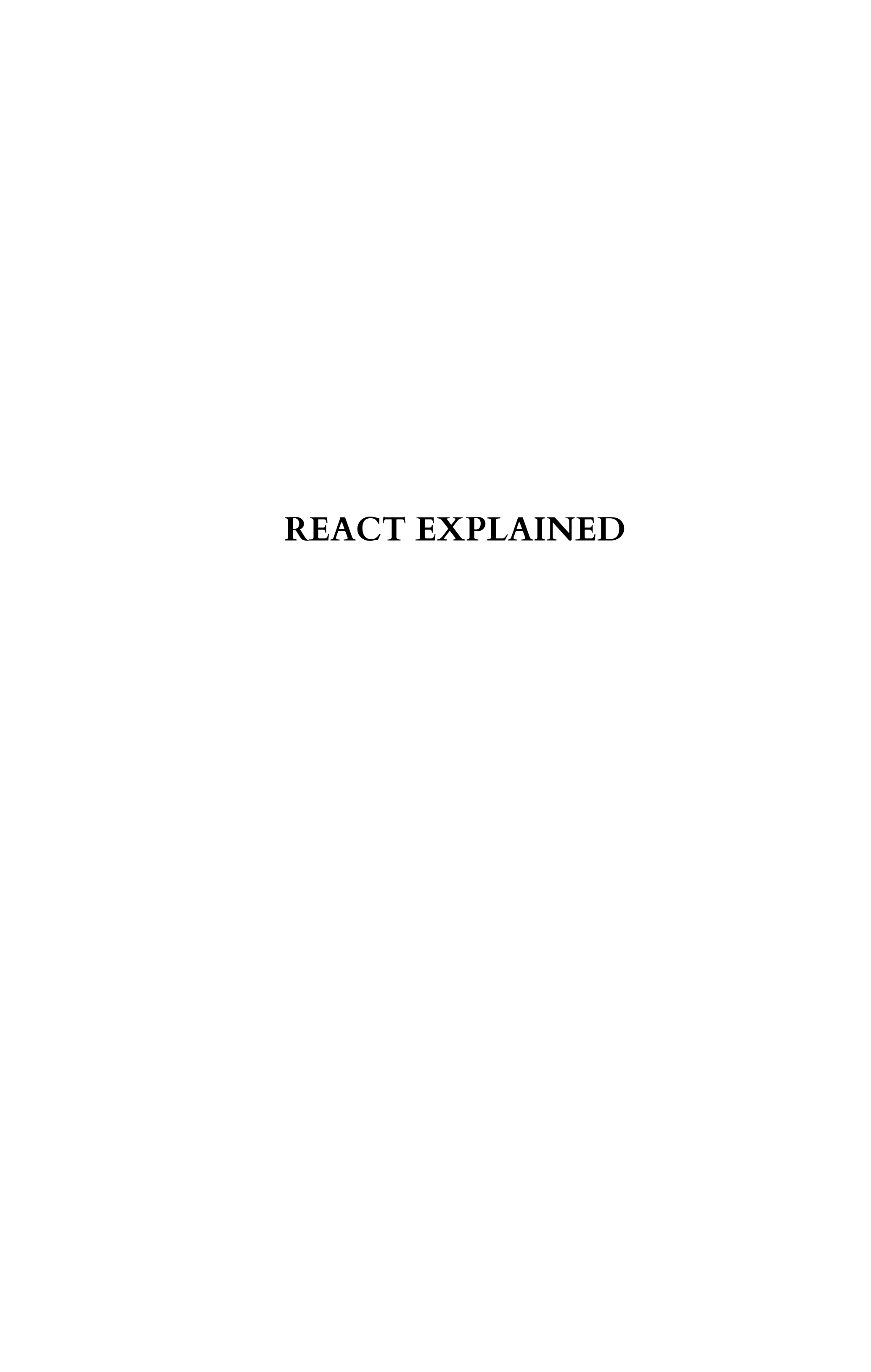
Thanks for reading React Explained. We wish you the best in your future endeavors with the software!

The Legal Details

This book is Copyright © OSTraining.

This book is published by OSTraining.

Proper names and contact information are used as examples in this book. No association with any organization or individual is intended, nor should it be inferred.



REACT EXPLAINED

REACT EXPLAINED

STEVE BURGE

React Explained Copyright © by Steve Burge. All Rights Reserved.

Contents

React Explained 1

Part I. Preparing to React

1. The JavaScript You Should Know for React Explained 9
2. 5 Exercises in Vanilla JavaScript 29
3. Developer Tools for React Explained 33
4. 5 Exercises with Developer Tools 43
5. A High Level Overview of React 51

Part II. React Explained

6. An Introduction to React Elements and Components 65
7. 5 Exercises in Writing React With Elements and Components 77
8. An Introduction to JSX 83
9. 5 Exercises in Writing React With JSX 99
10. An Introduction to Creating React Apps 103
11. 5 Exercises in Creating a React App 115
12. Props in React Explained 119
13. 5 Exercises in Working with Props 135
14. State in React Explained 139
15. 5 Exercises in Working with State 151
16. The Component Lifecycle Explained 155
17. 5 Exercises with the Component Lifecycle 175

React Explained

React is an incredibly popular choice for Javascript projects.

React exists in a large and constantly changing field of JavaScript libraries and frameworks. However, as it stands currently, React is a go-to choice for many developers.

Why is React so popular? I would pinpoint three major advantages for React: simplicity, ingenuity, and being at the right place at the right time.

Advantage #1. Simplicity

The simplicity of React is that it is a user interface library.

React does one thing and does it well. Unlike some other libraries such as Angular, which come with much more functionality out of the box, React does not. It offers you a component architecture for building user interfaces.

This single focus of React has led to a scenario of "React and Friends," where you have to use other libraries to handle things like routing and advanced state management that may come bundled with other similar frameworks, like Angular. However, this is not a bad thing. As my dad used to say:

> "Pick a tool that does one thing really well over a tool that tries to do everything."

Angular are also great tools. I am actually a big fan of Angular. But React's simplicity has allowed it to evolve quickly, while also staying fairly stable along the way. Although React does have a learning curve, there is less to learn with React than there is with more complex frameworks.

We are web developers working in an era of small, focused libraries. Modern developers pull in small libraries rather than rely on large frameworks to do everything for us. React fits into this era perfectly.

Thanks to React's simplicity, it can be used in many different environments.

When React first started, it focused on building web interfaces.

A library called ReactDOM was created to handle all of the DOM interactions. The core React library was left absent of anything DOM specific.

A library called React Native was created to help with building native applications for mobile devices. All of this code was kept out of the core React library, allowing Core React to work for the web and mobile devices.

A final example is the newer React 360 library to help with building VR environments and 360 experiences. Again, we see how the simplicity of the core React library has allowed it to be used in so many different environments.

Simple is not the same thing as easy.

Any good developer can appreciate simple solutions that solve difficult problems in elegant ways. While we can describe one aspect of React's popularity as simplicity, we must also recognize the genius of React behind the scenes.

Advantage #2. Ingenuity

React has some rather ingenious engineering behind the scenes.

React took a fundamentally different approach to handling some of the problems that confront modern JavaScript developers. The virtual DOM, Javascript XML (JSX) and one way data flow are some great examples of this.

One of the clever things React did differently from other libraries was to create a Virtual DOM.

A common problem in JavaScript development involves making changes to the Document Object Model, the API for interacting with HTML via JavaScript. Rather than supply helper functions for updating the DOM directly, React created it's own version of the DOM in memory that you can update parts of without updating the the entire thing. This proved far more performant than making updates directly to the DOM. We will go into the Virtual DOM in more depth in this book, but trust me when I say, it is rather ingenious.

Another problem JavaScript developers face is managing templates. Do you create UIs purely with JavaScript? Do you create them partially in HTML? How much HTML goes in your JavaScript and how much JavaScript related code goes in your HTML? Once again, React took a new approach to this with the creation of JSX. JSX looks like HTML written in our JavaScript, but with the help of build tools, it compiles down to vanilla JavaScript. This allows you to create UI components purely in JavaScript, while also having markup that looks and feels familiar.

It took me a while to get onboard the JSX train. I began coding JavaScript during the early days of Web Standards, when combining your JavaScript and markup to this extent was highly discouraged. However, today it makes a lot of sense to be able to build your entire UIs from within a JavaScript file without needing to use actual HTML. I imagine you will also grow to love (or at least appreciate) JSX. Either way, you will have to admit it is rather ingenious.

React also took a rather different approach to data flow.

Before React, two way data binding was common. This involved connecting a piece of data with a UI element so if something changed in either place, it would be reflected in the other. React changed this completely and went with a one way data flow model.

With this approach data flows into your app and down through the components. If a change is made to that data, something is trigged to re-send the updated data back down through your app. This follows a model with a single source of truth, while still leveraging event handlers you're already familiar with to trigger updates to data. We will expand on data flow quite a bit in this book, but it is a very clever model.

Part of the proof of React's ingenious is that other libraries and frameworks have adopted many of the approaches that React was the first to pioneer. However, simplicity and ingenuity alone would probably not have been enough to make React as popular as it is today.

Advantage #3. Right place, right time

Two important questions developers ask when considering a

framework are "Who is supporting it?" and "Who is using it?" From day one, the best answer to both of these questions for React was "Facebook." Developers at Facebook created React, and several of the related libraries, to help support their huge online application.

Being developed at Facebook counts as being in the right place.

Facebook is a huge application that does not have a sign of going away or switching to another JavaScript framework anytime soon. For this reason, most developers felt they could rely on React to continue to receive support and updates for some time to come.

Before React, Angular was probably considered the leading JavaScript framework. Angular has Google behind it, which many developers feel they could trust for similar reasons. Vue.js, a library similar to React, which came out after React largely had a single developer behind it, Evan You. For this reason, some developers had a concern that if Evan stopped working on the project it might not continue to receive the same updates and support, although Vue continues as a popular alternative to React.

We cannot diminish that being developed at Facebook was a part of what has allowed React to become so popular.

The timing of React's release is also crucial.

React came along at a time when Single Page Web Apps (SPAs) were becoming the standard and more and more complex. Concepts like component architecture, the virtual DOM, one way data flow, and JSX were solving problems that many developers had struggled to address on their own or felt other frameworks did not adequately addressing. Some of these solutions, like the Virtual DOM, were not even solutions many developers had ever considered.

React solved some serious problems with simple and ingenious solutions. On top of this, it wasn't too difficult to learn. React came along at the same time as many developers were learning the new features of ES6 (EcmaScript 2015). So for many, learning "advanced" or "new" JavaScript and learning React went hand in hand. In fact in the earliest versions of React, developers shifted from creating components using the "React" way to an "ES6" way that has since become "The React" way of doing things.

That React incorporated an ES6+ approach to writing JavaScript from the beginning only helped it.

React also entered the scene admits a growingly complex tooling landscape. I remember early on a lot of folks said they couldn't learn React because they would need to learn the command line and build tools like webpack. This scared a lot of people. In fact it still does. Many people loved Vue.js because they didn't need to use any build tools.

I will interject a personal opinion here: If you want to develop with JavaScript today, you should learn some basic tools. These include the command line basics, build tools like webpack, and transpiling tools like babel. However, you don't have to be proficient in Linux by any means, and webpack and babel continue to get easier to configure and use.

React had to address this issue, and it did so with Create React App. Create React App allows you to type on line into a command line tool and get a new React App setup with all the tooling working and hidden from your view. This way you could focus on learning React, which is not that hard, and not have to worry about the tooling, which can be tricky to learn if you are new.

You still have to learn some tooling as you get further along with React, but only because we are still living in a world where build tools like the ones we use are still needed. In the future they won't be necessary for the same reasons. However, React may still be useful even after browser support allow us to move away from so many build tools.

Who is Using React?

Before we wrap up our introduction to React, it can be insightful to look at who in the development and professional world is using React on projects.

Here is a small, partial list of major projects using React:

- Airbnb
- eBay
- Lyft
- Netflix
- PayPal

- Reddit
- Salesforce
- Twitter
- WordPress

Several major showcases of using React Native for mobile applications exist as well:

- Facebook
- Instagram
- Pinterest
- Skype
- Uber
- Walmart

You can tell from this list that React is a trusted library that can build interfaces for a range of applications. Making the decision to learn React will serve as a valuable tools, not just for your own projects, but for potential employment as well.

Let's Get Ready to React

Hopefully this introduction has helped you understand at a high level what has made React so popular. We have discussed how React's simplicity in design, ingenious engineering and simply being at the right place at the right time has helped cement it as the JavaScript library of choice for making User Interfaces with JavaScript. We have also looked at a few examples of major projects that use React in the real world.

Throughout the rest of this book we will dig deep into how React works and how to build applications with it. While we will focus on building for the frontend on the web, many of the skills you will learn will also apply to writing React on the server side, for native applications and even for VR and 360 environments.

So, I hope you are excited to dig in deeper. Let's get ready to React!

PART I

Preparing to React

This section goes over important technical skills that will help you better understand React.

Chapter 1 – "Important JavaScript to Know for React" may be a review for you or something you come back to for reference more than once as you learn React. We discuss everything from "What is an expression?" to "How keyword *this* and binding works" and "What .map(), .filter() and .reduce() do." All of the JavaScript explained in this section will be used at some point when building with React.

Chapter 2 – "Important Tools to Know for React" gives a high level overview of the most common development tools used with React. Although we explain what each tool does individually, these tools are often used together in one workflow with overlapping and interlacing parts as we will see later in this book.

Chapter 3 – "A High Level Overview of React" finally introduces us to React itself. We explain the basic building blocks of React, show how data flows through a React app, and get you comfortable beginning to read and write your first React apps.

1

The JavaScript You Should Know for React Explained

Welcome to React Explained!

In this first chapter we are going to review some aspects of JavaScript that will help us successfully work with React.

It is becoming easier to use React without a deep understanding of JavaScript or its related tools. However, it is still not completely possible, or recommended, for us to take on React without some background knowledge.

This chapter is not meant as an "Introduction to JavaScript." For resources on learning basic JavaScript, please visit the book website reactexplained.com.

This chapter is designed to introduce 12 characteristics of JavaScript that are not always well understood. These Javascript features will likely be at work behind the scenes in most of your React applications.

1. EcmaScript and JavaScript Versions
2. Statements vs Expressions
3. const, let, var, .freeze() and Immutability
4. Template Literals (Template Strings)
5. Arrow Functions
6. Classes
7. How "this" Works in JavaScript
8. Tertiary Conditionals
9. Spread Syntax and Deconstruction Assignment
10. .filter(), .map(), and .reduce()
11. DOM Node Creation
12. Exports and Imports

One of the great things about React is that you will find is that a lot of the code you write is basic, "vanilla" JavaScript. While React, ReactDOM and JSX provide some helpful shortcuts, we will still find ourselves writing our own Classes, specifying event handler code, making API requests, sorting and filtering data, and more. Since we will do all of this with vanilla JavaScript, the more JavaScript you know when working with React, the better.

#1. EcmaScript and JavaScript Versions Explained

JavaScript is based on a programming language standard called EcmaScript. Starting in 2015, the EcmaScript Standards Committee began an annual release cycle of new updates to the EcmaScript standard each year. This meant new annual features became available to JavaScript as well.

New features that were added in 2015 are referred to as EcmaScript 2015. Since EcmaScript 2015 was actually the sixth release of the standard, you also see it referred to as ES6. EcmaScript 2016 would be ES7, EcmaScript 2017 would be ES8, etc. While EcmaScript 2015 introduced a lot of new features to the language, later annual releases tend to just have a few new features each year.

The problem with new EcmaScript features is that although you can use them in your JavaScript, they may not be supported in browsers. This gives us two options for using new additions to the JavaScript language. We can wait for browsers to enable support for new features. Or we can use transpiling tools (covered in the next chapter) to convert new JavaScript code into JavaScript code that browsers already support.

It is also possible to rely on something called Polyfills, small bits of code that we can add to our projects that add support for specific features. Depending on the new JavaScript language feature you want to work with, you may use a transpiling tool, or you may find a polyfill.

For the examples in this book we will rely primarily on our build tool setup (including a transpiler) to handle and JavaScript features that do not have strong browser support.

#2. Statements vs. Expressions Explained

In loose terms, a statement in JavaScript is a block or line of code that does something. They usually end in semicolons (if you use them) or appear as blocks within curly braces. The following are examples of statements in JavaScript:

```
const title = "Welcome!";
function greet(title) {
  console.log(title);
}
```

In this example above, the first line is a statement, the entire function declaration is a statement, and the line logging the title is a statement.

Expressions produce a value or are a value themselves. You will often see expressions on the right side of an equal sign or as a parameter for a function. In both of these cases, the expressions resolve to values which can then be assigned or passed as such.

In the example below, we get a form from a page and log out title and content values when someone submits the form.

```
const form = document.querySelector( 'form' );
form.addEventListener( 'submit', displayPost );
function displayPost( event ) {
  const title = document
                  .getElementById( 'title' )
                  .value,
  const content = document
                  .getElementById( 'content' )
                  .value;

  event.preventDefault();

  console.log( title );
  console.log( content );

}
```

The expressions here are as follows:

- The selector `document.querySelector( 'form' )`
- The parameters ‘`submit`‘ and ‘`displayPost`‘
- The parameter ‘`event`‘ passed into `displayPost()`
- The selectors `document.getElementById( 'title' ).value` and `document.getElementById( 'content' ).value`
- The ‘`title`‘ and ‘`content`‘ variables when passed in to `console.log()`

In each of these cases–selectors, function references, variable names or strings of text–all return a value. This value in turn can be assigned to a variable or passed as a parameter into a function.

The other thing about expressions is that they appear as part of statements. Statements are the full block of code. Expressions are the part that returns a value. A single statement can also have multiple expressions.

It will become important to know when you are writing a statement and when you are writing an expression since JSX only accepts expressions within its tags.

#3. const, let, var, .freeze() and Immutability Explained

JavaScript has three ways to declare a variable: `var`, `let` and `const`. `var` has been around since the beginning of JavaScript and `let` and `const` were added with EcmaScript2015.

In this book we will use `const` by default, `let` when const is not appropriate, and pretty much avoid the use of var. This is a common approach in the React community.

The table below shows the similarities and differences between the three.

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

The most important thing to know from this table for the purposes of this book is that the `const` keyword does not allow it's value to be reassigned to another value. However, if the `const` value is an object or an array you can still edit items within that object or array.

```
const name = 'Zac Gordon';
let location = 'Washington DC';

name = 'React'; // Throws error
location = 'Internet'; // Allowed

const ids = [1, 2, 3];
const post = {
 id: 1,
 title: 'New Post'
};

ids.push(4, 5) // Allowed to add to array
ids.pop() // Allowed to remove from array
ids[0] = 0 // NOT allowed to reassign values

// NOT allowed to reassign object itself
post = { id: 2, title: 'New post' }

// Allowed to add new properties and methods
```

```
post.slug = 'new-post'
// Allowed to reassign properties and methods
post.id = 2
// Allowed to remove properties and methods
delete post.title
```

If you want to prevent items in an object or array from being changed, you can use the native `Object.freeze(objectOrArray)`.

```
'use strict'
const ids = [1, 2, 3]
const post = {
  id: 1,
  title: 'New Post'
}

Object.freeze(ids) // Freeze the ids array
Object.freeze(post) // Freeze the post object

ids.push(4, 5) // NOT allowed to add to array
ids.pop() // NOT allowed to remove from array
id[0] = 0 // NOT allowed to reassign values

// NOT allowed to add new properties
post.slug = 'new-post'
// NOT allowed to reassign values to properties
post.id = 2
// NOT allowed to remove properties
delete post.title
```

When we have a variable that we cannot change, the value of it is referred to as *Immutable*. A variable that can be changed is referred to as *Mutable*. In React development there is a strong pattern of Immutability, or programming in a way so that once data is assigned, it is not reassigned. With an Immutable approach, when you need data to change you would create a copy of it and modify the copied content, leaving the original content unchanged.

For this reason we use `const` by default to prevent our data from

being reassigned. However, remember, if you want your data truly immutable you will need to use `Object.freeze()` or an immutable JS library that does something similar. If for some reason you ever need to unfreeze an object or array you can use `Object.unfreeze(objectOrArray)`.

#4. Template Literals Explained

Template literals, also called "Template Strings," are a special type of string in JavaScript that allows for including variables within the string.

```
const name = 'Zac Gordon'
// Logs "Hi Zac Gordon! Welcome :)"
const welcomeMsg = `Hi ${name}! Welcome :)`
```

Notice that rather than using single or double quotes to wrap our string value, we are actually using the back tick character (`). The other thing we see is the pattern of `${variableName}` within the back ticks to reference a variable.

Template literals will also allow you to use line breaks when creating your strings.

```
const first = 'Zac'
const last = 'Gordon'
const longMsg = `Welcome ${first}!
      we
    see
  your
last name is ${last}`
```

In the example above, the spaces and line breaks are saved as part of the string structure and would appear intact if you logged out the data to the console. However, if you render the longMsg string to the DOM, it would appear as one normal string of text with no line breaks.

It is quite likely you will see template literals in React examples and applications.

#5. Arrow Functions Explained

Arrow functions, or “fat” arrow functions, are a shorthand for writing anonymous function expressions in JavaScript.

An expression in JavaScript is something that returns a value. We often see expressions appear on the right side of the equal sign (=), which then returns a value to the left side of the equal sign (=) where we commonly have a variable name.

The anonymous means that our function does not have a name, so it is usually executed called where it is written or assigned to a variable.

```
const render = title => console.log( title )
render( 'New Post' ) // Logs "New Post"
```

In this example above we are creating a function expression, passing in the parameter title and then logging out the title inside of the function body.

```
const render = ( id, title ) =>
    console.log( `${id}: ${title}` )
render( 1, 'New Post' ) // Logs "1: New Post"
```

In the example above we are doing the same thing with two parameters. Notice when two parameters (or no parameters) are passed that we have to wrap them inside of parenthesis ().

One line arrow functions will return the value by default.

```
const render = ( id, title ) => `${id}: ${title}`
// Logs "1: New Post"
console.log( render( 1, 'New Post' ) )
```

If you want to break an arrow function into multiple lines, you must manually return a value.

```
const render = ( id, title ) => {
  console.log( `Working with ID ${id}` )
  return `${id}: ${title}`
}
```

Arrow functions have one other important characteristic. They do not have binding for the keyword `this`. If you try to use `this` in an arrow function it will go outside the current function scope to find `this` defined.

