REACT EXPLAINED

React is a popular and powerful library for building websites and applications with JavaScript.

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

Some of the reasons for React's popularity include its 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. React offers a component architecture for building user interfaces. This is different from some other libraries, such as Angular, which come with much more functionality out of the box.

This single focus of React has led to a scenario of "React and Friends," where we 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 is also a great tool, and I am actually a big fan of Angular. But React's simplicity has allowed it to evolve quickly, while at the same time remaining 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 its own version of the DOM in memory that we 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 we create UIs purely with JavaScript? Do we create them partially in HTML? How much HTML goes in our JavaScript and how much JavaScript related code goes in our 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 us 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 JavaScript and markup to this extent was highly discouraged. However, today it makes a lot of sense to be able to build entire UIs from within a JavaScript file without needing to use actual HTML. I imagine we will also grow to love (or at least appreciate) JSX. Either way, it can't be negated that 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. 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 the 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 the app. This follows a model with a single source of truth, while still leveraging event handlers we'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. However, Vue continues as a popular alternative to React.

We cannot diminish the fact that being developed at Facebook is 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 address. 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). 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 entering 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 we want to develop with JavaScript today, we should learn some basic tools. These include the command line basics, build tools like webpack, and transpiling tools like babel. However, we 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 us to type into a command line tool and get a new React App set up. It does this with all the tooling working and hidden from view. This way we can focus on learning React, which is not that hard, and not have to worry about the tooling, which can be tricky for beginners to learn.

It's still important to learn some tooling as we 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 allows 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

We 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 tool, not just for our own projects, but for potential employment as well.

LET'S GET READY TO REACT

Hopefully this introduction has helped us understand at a high level what has made React so popular. We have discussed how React's simplicity in design, ingenious engineering, and how it 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 taught will also apply to writing React on the server side, for native applications, and even for VR and 360 environments.

ORGANIZATION OF THE BOOK

This book is divided into three sections: Preparing to React, React Explained and A React Project.

In the first section of the book, "Preparing to React," we will review some of the important JavaScript and Development Tools you will use when working with React. We will also talk about React at a very high level, defining concepts like Component Architecture and One Way Data Flow.

In the main part of the book, "React Explained," we go over each of the core concepts of React in depth. We cover one topic per chapter to keep everything as simple and clear as possible. Between each content chapter, we also have 5 exercises designed to give a hands on practice working with React and show some common patterns of how to use React.

The final section of the book, "A React Project," takes us through building an entire project with React. This will introduce us to some helpful packages often used along with React, like React Router and the Firebase Database and Authentication system. By the end of this project, we should feel more comfortable going out and tackling our own apps or website projects using React.

HELPFUL PREREQUISITES

It's not necessary to know a lot of JavaScript to learn React. In fact, some people get their introduction to writing JavaScript by working with React. However, the more JavaScript we know before learning React, the easier it will be to learn React.

To get a free course on JavaScript Basics, visit the book website: ReactExplained.com. In the first chapter of this book, “Helpful JavaScript To Know for React,” we will also review some of the newer and more important aspects of JavaScript to make sure learning JavaScript doesn’t distract us too much from learning React.

However, I try to write and teach in a way so that someone of any skill level can pick up what we’re talking about and follow along successfully, even if everything doesn’t make 100% sense on the first past.

So, are we excited to dig in deeper? Let’s get ready to React!

ABOUT THE OSTRAINING EVERYTHING CLUB

React Explained is part of the OSTraining Everything Club.

The club gives you access to all of the video classes, plus all 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 Everything Club today: https://ostraining.com.

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

ABOUT THE OSTRAINING TEAM

Zac Gordon is a professional educator, with a current focus on JavaScript development with and alongside WordPress at javascriptforwp.com. Zac has years of experience teaching at high schools, colleges, bootcamps, and online learning sites like Treehouse, Udemy and Frontend Masters. In addition to teaching, Zac also runs Web Hosting for Students, one of the world's largest hosting companies dedicated to students and teachers. You can also catch his free Office Yoga sessions on OfficeYoga.tv.

This book also would not be possible without the help of the OSTraining team. Thanks to Valentin for the cover and to Steve and Stacey for editing.

WE OFTEN UPDATE THIS BOOK

This is version 1.0 of React Explained. This version was released on March 5, 2018.

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 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.

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

React Explained

REACT EXPLAINED

Your Step-by-Step Guide to React

ZAC GORDON

OSTraining

CONTENTS

PART I.

PREPARING TO REACT

This section goes over important technical skills that will help provide a better understanding of React.

Chapter 1: "Helpful JavaScript To Know for React" may be a review or something to come back to for reference more than once as we 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: "Helpful Developer Tools to Know for React" gives a high level overview of the most common development tools used with React. Although we explore 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 us comfortable beginning to read and write our first React apps.

CHAPTER 1.

THE JAVASCRIPT YOU SHOULD KNOW FOR REACT

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 at 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 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 a lot of the code we 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 this is all done with vanilla JavaScript, the more JavaScript we 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, we 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 we can use them in our JavaScript, they may not be supported in browsers. This gives us two options for using new additions to the JavaScript language:

1. Wait for browsers to enable support for new features.
2. 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 we want to work with, we may use a transpiling tool, or we may find a polyfill.

For the examples in this book, we will rely primarily on our build tool setup (including a transpiler) to handle any 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. We 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. We can in turn assign this value to a variable or pass it 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 we are writing a statement and when we are writing an expression. It's important because 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 its value to be reassigned to another value. However, if the `const` value is an object or an array, we 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;
```

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

```
'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 we need data to change, we 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 that if we want our data truly immutable, we will need to use `Object.freeze()` or an immutable JS library that does something similar. If for some reason we ever need to unfreeze an object or array, we can use `Object.unfreeze(objectOrArray)`.

#4. TEMPLATE LITERALS EXPLAINED

Template literals, also called "Template Strings," are a special type of string in JavaScript that allow for us to include 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 also allow us to use line breaks when creating our 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 we logged out the data to the console. However, if we render the longMsg string to the DOM, it would appear as one normal string of text with no line breaks.

It is quite likely to 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 (=). This is where we commonly have a variable name.

The anonymous means that our function does not have a name, so it is usually executed 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 create a function expression, pass in the parameter title, and then log 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 we want to break an arrow function into multiple lines, we 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 we 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 we want `this` to refer to a class rather than a method. However, it can be confusing if we 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 use them.

#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, we will not create our 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 create a new class called "Example" that extends 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 simply return 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 we 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, in 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 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 we 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 we want to call it later (possibly more than once) and keep the value we defined.

Here is an example of when we 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 some times when we want to 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`. Commonly we will see them used when we 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 us 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 our conditional in the parenthesis. Our conditional statement is just checking to see if something is true or present. We 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 us to get comfortable with how to write them. It's useful for us to remember the syntax so we do not have to look up 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
```

We 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`.

We will absolutely see deconstruction assignment used in React apps, starting with the first lines where we 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 in case 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 we will see JavaScript for loops when using React.

`.reduce()` takes a 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. It remembers the `total` from the last iteration. We will often see this parameter named something generic like `memo`.

The second parameter is the name we 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 equal to 39, and the last time it is equal to 209.

The next parameter is simply the `index` of the current iteration. With simple reduce examples, we 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 we want to have multiple values, we may want to do a filter instead.

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

#11. DOM NODE CREATION EXPLAINED

If we want to understand what React is doing under the hood, it is also extremely helpful to become familiar 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. This is 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 here we are 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 the more we 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 we want a single file to export multiple items, rather than a single default export like above. This could be helpful if we 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 imported the name variable, as well as the render function, but we also renamed the render function to `display()`.

As mentioned, we 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 our imports and exports as they do not work at this time in browsers.

WHAT'S NEXT?

Now that we have reviewed quite a bit of JavaScript, let's turn our attention to talking about tools for React development.

CHAPTER 2.

HELPFUL DEVELOPER TOOLS TO KNOW FOR REACT

In the previous chapter, we 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

For those who are already comfortable with these tools, skip to the next chapter. However, for those who aren't familiar with these tools, 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 we can open a file by navigating to the file and typing "open readme.md" in the command line, and it will open the file. We 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 us 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. We 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 us a command line. Integrated command line tools will often give us 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, we need to know some command line basics. Here are some of the basics 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 our most important tools. Simply put, a code editor lets us edit code without injecting unwanted characters or formatting. A word processor and rich text editor are not tools for editing code and will inject extra characters and break our 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 for us to extend them via themes and plugins (or extensions or add-ons). These extensions allow us to do a lot more with our editor. For example, they can 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 we change a file name somewhere, it can automatically update reference to that file anywhere in our code.

Today the landscape between code editors and IDEs is blurring. Most developers are quite happy using code editors with popular extensions or add-ons and never use an IDE. However, IDEs do have some major benefits and are worth looking into.

For beginners who 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. Open the command line tool and check to see if Node is installed. Use the following command:

```
node -v
```

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

As we delve deeper into working with React, we will likely come across React running on the server side with Node. This is outside the scope of this book. However, if we were to continue exploring React Native, then we would actually write React code in Node – and not just use it for development tools.

For this book, make sure to have the latest version of Node installed.

#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 is the most popular. Although originally named for Node packages, today it manages frontend scripts for us too. The folks at Facebook created Yarn (as well as 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 set up an app for working with NPM, we usually start with opening that app directory in the command line and typing something like this:

```
npm init
```

This command will take us 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 and 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 we 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 our application code for production. Instead they just live on the computer.
2. Dependency: This means our application needs this package to run properly, and our build tool should bundle this package along with the final application code.
3. Development Dependencies: This means our application (or tooling setup) only needs this package for local development, and the package should not be bundled with our final source code.

For those who are wondering which way to install a package, don't worry! Most packages will tell us the best way to install their library in their installation instructions.

Then in our application code, we can import 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 are not usually included when application code is saved to something like github or the production server.

Luckily, as long as we have a `package.json` file, we 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, we will often see "npm install" as the first task when working with an application that we 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 us to create shortcuts for commands that we commonly run. A common example is to create a shortcut called “dev” to start up your development environment. We can then execute this entire command with the following (much shorter) command.

```
npm run dev
```

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

For those who 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 us up and running. We can find these tutorials 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 us 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 our 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 bundling tool called Webpack. When we first start with Create React App, we will not even see the webpack configuration files or code. However, if we create React applications from scratch or make more advanced applications, we 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" is that it has 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 we want transpiled and what browser versions we 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 the 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.

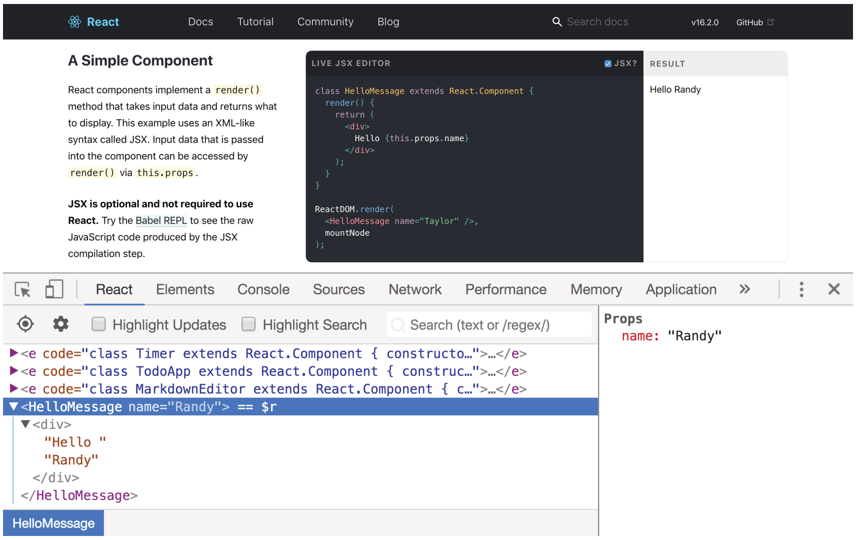
In addition to the webpack server, several other Node driven local development servers exist, like http-server. However, for the most part, we will leverage the webpack development server when writing React.

When launching or testing our final React app, we can usually use any server that allows client side JavaScript to run. If we have built additional server side functionality outside of our React apps, we will want to use a production server that supports both. Down the road, for those who get into server side React, there will be a need for 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 that since we are reading a book on React, we have already used a Web Inspector tool to explore things like the DOM, markup, CSS, and possibly more.



A Simple Component

React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

LIVE JSX EDITOR ✓ JSX?

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  mountNode
);
```

RESULT

Hello Randy

```
▶<e code="class Timer extends React.Component { constructo…">…</e>
▶<e code="class TodoApp extends React.Component { construc…">…</e>
▶<e code="class MarkdownEditor extends React.Component { c…">…</e>
▼<HelloMessage name="Randy"> == $r
  ▼<div>
     "Hello "
     "Randy"
   </div>
 </HelloMessage>
```

```
Props
  name: "Randy"
```

HelloMessage

React Dev Tools adds a new tab to the Web Inspector that shows off information about our React app. In the graphic above, notice that we have highlighted a block starting with <HelloMessage … and what we 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. Many will likely start to use this tab in place of the normal Elements tab. Note that we can drag and drop the placement of the React tab from under the double arrows (>>), where it initially appeared, to a more convenient place, such as the far left in this screenshot.

We will want to install React Dev Tools in either Chrome or Firefox before we 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. For those who are first seeing these tools, this may seem overwhelming. Don't worry, once we get into a workflow, we will start to feel more comfortable with these tools.

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

CHAPTER 3.

A HIGH LEVEL OVERVIEW OF REACT

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

In the next few pages, we'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 we will build with React are language-agnostic.

React has companion libraries that enable our interfaces to work in different locations. React has libraries to make our 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 our interfaces to work in client-side websites and applications in the browser. ReactDomServer, ReactNative, and React360 are also libraries some may want to explore for using React interfaces in other environments.

In addition to providing helper functions for building interfaces, React's architecture allows us to handle interactions with our 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. We 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. We create components 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 us write

“HTML markup” directly in our JavaScript. Since we are using React to create user interfaces, and user interfaces on the web involve HTML markup, it makes sense that we would see HTML like elements within our UI components. We will explore JSX in depth in this book.

Let’s look at some code from 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>
      <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>
  );
}
```

We 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.

We should only change the value of a prop 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, we have to discuss *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. Therefore, data is set at the highest component necessary for all children components to receive the information they need. Data cannot flow up to parent components.

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. Additionally, the update needs to be 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 it makes a good instance of when to rely on state.

```
class Counter extends React.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.

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 React.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 any time state is updated, a new value will automatically be 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 our apps. 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 we can call it from any child component, but the action will take place in the original component that set the state, which also has the ability to update it as well.

For those who are not familiar with passing functions as parameters, it is completely valid vanilla JavaScript.

Once we call the event handler from the child component, state will be updated in the parent component, and 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 we set and update 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 we 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 how to 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.

Chapter 4: "React Elements and Components Explained" dives into the most important function in React, `createElement()`. We look at how to use it to create the most basic building blocks in React: elements. Components are also explained as functions or classes that return a single element or nested group of elements.

Chapter 5: "5 Exercises in Writing React With Elements and Components" gives us practice writing the simplest React possible using React.createElement to make basic elements and functional components.

Chapter 6: "JSX Explained" introduces the amazingly helpful JSX language extension for JavaScript, which allows us to write what looks like HTML in our JavaScript. While this is a little controversial compared to previous JavaScript patterns with a separation of concerns between markup and behavior, it has become the standard for creating markup with React.

Chapter 7: "5 Exercises in Writing React With JSX" walks us through how to do what we did in Chapter 5, but this time we will do it using JSX syntax.

Chapter 8: "Create React App Explained" will take us through the tooling stack we will use to work with React. We explore how to set up and use Create React App and the various commands it comes with. This will allow us to easily spin up and manage React projects.

Chapter 9: "5 Exercises With Create React App" walks us through how to set up and use Create React App for spinning up and working on React projects.

Chapter 10: "Props in React Explained" shows us how data is passed through components using props. Props are similar to parameters passed to a function or attributes attached to markup. They also enforce the one way data flow in React.

Chapter 11: "5 Exercises in Working With Props" gives us practice passing information between components using props.

Chapter 12: "State in React Explained" demonstrates how we can manage data that changes in our code that we need to keep track of. We look at how to set and update state, as well as how to pass it down into other components via props. Since state only exists at a component level in React, we also talk about where to set state in our apps.

Chapter 13: "5 Exercises in Working With State" gives us practice setting up and modifying components state as well as how to pass the values of state and ability to update state down into children components.

Chapter 14: "The Component Lifecycle Explained" goes over how we can hook in custom code at different points when our app or components are running. This gives us a fine grain control of when and how our code should execute and lets us do things like work with asynchronous API requests or improve performance by controlling when things should update.

Chapter 15: "5 Exercises With the Component Lifecycle" gives us practice working with the different component lifecycle hooks.

CHAPTER 4.

REACT ELEMENTS AND COMPONENTS EXPLAINED

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

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

We create React elements using a function called `createElement()`.

REACT.CREATEELEMENT() EXPLAINED

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

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

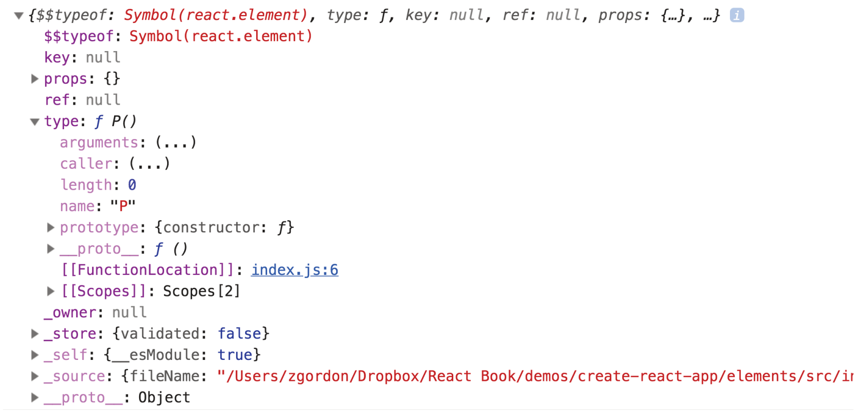
The method takes three parameters:

1. The type of element to create
2. The properties or attributes we want assigned to the element
3. Element children (can be 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:


```
▼{$$typeof: Symbol(react.element), type: ƒ, key: null, ref: null, props: {…}, …} ℹ
   $$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/ir
 ▶ __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. We will save the call inside of functions (and later classes), so we can reuse or call them in other places in our code.

This might result in a pattern like the one below:

```
function Welcome() {
  return React.createElement("h1", { className:"
welcome"}, "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 want to display a welcome message, we have a reusable component (or function) that we can easily call.

Note that we capitalize component names in React. We use `Welcome()` and not `welcome()`. We do this 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, we will nest elements 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 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. It 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. Start from the last line. Here we pass the `App()` component into

`ReactDOM.render()` to be converted into valid DOM elements. Then we add it 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 we 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 we learn to follow a component hierarchy, we will be able to find our way around most React applications.

So we can double check our 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 allow 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 we 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 its 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. If there's a situation where we do not want an actual element displayed to the page, we could use React.Fragment.

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 we 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 we use them 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, inside the function, we return the components, elements and any other JSX we want to have our component to have.

We will use arrow functions to create components until we learn how to create class based components to leverage State.

A BRIEF REVIEW OF ELEMENTS AND COMPONENTS

When working with React, some common terms will get used in different ways to mean different things. We can often use elements and components 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.

We can nest React elements and components, which allows us to build 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 in React, a file will contain a single element and that is all.

LET'S PRACTICE!

In this chapter, we learned about React elements and components. We're ready to start writing React code. Turn the page, and there are five practice exercises to get us started with React.

CHAPTER 5.

FIVE EXERCISES IN WRITING REACT WITH ELEMENTS AND COMPONENTS

Let's start writing React!

In this chapter, there are five exercises that will help you start writing React elements and components.

GETTING SETUP

You can download the practice exercises for this book on Github at the following url:

- https://github.com/zgordon/react-book.

The exercises for this chapter are in the "`4-elements-and-components`" directory. Inside of that folder, you will find two other folders:

- "`starter`" has incomplete starter files outlining the exercises for you to use for trying on your own.
- "`completed`" has all the completed examples for you to check your answers against.

You may find it helpful to open the entire "`4-elements-and-components`" folder in your code editor so you can easily access to starter and completed files.

PRACTICE EXERCISE #1

For the first exercise, create a simple paragraph element using `React.createElement()`.

Make sure the paragraph element:

- does not have any special classes or attributes.
- includes some simple text like “Hello React”.

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.

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.

Your final markup should look like this:

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

PRACTICE EXERCISE #2

The second exercise starts off in a similar manner as the first exercise. Create an element and save it as a variable.

The element you’re creating here should:

- be an h1 element with a class of “entry-header”.
- include 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><a href="http://reactjs.org/">React</a></h1>
```

PRACTICE EXERCISE #3

In this next exercise, create a component rather than a single element.

The component should:

- be called `Header`.
- return a `header` element with an ID of "main".

Then, inside the `header` element, 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></h
1>
  <p>Hello React.</p>
</header>
```

PRACTICE EXERCISE #4

From here, you continue with another component example. For this exercise, create 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 the final exercise, create a component called `App` that returns a `React.Fragment` with your `Header` and `List` components within it.

This will give you practice using `React.Fragment`. It will also give you practice creating components that return other components, which 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></h
```

```
1>
  <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 there's library called JSX that provides an easy to use shorthand for React.createElement.

For example, you can write the following vanilla React with JSX.

Vanilla React:

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

JSX:

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

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

In the next chapter, you'll learn the rules of JSX. Going forward, you will hardly ever (possibly never) need to write `React.createElement()` in its long form again.

CHAPTER 6.

JSX EXPLAINED

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

As we saw, creating components can involve a lot of nested function 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, we all have seen the markup language, HTML. XML is an extended version of HTML where we can create our own elements with names of our like, such as `<item>`, `<query>`, or pretty much anything else we want.

JSX stands for **J**ava**S**cript **X**ML. It is an extension for JavaScript that allows for writing what looks like HTML and XML in our JavaScript.

So we 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 displays 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.

Let's use the example JSX we looked at above again:

```
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!"
    )
    );
)
```

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

We will soon get very comfortable writing JSX and forget all about `React.createElement()`.

For now, let's look 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?

We will likely see JSX written in two places:

1. We can save it as a variable anywhere in your JavaScript code.
2. We will most commonly use it 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 want to reference that `heading` element, we could use the heading constant. While this is not the most common way of writing JSX, it is done often, so it's helpful to know we might see it used this way.

More likely though, we 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 its own element in JSX.

We are going to learn a lot more about the syntax and rules of JSX. For now, we want to remember that we will most commonly use it in the return statement of components; however, we can also save it as a variable anywhere in our JavaScript code.

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, we 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 lettering 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 set up 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 we write these in 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 using 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, and 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 we 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. We do this since we are once again writing JavaScript within JSX tags (starting and ending with `<div></div>`).

Remember, whenever we have JSX tags and want to call a JavaScript variable or write some vanilla JavaScript, we 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 we 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 that 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 message = 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. Then if it is true, it assigns the value of the code in the parenthesis after the question mark. If the check returns false, it will return the code in the parenthesis after the colon. Hopefully we notice this conditional test returns a value and is therefore 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 pull 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 prefer it. 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

In the above example, we have an if and an else condition. Sometimes we do not have an else statement. In these cases, we rely on an interesting pattern in JavaScript where any expression always returns to true (unless it explicitly returns a false 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 displayed. 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 we have a conditional check to do that only requires an if and no else.

- First, place the value we want to check.
- After that, use double ampersands.
- Finally, place the expression we 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 we need to map over content, we should pull it out of the return statement and into a variable or its function or class. In the example below, we clean up our code a little bit by pulling out the map into its 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>
  );
}
```

We 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. Thus, we should not take the following approach:

```
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 we 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, 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. We could shorten this 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 we may want to use the <React.Fragment> instead if the <> shorthand format is not supported.

One final note that should be mentioned is that it's 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 us comfortable with writing JSX and help us walk through some of the most important concepts and rules for how it works.

CHAPTER 7.

FIVE EXERCISES IN WRITING REACT WITH JSX

Now let's do a little practice with the JSX you have learned to solidify what was learned.

If you completed the last set of practice exercises, you will find these exercises familiar. In fact, they are the exact same exercises as before. However, this time you will complete them using JSX rather than `React.createElement()`.

GETTING SETUP

You should already have the practice exercises, but you can download them from Github if you do not already have them:

- https://github.com/zgordon/react-book

The exercises for this chapter are in the `"5-jsx"` directory. Just like with the last set of practice exercises, inside of this folder, you will find two other folders:

- `"starter"` has incomplete starter files outlining the exercises for you to use for trying on your own.
- `"completed"` has all the completed examples for you to check your answers against.

You may find it helpful to open the entire `"5-jsx"` folder in your code editor so you can easily access to starter and completed files.

PRACTICE EXERCISE #1

For the first exercise, create a simple paragraph element using JSX.

Make sure the paragraph element:

- does not have any special classes or attributes.
- includes some simple text like “Hello React”.

Save this element as a variable name `pEL` using `const`.

To test, call `pEL` the bottom of the exercises where `ReactDOM.render()` is called.