This can be helpful if you want `this` to refer to a class rather than a method. However, it can be confusing if you are expecting `this` to work as it would in a normal function.

In general, React apps use arrow functions where possible, unless there is a reason not to.

#6. Javascript Classes Explained

Classes in JavaScript are a special type of function. They use prototypal inheritance, which is different than "classical" inheritance found in other programming languages like Java.

Classes are often used in React to create components. Usually, you will not create your own classes, but rather extend default React classes.

Here is an example of how a class may be used in the context of React:

```
class Example extends Component {
  constructor(args) {
    super(args)
    this.state = {
      message: 'A message in state'
    }
  }

  render() {
    return `Message: ${this.state.message}`
  }
}
```

In the example above we are creating a new class called "Example" that is extending a class "Component" (not shown, but in React core).

The constructor method executes automatically when the class is instantiated. If a constructor function is not included with a class, JavaScript will use a default constructor automatically. However, in

React we commonly need to create a constructor function for our classes that extend React classes.

The `super()` function in JavaScript classes does a few things. First it calls the parent constructor method, which would appear inside of the Component class. Since we are passing an argument into `super(args)` called "args," this will also be available in the Component constructor class. The `super()` function will also make `this` available in the constructor method.

We then have a method called `render`. All classes in React will have a method called render that returns a valid React element. In our example we are simply returning a string of text rather than a DOM or React element.

In many cases we will be able to use functions in our React code, but there are times when classes are necessary, so it is important you understand their basic structure.

#7. How "this" Works in JavaScript Explained

The `this` keyword in JavaScript is a generic placeholder. By default, in strict mode, `this` is `undefined`. `this` is assigned to the window object by default without strict mode, however, all of the examples here we will assume strict mode is enabled (either manually or by a tool like webpack).

We generally use the `this` keyword inside of objects, functions and classes.

Inside an object, the this keyword will refer to the object itself.

```
const post = {
  id: 1,
  slug: `post-${this.id}`,
  title: 'First post'
}
post.slug // post-1
```

In the object above we use `this.id` to refer to `post.id`. In this example, this refers to the object itself.

Now let's look at `this` with a function. By default `this` is undefined in a function. However, we can assign properties (and

methods) to `this` manually. In this sense, `this` is being used similarly to any other variable, except that as we will be see, it can be overwritten from outside the function.

```
function render() {
  this.id = post.id
  console.log( this.id )
}
render({ id: 1 }) // 1
```

In the function example above we can see a pattern where we take a value from a parameter and assign it to a property on `this`.

```
function render() {
  console.log( this.id )
}
// undefined (or window in non strict mode)
render({ id: 1 })
```

In the example above we are not doing any manual assigning of `this`, so it remains `undefined`. However, we can leverage the `.call()` function in JavaScript to define what this should be at call time.

```
const post = { id: 1 }
function render() {
  console.log( this.id )
}
render.call( post ) // 1
```

The `.call()` method can be used on any function and simply calls the function. It is similar to just calling a function with parenthesis, like `render()`, except that you can pass a parameter to it that will become the new value for `this`. In the example above we are taking our post object and assigning it to `this` within the `render` function, even though this was previously `undefined`.

We also have a method called `.bind()` that assigns a value to `this` but does not call the function. We use `.bind()` when we want to

set `this` now, but call it later (possibly more than once) and keep the value for this we defined.

Here is an example of when you may want to use `bind()`.

```
const link = document.getElementById( 'link' )
const post = { id: 1, title: 'Hello bind()!' }

link.addEventListener( 'click',
  renderTitle.bind( post )
)

function renderTitle() {
  console.log( this.title )
}
```

By default when we assign an Event Listener in JavaScript it assigns `this` to be the target of the event. In this case, `this` would be assigned to the link. However, there are sometimes when you want assign `this` to something else, like data. In the example above we change the binding of `this` from link to the post object.

Certain patterns of writing React use `bind()` in a similar way when working with event handlers or other places where `this` needs to be explicitly set.

One last reminder here is that arrow functions do not track binding to `this`. So, commonly you will see them used when you want `this` to refer to something in a higher level of scope rather than the immediate function scope.

#8. Tertiary Conditionals Explained

Hopefully you are already familiar with conditional statements like these:

```
let loggedIn = true
if( loggedIn ) {
  console.log( 'Welcome!' )
} else {
  console.log( 'Please login' )
}
```

Tertiary operators, or tertiary conditionals, allow you to write simple conditional expressions. Since they are expressions they have to return a value, either in place or to a variable like in the example below.

```
let loggedIn = true
let message = (loggedIn) ? 'Welcome' : 'Please login'
console.log( message ) // "Welcome"
```

Notice the pattern here of writing your conditional in the parenthesis. Our conditional statement is just checking to see if something is true or present. You can write full conditional statements between the parenthesis, but a common pattern in React is just to check if something is available.

After the question mark (?) we have the value returned if the conditional statement is `true`. After the colon (:) we have the value to be returned if the conditional check is `false`.

Once we get into JSX we will revisit using tertiary operators, so it is a good idea for you to get comfortable with how to write them and remember the syntax so you do not have to lookup how to write them each time.

Remember, `(conditional) ? ifTrue : ifFalse`.

#9. Spread Syntax and Deconstruction Assignment Explained

The Spread syntax in JavaScript allows us to unpack iterable items, like arrays, into parameters or other arrays.

Here is a basic example of spreading an array into a predefined set of parameters:

```
const nums = [11, 22, 33]

function add(first, second, third) {
  return first + second + third
}

let total = add(...nums)
console.log(total) // Returns 66
```

Here is an example of spreading an array as a parameter where there are no defined parameters defined.

```
const postIds = [1, 2, 3]
const newPostIds = [4, 5, 6, 7]

postIds.push(...newPostIds)
console.log( postIds ) // Logs 1, 2, 3, 4, 5, 6, 7
```

You will likely see the spread syntax used with React applications.

The deconstruction assignment in JavaScript allows us to unpack items in an array or properties in an object and assign them to variables. This is used when getting data out from an array or object.

```
const library = {
  render: () =>console.log('Rendered'),
  save: () =>console.log('Saved'),
  update: () =>console.log('Updated'),
  push: () =>console.log('Pushed'),
}

const { render, push: notify } = library
render() // Logs Rendered
notify() // Logs Pushed
```

In the example above we have an object called `library`. Then later, we can get just the `render` and `push` methods to use in our app. We can do that using deconstruction assignment: placing the name of what we want to pull out between curly braces (`{ }`).

We can also rename a method or property during this process by referencing the correct name, followed by a colon (`:`), and then the new name we want to use. Notice how we used this method to rename `push` to `notify`.

You will absolutely see deconstruction assignment used in React apps, starting with the first lines where you import items from the React library.

#10. .filter(), .map() and .reduce() Explained

Although not a hard rule, in general, the React community leans towards a functional approach to development rather than procedural or a classical object oriented type approach.

At a basic level this means expect to see functions that create or return other functions or accept functions as parameters. It will also, in general, include working to keep our data immutable, as discussed in the section on `const`, `let`, `var`, `.freeze()` and Immutability.

JavaScript provides us with three helpful functional methods that we will see a lot in React code: `.filter()`, `.map()`, and `.reduce()`.

`.filter()` allows us to check if items in an array meet a certain condition. All of the items that do meet the condition will be returned in a new array.

```
let newPosts = posts.filter( post => {
  return post.title.includes( 'React' )
})
```

The example above will look through a collection of `posts`, grab all of the post where "React" appears in the title, and then assign all the matching posts to a new array called newPosts.

`.map()` allows us to call a function on each of the items in an array. It will also create a new array incase the function mutates the data in any way (we do not break our immutability practice).

```
let newPosts = posts.map( post => render( post ) )
```

In the example above we are mapping over all the `posts` and calling the `render()` function on each `post`. We will see a lot of examples of `.map()` like this. It is less likely you will see JavaScript for loops when using React.

`.reduce()` takes an collection of items and reduces them down into one value. A common example of `reduce()` is finding the average of an array of numbers.

```
const prices = [ 19, 39, 209 ]
let average = prices.reduce((total,price,index) => {
```

```
  total += price
  if( prices.length-1 === index ) {
    return total / prices.length
  } else {
    return total
  }
})
// Logs Average: $89
console.log( `Average: ${average}` )
```

`.reduce()` is a little more complicated than `.map()` and `filter()`. It takes a few parameters. The first parameter is referred to as an `accumulator` or memory value as it saves the returned value from each iteration and passes it into the next iteration. This is why we are able to add `price` to `total` in each iteration and it remembers the `total` from the last iteration. You will often see this parameter named something generic like `memo`.

The second parameter is the name you want to assign to the item in the iterable that is currently being run through the function. So the first time, `price` is equal to 19, second it is 39 and the last time it is 209.

The next parameter is simply the `index` of the current iteration. With simple reduce examples you do not need this parameter. We only need it because we want to check if we are on the last item in the array.

Within the `.reduce()` function we are adding the `price` each time and then checking to see if we are in the last item of the array. If we are on the last item we divide by the number of items in the array and return that value. Otherwise we return the `total` and keep iterating.

`.reduce()` must always return a single value. If you want to have multiple values, you may want to do a filter instead.

We will use `.filter()` and `.map()` all the time in React applications. Reduce is used less often, but it is the pattern behind the Redux state management library often used with React.

#11. DOM Node Creation Explained

If you want to understand what React is doing under the hood, it is extremely also be helpful to familiarize yourself with how Document Object Model Nodes are created, nested within one another, and then added to a page.

```
function createHeader(post) {
  const container = document.getElementById('page');
  const header = document.createElement('h2');
  const link = document.createElement('a');
  const text = document.createTextNode(post.title),

  header.classList.add('post-title');
  link.href = post.link;
  link.appendChild(text);
  header.appendChild(link);
  container.appendChild(header);
}
```

In the example above we see the important low level process of using the DOM API to create element nodes and text nodes, customize node attributes, and append them to the page wherever the ID of `page` exists.

We will never see this kind of code in our React apps. However, behind the scenes, React is executing code like this in order to create Nodes for adding to the DOM. For this reason, reviewing and understanding the example above is helpful for understanding what React is doing.

#12. Javascript Exports and Imports Explained

Exports and Imports were added to JavaScript with EcmaScript 2015. They allow us to export code from one JavaScript file and import it into another. React apps are almost always built using exports and imports.

As of the time of publication, browsers do not offer support for export or import. So, we will use a tool like webpack (discussed later) to manage the process of exporting and importing.

```
import React from 'react';
import MyComponent from './MyComponent';
import './App.css';

class Example extends React.Component {
  render() {
    return <MyComponent />;
  }
}
export default Example;
```

The example above is a very common piece of code that will make sense once we cover creating React components and using JSX.

The part we want to focus on is the first line where we import the React library by using keyword `import` followed by the name we want to assign to what we are importing (can be anything) and then the name of the package we want to install as it is referenced in our `package.json` file. (This should make more sense once we get into working with real examples).

In the second line import example we are doing the same as line one, but referencing a file path instead of a package name. This is the common pattern in React for referencing our own React components that we build. In this example, our tooling will look for either a file named `MyComponent.js` or a directory named `MyComponent` with an `index.js` file inside of it. Doing this type of importing will become second nature as you get comfortable with React.

In the third line import example we see that we are just importing in a CSS file. We don't give it a name, we simply import it. Our tooling setup will take care of handling the importing of CSS into a JavaScript file so we don't have to worry about how this actually works at this time. We just need to see an example of what importing a CSS (or SASS) file would look like.

Then we export out our main class using `export default`. If we then went to import our Example component using a line like the one below, we would have access to that class.

```
import Example from './Example';
```

There are some cases where you want a single file to export multiple items, rather than a single default export like above. This could be helpful if you were building a helper library or something similar.

```
const name = 'React';
const ids = [1,2,3,4,5];
function render() {
  console.log( 'Rendered' );
}

export { name, ids, render };
```

Here we are exporting out three separate values. Notice the use of object deconstruction here. Then in another file we can import the items we need, also using object deconstruction.

```
import { name, render: display } from './FileName';
```

In this line above we have importing the name variable as well as the render function, but also renamed the render function to `display()`.

As mentioned, you will get quite comfortable with importing and exporting data when working with React. It is also important to use a tool like webpack to handle your imports and exports as they do not work at this time in browsers.

Let's Practice

Now that we have reviewed quite a bit of JavaScript, let's do a few practice exercises to solidify some of the basics we have covered.

We won't practice everything above, just some of the most essentials. If you want a bit more of a primer on JavaScript, please check out the resources on the book's companion website.

2

5 Exercises in Vanilla JavaScript

In the previous chapter we looked at a lot of JavaScript theory. In this chapter we are going to do a few practice exercises with vanilla JavaScript that will include code similar to what we will see when we write React apps.

Practice #1 – const, let & freeze

In the React ecosystem we will use const by default and let when a variable needs to be reassigned. This exercise will help you get comfortable with when to use each. We also look at .freeze() for when you need to prevent an object or array from any changes.

In this first practice exercise, create a variable named username that should not be reassigned. Then create a variable named loggedIn that can be reassigned. Finally create a post object with an id and title that is frozen.

Test to see whether you can reassign each of the values (only the loggedIn should be able to be reassigned).

Log out the variables you created to test if the values are as expected.

Practice #2 – Template Literals

Template literals allow us to write strings with variables inside of them. In this practice exercise we will create a welcome message including a two variables for someone's first and last name.

To do this, create a variable firstName and assign your first name. Also create one for lastName with your last name.

Then create a template literal that will return "Hi firstName lastName!"‘

This should give you some practice with basic template literals that are often used in React apps.

Practice #3 – Arrow Functions

Arrow functions are a simplified syntax for writing functions in JavaScript. They do not track this so we want to be careful using them, but they are commonly used in simple ways in React apps.

To practice writing an arrow function, rewrite the following function using arrow function syntax into a new function named ArrowName.

```
function MyName(name) {
  return (
    `<p>${name}<p>`
  )
}
```

There are a few different ways you can write this, but try to do so in the simplest format you can get to work.
Finally, log out value of ArrowName() with your name as a parameter to make sure it works.
We will use arrow functions when creating components in React so it is good to get comfortable with them.

Practice #4 – filter() & map()

We will regularly filter and map over arrays of data when working in React. This exercise will help you practice both filtering and mapping over an array together.

First, start with an array of posts like this

```
const posts = [
  {
    id: 1,
    title: "First post"
  },
  {
    id: 2,
    title: "Second post"
```

```
  },
  {
    id: 3,
    title: "Last one"
  }
];
```

Then filter through the posts and find the titles that include the word "post." Then map through the filtered list of posts and log out the titles of each post.
If you can, try chaining the filter and the map functions together. This practice exercise will help you get comfortable with filtering arrays based on matching conditional statements as well as mapping over arrays to do something with the data.

Practice #5 – DOM Creation

In this practice exercise we are going look at simple DOM creation using document.createElement(), document.createTextNode() and appendChild().

While we will not write any of this type of code in this way when working with React, it is helpful to understand some of what is going on under the hood when we using React and JSX.

Starting off with the example index.html file that has a div with an ID of root.

Then try mapping over an array of posts like this one:

```
const postsArray = [
  {
    id: 1,
    slug: "#first-post",
    title: "First post"
  },
  {
    id: 2,
    slug: "#second-post",
    title: "Second post"
  },
  {
```

```
    id: 3,
    slug: "#last-one",
    title: "Last one"
  }
];

postsArray.map(post => {
  // Create post markup
  // Append markup to posts container
});
```

While you are mapping you can create markup for a post that includes an h2 with a link inside it and the title of the post inside that. So you will have final markup like this:

```
<h2><a href="#first-post">First post</a></h2>
<h2><a href="#second-post">Second post</a></h2>
<h2><a href="#last-one">Last one</a></h2>
```

As mentioned, we will not write this kind of code in React because React has much better methods for us to create markup and UIs. However, it is still helpful to know how basic DOM creation works with JavaScript so we can better understand what React does behind the scenes.

What's Next?

In these exercises above we practiced some of the helpful JavaScript we will use when working with React.

In addition to vanilla JavaScript, there are also some common development tools that will be helpful when working with React. In the next chapter we will go over some of these tools that help you write, edit, compile, and debug your React code.

3

Developer Tools for React Explained

In the previous chapter, we have covered some key JavaScript features you should know when working with React

Now, let's turn our attention to talking about tools for React development.

It is possible to build a React application with only a text editor. But I would not recommend this approach. A modern Javascript developer needs to be comfortable with a range of development tools.

In this chapter, we'll look at some of the common types of tools we will use when working with React.

1. Command Line Tools
2. Code Editors and IDEs
3. Node
4. Package Managers
5. Bundling Tools
6. Transpiling Tools
7. Local Development Servers
8. React Dev Tools

If you are already comfortable with these tools, you can skip to the next chapter. However, if some of these tools are new to you, I would suggest reading through the rest of the chapter to have at least a high level understanding of each type of tool.

#1. Command Line Tools

Command line tools allow us to execute code by typing commands into a text based interface called, the command line. Some commands have equivalents with user interfaces. For example, on a Mac, you can

open a file by navigating to the file and type "open readme.md" in the command line and it will open the file. You can also navigate to the file in Finder and double click it to open it.

Some commands however, do not have user interfaces. For example, we will use something called "Create React App." The code for this tool requires you to use the command line to interact with it. There is not an application or window we can open to click a button to "Create a React App." Many of the tools we will use with React are built this way. You have to use the command line to call the commands.

There are, in general, two types of command line tools: stand alone command line tools and integrated command line tools. Stand alone tools just offer you a command line. Integrated command line tools will often give you command line access within another tool, like a code editor. In the next section we will talk about code editors and how many have integrated command line tools.

In order to interact with the command line, you need to know some command line basics. Here are some of the basics you want to know:

- How to navigate to files or folders
- How to list out the content of files and folders
- How to create and delete files and folders
- The basic structure of commands
- How to run some basic commands

If you are not already comfortable with using the command line, I suggest checking out the site commandlinebasics.com to brush up on the basics.

#2. Code Editors and IDEs

Along with a command line tool, a Code Editor is one of your most important tools. Simply put, a code editor let's you edit code without injecting unwanted characters or formatting. A word processor or rich text editor are not tools for editing code and will inject extra characters and break your code.

Here are some popular code editors:

- Visual Studio Code
- Atom

- Sublime Text

These code editors allow for simple code editing. However, they also come with the ability to be extended via themes and plugins (or extensions or add-ons). These extensions allow you to do a lot more with your editor, for example show error hints, formatting help, and other shortcuts and features.

Integrated Development Environments (IDEs) are powerful code editors with more features built in out of the box. A popular IDE for JavaScript development is WebStorm from Jetbrains.

In addition to having more features built in out of the box, IDEs can also do things like keep track of file names and locations so if you change a file name somewhere it can automatically update reference to that file anywhere in your code.

Today the landscape between code editors and IDEs is blurring. Most developers are quite happy extending tools code editors and never using an IDE. However, IDEs do have some major benefits and are worth looking into.

If you have not worked with an IDE before I would suggest downloading the trial version of WebStorm and checking out the tutorial at webstormtutorials.com.

#3. Node Explained

Technically, Node is JavaScript that runs on the server. In contrast, most JavaScript runs in the browser.

While Node is technically a language and not a tool, many of the tools we will use in this book require Node in order to run.

So, one of the first steps to take when working with React is to install Node. Most Macs already have Node installed, but you will want to open your command line tool and check to see if you have Node installed with the following command:

```
node -v
```

If Node is not installed, you can go to nodejs.org and follow the instructions for downloading and installing Node.

As you delve deeper into working with React you will likely come

across React running on the server side with Node. This is outside the scope of this book, but if you do continue to explore React Native then you will actually write React code in Node and not just use it for development tools.

For this book, make sure that you have the latest version of Node installed on your computer.

#4. Package Managers Explained

When we build React applications we rely on multiple libraries. Two important libraries are React and React DOM, which this book covers in depth. We will likely use several other JavaScript libraries when building with React.

The best practice for working with JavaScript libraries is to leverage a package manager like NPM or Yarn. Node Package Manager (NPM) was the original package manager for JavaScript and it still the most popular. Although originally named for Node packages, today it manages frontend scripts for us too. Yarn was created by the folks at Facebook, who created React, so it is quite popular with React developers.

A package manager provides several things:

- A way to download a library for use in your application
- A way to update a library to the latest version
- A way to remove libraries from our applications

The popular package managers for JavaScript (NPM and Yarn) give us a command line interface to do each of these things.

To setup an app for working with NPM, you usually start with opening that app directory in the command line and typing something like this:

```
npm init
```

This command will take you through the process of creating a JSON package manager configuration file, often named `package.json` (and `package-lock.json`). This `package.json` will keep a list of all packages we have installed as well as the version we are using.

To install a library we would find it listed on a site like npmjs.com. This site is the primary resource for JavaScript libraries that you can install with a package manager like NPM or Yarn.

Once we find the library we can install it using a command like this::

```
npm install react
yarn add react
```

This process downloads the library we want, as well as any dependencies that library has, and saves them to a folder in our application that it creates called `node_modules`.

There are generally three different ways to install a package with a tool like NPM or Yarn.

1. Global: This installs the package for use on your entire computer, not just for a single application. This is more common for tools than it is libraries. Globally saved packages are not bundled with your application code for production. Instead they just live on your computer.
2. Dependency: This means your application needs this package to run properly and your build tool should bundle this package along with your final application code.
3. Development Dependencies: This means your application (or tooling setup) only needs this package for local development and the package should not be bundled with your final source code.

If you are wondering which way you should install a package, don't worry! Most packages will tell you the best way to install their library in their installation instructions.

Then in our application code we can import in these libraries in our JavaScript using `import` and the name of the library.

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

In the example above we are importing the React library from our React package saved in the `node_modules` folder. The use of imports only works with a bundling tool like webpack, which we will look at shortly. However, usually imports require a link to a file path. If a file

path is not given, a tool like webpack would fall back to looking in the `node_modules` folder for a library called `react`.

The `node_modules` folder in a large React application will get quite large. For this reason, the contents of a `node_modules` folder is not usually included when application code is saved to something like github or the production server.

Luckily, as long as you have a `package.json` file, you can easily install all of the necessary packages with the following command:

```
npm install
yarn install
```

This wonderful command will download all of the packages listed in the package.json configuration file into a "node_modules" folder that will be created if it doesn't already exist.