Your final markup should look like this:

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

PRACTICE EXERCISE #2

For the second exercise, you 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><a href=”http://reactjs.org/”>React</a></h1>
```

PRACTICE EXERCISE #3

In this next exercise, create a component rather than a single element.

The component should:

- be called `Header`.
- return a `header` element with an ID of "main".

Then, inside the header element, 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, you continue with another component example. For this exercise, create 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 the final exercise, create a component called `App` that returns a Fragment with your `Header` and `List` components within it.

This will give you practice using `React.Fragment`. It will also give you practice creating components that return other components, which 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></h
1>
  <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, your JSX "markup" is being passed to `React.createElement()`.

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

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

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

The next step from here is to learn how to use the Create React

App tool. This will give you a better integrated development environment for working with React.

CHAPTER 8.

CREATE REACT APP EXPLAINED

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 limitation is that we cannot use `imports` to reference other libraries or easily break up our code into small modules. There are more limitations, but these alone are worth moving from the approach we have been working with to using more powerful tools.

The typical React workflow involves using scripts for bundling, transpiling, linting, testing, running a development server, and often more. This can be a lot to set up, keep updated, and modify manually to suit our 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 settings 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. 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

For those who'd like some more information on any of these, please refer back to the chapter "Developer Tools for React."

There are also some alternative Create React App versions that include other tools. For example, Create React App Typescript includes Typescript, and Create React App Parcel uses Parcel instead of webpack. 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 set up 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 to 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.

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 we look inside the folder, it will contain *a lot* of directories in addition to the ones Create React App depends on to work.

The reason we see so many packages or directories is that often 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. What's included inside this folder is not just for React itself but also for the different tools that Create React App includes.

Later we will install our own packages. 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 we can swap out with our own favicon.ico file.

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

The `manifest.json` file, called a Web App Manifest, provides meta information about our site 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's home screen. To do this, the file includes information about our site, like what URL to open, what icon and name to use, as well as some other information.

It is a good idea to modify the `short_name`, `name`, `theme_color` and `background_color` to suite our 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 our React code, they will need access to this src directory. For this reason, we 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 further at this code later as we start building with Create React App. 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 other files to import and bundle. All of the imported code will be bundled together and compiled to the `public` directory.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerSer
viceWorker';

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

If we look at the code in our `index.js` file, we see it imports React and ReactDOM, which makes sense. Then it imports some CSS and our `App` component.

We also 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.

Notice 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, which will automatically be added to this list.

The other thing this file contains are NPM scripts we can run with our app. Commonly, we have `npm run dev` and `npm run build`, or similar commands, that will handle different ways of bundling our code, watching for changes and managing servers. Create React App has its 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)

We might want to change the `name` or `version` of our app here, but otherwise we should not need to modify this file.

README.md

This is another standard file. To start, Create React App gives us a generic read me file with a list of the file hierarchy and available scripts.

It is likely we will want to rewrite this for our own needs. I would not suggest shipping an app without updating this file.

yarn.lock

Yarn is a tool similar to NPM from the folks at Facebook, the same folks who also 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 set up and understand its 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 our files for any changes. It will also open `http://localhost:3000/` in our browser since that is the default URL and port that Create React App uses.

Now, if we make any changes to our src code, for example in the `App.js` or `App.css` file, those changes will be detected and our app will be re-bundled. The webpage showing our app will also be automatically reloaded.

Before we start working on a React App with Create React App, we always want to run `npm  start` so that our changes are detected. However, this command is meant for development, not for production.

When we are ready to bundle our app for production, run the following:

```
npm run build
```

This will create a new `build` directory that includes everything from the `public` directory. It will also include an assets folder containing the CSS and JS bundled from our `src` folder.

This build folder is the folder we would ship to production.

So, when working on developing and building our React app or site, make sure to run `npm start` first. Then, when we are complete with a sprint, or the entire project, run `npm run build` to get a final bundled version of the app.

USING CREATE REACT APP ON AN EXISTING PROJECT

So far we have been approaching Create React App as if we were starting a React project from scratch. However, there are instances where we want to fork or work on someone else's project who started it with Create React App.

If this is the case, we'll know they are using Create React App in one of two ways. First, they will hopefully tell us in the `README.md` file of the project. Second, we 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 we get rolling, we have to run `npm install`.

This will pull down all the dependencies we need to our node_modules folder. Since Create React App does this for us when we first create a project with it, we 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 we ever need to run any of our tests, `npm run test` will kick off that process. 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

As we have seen, Create React App offers a number of built in tools behind the scenes. With some projects, we may want to customize the configuration files for the built in tools.

When this happens for a project, we will need to run a one time `npm run eject` command. This command migrates all of the configuration files from being hidden and makes them all available to edit.

We cannot undo the `npm run eject` process.

The only reason to run eject is if we know the tool we 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 our React projects.

Remember, we do not have to eject from Create React App to build or launch our app, just if we 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 we start building a new React project, we will run `npx create-react-app project-name` to kick off the project.

After that first time though, `npm start` will be the command we run from the project directory to start watching our files for changes, and for starting up the development server.

When our project reaches points where we are ready to ship to staging or production, run `npm run build`, and send the `build` directory where it needs to go.

If testing is part of our workflow, we 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 we start working on a project someone else has started with Create React App, we will need to run `npm install` the first time before calling `npm start`.

LET'S PRACTICE!

Now that we have a fundamental understanding of what Create React App does and how its basic commands work, let's take some time to practice using it.

CHAPTER 9.

FIVE EXERCISES WITH CREATE REACT APP

Now that you have a basic understanding of what Create React Does, let's practice using it.

GETTING SETUP

You should already have the practice exercises, but you can download them from Github if you do not already have them:

- https://github.com/zgordon/react-book

The exercises for this chapter are in the "`6-create-react-app`" directory.

Unlike the past exercise files, this directory is completely empty.

For each practice exercise, you will run `npx create-react-app` in this directory. That will in turn create a new directory for each exercise.

You will know if you succeed with the exercises if your React app and files work and look as described below.

PRACTICE EXERCISE #1

The exercise below will help you establish comfort setting up a new project with Create React App and moving into it with the command line.

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 the section, "Setting Up Create React App," from the last chapter.

PRACTICE EXERCISE #2

This next exercise will help you gain confidence starting the Create React App development server and seeing the changes to your code reflected in the browser.

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 src/App.js and save to reload" to something else. On save, you should see the browser refresh with your new value.

PRACTICE EXERCISE #3

This practice exercise will help reinforce the skill of setting up React apps. It will also get you comfortable adding new component files to an app and having them show up working in the browser.

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.

PRACTICE EXERCISE #4

After you complete this fourth exercise, you should feel more comfortable running the build process for getting your app ready for production.

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 its own, different from the development server.

You can now also preview this built version using its 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 it did 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.

WHAT'S NEXT?

Throughout the rest of the book, you 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.

CHAPTER 10.

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. Therefore, they can 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 gets 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 we will often see it included by default in all components created with functions just in case.

Inside of our `Avatar()` and `UserName()` components, we see 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 a 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 flows down through an app, from parent components to children components. If we 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.This reinforces the practice of immutability for all data passed through our app as props. If we need to change prop values, there is a way to do it using something called State that we will look at in an upcoming chapter.

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:

```
<Site />
|
---------------------------
| | |
<Header /> <Content /> <Footer />
| |
---------------- -------------
| | | |
<SiteInfo /> <MainNav /> <Copyright /> <FooterNav
/>
```

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. Then we want to pass the data from that component down through the `<Header />` and `<Footer />` components as props. Finally, we want to pass it 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, we 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 so 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."

```
<Site />
|
---------------------------
| | |
<Header /> <Content /> <Footer />
|
------------------------------
| | |
<SiteInfo /> <Profile /> <MainNav />
|
--------------------------
| | |
<Avatar /> <UserName /> <Points />
```

In the example above, we're just looking at a part of the component hierarchy. In a large application, `<Content />`, `<Footer />`, `<SiteInfo />` and `<MainNav />` would all likely have children components as well. We are focused for this example on the `<Profile />` component and its children.

Let's imagine now that `<Avatar />` and `<Username />` both need access to a User object, but we don't need that User object 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, we need the User object in the `<Footer />` components, or one of its 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, we may need to move around certain functions or original setting of props to higher components if where we 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. Eventually it will become second nature when building React apps. We will also do some practice with this later to become more familiar with it.

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

There is one important rule about props with boolean values that allows us to write less code. It 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 there's 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 we might want to pass all or most of the props from one component into a child. We can use the JavaScript Spread operator to help with this.

Let's imagine for demonstration purposes that we had a `<Profile />` component that received 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 its 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. The spread operator can help us in those cases.

DESTRUCTURING PROPS

In addition to Spreading props, Destructuring props can be helpful as well.

One technique involves destructuring props as they are passed into a component function. With this, we will need to identify specific props, and not just the generic props object that contains all properties.

Compare the code below, which shows how we would have written it previously, with the code after destructuring (see the second example).

```
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 we need. Second, it makes props easier to write (ie `props.username` vs `username`).

With this style, in order to add any new props, we will use 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. It would 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. However, if we needed to pass on some props, and not others, to children, it can be helpful.

```
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 we might not resort to using these conventions on our own when just starting with React, they are patterns we will see. Thus, it can be helpful to know about them in advance.

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. 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 coming chapter, “States in React Explained.”

LET’S PRACTICE!

Now that we have learned 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.

First, let's do some practice with what we have learned about props.

CHAPTER 11.

FIVE EXERCISES IN WORKING WITH PROPS

Now that you have learned a bit about props in React, let's do some practice creating and passing props into components.

GETTING SETUP

At this point, let's assume you already have the practice files from https://github.com/zgordon/react-book.

The exercises for this chapter are in the "`7-props`" directory. Just like with the last set of practice exercises, inside of this folder, you will find a "`starter`" and "`completed`" directory.

PRACTICE #1

For the first practice, you will practice adding props to a component. To get set up, open the "`07-props/starter`" project, run `npm install`, and 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 on line 16 that the `<User />` component is already set up to use these props, they just need to be passed.

Once this is complete, 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 get you comfortable with the first step of working with Props, which is passing them into a component.

PRACTICE #2

In the second practice, you will take a step further with your practice of adding props to a component. To get set up, 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 you created as a prop into the `<Post />` component.

Finally, come down to where the `<Post />` component is set up. 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 set up being rendered on the page. You can try modifying the original post object you set up to test that it all works properly.

This practice exercise is great to help you set up more of the parts of props on your own. You define the variables used as props, set the props on a component, and then modify a component to work with props. This may be worth practicing a few times on your own.

PRACTICE #3

Now that you have practiced a little with adding and using props, you will have you write a bit more of the code on your own. You will also pass props down two levels of components rather than just one.

To get set up, 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 set up 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, you 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 set up, 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 its 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 set up 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 the final exercise, you pull together what you have done in the examples above into one larger example.

Set up 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 you did 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 set up 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 some practice. However, props are only part of the data flow architecture in React. Another essential part is state. State can help you update your props and components with dynamic data.

In the next chapter, you will explore State in React. You will see how it works and how it can make your apps way more powerful than just using Props alone. You will also look at how to write class based components (rather than functional components) to support the new features state brings.

CHAPTER 12.

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 our 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 lets 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 our 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 access and update state throughout our app.

By default in React, we do not have a global state object, but every component can have its own state object. This means that each component has the option to set and manage its own state.

If needed, a component can pass the value of its 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. Then we'll 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.

Here's an example of a functional component:

```
const MyApp = () => (
  <h1>Welcome</h1>
);
```

However, in order to demonstrate how state works, we will need to use classes for our components. *There is a new hooks system in React you may want to explore after completing this book that allows functional components to receive and update state as well.*

Here is an example of a component created via a class set up 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 it down a bit to see what is happening here.

First, we are naming our component `MyAppClass`, which makes it available for us to import and call it 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 the React core that offers base functionality for building components with classes. All components we build with classes should extend `React.Component` so they all include the built in methods that React offers, like setState and the Lifecycle Hooks we will learn about later.

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.

Although we have to return one element or component, we can of course nest other elements or components inside that main one like we saw 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, you've already set it up in the initial state object.

It is important to mention once again, that we never update `state` directly. Always call `setState()` to modify a value in `state`. If we do not, React will not be able to track the changes and appropriately update anywhere the value from `state` appears.

UPDATING STATE WITH EVENT HANDLERS

Most of the time we do not hard code `setState()` calls into our `render()` function. Instead, we 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 an initial value for our count state of `0`. Then we see a `handleClick()` function that will serve as our event handler for increasing the value of count and calling setState.

We can see here we call `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 the prop value 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 we 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.coun
t + 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, we've broken it 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 its 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.coun
t + 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 is to take the function that updates state, and pass it down as a prop, as well as 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 its own component. However, the event handler it will call when clicked is passed down to it via props. So the button component has no idea of what the event handler will do.

In the `Counter` class, we have broken the event handler out into its 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 gets reset to the default values.

So there are times when we might want to make state persistent, or to last between page refreshes and 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 our own code to do this, but several packages and libraries exist that help us easily do this.

In the case of local storage, the values of our 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 our 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 our app are shown. Session storage can be helpful if we know we want to wipe state as soon as a user is done using our site.

Using a database, like the popular Firebase, allows us to store our state values outside of the user's browser for more reliability. It should be noted here that there is a distinction between having a database store content for our sites that we've loaded 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 us 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.h
andleClick}>+</button>;
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("roo
t");
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 then 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.

CHAPTER 13.

FIVE EXERCISES IN WORKING WITH STATE

Although you have learned about state in theory, it's good to practice working with state and updating it. The following practice exercises will help you get comfortable with this.

GETTING SETUP

The exercises for this chapter are in the "`8-state`" directory. In there you will find "`starter`" and "`completed`" directories.

PRACTICE #1

In this exercise, you will practice creating a property in state and rendering it on the page.

To get set up, open the `starter` directory and run `npm install` and 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.

If done correctly, you will see your username displayed. This practice exercise will help you create values in state and render them to the page.

PRACTICE #2

In this practice exercise, you will work with updating the state using `setState` and an event handler. To get set up, 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 set up, and before the `render()` function, create an arrow function called `handleUsername()` that takes the event object as a parameter. It should 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`). You will get this value from an input form field when you 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 you 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.

If everything works, you will update the state whenever you change the value of the username input field.

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 you have practiced creating and setting state, you 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()`. Also 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`. This 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.

If everything works, the state should update whenever the value in the name fields change.

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, you practice passing both the value of state and the event handlers to update state down into children components. You also get to practice again 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 return an `<h2>` with `text` from `props`.

Second, create a functional component called `Button` that also accepts `props`. Have it return a `<button>` element. 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.

This exercise helps you put together passing state and the ability to update state down into children components.

PRACTICE #5

In the final exercise, you take your last practice exercise and add local storage support for state so that it keeps its value on page refresh and 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 you can use it in your component.

Come down into the `Practice5` `render()` method. Call `<SimpleStorage />` and set a `parent` prop equal to `this` just like what you saw in the previous chapter.

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.

This exercise gives you some practice keeping state persistent in React.

WHAT'S NEXT?

Now that you have worked a bit with setting, updating and passing around state, let's turn our attention to another important topic: the component lifecycle in React.

This will give you some new hooks you can use to update (or not update) your components once they are loaded to the page.

CHAPTER 14.

THE COMPONENT LIFECYCLE EXPLAINED

A lifecycle, in broad programming terms, refers to 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.

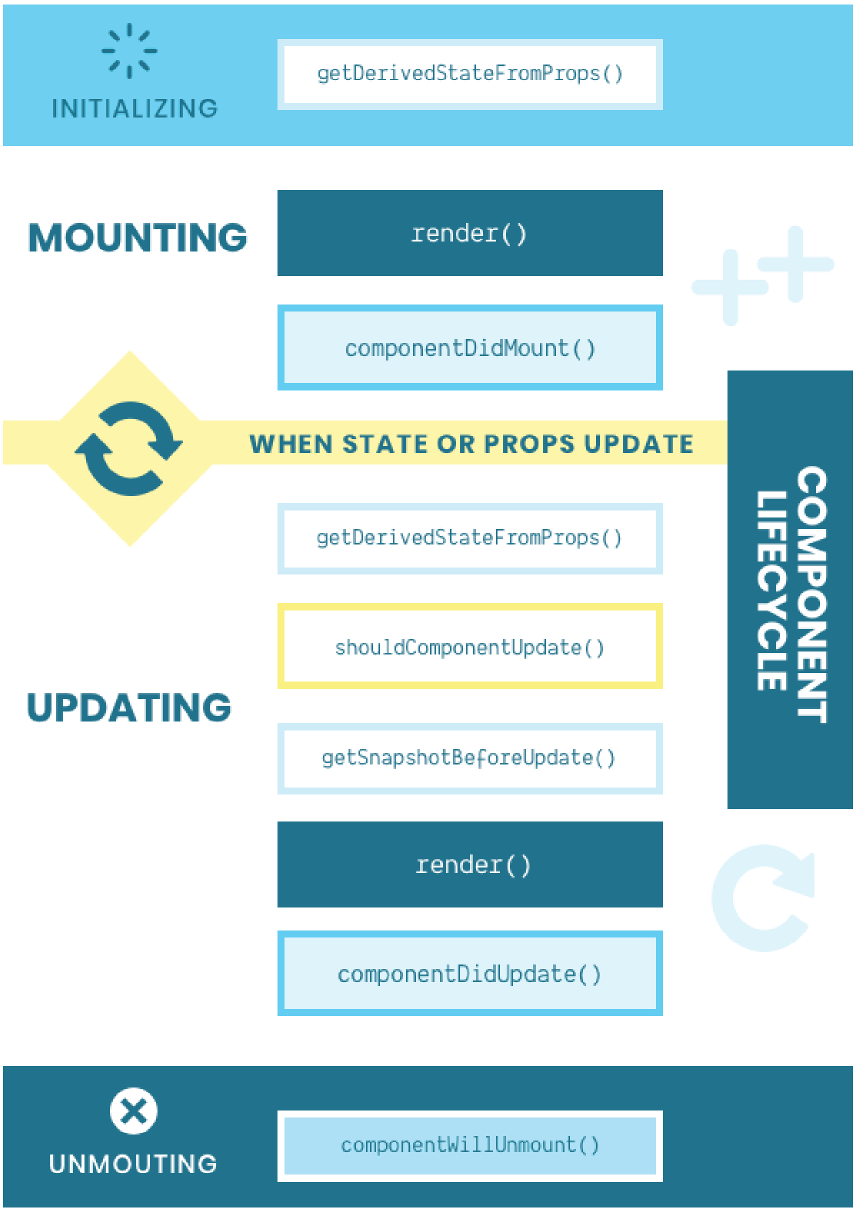
In this chapter, we will explore the component lifecycle hooks and when we might use them.

> 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 set up and

props and state are being passed and set up

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 these 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, it is being prepared to be ready. There is just one lifecycle hook available at this time.

getDerivedStateFromProps()

This lifecycle hook is executed while our component is being initialized, and before it is rendered to the page. It takes two parameters, which React automatically populates for us:

```
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 we will need to do something using this hook. In fact, the React documentation encourages us to use other hooks besides getDerivedStateFromProps() whenever possible.

However, if we 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 set up, and that is available within getDerivedStateFromProps() as well.

As mentioned, use of this component is rare. Make sure that 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 we 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, we 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 we 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 we 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 that 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()`, we 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 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 our production code, we 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 a good example of how componentDidMount might be used and some considerations to remember.

Another time to use `componentDidMount()` is when we have JavaScript that depends on content already being rendered on the page. For example, we want to call 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, we 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 we can select, modify, and attach event handlers to DOM elements outside of where React is loaded. The example uses 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>;
  }
}
```

We can see here that inside the `componentDidMout()` component, 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 changes, or any item in state within a component changes, 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:

- `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 lets us 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 do 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 we 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 `shouldComponentUpdate()` 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 that changed with this example is we placed 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 when an update to props or state occurs. This lets us get any DOM information from the page before the update occurs. We can then pass this information or anything else we need to `componentDidUpdate()`, which will be called right after the component updates.This gives us a tunnel to pass data from before a component updates to after.

This can let us get a "before" snapshot of anything on the page, often window, 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"]
  };

// dd 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 he
ight
componentDidUpdate(prevProps, prevState, snapsho
t) {
  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 we have multiple values, it may make sense to return an object or array. However, we 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 we will get values from the page, 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 gets called again when a change to props or state occurs and the lifecycle methods above have already run: `getDerivedStateFromProps()`, `shouldComponentUpdate()`, `getSnapshotBeforeUpdate()`.

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 we need to make modifications to the DOM, 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. In order 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, snaps
hot) {
  // 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.scrollHeight);
}

render() {
  return (
    <div
      id="posts"
      style={{
        overflow: "scroll",
        height: "100px",
        border: "1px lightgray solid"
      }}
    >
      <ol>
          {this.state.posts.map(post => <li>{pos
t}</li>)}
        </ol>
      </div>
    );
  }
}
```

The first difference here is that we are assigning the timer to `this.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.

We will not use `componentDidUpdate()` in every component, but it is one of the more commonly used components in React. Remember that we 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 gets called right before a component is removed from the DOM. This can be helpful for cleaning up anything allocated in memory, like timers and event listeners added to non-React elements in the DOM. We 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 its own component, and it counts 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.displ
ayTimer });
};
```

```
render() {
  return(
    <div>
      {this.state.displayTimer ? (
        <CountDown
toggleTimer={this.toggleTimer} />
      ):(

<button onClick={this.toggleTimer}>Start Timer</
button>
      )}
      </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.toggleTime
r();
}

componentWillUnmount() {
  clearInterval(this.timerID);
  console.log("Timer about to unmount!");
}

render() {
    return <p>Wait {this.state.count} more secon
ds..</p>;
  }
}
```

We will not need to use `componentWillUnmount()` often, but once again, it is good to know that it exists and how and when we 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 we will likely use:

- Mounting – `componentDidMount()`
- Updating – `componentDidUpdate()`

And here is a list of the less common hooks we 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()`. 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 get comfortable using some of the hooks in practice.

We will focus mostly on the more common hooks, but we want to make sure we have a chance to use some of the less common hooks as well.

CHAPTER 15.

FIVE EXERCISES WITH THE COMPONENT LIFECYCLE

Now that you have learned about the Component Lifecycle, let's do some practice hooking into the lifecycle methods.

GETTING SETUP

To get started with these exercises, open the "`9-component-lifecycle/starter`" directory in your code editor, run `npm install`, and then `npm start`.

You also have a "`9-component-lifecycle/completed`" directory if you get stuck and want to compare your code to some working code.

PRACTICE #1

In this first exercise, you 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 you 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, you 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 via the WordPress CMS REST API:

```
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. If everything is set up properly, you will see the posts render to the page once the API call is complete.

This practice exercise helps you 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, you 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, you will build a counter with a bar chart that counts up to 20. However, you 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, you will work with `the componentDidUpdate()` lifecycle hook to tell when values in props or state have changed.

To do this, you will build off of your 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 you whether or not state has been updated. If it has been updated, log out "State Changed!".

Now 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 you 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 you have an example of checking to see if state has been updated and if 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 now have a model for how to use `shouldComponentUpdate()`.

PRACTICE #5

In this final exercise, you will look at how to hook into `componentWillUnmount()` to take actions when a component is about to be removed from the DOM.

In this example, you modify your counter and bar chart components to automatically increase the points every 300 milliseconds until it reaches ten. Then you remove the bar chart component and display a message that the goal has been reached. When the bar chart is removed from the page, you also need to stop the timer, so it does not continue to count in memory.

To set up, 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.

WHAT’S NEXT?

Now that you have learned a good deal about React and have practiced using it, let’s turn our attention to building a more complex project than the simple examples you have worked with so far.

This will allow you to pull together everything you have learned. You’ll also learn some new practices and helpful libraries for working with React.

PART III.

A REACT PROJECT

In this section of the book, we will build and launch a complete project using React and some other helpful libraries and tools.

The project involves eleven steps:

Step 1 – "Listing Content From State" starts us off building a simple version of our final project that starts with all of our data hard coded in our state.

Step 2 – "Routing and Single Content Views" introduces the popular React Router. This tool will help us create what content loads based on the URL and Link components.

Step 3 – "Adding a Content Form" shows us how to build out a form for editing our content using the Quill editor. We will set up the form so that we can use it for adding new content as well as editing existing content.

Step 4 – "Adding Flash Messages" gives us a quick way to display messages to the user that quickly expire after a few seconds. We will use this for saved, edited and deleted messages.

Step 5 – "Updating Content" is the section where we will take our posts that already exist and load them into the same form we built before for adding new content. We will learn how to rewire existing code for new functionality while also not needing to repeat code.

Step 6 – "Deleting Content" is an important section where we talk about how to go about deleting data but also discuss possible UX considerations for confirming the deletion of data.

Step 7 – "Maintaining Persistent State With Local Storage" implements a helpful local storage package that will sync our state with local storage for offline use and caching purposes.

Step 8 – "Authenticating With a Firebase Database" goes over the important topic of how to authenticate with a 3rd party library using React. The library we use is Firebase, a free and powerful database and authentication library we can use with our React projects.

Step 9 – "CRUD and Live Syncing With Firebase" takes the integration with Firebase further and shows us how to create, read, update, and delete data from Firebase. We also look at how Firebase gives us a live sync with our data that doesn't need to be refreshed.