For this reason, you will often see "npm install" as the first task when working with an application that you need to edit. This step is necessary because when people share source code for applications they usually do not include the "node_modules" folder or contents.

The node_modules folder is usually not necessary for a completed, live application either. Using a build tool like webpack will allow us to take what code we need from our libraries and combine that with our application code, either in the same file or separate ones.

One last important aspect of working with package managers is that they allow you to create shortcuts for commands that you commonly run. A common example is to create a shortcut called "dev" to start up your development environment. You can then execute this entire command with the following (much shorter) command.

```
npm run dev
```

You will likely leverage shortcuts like this in all of your React applications. The most common types of shortcuts are for a build process, development server, testing and other similar tasks like that.

If you have not used a package manager before it is a good idea to practice setting up a package.json configuration file, installing some

packages, removing some packages, and trying a basic custom script shortcut.

NPM has a great set of tutorials to get you up and running that you can find in the NPM documentation under "Getting Started": docs.npmjs.com.

#5. Bundling Tools

Bundling tools take multiple JavaScript files and combine them into single files.

There are several benefits to using a bundling tool.

First, it is better at this time to make a single request in a website to a single JavaScript file than it is to make a dozen requests to a dozen different JavaScript files. This may change with the adoption of HTTP2, but as of the time of writing, limiting server requests is still a positive thing to do. With a bundle tool, we can take what was a bunch of separate JS files and combine them into just one or two files.

The second benefit of a build tool is that they let you use JavaScript imports and exports. As we mentioned when we introduced imports and exports, they do not have support in the browser at the moment. However, bundling tools do know how to work with import and export.

A bundling tool will create a "dependency map" of your application based on the use of imports. Then, the tool will combine all of those files into a single file. It is also possible (although outside the scope of this book) to break up a single file into several files for convenience or performance purposes.

In this book we will use the bundlingg tool called Webpack. When we first start with Create React App we will not even see the webpack configuration files or code. However, if you create React applications from scratch or make more advanced applications, you will likely need to deal with webpack configurations or commands.

#6. Transpiling Tools

Transpilers take code as an input and output converted code. At their most powerful level, transpilers can take code in one language and convert it into a completely different language. In the context of React

and JavaScript, a transpiler takes in modern JavaScript and outputs JavaScript older browsers can support.

I am using the term "modern JavaScript" here to refer to two things. The first aspect of "modern JavaScript" are features that are newly available or in development but are not yet supported in the browser. A transpiler will take code using these unsupported features and rewrite them in formats that older browsers can support. It does this by either rewriting our new code completely or including polyfills that provide the unsupported functionality and leave our code as is.

The second aspect of "modern JavaScript" refers to code that is included in our JavaScript but not technically part of the current or planned EcmaScript standard. The best example of this is JSX, which we will explore at length in this book. JSX is not technically part of the JavaScript language, but it goes inside of our JavaScript code. A transpiler then processes the JSX code and outputs working JavaScript with no JSX.

Since JavaScript continues to evolve, we will likely continue to use transpilers. In this book we will use a transpiler called Babel. Babel offers configurations to determine exactly what features you want transpiled and what browser versions you want to support. It also offers great default configurations that will automatically support most new JavaScript features and the latest few versions of browsers.

#7. Local Development Servers

At the most basic level, React will work in a single HTML file opened in a browser on your computer. However, to make our development easier, we will leverage a local web server. The webpack bundler we use along with React provides a great server that can also watch for changes in our code and automatically refresh the browser.

In addition to the webpack server, several other Node driven local development servers exist, like http-server. However, for the most part we will leverage webpack development server when writing React.

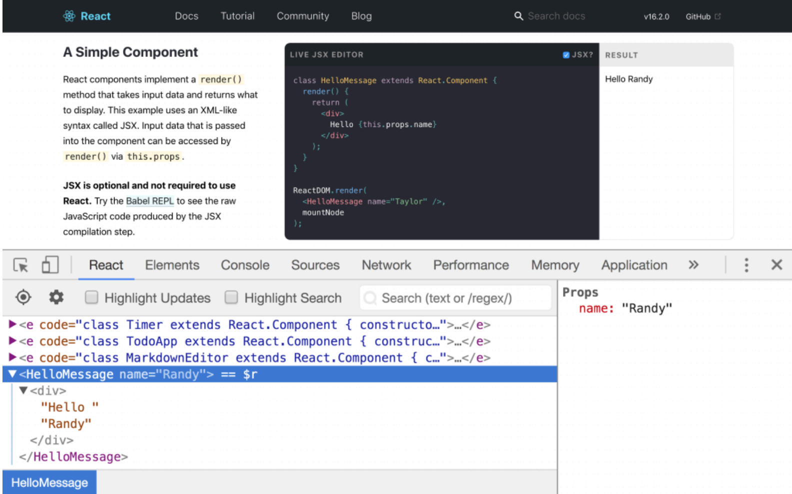
When launching or testing your final React app you can usually use any server that allows client side JavaScript to run. If you have built additional server side functionality outside of your React app you will want to use a production server that supports both. Down the road if

you get into server side React, you will need a server environment that supports Node.

#8. React Dev Tools

When we run a React app in the browser, there is a lot of information about our app available that we can explore with a browser extension. This extension is called React Dev Tools and is available for Chrome and Firefox.

We are going to assume if you are reading a book on React that you have already used a Web Inspector tool to explore things like the DOM, markup, CSS and possibly more.



React Dev Tools adds a new tab to the Web Inspector that shows off information about your React app. In the graphic above you can see that we have highlighted a block starting with <HelloMessage … and what you can't see is that I changed the Props name property from "Taylor" to "Randy" and it updated on the page.

With this tool we can interact with, troubleshoot and explore React apps in the browser. You will likely start to use this tab in place of the normal Elements tab. Note that you can drag and drop the placement of the React tab from under the double arrows (>>) where it

initially appeared to a more convenient place like on the far left in this screenshot.

You will want to install React Dev Tools in either Chrome or Firefox before you get up and rolling with React.

What's Next?

In this chapter, we took a look at helpful tools we will use when working with React. If these are new to you, this may seem overwhelming. Don't worry, once we get into a workflow, you will start to feel more comfortable with these tools.

Now, we're ready to turn our attention to React itself. In the next chapter, I'll give you a high-level overview of React's architecture, data flow, components and more.

4

5 Exercises with Developer Tools

Later on in this book we will learn about a tool called Create React App that simplifies how we use many of the tools covered in the previous chapter. However, it always helps to have some basic command line skills and understanding of what is going on behind the scenes with our JavaScript development tools.

These exercises are designed to help you get comfortable some of the skills and tools that help our JavaScript development process.

Practice #1

In this first exercise we will practice using some basic commands in a command line tool.

Open command line tool (like Terminal on the Mac or Cmder for the PC). Then try to follow along with the following steps from the command line:

1. Navigate to your root folder by typing “cd ~”
2. Then navigate to your Desktop by typing “cd Desktop”
3. Create a folder on your Desktop by typing “mkdir test-folder”
4. Navigate into the new folder by typing “cd test-folder”
5. Create a README.md file in the new directory by typing “touch README.md”
6. Finally, list out the contents of your directory using “ls”

Continue to practice this exercise until you can remember the necessary commands and get comfortable typing them. This will ensure you have the basic skills with the command line to use the other tools we will need.

Practice #2

In this exercise we will practice getting a project up and running that uses NPM for managing the JavaScript packages and libraries we will use in our projects.

To start, open the "chapt-2-dev-tools/practice-2" directory in a code editor with a command line tool.

Look inside the package.json file. You should see "moment", the popular JavaScript date and time helper library listed as a dependency.

To install moment (and any other dependencies that could be listed in the package.json file) we need to run "npm install."

Run "npm install" in the "practice-2" directory and you should see two things happen:

1. A node_module folder appears
2. A package-lock.json file appears

Navigate into the node_modules folder and you should see "moment" listed along with a number of other dependencies.

If you can get this far it demonstrates an important skill: the ability to setup a project in your development environment that uses NPM to manage dependencies.

In order to use dependencies in our code we will have to use a bundler tool, like webpack, which we will do in the next exercise. However, now when a project you want to use says in the setup instructions, run "npm install," you know what to do.

Practice #3

In this exercise we are going to look at how to use the webpack bundling tool so we can use imports and exports in our JavaScript code as well as combine multiple JavaScript files into one final bundled file.

To start, open the "chapt-2-dev-tools/practice-3/starter" directory in your code editor.

Open the package.json file. It should look similar to the previous practice exercise.

Run npm install to get moment, React and React DOM installed.

Then we want to install webpack and the webpack command line

helper library as developer dependencies. This means they are loaded to our local computer but not saved in our final bundled code like moment.

Run “npm install webpack webpack-cli —save-dev”.

Once it is done running you should see “webpack” and “webpack-cli” listed in package.json as devDependencies.

Next, we need to create a “webpack.config.js” file and place the following inside:

```
const path = require("path");

module.exports = {
  entry: "ENTRY_PATH",
  output: {
    filename: "OUTPUT_FILENAME",
    path: path.resolve(__dirname, "OUTPUT_DIRECTORY")
  }
};
```

Then change the ENTRY_PATH to “./src/index.js” to point our main JavaScript file.

Change OUTPUT_FILENAME to “main.js” and OUTPUT_DIRECTORY to “dist” to tell webpack to save the final bundled file to “dist/main.js.” This is the file our index.html file links to.

Now, we need to setup our webpack commands.

Open the package.json file and after line 4 “description” add the following:

```
"scripts": {
  "start": "",
  "build": ""
},
```

This will give us two new commands for our project:

- npm start
- npm run build

Inside of these we can call webpack cli commands.

Update the scripts with the following webpack commands:

```
"scripts": {
  "start": "webpack --watch --mode=development",
  "build": "webpack --mode=production"
},
```

Now, running "npm start" will get webpack to watch our files for changes and bundle our code so it is still slightly readable. When we run "npm run build" webpack will do a one time bundle of our code into a highly minified format for shipping to production.

Both commands will output the bundled file to "dist/main.js"

Try running "npm start". You should get some messages in the command line saying that your code has been compiled. Also, a "main.js" file should appear in your "dist" folder.

Open the "dist/main.js" file to see a bundled version of our "src/index.js" file.

Now open the "src/index.js" file and a make a change to the text in the last line that displays the date on the page. Once you make the change, you should see the command line tell you that webpack has rebundled your code.

If you open the "dist/index.html" file in the browser you should see your changes take place.

Stop webpack from watching your files by typing Crtl + C in the command line.

Now try running "npm run build." Once this is complete it should tell you the bundle was complete and give you the command line prompt again.

Open the "dist/main.js" file. It should look highly minified. In the future, if you make changes to your code you will need to run "npm run build" to get a production bundle of your code open.

This exercise shows you how to integrate webpack with NPM scripts to bundle our code and allow us to use imports in our code like we do with the moment library in this example.

Practice #4

In this exercise we will look at how to add Babel to our project for transpiling our newer JavaScript and React code.

To start, open the chapt-2-dev-tools/practice-4/starter" in your code editor.

Install the core Babel libraries:

```
npm install--save-dev @babel/core @babel/preset-env babel
```

Then install the Babel libraries for React and JSX:

```
npm install --save-dev @babel/preset-react
```

Finally we will install a Babel library for using class field properties, which is not yet fully supported:

```
npm install --save-dev @babel/plugin-proposal-class-prope
```

Now we can set these up in our ".babelrc" and "webpack.config.js" file.

Open up the ".babelrc" file, which controls our Babel configuration. Update the "presets" and "plugins" with the following values:

```
"presets": [
  "@babel/preset-env",
  "@babel/preset-react"
]
```

This tells Babel to support the latest few versions of popular browsers as well as support React and JSX. Then update the "plugins" array with the following:

```
 "plugins": [ "transform-class-properties" ]
```

This will allow our class properties to work.

Next we need to setup webpack to make sure that all of our JavaScript passes through the Babel configurations we just setup.

Open the webpack.config.js file and add a comma to the end of line 8 at the end of the output object. Then after it place the following:

```
module: {
  rules: [{
    test: /\.js$/,
    exclude: /node_modules/,
    loader: "babel-loader"
  }]
}
```

This will ensure that any file ending with ".js" will passed through our Babel settings. Anything in the "node_modules" will be excluded from this, which is good.

To test all of this, run "npm start" and open the "dist/index.html" in the browser. It should display an h1 saying "Title" and a message "Posted" with the current date.

If you look at the "src/index.js" file you can see that it is using moment, React and ReactDOM. We will learn more about what everything here is doing later in the book. For now we just need to know that this code would not work if we did not setup Babel properly.

In the future you may need to setup and use Babel in similar ways to how we used it here in order to support new features in JavaScript or related libraries.

Practice #5

In this last practice exercise we are going to setup a server to run our files, rather than just opening our index.html directly. This will also allow our tooling to automatically refresh the page whenever we make changes.

To start, install the Webpack Dev Server with the following:

```
npm install --save-dev webpack-dev-server
```

Then open the webpack.config.js after line 8, place the following:

```
devServer: {
  contentBase: "./dist"
},
```

This will tell webpack to use our “dist” folder as the root for the server and load our index.html automatically.
Finally we need to update our npm start command to load the server.
Open package.json and change the “start” command to the following:

```
"start": "webpack-dev-server --open --mode=development"
```

This will cause the webpack dev server to launch when we call “npm start.”
To test this, run “npm install” and then “npm start”.
This should open http://localhost:8080/ in your browser, with our code running.
If you go into the “src/index.js” file and change the text in the <h1> it should automatically refresh the page in the browser.
To stop the server, press Crtl + C.
Development servers like Webback Dev Server are very helpful for building and testing our sites. Remember though that if you want to have a build of your code to ship to production, you should still run “npm run build”.

Next Steps

The practice exercises from this chapter can help you get more comfortable with some of the basic tools and skills we will need for React development.

However, in a later chapter we will explore how the development tool, Create React App, combines all of the tools we have looked at here into one simpler tool.

So, now, let’s turn our attention away from JavaScript and Development Tools and to React itself.

5

A High Level Overview of React

In this chapter, we are going to get a broad introduction to React.

As you read the next few pages, you'll see React code for the first time. We won't start writing React in this chapter, but we will see example code. We'll take a close look at those examples so we can start getting into the React frame-of-mind.

The Key Concepts of React

Out of the box, React is a library for building user interfaces.

Although React is a JavaScript library, the interfaces you will build with React are language-agnostic.

React has companion libraries that enable your interfaces to work in different locations. React has libraries to make your code work in the browser, on the server, in native applications and even in 360 and Virtual Reality environments.

In this book, we focus on working with ReactDOM, the library that enables your interfaces to work in client-side websites and applications in the browser. ReactDomServer, ReactNative and React360 are also libraries you may want to explore for using React interfaces in other environments.

In addition to providing helper functions for building interfaces, React's architecture allows for you to handle interactions with your interfaces. These interactions can involve event handling, API calls, state management, updates to the interface, or more complex interactions.

React does not provide as many helper functions as some JavaScript frameworks. This is in large part why we call React a library and not

a framework. You will still need to write plenty of vanilla JavaScript when working with React.

React's Architecture Explained

In programming, a component is an independent, reusable piece of code. Components are created via a JavaScript function or class in React.

React uses a component architecture for building user interfaces and organizing code. The main file for a simple React app may look something like this.

```
// Import React and Other Components
import React from 'react';
import ReactDOM from 'react-dom';
import Header from './Header';
import MainContent from './MainContent';
import Footer from './Footer';

function App(){
  return (
    <div className="app">
      <Header />
      <MainContent />
      <Footer />
    </div>
 );
}

ReactDOM.render(
  <App />,
  document.getElementById("root")
);
```

We can see here a few components in use. `<Header />`, `<MainContent />` and `<Footer />` are all components. The `App()` function is a component as well and we can see on the last line of this example how we can use the ReactDOM library and the

ReactDOM.render() method to manage adding the UI we build to a webpage.

If we dig inside of the <Header />, <MainContent /> and <Footer /> components, we would likely see the use of more components as well as what looks like HTML markup.

```
import React from "react";
import Ad from "../Ad";
import logo from "../assets/logo.svg";

export default function Header() {
  return (
    <header className="app-header">
      <Ad />
      <img src={logo} alt="logo" />
      <h1 className="app-title">Site Name</h1>
    </header>
  );
}
```

In this <Header /> component above we can see that we are pulling in yet another component called <Ad />. Most React applications contain several layers of component nesting like we see with <App />, <Header /> and <Ad />.

We also see the use of HTML elements in our React code. This is possible thanks to a library called JSX, which lets you write "HTML markup" directly in your JavaScript. Since we are using React to create user interfaces, and user interfaces on the web involve HTML markup, this makes sense we would see HTML like elements within our UI components. We will explore JSX in depth in this book.

If we look at some code from for a simple React app built using React 360, React's VR library, the actual components we call would be different, but the component architecture is still present.

```
import React from 'react';
import {
  Text,
  View,
```

```
  VrButton,
} from 'react-360';

class Slideshow extends React.Component {
  // Code removed for brevity
  return (
    <View style={styles.wrapper}>
      <View style={styles.controls}>
        <VrButton onClick={this.prevPhoto}>
          <Text>{'Previous'}</Text>
        </VrButton>
        <VrButton onClick={this.nextPhoto}>
         <Text>{'Next'}</Text>
        </VrButton>
      <View>
        <Text style={styles.title}>
          {current.title}
        </Text>
      </View>
    </View>
  );
}
```

The code above creates several layers of 360 views with some buttons and text overlaid. While the actual code might not make complete sense, it should be clear that we have several nested components representing the view, button and text.

This is a good example because you can see how the same components are reused in different ways by passing them different parameters, or what React calls props. Understanding how data passes through React components is important for understanding the typical component architecture used for building with React.

React's Data Flow Explained

React will get and set data at the highest point necessary in a component hierarchy. This allows data to pass in a one-way direction down through an application.

Let's take a look at this example and imagine some of the types of data we would need for various components.

```
function App() {
  return(
    <React.Fragment>
      <Header />
      <Content />
      <Sidebar />
      <Footer />
    </React.Fragment>
  );
}
```

Something like the name of the site might need to be available to both the `<Header />` and `<Footer />`. The main content for the particular page would need to be passed to `<Content />`. Some additional widget data might need to go to `<Sidebar />`.

```
function App() {
  const siteTitle = getSiteTitle();
  const widgets = getWidgets();
  const mainContent = getPageContent();
  return(
    <React.Fragment>
      <Header siteTitle={siteTitle} />
      <Content mainContent={mainContent} />
      <Sidebar widgets={widgets} />
      <Footer siteTitle={siteTitle} />
    </React.Fragment>
  );
}
```

This convention of making up attribute names and assigning them a value is how we pass data into a component.

Now the `<Header />` and `<Footer />` have access to the `siteTitle`, the `<Content />` has access to the `mainContent`, and `<Sidebar />` has access to the `widgets` it needs.

An important note is that this pattern of passing data into a

component only passes the data one level. Components inside of `<Header />` will not automatically get access to `siteTitle`.

```
function Header(props) {
  return(
    <header>
      <p>We can see the {props.siteTitle} here.</p>
      <PageHeader siteTitle={props.siteTitle} />
      <PageSubHeader />
    </header>
  );
}
```

You can see here that inside `<Header />` we can call `props.siteTitle` and have access to that value we passed into it. However, if we wanted to have access to `siteTitle` within the `<PageHeader />` component we would have to manually pass that information down as well.

> When a component receives a value as a prop, it should not modify it.

Props should pass through a component tree as immutable data. This ensures that any component that references a prop, references the same value as other components receiving that prop.

The value of a prop should only be changed in the component that originally set the value of the prop and started passing it down through the component tree. In our example code above, the `<App />` component could change the value of `siteTitle`, but the `<Header />` or `<PageHeader />` components should not.

To understand the flow of how dynamic data gets updated in a React app involves discussion of *state* and how event handlers can be passed as props.

React Component States Explained

As we have learned, data flows down unchanged through components as props. Data is set at the highest component in the component tree

necessary for all children components to be passed the information need as props.

In some cases, this data is received once and does not need to change. In many cases though that data must remain dynamic and have the ability to update at any given time and have that update reflected in all children components.

To keep track of data that changes in React we have a React state object and a helper function to update the state value.

Here is an example of a counter that would update itself. The value of the counter is a value that is dynamic within this component and therefore makes a good instance of when to rely on state.

```
class Counter extends Component {
  state= {
    counter:0
  };

  handleCount = () => {
    this.setState({
      counter: this.state.counter + 1
    });
  };

  render() {
    return (
      <div>
        <h1>{this.state.counter}</h1>
        <button onClick={this.handleCount}>
          Count Up!!
        </button>
      </div>
    );
  }
}
```

Now it is important to note that this state is scoped to just this component. The value of state in counter would not be available to child or parent components.

So in a more complex example, like below, we would have to pass the value of count down as a prop into the child element.

```
class Counter extends Component {
  state= {
    count:0
  };

  handleCount = () => {
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return (
      <div>
        <PageHeader count={this.state.count} />
        <button onClick={this.handleCount}>
          Count Up!!
        </button>
      </div>
    );
  }
}
```

The <PageHeader /> count prop gets updated every time we update the state in the <Counter /> component

```
function PageHeader(props) {
  return <h1>{props.count}</h1>;
}
```

The nice thing about this approach is that anytime state is updated, a new value will be automatically passed down into any child components with the value of a prop set to state.

This allows us to have a single point of truth for dynamic data. The source of truth is the value in state, managed from a single component. All instances of this value in children components are

immutable values received as props that should not be changed outside of this component.

Components that appear above this component in the hierarchy would not have access to this data as it is only passed *down* via props. We see again why we try to set and manage state from components higher in the hierarchy so that the data is available to everything that needs it.

There are some other architecture patterns, like higher order components and the context API, which circumvent needing to manually pass tons of props through your app. For now though, we want to make sure we understand this high level overview of how things generally work before we start taking shortcuts.

Updating Component State from Child Components

Now, what happens when we want to trigger state to be updated from a child component?

Imagine, for instance, that with the example above we wanted to have a <Button /> component rather than a hard coded button in our main <Counter /> component? This is actually quite common in complex apps.

The solution to this, in the React world, is to pass the event handler function that updates the state with setState down as a prop. Then it can be called from any child component, but the action will take place in the original component that set the state and has the ability to update it as well.

If you are not familiar with passing functions as parameters, it is completely valid vanilla JavaScript.