Step 10 – "Deploying The Project" completes the process with a demonstration of how we can build our app for deployment. We also look at how to integrate with the popular hosting provider Netifly that lets us ship to staging and production all from the command line.

Step 11- "Refactoring Your Code" gives a look at the common process of improving a code base even after it has shipped to production. We will look at how to make our code more efficient and cleaner without changing any functionality.

CHAPTER 16.

REACT PROJECT INTRODUCTION

Now that you have learned the core functionality of React and completed some practice exercises with it, you will build a larger project from scratch. You will use what you have learned, along with some new libraries and techniques.

WHAT YOU WILL BUILD

The project you will build is a website that displays blog posts pulled from a database. You will also add the ability to log in to the site to add, edit, and delete posts.

Here is a rough outline of the steps:

1. Build a static version of the site
2. Add in routing with the React Router library
3. Build a Post Form using the Quill editor library
4. Add the ability to edit and delete posts
5. Connect to a Firebase database

This project will allow you to continue to practice what you have learned so far. You will also learn some new things along the way, including some helpful React related libraries and tools.

STARTING THE PROJECT

You will use Create React App to start our project.

Inside of the main files repo found here, https://github.com/zgordon/react-book, you will see a `project` folder.

Inside of the main `project` folder, there is a folder for each completed step of the project. If you want a starter folder, you can start with the previous steps folder.

However, it is recommended that you create your own directory for this project and follow along start to finish with your own project folder.

To do this, open the project folder and run the following:

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

This will spin up a new directory called `my-project` that you can continue to use for the duration of the project.

WHAT'S NEXT?

In the next chapter, you will build a static version of your blog site. This will involve starting with a few posts in state and then rending a list and single page view for the posts.

CHAPTER 17.

REACT PROJECT STEP #1. LISTING CONTENT FROM STATE

In this first step, you are going to create a basic site that pulls some posts from state and displays them on the page.

GETTING STARTED

You should already have a project directory you created with Create React App in the project introduction. If you do not already have this folder, you can run the following:

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

Once you have this directory created, go ahead and open it in your code editor.

Then run the following to start the development server:

```
npm start
```

Now you should be able to follow along with the changes you will make next.

STARTING WITH A FRESH APP COMPONENT

Open the App.js file and delete its contents. Then start over with the following simple base:

```
import React, { Component } from "react";
import "./App.css";

class App extends Component {
  render() {
    return (
      <div className="App">
        APP HERE
      </div>
    );
  }
}
export default App;
```

This should display APP HERE on the page and not much else. Next, add in a header component for your app.

CREATING THE HEADER COMPONENT

Create a new directory in the `src` folder called `components`. Inside of the components directory, create a new file called `Header.js`. This will serve as the header and navigation for your app.

Inside of the `Header` component, import React from "react". Then create a functional component called `Header` and export it.

Have the component render a `<header>` element with a class of `"App-header"`. Then, inside of that, place a `<ul>` with a class of `"container"`.

Finally, place an `<li>` element with the words "Site Title" inside of them.

```
import React from "react";

const Header = props => (
  <header className="App-header">
    <ul className="container">
      <li>Site Title</li>
    </ul>
  </header>
);
export default Header;
```

Your final code should look something like the component above.

Once you have this component created, go back to the `App.js` file and `import Header from "./components/Header"`. Then call `<Header />` inside of the `App` component main `<div>`.

You should see the `Header` component load on the page.

CREATE THE POSTS COMPONENT

Now you will create a new component file named "`Posts.js`" in the components folder.

Import React at the top and create a functional component named `Posts` that destructures posts from props. Have `Posts` return an `<article>` with the classes `"posts"` and `"container"`. At the end of the file, make sure that the `Posts` component is the default export.

```
import React from "react";

const Posts = ({ posts }) => (
  <article className="posts container">
    <h1>Posts</h1>
  </article>
);
export default Posts;
```

Now you want to display a list item if no posts are displayed. It should go after the `<h1>` and look something like this:

```
<article className="posts container">
  <h1>Posts</h1>
    <ul>
      {posts.length < 1 && <li key="empty">No
posts yet!</li>}
    </ul>
  </article>
```

Next you want to map over the posts and display an `<h2>` with the title of each post. It will look something like this:

```
<ul>
  {posts.length < 1 &&<li key="empty">No posts y
et!</li>}
    {posts.map(post => (
      <li key={post.id}>
        <h2>{post.title}</h2>
      </li>
    ))}
  </ul>
```

Note that you have to set keys for all of the list items, and you will use the post id for that.

CALLING <POSTS /> FROM <APP />

To keep your app simple for the moment, you will load some posts into state in the `<App />` component. Come into `App.js` and add the following posts into state:

```
class App extends Component {
  state = {
    posts: [
      {
        id: 1,
      title: "Hello React",
    content: "Lorem."
  },
{
  id: 2,
    title: "Hello Project",
    content: "Tothe."
  },
{
  id: 3,
    title: "Hello Blog",
    content: "Ipsum."
  }
]
};
  render() {
// Do not change the render method yet
  }
}
```

Next you will import the Posts component at the top of `App.js` and call `<Posts>` after the `<Header>`. Make sure to set the posts props to the props in state:

```
<Posts posts={this.state.posts} />
```

Now you should see the posts listed out to the page.

ADD SOME BASIC CSS

To make your app a little bit nicer, add the following CSS into your App.css file:

https://github.com/zgordon/react-book/blob/master/project/blog-step-01/src/App.css

This will give you some good default styles for the rest of your project.

WHAT'S NEXT?

Now that you have your posts listed out, and your basic app set up, you are going to add routing and single page views to your app.

CHAPTER 18.

REACT PROJECT STEP #2. ROUTING AND SINGLE CONTENT VIEWS

In order to help you handle single page views and navigate between pages on your site, let's introduce routing. Routing allows you to use true URLs in your app (like app.com/page) so that you don't have to worry about writing all the event handlers to deal with this.

Rather than writing your routing from scratch, you will use the popular React Router library. This will save you a lot of time and introduce you to React Router, an important library in the React ecosystem.

If you followed along successfully with the last chapter, you can continue on with the same code base. Or, if you would like to start fresh, you can start from the complete step 1 files and continue from there. Make sure to run `npm install` if you are starting from one of the example directories.

SETTING UP REACT ROUTER

The first step for setting up React Router is to install the package. Open your project folder, and then run the following:

```
npm install react-router-dom
```

Now start up your development environment with the following:

```
npm start
```

To use React Router, open up App.js and import the following:

```
import {
  BrowserRouter as Router,
  Switch,
  Route
} from "react-router-dom";
```

This will give you the necessary components you need to set up your routing.

ADD SLUGS TO POSTS IN STATE

React Router wants you to have true permalinks as fall backs. To create proper links, you will want to add slugs to your posts in state.

Modify the state in the `App` component as follows:

```
posts: [
  {
    id:1,
    slug:"hello-react",
    title:"Hello React",
    content:"Lorem."
  },
  {
    id:2,
    slug:"hello-project",
    title:"Hello Project",
    content:"Tothe."
  },
  {
    id:3,
    slug:"hello-blog",
    title:"Hello Blog",
    content:"Ipsum."
  }
]
```

Now you can continue with your Router setup.

ADDING ROUTER WRAPPER AND ROUTES

The next step is to come down into your `App` component render return and wrap the app div in a `<Router>` component like this:

```
<Router>
  <div className="App">
    {/* Don't change inside here yet */}
  </div>
</Router>
```

This will identify your app as being managed by the React Router.

Now you want to only call your `<Posts />` component when the main route or root of your site is accessed. Remove your current `<Posts />` component call with the following:

```
<Switch>
  <Route
    exact
    path="/"
    render={() => <Posts
posts={this.state.posts} />}
  />
</Switch>
```

What this does is check to see if you are on the main route of your site (locally http://localhost:3000/). If it is, you will call the `<Posts />` component as you had previously.

Next let's set up links around your post titles to link to a single view component.

ADDING LINKS TO POSTS AND HEADER

Open up the Posts component and import the following at the top:

```
import { Link } from "react-router-dom";
```

React Router provides a `<Link>` component to use wherever you want links in your apps that are tied to routes or URLs.

Then modify the `<h2>` in the posts map as follows:

```
<h2>
  <Link to={`/post/${post.slug}`}>{post.title}</
Link>
</h2>
```

The code above creates a link to a URL like /post/slug using the post slug you set in state.

If you open up your app now and click on one of the links, you should see the URL of the site change and no content display.

You will get to creating the single post view next, but first let's add a link to the root of your site in the header.

Open `Header.js` and import Link again from react-router-dom.