Once the event handler is called from the child component, state will be updated in the parent component, the new value of state will be passed down through the component hierarchy via props.

Here is an example of what that would look like.

```
class Counter extends Component {
  state= {
    count:0
  };
```

```
  handleCount = () => {
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return (
      <div>
        <PageHeader count={this.state.count} />
        <Button handleCount={this.handleCount} />
      </div>
    );
  }
}

function PageHeader( props ) {
  return(
    <h1>{props.count}</h1>
 );
}

function Button( props ) {
  return(
    <button onClick={props.handleCount}>
      Count Up!!
    </button>
  );
}
```

Here we can see a simple example of how React handles data flow. There is a single point of truth for data. This exists in state that is set and updated from a single component. Then data is passed in a one way flow down through a nested component tree via props.

If state needs to be updated from a component other than where it was originally set, then an event handler can be passed down to the necessary child component as a prop. This keeps data immutable and flowing one way because even if a child component triggers a change, that change takes place higher up in the original component.

When we assign the value of a prop to something from state, like below, that prop value will automatically update whenever state changes.

```
<PageHeader counter={this.state.count} />
```

Any other child component that references that prop value will also receive the update automatically. This is the beauty of data flow in React.

This can take a little while to get used to depending on how you have approached problems like these with JavaScript in the past. However, this should all serve as a good starting point for us to be able to dig deeper into explaining React.

What's Next?

From here we will begin looking at how to build Elements and Components with React and add them to the pages. This will get us hands on with building User Interfaces and practicing our React skills.

We then proceed into more depth on the concepts we outlined above, learning how to put them into practice so we know how to use them on our own. So, review any concepts above that feel worthy of a second glance, then let's jump in to writing some React of our own.

PART II

React Explained

This is the heart of this book. In the coming chapters we will explore the foundations of React.

6

An Introduction to React Elements and Components

In the previous chapter, we saw our first examples of React code.

In this chapter, we're going to dig deeper as I introduce you to React elements and components. As soon as this chapter is complete, you'll be ready to start writing your first React code.

React Elements Explained

A React element is a specific way of referring to a piece of the user interface.

Technically, a React element is just a JavaScript object with specific properties and methods that React assigns and uses internally.

Companion libraries can then take React elements and translate them into elements native to whatever environment you're running React.

For example, when we use ReactDOM, React elements are turned into DOM elements. On the other hand, when we use React Native, React elements are turned into native Android and iOS UI Elements.

However, React elements are not the same thing as DOM elements. This is an important distinction. React elements are agnostic to the environment in which they are ultimately rendered.

React elements are created using a function called `createElement()`.

React.createElement() Explained

The `.createElement()` method is part of the Top-Level React API and used to generate React elements.

```
createElement( 'type', [ properties ], [ children...] );
```

The method takes three parameters:

1. The type of element to create
2. The properties or attributes you want assigned to the element
3. Element children (can be a strings of text or other elements)

The example below gives a basic demonstration of .createElement in action.

```
const hello = React.createElement(
  "p",
  { className: "featured" },
  "Hello"
);
```

Internally, this would create an object that looks something like this.

```
▼{$$typeof: Symbol(react.element), type: f, key: null, ref: null, props: {…}, …}
   $$typeof: Symbol(react.element)
   key: null
 ▶ props: {}
   ref: null
 ▼ type: ƒ P()
     arguments: (...)
     caller: (...)
     length: 0
     name: "P"
   ▶ prototype: {constructor: ƒ}
   ▶ __proto__: ƒ ()
     [[FunctionLocation]]: index.js:6
   ▶ [[Scopes]]: Scopes[2]
   _owner: null
 ▶ _store: {validated: false}
 ▶ _self: {__esModule: true}
 ▶ _source: {fileName: "/Users/zgordon/Dropbox/React Book/demos/create-react-app/elements/src/i
 ▶ __proto__: Object
```

However, if we were to pass this object into the companion ReactDOM library we would get something like this:

```
<p class="featured">Hello</p>
```

This demonstrates again how React elements are JavaScript objects with properties and methods unique to React that can be passed to

companion libraries and converted into elements native to those environments. In this case, a React object is passed to ReactDOM and converted to valid DOM elements.

Here is the full code for how we would make that work with ReactDOM.

```
import React from "react";
import ReactDOM from "react-dom";

const hello = React.createElement(
  "p",
  { className:"featured" },
  "Hello"
);
ReactDOM.render(
  hello,
  document.getElementById("root")
);
```

The code above first imports both the React and ReactDOM libraries.

Then it calls `ReactDOM.render()`, which takes a React element as the first parameter. The ReactDOM.render() will then convert this React element into a valid DOM element.

The second parameter tells ReactDOM where on the page it should add this newly created DOM element.

In this case, we are telling ReactDOM to render our paragraph element inside of the HTML element with an ID of root.

Creating Components with .createElement() Explained

Most of the time we will not call `.createElement()` inline like we did in the example above. Most of the time we will save the call inside of functions (and later classes) so they can be reused or called in other places in our code.

This might result in a pattern like the one below.

```
function Welcome() {
  return React.createElement("h1", { className:"welcome"}
}
```

```
ReactDOM.render(
  Welcome(),
  document.getElementById("root")
);
```

Here we see the `createElement()` call assigned to a new function called `Welcome()`. `Welcome()` would be considered a "component" in React.

> A React component is a function or class that returns a valid React element.

Now, whenever we wanted to display a welcome message, we have a reusable component (or function) that we can easily call.

Note that component names in React are capitalized. `Welcome()` is used and not `welcome()`. This is done to to distinguish normal functions from ones that return React elements.

In case it isn't clear, the code above would result in the following DOM element being added to the page.

```
<h1 class="welcome">Welcome!</h1>
```

After we introduce JSX in the next chapter, we will no longer call `createElement()` manually, however, there are a few more aspects of element and component creation we should explore before learning the easy way to do things with JSX.

Nested Elements and Components Explained

In the previous examples we called `createElement()` and passed a string of text as the child parameter. More commonly, elements will be nested within each other.

Here is an example of a nested createElement() call in action.

```
function Welcome() {
  return React.createElement(
    "h1",
    { className: "welcome" },
```

```
    React.createElement(
      "a",
      { href: "https://reactjs.org/" },
      "Welcome!"
    )
  );
}

ReactDOM.render(
  Welcome(),
  document.getElementById('root')
);
```

Here we see a React component called `Welcome()` that returns an h1 React element. In turn, that h1 element has a React a element passed as the child value.

Once passed through ReactDOM, this component would render as follows.

```
<h1 class="welcome">
  <a href="https://reactjs.org/">Welcome!</a>
</h1>
```

This pattern of nesting elements is quite common and can go many layers deep depending on the UI being built. We can also nest components within one another and pass them as children parameters to `createElement()`.

Read through this example below that contains a number of examples of elements and components being nested.

```
import React from "react";
import ReactDOM from "react-dom";

function Welcome() {
  return React.createElement(
    "h1",
    { className: "welcome" },
    React.createElement(
      "a",
```

```
      { href: "https://reactjs.org/" },
      "Welcome!"
    )
  );
};

function Footer() {
  return React.createElement(
    "footer",
    { className: "entry-footer" },
    Divider(),
    React.createElement("p", {}, "Goodbye ?")
  );
};

function Divider() {
  return React.createElement( "hr" );
};

function App() {
  return React.createElement(
    "article",
    { className: "post" },
    Welcome(),
    Divider(),
    React.createElement(
      "div",
      { className: "entry-content" },
      React.createElement("p", {}, "Main content")
    ),
    Footer()
  );
};

ReactDOM.render(
  App(),
  document.getElementById("root")
);
```

Let's work backwards through the code above, starting from the last line where we pass the `App()` component into `ReactDOM.render()` to be converted into valid DOM elements and add to the page inside of whatever element has the id of "root."

If we look at the `App()` function, or component, we can see it is returning an "article" element, that `ReactDOM.render()` will convert into a valid HTML `<article>` tag. The article tag has a class of "post" and four children elements:

- A Welcome() component
- A Divider() componet
- A "div" element created inline with `createElement()` that contains a "p" element as a child, also created with `createElement()`
- A Footer() component

As we go further with React we will not usually call React.createElement manually like this, however, the example above shows how you can pass both components and `createElement()` calls as children parameters to `createElement()`.

To fully understand our app we would next have to explore the Welcome(), Divider() and Footer() components. This process of starting with a high level component and then moving down the nested component tree to see what it contains is a very common practice. If you learn to follow a component hierarchy you will be able to find your way around most React applications.

So you can double check your interpretation of the code above, here is what the code would ultimately produce in the browser once passed through `ReactDOM.render()` and added to the page.

```
<article class="post">
  <h1 class="welcome">
    <a href="https://reactjs.org/">Welcome!</a>
  </h1>
  <hr>
  <div class="entry-content">
    <p>Main content</p>
  </div>
```

```
  <footer class="entry-footer">
    <hr>
    <p>Goodbye ?</p>
  </footer>
</article>
```

Reread through the code until this all makes sense. Please note that this is not the cleanest way of nesting React elements and components, but it does allows us to see the flexibility of what is possible.

Breaking Up Components Into Separate Files Explained

I don't recommend storing all of our components in a single file. That is generally a bad idea for any large JavaScript application. Instead, we can break them into separate files using `exports` and `imports`.

Officially, React does not have a defined set of naming conventions and file architecture to follow. However, many React apps follow similar conventions. Throughout this book we will explore some different conventions for this.

Refactoring our example from above we could create the following files:

```
/src
|-- index.js
|-- App.js
|-- Welcome.js
|-- Divider.js
|-- Footer.js
```

A few common conventions are followed here:

- index.js – Imports App() and calls ReactDOM.render()
- App.js – Imports the other components and exports App()
- Welcome.js – Exports the Welcome() component
- Divider.js – Exports the Divider() component
- Footer.js – Exports the Footer() component
- We have an index.js file that would be the main entry point for our app
- Component file names match the names of the components they contain

- Files exporting components are capitalized

For larger applications, we often also see certain components or aspects of the app moved into their own directories.

For now, we simply need to know that we follow the same best practices for writing React code that we do for writing JavaScript in general: organize your code into logical, modular, parts.

A Note on Fragments

As we have seen, components must return a single element. This element can have other elements or components nested within it, but a component cannot return two sibling elements.

This is not a problem most of the time, but there are instances where you don't want to add additional parent elements just to get around the sibling elements issue.

Let's take an example where an `App` component returns a `Header`, `Content` and `Footer` component. In order to do this we would have to wrap all the components in a div or something like that.

```
function App() {
  return (
    React.createElement("div", {},
      Header(),
      Content(),
      Footer()
    )
  )
}
```

However, let's imagine we did not want to have another div or parent element at all there. To solve this problem, React has the React.Fragment.

A Fragment is a DOM node that exists in memory as a wrapper node, but disappears and leaves no markup once it is added to the page. It is not unique to React as Fragments are a valid part of the DOM API. However, React has it's own version that looks like this.

```
function App() {
  return (
    React.createElement(React.Fragment, {},
      Header(),
      Content(),
      Footer()
    )
  )
}
```

The code above solves the problem of returning a single element. However, that element does not return any markup. If we were to render this to a page with ReactDOM.render() we would just see Header, Content and Footer rendered with no parent element.

Using React.Fragment is not required but it is good to remember it exists incase you ever have a situation where you do not want an actual element displayed to the page.

Writing Functional Components with Arrow Functions

When we create components using functions in React, it is possible to use arrow functions for a shorter syntax. This is not required, but it is a pattern you will see done.

Arrow functions in JavaScript do not keep track of the `this` keyword. However, we do not need the this keyword when building simple components. For this reason, arrow functions are an acceptable syntax for creating components, as long as we don't need to scope or bind `this`.

Here is what the arrow function looks like when used for creating components.

```
const App = () => {
  return (
    <div className="App">
      <Header />
      <Content />
      <Footer />
    </div>
  );
```

```
};
ReactDOM.render(
  <App />,
  document.getElementById("root")
);
```

Notice that we are using fat arrow functions here without any parameters so we have an empty `()` parenthesis. Then our normal return() inside of the function with the components or other JSX we want to have our component return.

We will use arrow functions to create components until we learn about State and components that will need binding of this later on in this book.

A Brief Review of Elements and Components

When working with React, some common terms will get used in different ways to mean different things. Elements and components can often be used this way.

Remember, a React element is simply a JavaScript object with certain properties and methods unique to React that is created using `React.createElement()`. When writing React for the web, React elements usually map to HTML elements.

A component in React is a function or class that returns a valid React element.

React elements and components can both be nested to allow for building complex user interfaces. Although we can store multiple components in a single file, it is generally a good idea to break up our apps into modular files. Often time in React a file will contain a single element and that is all.

What's Next?

In this chapter, you learned about React elements and components. You're ready to start writing React code. Turn the page, and we have 5 practice exercises to get you started with React.

7

5 Exercises in Writing React With Elements and Components

Let's start writing React!

We have 5 exercises for you that will help you starting writing React.

You can download the practice exercises for this book at https://github.com/zgordon/react-book. The exercises for this chapter are under "chpt-4-elements-and-components." You will find "practice-starter" with comments outlining the exercises. Then the "practice-completed" has all the completed examples for you to check your answers against.

Here is a brief overview of the practice exercises with some insight into each one.

Practice Exercise #1

The first exercise involves create a simple paragraph element using `React.createElement()`. The paragraph element should not have any special classes or attributes and some simple text like "Hello React."

You should save this element as a variable using `const`. Then at the bottom of the exercises where ReactDOM.render() is called, add your element variable name there to test that you created it properly.

You can open the index.html file in the browser to test that everything works properly. If you get stuck, check the completed code for a little help.

You final markup should look like this:

```
<p>Hello React.</p>
```

Practice Exercise #2

The second exercise is similar to the first in the you will start off creating an element and saving it as a variable. The element we're creating as an h1 element with a class of "entry-header." However, this element has a link element inside of it that links to the React website and includes the text of "React."

Like the example above you will have to add your element variable to the ReactDOM.render() call in order to test it.

Your final markup should look like this:

<h1>React</h1>

Practice Exercise #3

In the next exercise we will create a component rather than a single element. The component is called `Header` and it should return a `header` element with an ID of "main." Inside of the header element you should pass in the p element and the h1 element you created from Exercises #1 and #2.

To test you will add Header() to ReactDOM.render(). Your final markup should look like this:

```
<header id="main">
  <h1><a href="http://reactjs.org/">React</a></h1>
  <p>Hello React.</p>
</header>
```

Practice Exercise #4

From here we continue with another component example. This exercise has you creating a component called `List` that returns an unordered list with three list items within it. Each list item should be a link to a React resource. The `ul` element should also include both a custom class and ID attribute.

When you call `List()` in `ReactDOM.render()` it should return markup like this:

```
<ul class="react-links" id="top">
  <li>
```

```
    <a href="http://reactjs.org/docs">
      React Docs
    </a>
  </li>
  <li>
    <a href="https://reactjs.org/docs/react-dom.html">
      ReactDOM Docs
    </a>
  </li>
  <li>
    <a href="http://reactexplained.com/">
      React Explained Book
    </a>
  </li>
</ul>
```

Practice Exercise #5

In our final exercise we create a component called `App` that returns a `React.Fragment` with our `Header` and `List` components within it. This will give us practice using `React.Fragment` as well as creating components that return other components that in turn return individual elements. This is a fairly common practice in React.

The final markup for this will look something like this:

```
<header id="main">
  <h1><a href="http://reactjs.org/">React</a></h1>
  <p>Hello React.</p>
</header>
<ul class="react-links" id="top">
  <li>
    <a href="http://reactjs.org/docs">
      React Docs
    </a>
  </li>
  <li>
    <a href="https://reactjs.org/docs/react-dom.html">
      ReactDOM Docs
    </a>
```

```
  </li>
  <li>
    <a href="http://reactexplained.com/">
      React Explained Book
    </a>
  </li></ul>
```

What's Next?

After you have successfully completed the practice exercises, you should feel comfortable creating basic elements and components using React. I would encourage you to try creating some of your own elements and components.

As you may have noticed, using React.createElement can become quite cumbersome, especially when creating nested components and elements.

Luckily we have a library called JSX that gives us an easy to use shorthand for React.createElement.

Rather than do something like this with vanilla React:

```
const boldLinkPEl = React.createElement(
  "p",
  { className: "featured" },
  React.createElement(
    "a",
    { href:"https://reactexplained.com/" },
    React.createElement(
      "strong",
      {},
      "Important Link"
    )
  )
)
```

We can write this with JSX:

```
<p className="featured">
  <a href="https://reactexplained.com/">
    <strong>
```

```
      Important Link
    </strong>
  </a>
</p>
```

You'll see that this looks an awful lot like HTML right inside our JavaScript. This is exactly what JSX is. JSX gives us the ability to write what looks like HTML, but is actually just a shortcut for writing `React.createElement`.

In the next chapter we will go over the rules of JSX. Going forward we will hardly ever (possibly never) need to write `React.createElement()` in it's long form again.

8

An Introduction to JSX

In the previous chapter, we created React elements and components using `React.createElement()`.

As we saw, creating components can involve a lot of nested functions calls and object definitions. These do not make for code that is easy to read and write.

JSX is a separate JavaScript library from React that serves as a shorthand for calling `React.createElement()` to create elements and components. JSX looks like HTML. But, thanks to Babel, it processes as valid JavaScript.

Most React apps use JSX. It is possible to use JSX without React, but it is most commonly used alongside React.

In this chapter, we'll look at the rules and syntax for JSX.

What is JSX?

Hopefully you have seen the markup language, HTML. XML is an extended version of HTML where you can create your own elements with names of your like `<item>`, `<query>`, or pretty much anything else you want.

JSX stands for **J**avaScript **X**ML. It is an extension for JavaScript that allows for writing what looks like HTML and XML in your JavaScript.

So you may have some code that looks like this:

```
const Heading = () => (
  <h1>
    <a href="https://reactjs.org/">
      React!
    </a>
```

```
  </h1>
)
```

Within the React ecosystem we could take this `Heading()` function and use it as a component that display an h1 with a link inside when passed through `ReactDOM.render()`.

There are important rules to JSX we need to know, but first let's look at what is going on under the hood when we write JSX.

JSX is Just React.createElement() Under the Hood

Behind the scenes, when JSX gets transpiled (by Babel in our case), it uses `React.createElement()` to create the same elements it represented in its XML style. Since `React.createElement()` is a basic JavaScript function we can call it in the browser without needing it to be transpiled further.

So taking the following example again with JSX:

```
const Heading = () => (
  <h1>
    <a href="https://reactjs.org/">
      React!
    </a>
  </h1>
)
```

When we transpile this using a tool like Babel React JSX Transform, we get the following code:

```
const Heading = () => (
  React.createElement(
    "h1",
    null,
    React.createElement(
      "a",
      { href: "https://reactjs.org/" },
      "React!"
    )
```

```
  );
)
```

You can see here that in the place of the XML style markup in the JSX example, we have the `React.createElement()` function being called.

You will soon get very comfortable writing JSX and forget all about `React.createElement()`. Let's look now at where we tend to see JSX written, since we can't throw it just anywhere without knowing what we're doing.

Where Can We Write JSX?

You will likely see JSX written in two places:

1. It can be saved as a variable anywhere in your JavaScript code.
2. It is usually included in the return statement of Component functions and classes.

Let's take a look at each of these:

```
const heading = <h1>Heading</h1>;
```

Now anytime we wanted to reference that `heading` element we could using the heading constant. While this is not the most common way of writing JSX, it is done often, so it's helpful to know you might see it used this way.

More likely, though you will see JSX written in the return statement of a component like so:

```
const Heading = () => (
  <h1>Heading</h1>
)
```

What is nice about this approach is that if a function or class returns valid JSX, we can then call it as a JSX element like so:

```
const Heading = () => (
```

```
  <h1>Heading</h1>
)

const Post = () => (
  <div className="post">
    <Heading />
    <p>Post content here.</p>
  </div>
)
```

Notice that we can use the heading function as it's own element in JSX.

We are going to learn a lot more about the syntax and rules of JSX, but for now we want to remember that we will most commonly use it in the return statement of components and sometimes saved as variables.

Opening/Closing Tags and Self-Closing Tags

Just like with HTML and XML, we have two types of JSX tags. The first type of tag includes an opening and closing tag like this:

```
<tag>Some text</tag>
```

The other type of tag is self closing, like the following:

```
<tag property="value" />
```

Since JSX includes the basic HTML tags by default, you would likely understand the following JSX:

```
<div>
  <p>This is a paragraph<p>
  <img src="/path-to/image.png" />
</div>
```

The above example shows both opening/closing paired tags in action as well as a self closing tag for the img.

HTML Tags Are Lower Case

As we have seen, when we pass an HTML tag to JSX it will create the corresponding HTML tag using `React.createElement()`. When we write HTML in JSX we want to use lowercase letting like so:

```
<div>
  <h1>Heading Element</h1>
  <p>This is a paragraph<p>
</div>
```

This may seem intuitive, but it is important to point out this convention. This also distinguishes basic JSX tags from custom ones we setup on our own.

For example, we can see here the difference at a glance between default HTML JSX tags and custom component tags we have made.

```
<div>
  <Heading />
  <p>This is a paragraph<p>
</div>
```

These rules for capitalization are not necessarily required, but they are best practices and should basically be considered required syntax.

JSX Capitalization Rules Explained

When working with JSX we want to follow the following conventions:

1. Variable assignment should be lower case
2. Function returning valid JSX should be uppercase
3. Class returning valid JSX should be uppercase

In the example below we create our own capitalized JSX element by assigning it to a constant.

```
const welcome = <p>Welcome!</p>;
```

Then later in our code we could call our welcome message just like

a normal variable. The important thing to point out is that these are written lower case.

When we're working with components though we want to capitalize the name.

Here is something similar using a function:

```
const Welcome = () => {
  return <p>Welcome!</p>;
}
```

We could also write this using a class, which we will discuss later in the book

```
class Welcome extends React.Component {
  render() {
    return <p>Welcome!</p>;
  }
}
```

Whether we use a function or class to create our components we will always name them uppercase and call them using the JSX syntax like below:

```
<Welcome />
```

Otherwise, if you are using any of the default HTML tags, remember leave them lowercase.

Writing JavaScript Between Curly Braces {}

Since JSX processes most of what it receives through `React.createElement()` it is important to be able to interrupt that process if we want to run JavaScript (and not just write JSX tags).

As we will see, this happens quite a lot. A few examples include passing a variable into JSX, writing a short event handler, or even writing conditional logic. In each of these cases, we use curly braces {} to escape from the JSX and write plain old JavaScript or React code.

```
const Welcome = () => {
```

```
  const name = "Zac Gordon";
  return <p>Welcome {name}!</p>;
}
```

In the example above we are adding a variable named `name` into our JSX.

It is important to note in this example where the JSX begins and where it ends. The JSX does not actually start until we see the <p> in the return statement. Then it ends with the closing </p> tag. Before and after that we can write normal JavaScript. However, within those <p> tags we can only write more JSX tags, not normal JavaScript.

The curly braces tell our transpiler to process what is between the curly braces as normal JavaScript.

So in the example above we wrap our name variable within curly braces since we want that variable (normal JavaScript) to not be processed as JSX, but as JavaScript.

Here is another example of how we could combine Vanilla JavaScript within JSX.

```
const name = "Zac Gordon";
const heading = <h1>Welcome {name}!</h1>
const Welcome = () => {
  return (
    <div>
      {heading}
      <p>An additional welcome message</p>
    </div>
  );
}
```

The example above shows two instances of using curly braces to interrupt the processing of JSX in order to process vanilla JavaScript.

In the first instance we are using the `name` variable inside of the heading JSX. The JSX for the heading starts with the <h1> and ends with the </h1>. So if we want to reference normal JavaScript within that we need to use the curly braces.

The second instance is referencing our `heading` variable. Now you might think that the `heading` variable is JSX. That is not really

true. Technically it is a JavaScript variable that *contains* JSX. However, in order to reference `heading` we are calling normal JavaScript.

So within the Welcome component, when we want to call our `heading` we have to place it between curly braces since we are once again writing JavaScript within JSX tags (starting and ending with `<div></div>`).

Remember, whenever you have JSX tags and want to call a JavaScript variable or write some vanilla JavaScript, you have to escape it with curly braces.

However, we cannot really write *any* JavaScript. We can only write JavaScript expressions within curly braces.

Only JavaScript Expressions Can Go Between Curly Braces {}

As mentioned in the chapter on Important JavaScript to Know for React, an expression in JavaScript is something that returns a value or is a value.

JSX will only accept expressions between curly braces. That means we cannot write full statements as we might expect, only bits of JavaScript that return a value.

Here are a few common types of expressions you may use:

- Variable and object values
- Function calls and references
- Conditional expressions*

Here is an example with the first two types of expressions in action:

```
const site = "React Explained";
const user = {
  first: "Zac",
  last: "Gordon"
};

const getFullName = user => {
 return `${user.first} ${user.last}`;
}

const Welcome = () = {
```

```
  return (
    <h1>
      Hi {getFullName(user)}! Welcome to {site}
    </h1>
  );
}
```

While we would not likely see an example exactly like this in production, it does demonstrate how you can use function calls, objects and strings within curly braces inside JSX.

Conditional Expressions in JSX Explained

Because we can only pass JavaScript expressions into curly braces, it is important to point out the only type of conditional statements we can write inside of JSX are conditional expressions.

Traditionally when we write a conditional statement we might do something like this:

```
const Welcome = props => {
  const isLoggedIn = true;
  if (isLoggedIn) {
    return <p>Welcome!</p>;
  }
  return <p>Please login!</p>;
}
```

This is not using conditional expressions, but it works because it does not appear *within* our JSX. Rather, our JSX appears as an expression within our normal JavaScript.

However, if we tried to do the following, where we place our normal conditional statement within our JSX it will **not** work.

```
const Welcome = () => {
  const isLoggedIn = true;
  return (
    <div>
      {if (isLoggedIn) {
        <p>Welcome!</p>
```

```
      } else {
        <p>Please login!</p>
      }}
    </div>
  );
}
```

This will not work for a few reasons. First, the JavaScript inside of the curly braces is not an expression, but a statement. Once we start writing our JSX (with the first opening <div> tag) we can only include expressions within the curly braces. Second, we are trying to include JSX inside of our curly braces, which won't work either.

This is actually a common stumbling block when starting to work with conditionals and JSX. The most common work around involves using a conditional ternary operator, like so:

```
const welcomeMessage = isLoggedIn ? (
                         <p>Welcome!</p>
                       ) : (
                         <p>Please login!</p>
                       );
```

Note this is usually written in one line, but broken up here into multiple lines to fit on the page.

This checks if `isLoggedIn` is true or false and assigns the value of the code in the parenthesis after the question mark if it is true. If the check returns false, the code in the parenthesis after the colon will be returned. Hopefully you notice, this conditional test returns a value and is therefor an expression, which can be used within curly braces in JSX.

This allows us to rewrite our conditional statement in the following way:

```
const Welcome = () => {
  const isLoggedIn = true;
  return (
    <div>
      { isLoggedIn ? (
        <p>Welcome!</p>
```

```
      ) : (
        <p>Please login!</p>
      )}
    </div>
  );
}
```

We see that `isLogged` in is checked and the proper JSX is returned. However, rather than be assigned to a variable, the value is rendered along with anything else in the return statement.

It is also common to see the same conditional pattern used, but abstract the conditional expression out of the return statement and assign it to a variable like so:

```
const Welcome = () => {
  const isLoggedIn = true;
  const welcomeMessage = isLoggedIn ? (
    <p>Welcome!</p>
  ) : (
    <p>Please login!</p>
  );
  return (
    <div>
      {welcomeMessage}
    </div>
  );
}
```

Some developers find this pattern cleaner and preferred. Whatever approach you decide to take, remember that anything that appears within JSX must evaluate to an expression, and this includes our conditional checks.

Conditional Checks with && (AND) Explained

While the above mentioned approached to conditional statements where an if and else condition applies, sometimes we do not have an else statement. In these cases we can rely on an interesting pattern

in JavaScript where any expression always returns to true (unless it explicitly returns a falsy value).

This allows us to do something like this:

```
const Welcome = () => {
  const isLoggedIn = true;
  return (
    <div>
      { isLoggedIn && ( <p>Welcome!</p> )}
    </div>
  );
}
```

In the example above, `isLoggedIn` returns as true. The double ampersands checks to make sure that both the first and second conditional check are both true. Any JSX we place inside of parenthesis should return true as well. Therefore, since both tests pass, the value of the expression in curly braces is returned and displays. If `isLoggedIn` returns false, the expression with JSX will still return true, but the overall conditional check fails due to the double ampersand requiring both checks to be true.

Again, this is a nice solution when you have a conditional check to do that only requires an if and no else. Place the value you want to check first, use double ampersands then after that, place the expression you want to execute inside parenthesis.

Mapping Over Arrays with JSX Explained

Standard for loops are not expressions and therefore cannot be used within our JSX. Most JavaScript developers also prefer to map over arrays rather than run for loops as discussed in the earlier chapter on JavaScript.

The nice thing about maps in JavaScript is that they are expressions, so we can use them within our JSX.

Let's imagine that we want to map over an array of posts and display the title for each one. We will assume for sake of simplicity that all of the posts we need are being passed as part of the props parameter, which we will get into more later.

```
const ListPosts = props => {
  const posts = props.posts;
  return (
    <ul>
      {posts.map( post => <li>{post.title}</li>)}
    </ul>
  );
}
```

While the above code will work as expected, best practices suggest that when you need to map over content is should be pulled out of the return statement and into a variable or it's function or class. In the example below we clean up our code a little bit by pulling out the map into it's own value.

```
cont ListPosts = props => {
  const posts = props.posts.map( (post) =>
    <li>{post.title}</li> );
  return (
    <ul>
      {posts}
    </ul>
  );
}
```

In the future chapters we will look more at component architecture and best practices on breaking out code that does multiple things into smaller modules. We will also discuss props in depth in their own chapter.

JSX Keys and Lists Explained

Due to how React intelligently renders and re-renders nodes in the DOM, we need to place "keys" in all list items that we create with JSX.

```
const ListPosts = props => {
  const posts = props.posts.map( post => {
    return (
      <li key={post.id.toString()}>
```

```
        {post.title}
      </li>
    )
  });
  return (
    <ul>
      {posts}
    </ul>
  );
}
```

You can see here that for each list item we have assigned it an attribute of "key" with a value of the post ID.

This allows React to keep track of items within a list and only update certain items when needed, rather than the entire list of items each time. The ability to do this is one of the primary benefits of a library like React. However, in order for this to work, we need to add keys to list items.

It is possible to use an index if no unique ID is available, however, this has serious performance drawbacks so the following approach should not be taken:

```
const ListPosts = props => {
  const posts = props.posts.map( (post, index) => {
    return (
      <li key={index}>
        {post.title}
      </li>
    )
  });
  return (
    <ul>
      {posts}
    </ul>
  );
}
```

Rather than taking this approach with index, I would suggest you

add unique identifiers to your data, possibly using symbols or another approach that makes sense based on your data.

Fragments with JSX Explained

We learned in the previous chapter the React gives us a special `React.Fragment()` element we can use when we want a component to return multiple elements.

With JSX we get a few shorter ways of using Fragments.

```
const App = () => {
  return (
    <React.Fragment>
      <Header />
      <Content />
      <Footer />
    </React.Fragment>
  );
}
```

In the example above we are using the longest method of writing fragments. This could be shortened a bit like this:

```
const App = () => (
  <>
    <Header />
    <Content />
    <Footer />
  </>
);
```

However, it is important to note that some tools do not yet support this syntax so you may want to use the <React.Fragment> instead if the <> shorthand format is not supported.

One final note should be mentioned that it is possible to add keys to React Fragments if necessary. The React docs give the following example:

```
const Glossary = props => (
```

```
  <dl>
    {props.items.map(item => (
      // Without the `key`
      // React will fire a key warning
      <React.Fragment key={item.id}>
        <dt>{item.term}</dt>
        <dd>{item.description}</dd>
      </React.Fragment>
    ))}
  </dl>
)
```

The example above can be helpful because it allows React to update just a single instance of the definition list without updating the entire thing. We will see over time how this offers great performance benefits.

Let's Practice!

Now that we have covered a number of the important rules for working with JSX, let's practice writing some. This will get you comfortable with writing JSX and help you walk through some of the most important concepts and rules for how it work.

9

5 Exercises in Writing React With JSX

As we will see there are more rules to JSX, but the above gives us a solid introduction. Now let's do a little practice to solidify what we have learned.

You should already have the practice exercises, but you can download them here if you do not already have them: https://github.com/zgordon/react-book.

The exercises for this chapter are under "chpt-5-jsx." Just like with the last chapter you will find blank starter files with comments of what to do as well as completed files with working code. All you have to do to test is open the index.html file in your browser.

If you completed the practice exercises from the last chapter you will find these very familiar. In fact, they are the exact same exercises as before, but this time we will complete them using JSX rather than `React.createElement()` directly.

Practice Exercise #1

The first exercise involves create a simple paragraph element using JSX. The paragraph element should not have any special classes or attributes and some simple text like "Hello React."

You should save this element as a variable name `pEL` using `const`. To test, call `pEL` the bottom of the exercises where ReactDOM.render() is called.

You final markup should look like this:

```
<p>Hello React.</p>
```

Practice Exercise #2

For the second exercise we will practice nesting elements. Create a `const` named `h1LinkEl`. Give it the value of an h1 element with a class of "entry-header." Inside of the h1, create an anchor element with a link to the React website and the text "React."

Like Example #1 above you will have to add your element variable to the ReactDOM.render() call in order to test it.

Your final markup should look like this:

<h1>React</h1>

Practice Exercise #3

In this exercise we will create a component rather than a single element. The component is called `Header` and it should return a `header` element with an ID of "main." Inside of the header element you should pass in the paragraph element and the h1 element you created from Exercises #1 and #2.

To test you will add `<Header />` to ReactDOM.render(). Your final markup should look like this:

```
<header id="main">
  <h1><a href="http://reactjs.org/">React</a></h1>
  <p>Hello React.</p>
</header>
```

Practice Exercise #4

From here we continue with another component example. This exercise has you creating a component called `List` that returns an unordered list with three list items within it. Each list item should be a link to a React resource. The `ul` element should also include both a custom class and ID attribute.

When you call `<List />` in `ReactDOM.render()` it should return markup like this:

```
<ul class="react-links" id="top">
  <li>
    <a href="http://reactjs.org/docs">React Docs</a>
```

```
  </li>
  <li>
    <a href="https://reactjs.org/docs/react-dom.html">
      ReactDOM Docs
    </a>
  </li>
  <li>
    <a href="http://reactexplained.com/">
      React Explained Book
    </a>
  </li>
</ul>
```

For bonus points, make up a unique key for each list item.

Practice Exercise #5

In our final exercise we create a component called `App` that returns a Fragment with our `Header` and `List` components within it. This will give us practice using `React.Fragment` as well as creating components that return other components that in turn return individual elements. This is a fairly common practice in React.

The final markup for this will look something like this:

```
<header id="main">
  <h1><a href="http://reactjs.org/">React</a></h1>
  <p>Hello React.</p>
</header>
<ul class="react-links" id="top">
  <li>
    <a href="http://reactjs.org/docs">React Docs</a>
  </li>
  <li>
    <a href="https://reactjs.org/docs/react-dom.html">
      ReactDOM Docs
    </a>
  </li>
  <li>
    <a href="http://reactexplained.com/">
```

```
      React Explained Book
    </a>
  </li>
</ul>
```

What's Next?

After you complete the exercises above you should feel comfortable creating basic elements and components with JSX. You should also understand that behind the scenes, our JSX "markup" is being passed to `React.createElement()`.

I would encourage you to try creating some of your own elements and components as well as additional practice.

There are more rules to writing JSX, but we need to learn about some more important React features before they will make much sense. However, before we start digging deeper into React we have to get a better development setup.

Up to this point we have been linking to React, ReactDOM and Babel from script tags in our index.html file. This is fine for practice, but it is not ideal for development.

Our next step from here is to learn how to use the Create React App tool. This will give us a better integrated development environment for working with React.

10

An Introduction to Creating React Apps

Up until this point we have been loading the React and ReactDOM (as well as Babel) libraries via a script tag in our HTML file. While this works fairly well for learning and playing around when we're getting started, it has several limitations. The most important is that we cannot use imports to reference other libraries or easily break up our code into small modules and have them easily tracked. There are more as well but these alone are worth moving from the approach we have to using more powerful tools.

The typical React work flow involves using scripts for bundling, transpiling, linting, testing, running a development server and often more. This can be a lot to setup, keep updated, and modify manually to suite your exact needs.

Create React App gives us all of these scripts together in one tool. By default it will also hide away the configuration files and settings for these various scripts, giving us a simpler file hierarchy and cleaner work environment.

It also provides the ability to transition away from the simple, default interface and expose all of the underlying scripts and their configurations if developers need to work with setting different from what Create React App offers.

However, in many cases, especially for us working through this book, Create React App will meet our needs for developing React apps. So, let's take a look at what it specifically does and how we can use it.

What Create React App Includes

Bundled with Create React App, we get the following:

- webpack – A bundler
- webpack Dev Server – The webpack Node development server
- Babel – A transpiler
- Several polyfills
- ESLint – A linter
- Jest – The React testing library

If you'd like some more information on any of these, please refer back to the chapter on Important Tools for React.

There are also some alternative Create React App version that include other tools. Create React App Typescript includes Typescript and Create React App Parcel uses Parcel instead of webpack, for example. In this book we will use the original Create React App and the tools it offers.

As mentioned, all of these tools and their configurations are hidden out of the way, so when we first setup and use Create React App we will not see the configuration files for these tools.

Setting Up Create React App

In order to use Create React App, we first need to install it. To do that we want to have the latest versions of Node and NPM. It is a little outside the scope of this book to walk through the installation of Node and NPM, so please make sure that you have the latest version of these tools installed and running on your computer before proceeding.

To create a React app (or site) with Create React App we would run the following command in the command line:

```
npx create-react-app my-project
```

This will run the Create React App code and create a new directory called `project-name` in the directory where we originally ran the command.

So we may be in a folder called projects, run the command above and then end up with the following basic hierarchy.

```
|-- projects/
|   |-- my-project/
```

```
|   |   |-- node_modules/
|   |   |-- public/
|   |   |-- src/
|   |   |-- .gitignore
|   |   |-- package.json
|   |   |-- README.md
|   |   |-- yarn.lock
```

Let's break down the purpose of each of these folders and files that Create React App gives us out of the box.

Create React App Files and Directories Out of the Box

In the previous section we listed out the various directories and root level files that Create React App generates. Now we are going to go into more depth with each of them.

node_modules

The `node_modules` folder is where NPM will save all dependent files and libraries for our project. If you look inside the folder it will contain *a lot* of other directories that contain different that Create React App depends on to work.

The reason we see so many packages or directories is that often time a single library will require several other libraries as dependencies so we are seeing a flat listing of all of the large and smaller libraries here not just for React itself but also for the different tools that Create React App includes.

Later we will install our own packages and both those packages and any packages they depend on will be stored in the node_modules folder.

When Create React App (using webpack behind the scenes) bundles our code together it will bundle together all of the necessary packages from the `node_modules` folder as well. For this reason, we only need the `node_modules` folder in our local development environment. When we deploy to our production environment, we do not want to include the `node_modules` folder.

public

The public directory is where all of our public facing, non JavaScript code will go, particularly the main `index.html` file.

If we look in public to start we will see the following:

```
|-- public/
|   |-- favicon.ico
|   |-- index.html
|   |-- manifest.json
```

The `favicon.ico` and `index.html` should make sense. The `favicon.ico` is the React logo, which you can swap out with your own favicon.ico file.

The `index.html` is not something you will generally need to edit because most of your UI will be generated from React, not from hard coded HTML.

The `manifest.json` file, called a Web App Manifest, and it provides meta information about our site our app. It is most commonly used for devices to offer an “Add to Home Screen” option that will load an icon for our site to our device home screen and allow it to know information about our site, like what URL to open, what icon and name to use and other information.

It is a good idea to modify the `short_name`, `name`, `theme_color` and `background_color` to suite your project needs.

As mentioned, as we build our project and add more React code, we will see additional files added here automatically during the build process.

src

The `src` directory will contain all of our pre-bundled React code. This is our working development directory. Unlike our `public` directory, the `src` directory will not be shipped to production.

In order for another developer to make changes to your React code they will need access to this src directory. For this reason, you will often see this src folder included in version control.

```
|-- src/
```

```
|   |-- App.css
|   |-- App.js
|   |-- App.test.js
|   |-- index.css
|   |-- index.js
|   |-- logo.svg
|   |-- registerServiceWorker.js
```

If we look at the file architecture we will see that we have a few files designated to an `App` component. This includes a CSS file, a JavaScript file and another JS file holding the tests for our component.

The code within the `App.js` file should look something like this:

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img
            src={logo}
            className="App-logo"
            alt="logo" />
          <h1 className="App-title">
            Welcome to React
          </h1>
        </header>
        <p className="App-intro">
          To get started, edit
          <code>src/App.js</code> and save to
          reload.
        </p>
      </div>
    );
  }
}
```

```
export default App;
```

We haven't looked at working with using classes for creating components yet, but everything else should look pretty straight forward. We import React, an image and some CSS. Then we make a component with a header and welcome text.

Finally the component is exported.

We will look at this code more later as we start building with Create React App, so let's turn our attention next to the `index.js` file. This is our entry point for our app and where Create React App will start looking to for import statements to pull together all of our code and bundle into compiled code it sends to the `public` directory.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker'

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
registerServiceWorker();
```

If we look at the code in our `index.js` file we will see it imports React and ReactDOM, which makes sense. Then it imports some CSS and our `App` component.

Then we see something new, we import `registerServiceWorker`. We have not discussed service workers yet in this book. They are a Web API that leverages caches to allow for sites to work offline or faster on slow connections. With Create React App we get service workers out of the box without any configuration.

You can see that at the bottom of this file the `registerServiceWorker()` function is called to kick things off. While we're starting off we don't need to worry about this and will just let it run in the background.

The other thing we see in this file is `ReactDOM.render()` loading our `<App />` component to the div with an ID of "root" in our /src/ index.html file.

This `index.js` file represents a common pattern for React apps. It pulls in a single component that will in turn call and handle all other components. It also loads any high level functionality like service workers. Later we will see things like routing handled at this high level `index.js` file too.

.gitignore

The `.gitignore` file is important for React apps since a number of the files and directories should not be version controlled, particularly the node_modules folder. Any directories or files listed in this file will be excluded from being pushed to Git or any other version control that recognizes `.gitignore` file.

package.json

This is one of the most important files for a React app that involves multiple developers or is shared. It contains a number of important things, but two are most important.

The first is a list of all packages needed for development. Since the `node_modules` directory is not included in version control, the package.json will contain a list of all of these packages. One of the first steps when working with someone else's (or a shared) React app is to run `npm install`. This will look through the `package.json` file and download all the necessary packages to a `node_modules` folder.

To start, Create React App only has three dependencies listed here: React, ReactDOM, and React Scripts. As we build our apps we will install more dependencies and they will be automatically added to this list.

The other thing this file contains are NPM scripts you can run with your app. Commonly we have npm run dev and npm run build or similar commands that will handle different ways of bundling or watching your code for changes and managing servers. Create React App has it's own set of commands it gets from React Scripts:

- `npm start` – Starts our development server
- `npm build` – Builds a production build of our app to public
- `npm test` – Runs any tests we have setup
- `npm eject` – Will extract out all the hidden configurations and stop using Create React App (not reversible)

You might want to change the `name` or `version` of your app here, but otherwise you should not need to modify this file.

README.md

Another standard file here. To start, Create React App gives you a generic read me file with a list of the file hierarchy and available scripts.

It is likely you will want to rewrite this for your own needs. I would not suggest shipping your app without updating this file.

yarn.lock

Yarn is a tool similar to NPM from the folks at Facebook who built React. In this book we will primarily use NPM, but many developers prefer yarn. The yarn files are very similar to the package.json files that NPM uses.

Using Create React App for Development and Production

Now that we have Create React App setup and understand it's various parts, let's look a bit more at the common commands we will use with it.

From within the project directory run the following:

```
npm start
```

This will spin up a webpack development server and watch your files for any changes. It will also open `http://localhost:3000/` in your browser since that is the default URL and port that Create React App uses.

Now, if you make any changes to your src code, for example in the `App.js` or `App.css` file, those changes will be detected and your

app will be re-bundled. The webpage showing your app will also be automatically reloaded.

Before you start working on a React App with Create React App you always want to run `npm start` so your changes are detected. However, this command is meant for development, not for production.

When you are ready to bundle your app for production you will want to run the following:

```
npm run build
```

This will create a new `build` directory that includes everything from the `public` directory as well as an assets folder containing the CSS and JS bundled from your `src` folder.

This build folder is the folder you would ship to production.

So, when you need to work on developing and building your React app or site, make sure that you run `npm start` first. Then when you are complete with a sprint or the entire project, run `npm run build` to get a final bundled version of your app.

Using Create React App on an Existing Project

So far we have been approaching Create React App as if you were starting a React project yourself from scratch. However, there are instances where you want to fork or work on someone else's project who started it with Create React App.

If this is the case you should know they are using Create React App in one of two ways. First, they will hopefully tell you in the README.md file of the project. Second, you can open the package.json file and see if they have a setup like the following:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject"
}
```

If the following commands exist in the package.json it is more than

likely they are using Create React App. So, everything we have said so far applies and everything we will continue to cover in this chapter applies.
However, before getting rolling you have to run `npm install`. This will pull down all the dependencies you need to your node_modules folder. Since Create React App does this for you when you first create a project with it, you will have to do this step manually when working with a project that has already been created.

Running Tests with Create React App

Testing with JavaScript and React is a little outside the scope of this book. However, Create React App does ship with the Jest testing library, the preferred testing library for React apps.

If you ever need to run any of your tests, `npm run test` will kick off that process for you. We are not going to explore those options in this book, but once you are comfortable with testing, it should all make sense and be a helpful integration.

Ejecting from Create React App

Create React App offers a number of built in tools behind the scenes as we have seen. With some projects you may want to customize the configuration files for the built in tools.

When this happens for a project, you will need to run a one time `npm run eject` command that migrates all of the configuration files from being hidden and makes them all available for you to edit.

You cannot undo the `npm run eject` process.

The only reason to run eject is if you know the tool want to customize and feel comfortable making and maintaining changes to it along with the other tools. We are not going to look into running eject in this book as it will not be necessary for our needs and for likely most of your React projects.

Remember, you do not have to eject from Create React App to build or launch your app, just if you want to customize the underlying tools settings in a way that is not possible.

How You Will Likely Use Create React App

In most cases when you start building a new React project you will run `npx create-react-app project-name` to kick off the project.

After that first time though, `npm start` will be the command you run from the project directory to start watching your files for changes and starting up the development server.

When your project reaches points where you are ready to ship to staging or production, you will run `npm run build` and send the `build` directory where it needs to go.

If testing is part of your workflow you will likely run `npm run test` regularly. Calling `npm run eject`, however, should rarely be necessary and a command that means the end to working with Create React App on that project.

On the chance that you start working on a project someone else has started with Create React App, you will need to run `npm install` the first time before calling `npm start`.

11

5 Exercises in Creating a React App

Now that we have a basic understanding of what Create React Does and how to use it, let's do some practice.

For these practice exercises you will need a generic `practice` folder. Then for each exercise you will spin up a new Create React App instance from that `practice` folder.

Practice Exercise #1

Inside of a practice folder call `npx create-react-app exercise-1` from the command line.

Then navigate into the new folder using `cd exercise-1` and run `ls`.

You should see the list of default React files outlined in section, "Setting Up Create React App," above.

This exercise will help you establish comfort setting up a new project with Create React App and moving into it with the command line.

Practice Exercise #2

Inside of a practice folder call `npx create-react-app exercise-2` from the command line.

Open up the `exercise-2` directory in your code editor.

Run the command `npm start` from inside the `exercise-2` directory. Open the URL it gives you for the development server in your browser. To stop the development server, type `Ctrl + C` in the command line.

Then in your code editor, change the text of the `p` tag in the `/src/App.js` file from "Edit <code>src/App.js</code> and save to reload" to

something else. On save you should see the browser refresh with your new value.

This exercise will help you gain confidence starting the Create React App development server and seeing the changes to your code reflected in the browser.

Practice Exercise #3

Inside of a practice folder call `npx create-react-app exercise-3` from the command line.

Open up the `exercise-3` directory in your code editor.

Run the command `npm start` from inside the `exercise-3` directory. Open the URL it gives you for the development server in your browser. To stop the development server, type `Ctrl + C` in the command line.

Then in your `src` directory, add a new file named `Hello.js` with the following code:

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

class Hello extends Component {
  render() {
    return <p className="Hello">Hello!</p>;
  }
}
export default Hello;
```

Then open your `src/index.js` file and change the references to App on line 4 and <App /> in line 7 to the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import Hello from './Hello';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <Hello />,
```

```
  document.getElementById('root')
);
```

When you save the changes to the `src/index.js` file you should see the changes reflected in the browser.

This exercise will help reinforce the skill of setting up React apps, while also getting you comfortable adding new component files to an app and having them show up working in the browser.

Practice Exercise #4

Inside of a practice folder call `npx create-react-app exercise-4` from the command line.

Open up the `exercise-4` directory in your code editor. Then change the text in the `p` tag in the `/src/App.js` file from "Edit <code>src/App.js</code> and save to reload" to something else.

Then run npm run build in your project directory. It should create a build folder.

As suggested, then run the following two commands:

```
yarn global add serve
server -s build
```

This should in turn give you a link to open up the built version of the site on a server of it's own, different from the development server.

If you are able to complete this exercise you should feel more comfortable running the build process for getting your app ready for production.

You can now also preview this built version using it's own little server, although doing that is completely optional.

Practice Exercise #5

Inside of a practice folder call `npx create-react-app exercise-5` from the command line.

Then `cd` into the `exercise-5` directory from the command line.

Run the following command to eject your configuration file settings and leave Create React App:

```
npm run eject
```

You will have to confirm “y” that you want to eject from Create React App. *If you are using git, you will have to make sure you have no untracked changes before doing this.*

Once the eject command has run you should see a `config` and `scripts` directory in your `exercise-5` folder.

Now run `npm run start` just to make sure everything is still working. You should see the development server start as done in Practice #2. To stop the server, type `Ctrl + C` in the command line.

There is no undoing the eject command so only do it when you know what you are doing or have support from someone who does. However, practicing this exercise on a practice project is a good idea to see how it works without breaking an existing project.

Next Steps

For the rest of the book we will be using Create React App to spin up new little React projects. If you would like to run through the exercises above a few additional times it can be good practice.

Create React App is a great call for starting React projects. It should work as a solution for most of your projects.

On the chance that you are working with a team or project that requires a different setup than what Create React App provides, you can still expect to have basic commands like `npm start` or `npm run build` or some equivalent available to call.

12

Props in React Explained

As we have learned, React is a user interface library.

We have `React.createElement` at the heart of creating HTML interfaces on the fly with JavaScript. Then JSX serves as a syntactical extension to make calling createElement look more like writing HTML.

Up to this point we have created and nested components, but we have not looked at how data is passed between and shared between components. Without understanding this important aspect of React we would not be able to build very interesting or dynamic interfaces.

Props are the most basic (and common) way to pass data between components.

Props is short for properties. React elements are technically objects and can therefor have properties attached to them. These properties contain data that is made available to use in that given component.

Basic Props Syntax

Let's take a look at a basic example of props in action and then break down some important aspects.

```
// Display a user profile
function Profile(){
  function getCurrentUser() {
    // Do things to get the user;
    return user;
  }
  return (
    <div className="profile">
      <Avatar user={getCurrentUser()} />
```

```
      <UserName user={getCurrentUser()} />
    </div>
  );
}

// Display a user Avatar img
function Avatar( props ) {
  return (
    <a href={props.user.url}>
      <img
        className="avatar"
        src={props.user.avatarURL}
        alt="Avatar for {props.user.fullName}"
      />
    </a>
  );
}

// Display the user name in a link
function UserName( props ) {
  return(
      <a href={props.user.url}>
        {props.user.fullName}
      </a>
    </div>
  );
}
```

Let's start in the `<Profile />` component. We have a function here that we are pretending get's the current logged in user for an app (although we haven't actually written the function). This is a fairly common example of a function that would exist in an app.

However, the problem we have is that we need that user information available to both the `<Avatar />` and `<UserName />` components. To pass this data using JSX we will add what look like HTML attributes to our components. In this example we are adding a `user` prop or property to our `<Avatar />` and `<UserName />` components.

Then when we look at our `Avatar()` and `UserName()` functions

we will see that we are adding a parameter called `props`. The `props` parameter is actually available to all custom components we create. We do not see it with our `Profile()` component because we are not leveraging it, but you will often seen it included by default in all components created with functions just in case.

Inside of our `Avatar()` and `UserName()` components we that `props.user` is now something we have access to. We now have a working example of taking data from one component and passing it into child component.

There is more we have to learn about using components, but this example serves to show us the very basics. What we have to emphasize before proceeding to learn more about props is that data only moves in a one way direction in data props, and that is from parent children down to children.

One Way Data Flow Through Components

Historically, JavaScript applications have had a two way flow of data. This makes more sense with an MVC or MV* architecture where different views or controllers had to have a live record of data in other views or controllers.

React has a different model. React does not have two way data binding. It has a one way flow of data.

Data in React apps only flow down through an app, from parent components to children components. If you have worked with two way data binding or MC* architecture in the past this can take a while to get used to. However, the React one way data flow model is actually beautifully elegant and removes much of the complexity and places where something can go wrong with two way data binding.

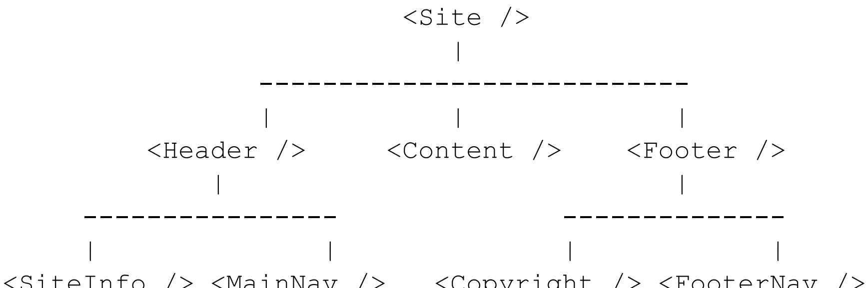
What this means at a practical level is that once data is passed down from a parent to a child, that prop value should not be changed. If prop values need to be changed, there is a model for doing that we will explore in the next chapter using state. This reinforces the practice of immutability for all data passed through our app as props.

Setting Props at the Highest Component Level Necessary

Since we cannot pass data up the component hierarchy, from children

to parent components, we want to start passing props at a level in the component hierarchy where that data will reach all necessary children components that need the data.

Let's take a look at a simple site built using React as an example



Let's imagine now that the <SiteInfo /> component and the <Copyright /> component both need to have access to the site name.

To get the site name we might write a function called getSiteName() that would pull the site name from wherever it is stored. However, the question arises of where should we write that function?

We could write the function twice, once in the <SiteInfo /> component and again in the <Copyright /> component, but obviously duplicating our code is a bad idea. We could also write a helper library and import that into our React code and call the function in each of the <SiteInfo /> and <Copyright /> components. While possible, this is not the normal practice in React as we are still duplicating that function call. Plus, in most cases, we want to keep all our necessary functions within actual components, not in separate libraries for easier tracking and troubleshooting.

So, the most common practice would be to write the getSiteName() function inside of the <Site /> component and then pass the data from that component down through the <Header /> and <Footer /> components as props and then finally into the <SiteInfo /> and <Copyright /> components. As we

can see, the `<Site />` component is the first parent component that shares both `<SiteInfo />` and `<Copyright />` as children.

```
// Setup the main site component
function Site() {
  function getSiteName() {
    // Do something to get siteName
    return siteName;
  }
  return (
    <Fragment>
      <Header siteName={getSiteName()} />
      <Content />
      <Footer siteName={getSiteName()} />
    </Fragment>
  );
}

// Display the Header component
function Header( props ) {
  return(
    <header>
      <SiteInfo siteName={props.siteName} />
      <MainNav />
    </header>
  );
}

// Display the SiteInfo component
function SiteInfo( props ){
  return (
    <div className="site-info">
      {props.siteInfo}
    </div>
  );
}
```

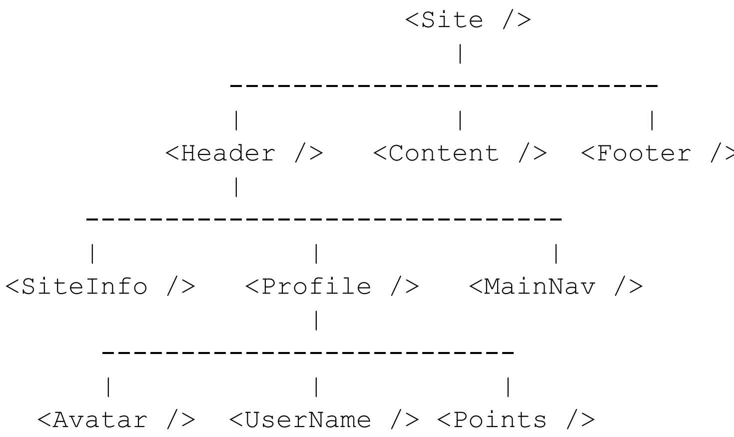
In the example above you can see how props would be passed from the `<Site />` component down through `<Header />` and finally used

for `<SiteInfo />`. The same would also be done for `<Footer />` and `<Copyright />`.

We also see here "Setting Props at the Highest Component Necessary." We have placed the function call for the User object high enough in our component hierarchy that all necessary child components have access to the data via props.

However, *this does not mean* that we should "Set Props at the Highest Component Possible."

Let's take a look at another example to reinforce the difference between the "highest component necessary" and "highest component possible."



In the example above, we're just looking at a part of the component hierarchy as `<Content />`, `<Footer />`, `<SiteInfo />` and `<MainNav />` would all likely have children components as well. We are focused for this example on the `<Profile />` component and it's children.

Let's imagine now that `<Avatar />` and `<Username />` both needed access to a User object, but that User object was not needed anywhere else on the site.

What we would do in this case is write a function like `getCurrentUser()` inside of our `<Profile />` component.

Then we would pass the value returned from that down into our `<Avatar />` and `<Username />` components. This is exactly what we did in our example at the start of this chapter.

What is important to point out here though is that we would not add the `getCurrentUser()` function to the `<Site />` component just because it is the highest component possible where we could place it. We only need the User object available in `<Avatar />` and `<Username />` and we only have to go up to their common parent in order to achieve this. This leads to self contained components that contain all data that children components may need to share.

If down the road, however, the User object was also needed in the `<Footer />` components or one of it's children then we would need to move the `getCurrentUser()` function up into the `<Site />` component. It is important to know that while developing and extending a React app you may need to move around certain functions or original setting of props to higher components if where you need that information changes.

This process of setting props at the first shared parent component, or the highest component necessary, is a practice that will take a little while to figure out at first but eventually become second nature when building React apps. We will also do some practice with this at the end of the chapter.

Passing Props Down Through Multiple Components

In large applications it is not uncommon to pass props down through multiple layers of components. Sometimes components may not directly use the props they are passed, they just pass them on to children components.

For performance reasons we want to avoid passing props down through too many layers of components if it is not necessary. The specific reasons for this are a bit beyond our scope at this time, but it has to do with components being unnecessarily re-rendered or updated on the page if there are ever changes to the props that it passes.

We will look at some design patterns with React later on that help with this, but for now, we start off practicing how to pass props down at least one or two levels of children components.

Boolean Props Default to True

One important rule about props with boolean values that will allow you to write less code involves boolean props being set to true by default.

Imagine for example if we wanted to pass a prop of `loggedIn` into a component. Then we could conditionally load one of two different component based on whether or not a user was logged in or not.

```
function App() {
  return (
    <div className="App">
      <Header loggedIn={true} />
      <Content />
    </div>
  );
}
const Header = ({ loggedIn }) => (
  <div>{loggedIn ? <Profile /> : <LoginForm />}</div>;
);
```

This is pretty basic React. We can shorten it in one way if the props value for loggedIn is true. If any boolean prop has a default value of true we can simply leave off the prop value and React will set it to true for us.

```
function App() {
  return (
    <div className="App">
      <Header loggedIn />
      <Content />
    </div>
  );
}
const Header = ({ loggedIn }) => (
  <div>
    {loggedIn ? <Profile /> : <LoginForm />}
  </div>
);
```

Notice the only difference in the two versions of this code is that `<Header loggedIn={true} />` has been shortened to just `<Header loggedIn />`.

From now on, if you have a prop without a value know it will get set to true.

Passing Props with Spread

Since we have pointed out it is not always a performant option to pass props down multiple levels, there are some instances when you might want to pass all or most of the props from one component into a child. Using the JavaScript Spread operator can be helpful with this.

Let's imagine for demonstration purposes that we had a `<Profile />` component that recevied a bunch of props and had to pass them down into a generic child component called `<Card />`.

Normally we would start with this.

```
function App() {
  return(
    <div className="App">
      <Profile
        name="Zac Gordon"
        url="https://zacgordon.com"
        bio="Educator, Yogi, ...more"
      />
    </div>
  );
}

const Profile = props => {
  return(
    <div className="profile">
      <Card
        name={props.name}
        url={props.url}
        bio={props.bio}
      />
    </div>
  );
```

```
};

const Card = ({ name, url, bio }) => {
  return(
    <div className="card">
      <a href={url}>{name}</a> - {bio}
    </div>
  );
};
```

Notice the repetition of having to manually pass props into <Profile /> and then again pass them all into <Card /> as well. With the Spread operator we can shorten the <Profile /> component to just this.

```
const App = () {
  return(
    <div className="App">
      <Profile
        name="Zac Gordon"
        url="https://zacgordon.com"
        bio="Educator, Yogi, ...more"
      />
    </div>
  );
}

const Profile = props => {
  return(
    <div className="profile">
      <Card {...props} />
    </div>
  );
};

const Card = ({ name, url, bio }) => {
  return(
    <div className="card">
      <a href={url}>{name}</a> - {bio}
    </div>
```

```
  );
};
```

Notice that instead of listing each prop manually, we can write {...props} which will spread each prop into it's own prop to be passed into the component.

It is important to point out, **passing all props is generally considered a bad practice**, however, there are some times when it may be necessary and the spread operator can help us in those cases.

Destructuring Props

In addition to Spreading props, Desctructuring props can be helpful as well.

One technique involves destructuring props as they are passed into a component function. This involves identifying specific props that you will need, not just the generic props object that contains all properties.

So where we may have written this previously:

```
const User = props => {
  return (
    <p>
      <a href={props.userURL}>
        @{props.username}
      </a> - {props.name}
    </p>
  );
}
```

With destructuring we can write this:

```
const User = ({username, name, userURL}) => {
  return (
    <p>
      <a href={userURL}>
        @{username}
      </a> - {name}
    </p>
```

```
  );
}
```

We could also do the destructuring inside of the function if we want to keep access to `props`.

```
const User = props => {
  const {username, name, userURL} = props;
  return (
    <p>
      <a href={userURL}>
        @{username}
      </a> - {name}
    </p>
  );
}
```

This may seem simple, but it has a few advantages. First, it clearly identifies exactly what props you need. Second, it makes props easier to write (ie props.username vs username).

You do have to remember with this style to add any new props you will use to the destructuring in the parameters section.

If we only want to destructure some of the props we can do a combination of destructuring along with spreading the rest of the properties into a new property like this.

```
function User({name, ...rest}) {
  return (
    <p>
      <a href={rest.userURL}>
        {name}
      </a>
    </p>
  );
}
```

The benefit of this approach may not be apparent in this specific example, but if we needed to pass on some props and not others to children it can be helpful in that instance.

```
function User({name, ...rest}) {
  return (
    <p>
      {name}
      <Card {...rest} />
    </p>
  );
}
```

In the example above we need to use name in the User component, so we destructure that one prop. However, we need to pass the rest of the props down into a child component so we spread the rest of the props into a variable named rest.

While you might not resort to using these conventions on your own when just starting with React, they are patterns you will see, so it can be helpful to know about them in advanced.

Component Props vs. Element Attributes

One last important topic to mention about props is how they work with React Elements. Props with components, behave as props. However, with React Elements, “props” behave as HTML attributes.

In these instances, props that match HTML attributes will be automatically assigned as those attributes.

```
const Header = () => {
  return (
    <h1
      id="primary"
      className="primary"
      title="A title to apply"
      random="BLAH"
    >
      The Header
    </h1>
  );
};
```

In the example above we are adding what looks like Props to a React

Element. In this case, all of the "props" added are actually just converted into HTML attributes.

For example, the component above will produce the following in the DOM:

```
<h1 id="primary" class="primary"
  title="A title to apply" random="BLAH">
  The Header
</h1>
```

We can see that most of these are valid HTML attributes, so it works as we might expect. However, notice that "random" has also been added as an attribute even though it is not a valid HTML attribute.

React does not assess whether something added in this way is valid HTML, so it's important for us to remember when working with React Elements, that what looks like props are actually treated as attributes.

Updating The Value of Props

Hopefully in learning about props in React you have asked yourself the question, "What happens when I need to update the value of a prop?" Or the more complex, but also important question, "How can I update props set in a parent component from within a child component."

These are really important questions and part of the React workflow. However, in order to understand how this works, we have to introduce the topic of State in React. We will do this in the next chapter.

So for now, let's do some practice with setting a prop value that does not change and passing it down through components.

What's Next

Now that we have learned about how to set hardcoded values for props and pass them between components we have to look at how to update the value of props and pass dynamic data through components. This involves an introduction into State.

In the next chapter we will explore State in React, how it works, and how it can make our apps way more powerful than just using

Props alone. We will also look at how to write class based components (rather than functional components) to support the new features state brings.

First, let's do some practice with what we have learned about props.

13

5 Exercises in Working with Props

Now that we have learned a bit about Props in React, let's do some practice creating and passing props.

Practice #1

In this first practice we will practice adding props to a component. To get setup, open the `07-props/practicestarter` project and run `npm install` then `npm start`.

Then find the `<User />` component on `line 11` in the `src/Practice1.js` file.

Pass in `id` and `username` as props to the component. You should see in on line 16 that the <User /> component is already setup to use these props, they just need to be passed.

Once this is complete, you can open the `src/index.js` file and uncomment `line 4`:

```
import Practice1 from "./Practice1";
```

Finally, make sure that you call `<Practice1 />` on `line 13` in place of "Call Practice Component Here."

You should see the user ID and username render in the browser.

This practice exercise will help you get you comfortable with the first step of working with Props, passing them into a component.

Practice #2

In our second practice we will take a step further with our practice of adding props to a component. To get setup, open the `07-props/`

practice-starter project and run npm start if the server is not already running.

Open up src/Practice2.js in your code editor. Add two properties to the post object on line 7. One for id and one for title.

Then come down inside the Practice2() return and call the <Post /> component inside of the div being returned.

Pass in the post object we created as a prop into the <Post /> component.

Finally come down to where the <Post /> component is setup and modify it to receive props and return both the id and title props within the paragraph tag it returns.

Once you have all this working, open the src/index.js file and uncomment line 5.

```
import Practice2 from "./Practice2";
```

Finally, make sure that you call <Practice2 /> on line 14 in place of the <Placeholder /> or<Practice1 />component.

You should see the Post title and id you setup being rendered on the page. You can try modifying the original post object you setup to test that it all works properly.

This practice exercise is great to help you setup more of the parts of props on your own. We define the variables used as props, set the props on a component and then modify a component to work with props. This may we worth practicing a few times on your own.

Practice #3

Now that we have practiced a little with adding and using props, we will have you write a bit more of the code on your own as well as pass props down two levels of components rather than just one.

To get setup, open the 07-props/practice-starter project and run npm start if the server is not already running. Open up src/Practice3.js in your code editor.

Find the <Post /> component on line 11 and pass in the title and author as props.

Then setup Post on line 21 to receive props and pass `title` to `<Heading />` and `author` to `<Byline />`.

Finally, make a component Heading that accepts props and displays the `title` with <h1> tags. Also make a Byline component that returns a paragraph with the `author` displayed from props.

Once you have all this done, open the `src/index.js` file and uncomment the line

```
import Practice3 from "./Practice3";
```

Finally, make sure that you call `<Practice3 />` in `ReactDOM.render()`.

In the browser this should display the `title` and `author` on the page and give you experience passing props down through multiple levels of components.

Practice #4

In this practice we will work with spreading and destructuring props. This will expose you to some common patterns of working with React that you will no doubt see in other projects and likely use in your apps as well.

To get setup, open the `07-props/practice-starter` project and run `npm start` if the server is not already running. Open up `src/Practice4.js` in your code editor.

First, look for the `<User />` component called within Practice4 on line 18. Spread the user object into the `<User />` component so each user property becomes it's own prop.

Then, inside of the User on line 26, destructure `firstName` and `username` from the props and use them in the component where you see FIRSTNAME_HERE and USERNAME_HERE.

Of course you still have to wire up the `src/index.js` file and make sure you have `<Practice4 />` imported and `<Practice4 />` called in `ReactDOM.render()`.

If everything is setup correctly, you should see firstName and username work as variables inside of the User component.

Again, this practice is meant to help you get comfortable with spreading and destructuring with props.

Practice #5

In our final exercise we pull together what we have done in the examples above into one larger example.

Setup by opening the `07-props/practice-starter` project and make sure `npm start` is running. Then open `src/Practice5.js` in your code editor and make sure `<Practice5 />` is wired up in `src/index.js`.

Within `src/Practice5.js`, you first want to spread the user object into the <User /> component like we have done in the previous exercise.

Then follow the instructions for destructuring the user props in User and passing the proper props into `<FullName />`, `<Username />`, `<Social />` inside User on line 29.

To get `<FullName />`, `<Username />`, `<Social />` working, you will need to follow the instructions for each component to set it up, destructure the necessary props, and call the props as outlined in the comments for each component.

Once everything has been setup correctly, you will see the Full Name of the user displayed in an <h1>, the username in a <p> tag and finally a list of social links.

What's Next

Hopefully you had fun practicing with props. They can be a little tricky when starting, but will become second nature after too long. However, props are only part of the data flow architecture in React. Another essential part is state.

In the next chapter we will introduce state in React and learn how it can help us update our props and components with dynamic data.

14

State in React Explained

State is a generic term in JavaScript development that refers to keeping dynamic data that might change in an object named state. Then, whenever your JavaScript needs to get data, it does so via the state object.

Along with a state object there is usually a function (or set of functions) that let's you set or update values stored in the state object. Setting or updating a value in state then triggers a reaction where the new value of state gets immediately updated through your entire app.

Many JavaScript frameworks will have a global state object for the entire app. The popular library, Redux, handles offering a global state object along with ways to update state throughout your app and keep a record of changes.

By default in React, we do not have a global state object, but every component can have it's own state object. This means that each component has the option to set and manage it's own state.

If needed, a component can pass the value of it's state down to child components via props. If the value of state changes in that component the new value will automatically update any child component receiving the value as a prop.

The discussion of state can get a little complex, but the main takeaway is if a React Component has data that it needs to change, we will store that data in a state object and use a helper function to update the data.

State and Class Based Components

Up to this point we have used functions to create components in React.

An example of a functional component:

```
const MyApp = () => (
  <h1>Welcome</h1>
);
```

However, in order for state to work we will need to use classes for our components. *There is a new hooks feature coming to React that will allow functional components to receive and update state.*

Here is an example of a component created via a class setup to use state:

```
class MyAppClass extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "React"
    }
  }

  render() {
    return (
      <h1>`Welcome ${this.state.name}</h1>
    )
  }
}
```

Luckily we can use the simplified Class Fields syntax supported in Create React App to get rid of the constructor method:

```
class MyAppClass extends React.Component {
  state = {
    name: "React"
  }

  render() {
    return (
```

```
      <h1>`Welcome ${this.state.name}</h1>
    )
  }
}
```

Now, let's break down a bit what is happening here.

First, we are naming our component MyAppClass, which makes it available to be imported and called elsewhere as <MyAppClass />. This is the same as components built with functions.

We are also extending another class called React.Component. This is a class in React core that offers base functionality for building components with classes. All components you build with classes should extend React.Component.

Next we are setting up our state object using the fields syntax. State is an object. In this case state has one property, name.

Below that we have a `render()` method. All class based components must have a render() method in React that returns a React element.

This is similar to how all function based components have to return a React Element. However, with classes, it is a method named render() that always returns the React Element.

Of course, this element can be nested, like with functional components.

We also find within render() a call to this.state.name. This is the format for calling state in a class method.

Again, state is a container for any data that a component may need to change. So an example of a component with more state properties would look like this:

```
class Profile extends React.Component {
  state = {
    name: "React Explained",
    url: "http://reactexplained.com"
  }

  render() {
    return (
      <h1>
```

```
        <a href={this.state.url}>
          {this.state.name}
        </a>
      </h1>
    );
  }
}
```

Updating State with setState in a Class Based Component

In the example above we saw state in use in a static example where we were not changing state.

When we change state in React, we use a function called setState, which takes as a parameter an object with any properties in state that we want to change.

We can call setState via this.setState since it is inherited from React.Component.

```
class Name extends React.Component {
  state = {
    name: "React Explained",
  }

  render() {
    this.setState({name: "New Name"});
    return (
      <h1>{this.state.name}</h1>
    );
  }
}
```

In this example above we are calling this.setState() inside of render, right before returning the React Element. The object we pass identifies the property we want to update, name, and the new value, “New Name.”

We can also change multiple values of state at once if needed with one setState call.

```
class Name extends React.Component {
  state = {
    name: "React Explained",
    url: "https://reactexplained",
  }

  render() {
    this.setState({
      name: "New Name",
      url: "https://newurl.com",
    });
    return (
      <h1>
        <a href={this.state.url}>
          {this.state.name}
        </a>
      </h1>
    );
  }
}
```

As we can see, setState will take an object with any properties from state along with their new values.

Make sure that before you call setState on a property, that it has already been setup in the initial state object.

Updating State with Event Handlers

Most of the time we do not hard code setState() calls into our render() functions but rather handle them with event handlers.

Here is a simple example showing how we could attach an event handler to a button to update a value in state.

```
class Counter extends React.Component {
```

```
  state= {
    count: 0
  };

  handleClick = e => {
    e.preventDefault();
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return(
      <>
        <h1>{this.state.count}</h1>
        <button onClick={this.handleClick}>
          +
        </button>
      </>
    );
  }

}
```

In the example above we have a initial value for our count state of 0. Then we see an handleClick function that will serve as our event handler for increasing the value of count and calling setState.

We can see here we are calling setState and update the value of count to be the current state of count plus one.

We wire this function into our UI in render(). Look for the button element with a property of onClick set to this.handleClick.

React event handlers are usually attached by setting a property for an element such as onClick, onSubmit, onChange, etc, equal to a function.

It is also possible to write the event handler function inline when you set the onClick value.

```
class Counter extends React.Component {
  state= { count: 0 };
```

```
  render() {
    return(
      <>
        <h1>{this.state.count}</h1>
        <button
          onClick={e => {
            e.preventDefault();
            this.setState({ count: this.state.count + 1 }
          }}
        >
         +
        </button>
      </>
    );
  }
}
```

In the example above we use an anonymous arrow function to call setState inline. It can actually be condensed into just one line of code, but for readability in the book it has been broken down onto multiple lines.

When we want to update state via event handlers we will generally follow an approach like the simple example above.

We will see plenty of examples of event handlers updating state throughout this book. Let's take a look next at how we can pass the value of state down into child components using props.

State and Props

There are many instances when a component with state needs to pass the value of it's state down to a child component.

To do this we use the same props method we looked at in the previous chapter. The nice benefit is that whenever we call setState on a property in state it will cause the new value to automatically update in any child component referencing the value as a prop.

Here is our example from above broken into two components to demonstrate how this works.

```
const Name = props => <h1>{props.count}</h1>;

class Counter extends React.Component {
  state= { count: 0  };

  render() {
    return(
      <>
        <Name count={this.state.count} />
        <button
          onClick={e => {
            e.preventDefault();
            this.setState({ count: this.state.count + 1 });
          }}
        >
          +
        </button>
      </>
    );
  }
}
```

In this example above we have a new functional component named Name that just displays a prop called count. Note that if a component does not need state, we will continue to use functions to create them, not classes.

When we call <Name /> in render() we set the props of count equal to the current value of count in state. Then whenever the event listener is called and state gets updated, an update value also gets passed into <Name /> and immediately updated on the page.

The same would be true if we were to pass the value of count down into further child components. Whenever the original state gets updated, that change will be reflected anywhere down the line it is referenced as a prop.

Updating Parent State from Child Components

One of the problems that comes up is how to have a child component update the state value of a parent component. Since individual components all manage their own state it is not really possible for one state to call setState on another component.

The way around this involves taking the function that updates state and passing it down as a prop as well into child components.

This allows child components to update the state in parent components by calling a function from that parent component that was made available via state.

Here is an example of how that could look.

```
const Name = props => <h1>{props.count}</h1>;

const Button = props => (
  <button onClick={props.handleClick}>+</button>;
);

class Counter extends React.Component {
  state = { count: 0 };

  handleClick = e => {
    e.preventDefault();
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return (
      <>
        <Name count={this.state.count} />
        <Button handleClick={this.handleClick} />
      </>
    );
  }
}
```

In this example we have made our button into it's own component. However, the event handle it will call when clicked is passed down to it via props. So the button component has no idea what it is actually updated.

In the Counter class we have broken the event handler out into it's own function again.

Then when we call our Button component, we pass in our handleClick function. Interestingly, when that function gets called inside a child component, it will still execute in the context of the Counter class and update the counter state in the correct component.

Making State Persistent

State is a wonderful tool for managing data that changes. However, when we refresh the page our state get's reset to the default values.

So, there are times when we might want to make state persistent, or last between page refreshes or tabs closing.

A few options exist for doing this. Some of the most common are using local storage, session storage or a database. It is possible to write your own code to do this, but several packages and libraries exist that can help you easily do this.

In the case of local storage, the values of your state will be saved in local storage so if someone refreshes the page or even closes the tab and comes back, it will remember the last value of state your app had used.

With session storage the value of state is made persistent until the user closes the browser tab. At this point the current value of state is wiped and the default state values from your app are shown. Session storage can be helpful if you know you want to wipe state as soon as a user is done using your site.

Using a database, like the popular Firebase, allows you to store your state values outside of the user's browser for more reliability. It should be noted here the distinction between having a database store content for your site that you load via HTTP requests and keeping just the latest state of an application stored in state.

It is outside of the scope of this chapter to get into how to integrate

each of these approaches, but many tutorials and npm packages exist to help you easily implement these different solutions.

To briefly show an example of a simple implementation of making state persistent using local storage, we can install the following package:

npm install react-simple-storage

This will install a package that syncs our state into local storage and checks the local storage for our state on initial page load if it is available.

We can see this in action with our counter example before. All we need to do is add <SimpleStorage parent={this} /> to the top of our main app component and everything will work.

```
import React from "react";
import ReactDOM from "react-dom";
import SimpleStorage from "react-simple-storage";

const Name = props => <h1>{props.count}</h1>;
const Button = props => <button onClick={props.handleClic
class Counter extends React.Component {
  state = { count: 0 };

  handleClick = e => {
    e.preventDefault();
    this.setState({
      count: this.state.count + 1
    });
  };

  render() {
    return (
      <>
        <SimpleStorage parent={this} />
        <Name count={this.state.count} />
        <Button handleClick={this.handleClick} />
      </>
    );
  }
}
```

```
const rootElement = document.getElementById("root");
ReactDOM.render(<Counter />, rootElement);
```

Here we can see at the top of our example we import SimpleStorage from "react-simple-storage." Then in our main Counter component we call <SimpleStorage parent={this} />. This single component will kick everything off for us and make state map to local storage.

When we run this code above and refresh our browser we will see the value of count stays up to date.

If we had a main index.js file for our app or a main App.js component, we would likely put the <SimpleStorage parent={this} /> call within there.

As mentioned, there are several different approaches to making state persistent. It may be that your app doesn't need any of them, or you may decide to explore some of the packages and tutorials available to help you make state persistent.

Let's Practice

We have explored some of the main fundamentals of state so far, but let's start practicing with them a bit to really solidify how to work with state in a practical application.

15

5 Exercises in Working with State

Practice #1

In this exercise we will practice creating a property in state and rendering it on the page. To get setup, open the completed-starter project and run npm install then npm start.

Then open the src/Practice1.js file.

Before the render function create a state object with a property of username set to a common username you use.

Then in the render() function, replace USERNAME_HERE with value of username from state.

This practice exercise will help you create values in state and render them to the page.

Practice #2

In this practice exercise we will work with updating the state using setState and an event handler. To get setup, open the starter files and run npm start if the server is not already running.

Open index.js and make sure Practice2 is imported and called in ReactDOM.render(). Make sure npm start is running.

Next open Practice2.js.

After the state is setup and before the render function, create an arrow function called handleUsername that takes the event object as a parameter. It would look something like this to start:

```
handleUsername = e => {}
```

Have the function handleUsername set the new value of username

equal to the event target value (e.target.value). We will get this value from an input form field when we attach it as an event handler.

Then come down inside the render() function and set the onChange prop for the <input /> element equal to the handleUsername function we just created. Remember to call it using the this keyword.

Set the placeholder value of the <input /> equal to the username in state.

Once you have all this working, open the src/index.js file and uncomment line 5.

import Practice2 from "./Practice2";

Finally, make sure that you call <Practice2 /> on line 14 in place of the <Placeholder /> or<Practice1 />component.

This exercise gives you practice updating state using setState and event handlers. This is an essential part of working with state. This exercise also introduced how to use the onChange event with an input field.

Practice #3

Now that we have practiced creating and setting state we will practice how to update the value of state from a child component.

Open index.js and make sure Practice3 is imported and called in ReactDOM.render(). Make sure npm start is running.

Next open Practice3.js.

Update the UserForm component to accept props. Then update PROPS_ID, PROPS_LABEL and PROPS_ONCHANGE to get their values from props.

Next come down underneath the handleFirst function and create another one called handleLast that will control changing the value of last name in state.

Finally, come down into the render() method and call <UserForm />. Set the following props:

- id = "firstName"
- label = "First Name"
- onChange = handleFirst

Then call <UserForm /> again with the correct props for a Last Name form.

This exercise shows how to pass event handlers to update state down into children components. It also shows how to use a single component that can accept different event handlers for different functionality.

Practice #4

In this exercise we practice passing both the value of state and the handlers to update state down into children components. We also look again at how to create a simple component that can accept different event handlers to cause different interactions.

Open index.js and make sure Practice4 is imported and called in ReactDOM.render(). Make sure npm start is running.

Next open Practice4.js.

First, create a functional component called Header that accepts props. Have it display an <h2> with text from props.

Second, create a function component called Button that also accepts props. Have it return a <button>. Set the onClick value equal to onClick from props. Set the text for the button equal to text from props.

Next, inside Practice4(), create a state object with a count property set to 0.

Still inside Practice4(), create an increment function that sets the value of count in state to count plus one. Create a decrement function that decreases the value of count in state by one. Then create a reset function that updates the count state to 0.

Then in the render() return, call <Header /> and set the prop of text equal to the value of count from state.

Then call <Button /> three times. The first time set the onClick prop to decrement and the text prop equal to "–". On the next one, have onClick set to increment and "+" for the text. Finally, set the last <Button /> onClick set to reset and the button text say "Reset."

Once done, this should display a counter on the page with +, – and Reset buttons that will increase and decrease the value of the counter. All of this runs from states and involves reusable children components.

Practice #5

In our final exercise we take our last practice exercise and add local storage support for state so that it remains on page refresh or when leaving and coming back to the page.

Open index.js and make sure Practice5 is imported and called in ReactDOM.render().

Make sure that Create React App is not currently running.

Install the Simple Storage package using the following:

```
npm install react-simple-storage
```

Once the package is installed, run `npm start` and open

Then make sure that SimpleStorage is imported from "react-simple-storage" so we can use it in our component.

Come down into the Practice5() render() method. Call <SimpleStorage /> and set a parent prop equal to "this".

This should automatically make sure that count from state is saved in local storage. Try increasing the count and then refreshing the page. It should keep the original value.

16

The Component Lifecycle Explained

A lifecycle, in broad programming terms, refers the entire time an application or piece of code is running, from when it is called and initializes to when it completes and stops.

Components in React all have their own lifecycles. This starts with when they are called to be rendered, lasts while they are displayed and completes when they are no longer being called.

React provides functions we can hook into and use to call our own code during each stage of the component lifecycle. Some of these lifecycle hooks we use quite often for certain use cases and others are for more fringe cases.

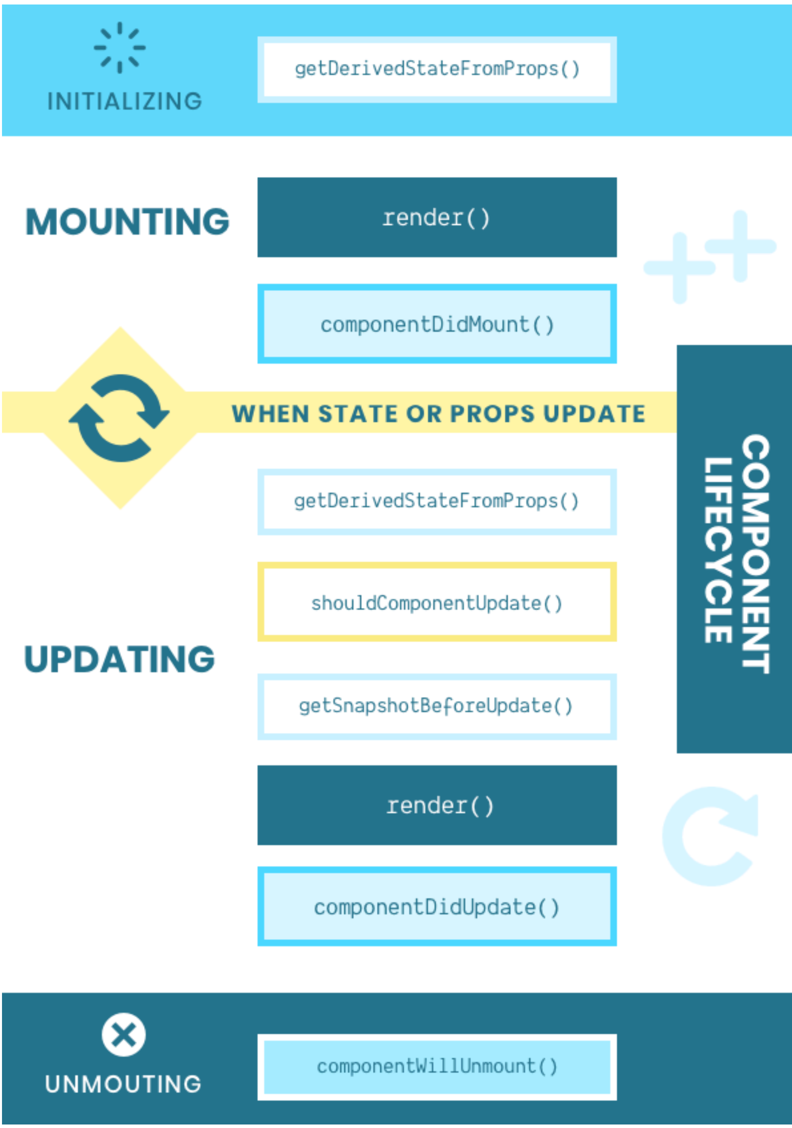
In this chapter we will explore the component lifecycle hooks and when we might use them.

> It is important to note that access to the component lifecycle is only available by default when we create our components with classes instead of functions.

The Component Lifecycle

To start off, let's look at the entire component lifecycle. This starts when a component is first called to be rendered to the page and ends when it is no longer being rendered.

Here is an illustration of the component lifecycle.



We can see from the illustration that the component lifecycle breaks down into four stages:

1. Initializing – When the component is being setup and props and state are being passed and setup

2. Mounting – The period of time when the component is actually rendered to the page and immediately after it has been rendered
3. Updating – This stage optionally kicks in any time there is an update to a prop passed into the component or state within the component
4. Unmounting – The final stage where the component is removed from the page

Each stage of the component lifecycle includes access to different lifecycle hook functions. Over the next few sections we will look at each of in more depth, paying particular attention to the most commonly use lifecycle hooks.

Initializing Lifecycle Hooks

In this first lifecycle phase, the component itself is not yet available, but rather, is being prepared to be ready. There is just one lifecycle hook available at this time.

getDerivedStateFromProps()

This lifecycle hook is executed while your component is being initialized, before it is rendered to the page. It takes two parameters, which React automatically populates for you:

```
getDerivedStateFromProps(props, state)
```

At this time in the component lifecycle, the only thing that is available are the props being passed into the component as well as the default value of any items in state. The component itself is not even yet available.

It is rare that you will need to do something using this hook. In fact, the React documentation encourages you to use other hooks besides getDerivedStateFromProps() whenever possible.

However, if you ever need to get access to the props and state of a component before it is rendered here is what that will look like.

```
const App = () => <App loggedIn="false" />
```

```
class Demo extends React.Component {
  state = {
    count: 0
  };
  static getDerivedStateFromProps(props, state) {
    console.log(props); // { loggedIn: false }
    console.log(state); // { count: 0 }
  }
  render() {
    return <p>getDerivedStateFromProps() Example</p>
  }
}
```

We can see that we first have a component App that passes a prop of `loggedIn` down to our Demo component. This is so we can see that the prop value is available with getDerivedStateFromProps.

We also have a default state setup and that is available within getDerivedStateFromProps() as well.

As mentioned, use of this component is rare. Make sure that if you are not doing things like making API calls or modifying props or state data from this method.

Mounting Lifecycle Hooks

The mounting phase of the component lifecycle is one that we use quite often. There are two lifecycle hooks that get called in this stage: render() and componentDidMount().

render()

When we introduced state in a previous chapter we began using classes to make our components. We learned that in every class based component we have to call render() and have it return a React element.

What we did not point out before is that render() is actually a lifecycle hook that gets called a few times during the component lifecycle. The first time it is called is when our component is first loaded to the page.

There is nothing special to render() that we have not seen in action

already. It is simply the function that holds the React element you want that component to load.

```
class Demo extends React.Component {
  render() {
    return(
      <div className="demo">
        <p>Demo</p>
      </div>
    )
  }
}
```

This example above should feel familiar. We call render, and inside of it return our React element.

One thing that can be noted is that if we need to write JavaScript inside of our render() function, make sure to do it before the return statement like so:

```
class Demo extends React.Component {
  render() {
    const name = "React";
    const className = "react";
    return(
      <div className={className}>
        <p>{name}</p>
      </div>
    )
  }
}
```

Other than that, you should already know how to use render. We will see later, that render is also called again when the component gets updates.

componentDidMount()

This lifecycle hook gets called immediately after the render() method is called and the component is loaded to the page.

Since the component is already rendered, code you execute here will not hold up the initial rendering of the component.

One common use for this hook is to make an API call to fetch data. Since render has been called, the API call will not hold up the loading of the page. Once the API call has returned, if you call setState() or use an event handler from props that calls setState() it will trigger the component to call render() again with the new data available.

This example below shows componentDidMount() in action with an API call to a site that we will assume returns a list of posts with JSON.

```
class Posts extends React.Component {
  state = {
    posts: []
  };

  componentDidMount() {
    fetch("https://site.com/api/posts")
      .then(response => response.json())
      .then(posts => {
        this.setState({ posts: posts });
      })
      .catch(error => console.error(error));
  }

  render() {
    return (
      <ul>
        {this.state.posts.map(post => (
          <li key={post.id}>
            {post.title}
          </li>
        ))}
      </ul>
    );
  }
}
```

The flow of data here is the render() method is called first and an empty

list will appear on the page. Then componentDidMount will be called and the fetch() call will kick off.

When the fetch returns posts, we update the posts in state with the posts we got back. This will trigger the render() function to be called again and the new list of posts in state show up on the page.

Since render has already been called, there will be no content rendered from Posts until the API call has returned. This brings up an important point.

When working with API calls or Promises in componentDidMount() you may have to think about giving an indication in the UI that something is happening.

In a simple refactor of the example above, we can add `isLoaded` to state as a boolean value set to false by default. Once the API call returns, we set can set it to true.

Within our render() method we can then check to see if the posts have loaded or not and display a message "Loading Posts…" if the API call has not returned.

```
class Posts extends React.Component {
  state = {
    isLoaded: false,
    posts: []
  };

  componentDidMount() {
    fetch("https://site.com/api/posts")
      .then(response => response.json())
      .then(posts => {
        this.setState({
          posts: posts,
          isLoaded: true
        });
      })
      .catch(error => console.error(error));
  }

  render() {
    return (
```

```
      <ul>
        {this.state.isLoaded ? (
          this.state.posts.map(post => (
            <li key={post.id}>
              {post.title}
            </li>
          ))
        ) : (
          <li>Fetching Posts...</li>
        )}
      </ul>
    );
  }
}
```

In your production code you may have to consider further feedback for users, like what happens if the posts don't load or the request fails. However, API calls serve as good example of how componentDidMount might be used and some considerations to remember.

Another time to use componentDidMount is when you have JavaScript that depends on content already being rendered on the page. For example, calling an external library that turns lists of images into slideshows or vertical blog posts into a masonry grid.

Here is an example of adding Masonry to a list of posts. Since we need the posts to be loaded before we instantiate Masonry, componentDidMount, is the perfect hook to use to add our code.

```
class Posts extends React.Component {
  state = {
    posts: ["Post 1", "Post 2", "Post 3"]
  };

  componentDidMount() {
    const grid = document.querySelector(".grid");
    const msnry = new Masonry(grid, {
      itemSelector: ".grid-item",
      columnWidth: 200
    });
```

```
  }

  render() {
    return (
      <div className="grid">
        {this.state.posts.map(post => {
          return <div className="grid-item">{post}</div>;
        })}
      </div>
    );
  }
}
```

Notice that we are using document.querySelector() to select the grid element from the DOM. This is not something we commonly do in React, but it is possible within componentDidMount().

In some rare cases you may even need to select, modify or attach event listeners to DOM elements outside of React but loaded on the same page.

Imagine for example that our initial HTML included the following:

```
<!-- This div is where React is rendered -->
<div id="root"></div>

<!-- This button is not part of React -->
<button id="not-react">Non-React Button</button>
```

All of our React code will load inside <div id="root"></div>. However, we have a button on the page that is loaded outside of React.

This example below will show how you can select, modify and attach event handlers to DOM elements outside of where React is loaded using vanilla JavaScript DOM methods.

```
class Posts extends React.Component {
  state= {
    count: 0
  };

  handleNotReactClick = e => {
```

```
    e.preventDefault();
    this.setState({ count:this.state.count+1 });
  };

  componentDidMount() {
    const button = document.getElementById("not-react");
    button.addEventListener(
      "click",
      this.handleNotReactClick
    );
    button.innerHTML = "React Controlled";
  }

  render() {
    return <p>Count: {this.state.count}!</p>;
  }
}
```

You can see here inside the componentDidMout() component that we are selecting the <button> from the page with an id of “not-react” and attaching an event listener in React to the button. The event handler is in our React code and will update the component state. All of this can happen once the component is rendered and we hook into componentDidMount().

Updating Lifecycle Hooks

Whenever a prop passed into a component or any item in state within a component change, the Update phase of React’s lifecycle kicks off.

The Updating phase will call our render() method again, as well as another common hook componentDidMount(). There are also a few less common methods that we will look at below as well:

- getDerivedStateFromProps()
- shouldComponentUpdate()
- getSnapshotBeforeUpdate()
- render()
- componentDidUpdate()

In general, these lifecycle hooks are helpful for when we need to run

code or check for conditions based on updates that happen after the component is initially rendered.

getDerivedStateFromProps()

We looked at this method above during the Initialization phase. Like during the Initialization phase, getDerivedStateFromProps() in the Updating phase executes before the component with updated props or state is re-rendered to the page.

shouldComponentUpdate()

This hook let's you check the updated values of props and state to see if the component should actually be re-rendered. If the new prop or state values to not match certain conditions it may not be worth re-rendering the component.

This can help with performance if props or state are constantly updating, but not all of those updates require the current component or children components to change themselves. It is important to note that if a component returns false in shouldComponentUpdate() then no children components will update either.

Let's take a look at shouldComponentUpdate() in action.

```
class Counter extends React.Component {
  state = {
    count: 0
  };

  componentDidMount() {
    setInterval(() => {
      this.setState({ count: this.state.count + 1 });
    }, 500);
  }

  shouldComponentUpdate(nextProps, nextState) {

    // Check if count is odd or even
    if (nextState.count % 2) {
      return false; // Does not re-render
```

```
    } else {
      return true; // Does re-rerender
    }
  }

  render() {
    return <p>{this.state.count}<p>;
  }
}
```

In the example above we are using componentDidMount() to start a timer that updates the count in state by one every half a second.

In shouldComponentUpdate we receive two arguments: the new value of props and the new value of state. We are just looking at the state in this example.

If the new state count is divisible by 2 with a remainder, giving us an odd number, then we will return false and not re-render the component. Otherwise, we have an even number and we will re-render the component.

This simple example shows how you can use conditional checks with new prop and state values as well as current prop and state values to determine whether a component should update.

In this example below we show how returning false from component did render will cause children components to not re-render as well.

```
class Counter extends React.Component {
  state = {
    count: 0
  };

  componentDidMount() {
    setInterval(() => {
      this.setState({ count: this.state.count + 1 });
    }, 500);
  }

  shouldComponentUpdate(nextProps, nextState) {
```

```
    if (nextState.count % 2) {
      return false;
    } else {
      return true;
    }
  }

  render() {
    return <Header count={this.state.count} />;
  }
}

const Header = props => (
  <p>{props.count}</p>
);
```

The only thing we have changed with this example is placing the count in a child component. However, if you run the code you will see that Header only re-renders if the state is even, even though the props in Header actually change each time the count in Counter state changes.

You will not likely need to use shouldComponentUpdate() in your React apps, however, for performance reasons and certain use cases, it is good to know this particular lifecycle hook exists.

getSnapshotBeforeUpdate()

This is another fringe use case lifecycle hook that works a little differently than the others. It executes right before the component is re-rendered and it let's you get any DOM information from the page and pass it to another hook, componentDidUpdate(), called right after the component is re-rendered.

This can let you get a "before" snapshot of anything on the page, often window or cursor or positioning information, and compare it to the state of the page after the component is updated.

In the example below we are displaying a list of posts from state. Every .5 seconds a new post is added to state. All of the posts are displayed inside of a div with a fixed height of 100px with a scrollbar.

Right before each new post is rendered, inside getSnapshotBeforeUpdate(), we get the full height of the post container, including what needs to be scrolled to see.

This value returned from getSnapshotBeforeUpdate() is then passed as the third parameter into componentDidUpdate(), a new component we will explore in more depth shortly. However, at this point, the new post has been rendered so the scrollable height of the post container has changed. We then log these two values out in componentDidUpdate() to compare them.

```
class Posts extends React.Component {
  state = {
    posts: ["First Post"]
  };

  // Add a new Post to state every .5 seconds
  componentDidMount() {
    setInterval(() => {
      this.setState({
        posts: [...this.state.posts, "New Post"]
      });
    }, 500);
  }

  // Before new post is rendered from state update
  // Get the current height of the post container
  getSnapshotBeforeUpdate(prevProps, prevState) {
    const posts = document.getElementById("posts");
    return {
      height: posts.scrollHeight
    }
  }

  // After new post is rendered from state
  // Get the snapshot height and compare to new height
  componentDidUpdate(prevProps, prevState, snapshot) {
    const posts = document.getElementById("posts");
    const newHeight = posts.scrollHeight;
```

```
    console.log(`Prev height: ${snapshot.height}`);
    console.log(`New height: ${newHeight}`);
  }

  render() {
    return (
      <div
        id="posts"
        style={{
          overflow: "scroll",
          height: "100px",
          border: "1px lightgray solid"
        }}
      >
        <ol>
          {this.state.posts.map(post => <li>{post}</li>)}
        </ol>
      </div>
    );
  }
}
```

The example above shows how any value we get inside of getSnapshotBeforeUpdate() before a component is re-rendered can be passed into componentDidMount(). If you have multiple values it may make sense to return an object or array, however, you can also just return a single value.

As we see, componentDidMount() will receive the previous props and previous state, so do not use getSnapshotBeforeUpdate() just to pass prop or state values to componentDidMount. It is more likely you will get values from the page or window or something that is going to change like that.

As mentioned, getSnapshotBeforeUpdate(), is one of our less commonly used lifecycle hooks, but it is good to know it exists and how to use it.

render()

We have already looked at the render method in the Mounting lifecycle phase. This method get's called again when a change to props or state occurs and the lifecycle methods above have already run: getDerivedStateFromProps(), shouldComponentUpdate(), getSnapshotBeforeU

Note that render will only be called on an update if shouldComponentUpdate() returns true, which it does by default.

We don't really need to look at render() again too much, but now it should make sense how a change in a prop or state value causes a component to be re-rendered. It is thanks to the Component Lifecycle.

componentDidUpdate()

The last of the Updating phase lifecycle hooks is componentDidUpdate(). It gets called immediately after render() executes due to an update.

This component can come in handy if you need to make modifications to the DOM or components or state based on changes that just took place. It is important to note though that calling setState() in componentDidUpdate() will cause the Update phases to start over again. So to avoid creating an infinite loop of updates, make sure to wrap any setState calls inside of a conditional statement.

As we learned previously, componentDidUpdate receives three parameters automatically: previous props, previous state, and snapshot.

Here is a familiar example where we add a new post to the page every .5 seconds. However, this time we are doing a few more things inside of componentDidUpdate().

```
class Counter extends React.Component {
  state = {
    posts: ["First Post"]
  };

  timerID;

  componentDidMount() {
    this.timerID = setInterval(() => {
```

```
      this.setState({
        posts: [...this.state.posts, "New Post"]
      });
    }, 500);
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    // Stop timer after 20 posts
    if (this.state.posts.length >= 20) {
      clearInterval(this.timerID);
    }

    // Scroll to bottom of postsContainer
    const postsContainer = document.getElementById("posts
    postsContainer.scrollTo(0, postsContainer.scrollHeigh

  }

  render() {
    return (
      <div
        id="posts"
        style={{
          overflow: "scroll",
          height: "100px",
          border: "1px lightgray solid"
        }}
      >
        <ol>
          {this.state.posts.map(post => <li>{post}</li>)}
        </ol>
      </div>
    );
  }
}
```

The first difference here is that we are creating assigning the timer to timerID. This allows us to stop the timer inside of componentDidUpdate() after 20 posts appear in this.state.posts.

The second thing we do inside of componentDidUpdate() is get the

new scrollable height of the posts container after the new post has been added. Then we automatically scroll the user to the bottom of the post container to see the latest post.

You will not use componentDidUpdate() in every component, but it is one of the more commonly used components in React. Remember that you can also get access to previous prop and state values, which can be helpful for some projects.

Unmounting Lifecycle Hooks

The final stage of the Component Lifecycle involves a single hook that get's called right before a component is removed from the DOM. This can be helpful for cleaning up anything allocated in memory, like timers or event listeners added to non-React elements in the DOM. You will not need to use this lifecycle hook often, but at times it will be essential.

componentWillUnmout()

In this example below we create an App component with a button to start a count down timer. The count down timer is it's own component and it count's down for 3 seconds. After 3 seconds it is removed from the page.

Inside of CountDown we call componentWillUnmout() to stop the timer we started and log a message letting us know the component is about to be removed.

```
class App extends React.Component {
  state= {
    displayTimer: false
  };

  toggleTimer=()=> {
    this.setState({ displayTimer:!this.state.displayTimer }
  };

  render() {
    return(
      <div>
```

```
        {this.state.displayTimer ? (
          <CountDown toggleTimer={this.toggleTimer} />
        ):(
          <button onClick={this.toggleTimer}>Start Timer<
        )}
      </div>
    );
  }

}

class CountDown extends React.Component {
  state= {
    count: 3
  };

  timerID;

  componentDidMount() {
    this.timerID=setInterval(()=> {
      this.setState({ count:this.state.count-1 });
    }, 500);
  }

  componentDidUpdate() {
    if(this.state.count===0) this.props.toggleTimer();
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
    console.log("Timer about to unmount!");
  }

  render() {
    return <p>Wait {this.state.count} more seconds..</p>;
  }
}
```

You will not need to use componentWillUnmount often, but once

again, it is good to know that it exists and how and when you might need to use it.

A Review of the Component Lifecycle Hooks

We can do a lot with React just using state and props. However, to create truly interactive and well architectured UIs, we will need to use Component Lifecycle Hooks.

Here are the most common hooks you will likely use:

- Mounting – componentDidMount()
- Updating – componentDidUpdate()

And here is a list of the less common hooks you may need at different times:

- Initializing – getDerivedStateFromProps()
- Updating – getDerivedStateFromProps()
- Updating – shouldComponentUpdate()
- Updating – getSnapshotBeforeUpdate()
- Unmounting – componentWillUnmout()

If all of these options seem a little overwhelming at first, just focus on learning and practicing componentDidMount and componentDidUpdate() and know the rest you can come back and review when a necessary use case presents itself.

Let's Practice!

Now that we have learned about the Component Lifecycle, let's practice using some of the hooks in practice.

We will focus mostly on the more common hooks, but make sure you have a chance to use some of the less common hooks as well.

17

5 Exercises with the Component Lifecycle

Now that we have learned about the Component Lifecycle, let's do some practice hooking into the lifecycle methods.

To get started with these exercises, open the "9-component-lifecycle/starter" directory in your code editor and run "npm install" and then "npm start".

Practice #1

In this first exercise we will practice hooking into the getDerivedStateFromProps().

Make sure <Practice1 /> is called in the "src/index.js" ReactDOM.render().

Open up "src/Practice1.js".

Notice the Practice1 component calls <Header /> and passes a prop of sitename.

Inside of the Header class, call static getDerivedStateFromProps(props, state) {}. Log out props and state inside of that method just to test what the parameters receive. Props should result in the sitename and state should give you the username.

Then create a new object called newState with a property of username set to some username of your choice.

Finally, return newState inside of getDerivedStateFromProps. This will override the previous value of state with the newState you created.

This practice exercise shows us how to use the rarely used getDerivedStateFromProps(). Here you can get the props and state before a component mounts to the page. Remember to always return the existing value of state or a new one.

Practice #2

In this exercise we will practice making an API call inside of componentDidMount().

Make sure <Practice2 /> is imported in "src/index.js" and called inside of ReactDOM.render().

Open up "src/Practice2.js".

Inside of the Practice2 component, call componentDidMount().

Then make a fetch request to the following demo API that will return three posts:

https://dev-react-explained-api.pantheonsite.io/wp-json/wp/v2/posts

Then take the response from that promise and call response.json(). That will give you the three posts. So with that response you can call setState and update the value of posts with what the API gives you.

It might be a good idea to catch any errors and log them out as well.

Once the component is mounted, it will make the API request and update the default post in state with the posts from the API.

This practice exercise helps us learn how to accomplish the common task of making an API request within a component and updating state with the content returned from the API call.

Practice #3

In this exercise we will practice working with the not too commonly used shouldComponentUpdate lifecycle method to set a conditional statement for whether a component should update or not.

In this example we will build a counter with a bar chart that counts up to 20. However, we only want the bar chart component to update when the count is divisible by five.

Make sure <Practice3 /> is imported in "src/index.js" and called inside of ReactDOM.render().

Open up "src/Practice3.js". Take a look over the code and what happens in the browser. By default, <BarChart /> should update with each point increase.

Then add shouldComponentUpdate(nextProps, nextState) {} in the BarChart component.

Inside shouldComponentUpdate() check to see if the new points in props is divisible by five with a remainder like so:

```
if (nextProps.points % 5) {}
```

If this passes it means that the number is not divisible by five. If that is the case then return false and the component will not update.

If there is no remainder then points is divisible by five and the <BarChart /> should update, so return true.

This single conditional statement will cause the BarChat to only animate and update when points is divisible by five.

You may not need to use this component often, but it is important to know how to use it when you want to control whether a component updates based on new values in props or state.

Practice #4

In this exercise we will work with the componentDidUpdate() lifecycle hook to tell when values in props or state have changed.

To do this, we will build off of our last practice exercise and log out details when a component has been updated.

Make sure <Practice4 /> is imported in "src/index.js" and called inside of ReactDOM.render().

Open up "src/Practice4.js". The code should look familiar from the last exercise.

Inside of the Practice4 component, call componentDidUpdate(prevProps, prevState) {}. Log out the previous state of points (as this component does not receive props). Also log out the current state of points (this.state.points).

Next, write a conditional statement to check if prevState.points !== this.state.points. This will tell us whether or not, state has been updated. If it has been updated, log out, "State Changed!".

Now, let's do the same thing, but with props.

Come down into the BarChart component and call componentDidUpdate(prevProps, prevState) {} again. This component does not have state, but it does have props, so we will work with those.

Log out points from the previous props. Also log out points from the current props.

Finally write another conditional statement that checks if the two are different:

```
if (prevProps.points !== this.props.points) {}
```

If this passes then the new props are different from the last time render was called and you should log out a message "Props Changed!"

Now we have an example of checking to see if state has been updated and props have been updated. It would also be possible to combine the two in a component that has both state and props.

Whenever you need to hook into a component to do something when it updates, you should now have a model for how to use shouldComponentUpdate().

Practice #5

In our final exercise we will look how to hook into componentWillUnmount() to take actions when a component is about to be removed from the DOM.

In this example we modify our counter and bar chart components to automatically increase the points every 300 milliseconds until it reaches ten. Then we remove the bar chart component and display a message that the goal has been reached. When the bar chart is removed from the page we also need to stop the timer so it does not continue to count in memory.

To setup, make sure <Practice5 /> is imported in "src/index.js" and called inside of ReactDOM.render().

Open up "src/Practice5.js". Take a look at the code and run it in the browser. You should see the bar chart start counting up to ten automatically and then the bar chart is removed. However, the timer does not stop.

Come down to where <BarChart /> is called on line 34 and add a stopTimer prop with a value equal to this.stopTimer.

Then come down into the BarChart component. Call componentWillUnmount() {} inside of BarChart. Inside of that log out the <BarChart /> is unmounting.

Finally, call this.props.stopTimer(). This will cause the timer to stop when the bar chart is no longer loaded. Now the example will count to ten and then stop when the bar chart is removed.

This is a good example for when you might want to use componentWillUnmount(). If a timer or other process has started on the page that will continue to exist in memory, then it can be a good practice to stop or remove the process when components using it are no longer on the page.

Next Up

Now that we have learned a good deal about React and practiced using it, we will turn our attention to building a more complex project than the simple examples we have worked with so far.

This will allow us to pull together everything we have learned as well as learn some new practices and helpful libraries for working with React.