Then change the list item of My Site to the following:

```
<li key="home">
```

```
  <Link to="/">My Site</Link>
</li>
```

This will give you a link to your homepage and the list of all posts.

If you now click on a post title, it will take you to a blank page. Clicking on the My Site link in the header will take you back to the homepage.

Next let's build a single post view component.

SINGLE POST COMPONENT

Create a new file `"/src/components/Post.js"` with a functional component called `Post` that displays an `<h1>` with the post title and a div with the post content.

It should look something like this:

```
import React from "react";

const Post = ({ post }) => (
  <article className="post container">
    <h1>{post.title}</h1>
    <div>{post.content}</div>
  </article>
);
export default Post;
```

This simple component will handle loading your single post view. Now you just have to wire the Router up to load this component when the correct URL is accessed.

CONFIGURING SINGLE POST ROUTE

Come back into `App.js` and import Post from `"./components/Post"`.

Next, after the `<Route>` for the `Posts`, create a new `<Router>` like the following:

```
<Route
  path="/post/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    return <Post post={post} />;
  }}
/>
```

This route is a little more complicated, so let's break down what is happening.

First, you are defining the path that will load your `Post` component. The path is equal to /post/ and then the post slug.

If this path matches, render will be called. Inside of your render setting, you are checking to find the post that matches the one in the URL. React Router automatically provides you with a prop of match, which lets you find the current slug from the URL with the following:

```
props.match.params.postSlug
```

Finally, once you get the correct post from state, load that as a prop into your `<Post />` component.

Now, when you test your site in the browser, you should be able to click on a post and see that post loaded. This gives you simple routing using the basic conventions of React Router.

SETTING UP A 404 PAGE

So far the URLs in your site work pretty well. The homepage loads Posts, and if the URL is /post/slug, it will load the Post component with the specific post matching the slug in the URL.

However, what happens if someone accesses a URL that does not match a post? Currently your site will break in a number of different ways.

So let's set up a 404 fall back page to load if an incorrect URL is accessed.

Create a new component called `"NotFound.js"` in the components folder. Set up the component to look something like this:

```
import React from "react";
import { Link } from "react-router-dom";

const NotFound = () => (
  <article className="not-found container">
    <h1>404!</h1>
      <p>
        Content not found. <Link to="/">Return
to posts</Link>
    </p>
  </article>
);
export default NotFound;
```

Now import that `NotFound` component into `App.js`. Right after the `Post` route, add the following:

```
<Route component={NotFound} />
```

Here you see a simplified use of `Route` where you can just give it the name of a component to load if no props need to be passed.

You will also want to update the `Post` route to return `NotFound` if no posts match the slug accessed. Here is what the updated Post Route should look like:

```
<Route
  path="/post/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    if (post) return <Post post={post} />;
    else return <NotFound />;
  }}
/>
```

This will ensure that if someone accesses a URL that does not exist, they will see our 404 `NotFound` component.

WHAT'S NEXT?

Now that you have routing and single page views in your app, let's move on to creating a form that will let you add new posts manually rather than having to hard code them from state.

CHAPTER 19.

REACT PROJECT STEP #3. ADDING A CONTENT FORM

In this step, you are going to create a form so that you can manually add new posts to state. This begins your CRUD operations of Create, Read, Update, and Delete that are always helpful to know how to set up as part of interfaces and apps.

To do this, you will use a simple input field for the title and an editor called Quill that will let you add rich text to your text field. Other rich text editor options exist, but Quill is a popular and easy to use option.

GETTING STARTED

If you are following along with the steps of the project from the last chapters, you can continue with your same code. You can also start fresh with the `"/projects/step-2/"` completed files, and by runing `npm install`, and then `npm start`.

SETTING UP THE QUILL EDITOR

To get the Quill editor loaded, first install the package. To do this, run the following:

```
npm install react-quill
```

This will install the Quill editor as an easy to use React component.

CREATING A POSTFORM COMPONENT

Create a new file in the components folder named "`PostForm.js`" and import the following at the top:

```
import React, { Component } from "react";
import { Redirect } from "react-router-dom";
import Quill from "react-quill";
import 'react-quill/dist/quill.snow.css';
```

This will give you React, a `Redirect` component from React Router, and the `Quill` editor component. It also imports the needed CSS for the Quill editor to work properly.

Next create a `PostForm` class based component like this:

```
class PostForm extends Component {
  render() {
    return (
      <form className="container">
        <h1>Add a New Post</h1>
        {* Title Fields Here *}
        {* Quill Editor Here *}
        <p>
          ;<button type="submit">Save</button>
        </p>
      </form>
    );
  }
}
export default PostForm;
```

You will come back and add more to this form shortly. But first, let's add a new route to your app, as well as a link in the header that links to the post form.

ADDING ROUTE AND LINK TO POST FORM

To start off, open the `Header.js` component and add in a new link to your post edit form.

Add the following link to after the My Site link:

```
<li>
  <Link to="/new">New Post</Link>
</li>
```

Now you need to come into your `App.js` and add a new `<Route>` for the link you created to `/new`.

After the Route to your single post view, add the following route:

```
<Route
  exact
  path="/new"
  component={NewPostForm}
/>
```

You will come back and edit this further at a later point, but this should let you click on "New Post" in the header and see the new post form load.

Now you will build the rest of the form.

BUILDING THE POST EDIT FORM FIELDS

Now let's come back into `PostForm.js` and add the form elements for the post title and content.

To start, add in title and content into your `PostForm` component state:

```
state: {
  title: "",
  content: "",
}
```

Next add in an input field and label for the title:

```
<p>
  <label htmlFor="form-title">Title:</label>
  <br />
  <input
    id="form-title"
    value={this.state.title}
    onChange={e => this.setState({ title: e.targ
et.value })}
  />
</p>
```

This will let you add a title for your post. It will also update the title in state whenever the value of the field is changed.

Below that, add the code for your Quill editor. That will look like this:

```
<p>
  <label htmlFor="form-content">Content:</label>
</p>
<Quill
  onChange={(content, delta, source, editor) =>
{
    this.setState({ content: editor.getContents(
) });
  }}
/>
```

The `Quill` editor field is slightly more complicated to set up than the normal input field. Notice that `Quill` makes available a number of variables by default into the `onChange` event handler.

You can use the editor variable passed in along with the `getContents()` method to get the contents from the `Quill` editor and set it to the content in state.

Again, this code is specific to `Quill` and saves Deltas, not normal HTML, as the content. Later you will have to go back and update your single page view to display those saved Deltas rather than normal HTML.

The final field you need to add is the submit button. It should look something like this:

```
<p>
  <button type="submit">Save</button>
</p>
```

Now, when you click on "New Post" in the header, you should see your completed post form. The final step is to hook up the event handlers to actually save the post.

ADDING THE POST FORM EVENT HANDLER

Within your `PostForm` component, let's write an event handler called `handleAddNewPost()`. The first thing the handler should do is prevent the event default from happening, which will prevent the form from submitting and refreshing the page.

Next you want to write a conditional statement to check if they have entered in a title. If there is not a title, you can display an alert message saying a title is needed.

If there is a title, then you want to create a new post object and assign it the title and content values from state. Just to test, let's start off logging this post to the screen.

So you should have an event handler in your `PostForm` component that looks something like this for the moment:

```
handlePostForm = e => {
  e.preventDefault();
  if (this.state.title) {
    const post = {
      title: this.state.title,
      content: this.state.content
    };
    console.log(post);
  } else {
    alert("Title required");
  }
};
```

Before this will work, you need to modify your form element to call the function on submit. That should look like this:

```
<form className="container" onSubmit={this.handl
ePostForm}>
```

Now when you enter in a title and content into the form, you should see a post object logged out in the console. The title should be the text you entered, while the content will be a Delta representation of the content you entered.

SAVING A NEW POST

Your `handlePostForm()` will currently log out your new post object. However, you want to save it back into the state for your main App component. You do this so it can be used throughout the entire app.

You will add the function to save a new post to state in your `<App />` component. Then you will pass it down as a prop into your `<PostForm />` component.

In your `App` component, create a method `addNewPost`. This will take `post` as a parameter. Then you have to do a few minor actions, like creating a new ID and slug for your post. You will add these two properties to your post. Finally, you will add this new post to the existing list of posts.

Your final method should look something like the following:

```
addNewPost = post => {
  post.id = this.state.posts.length + 1;
  post.slug = encodeURIComponent(
    post.title
      .toLowerCase()
      .split(" ")
      .join("-")
  );
  this.setState({
    posts: [...this.state.posts, post]
  });
};
```

Notice in the first line that you set a new ID by simply getting the length of the current array of posts and add one. This is not a production ready solution, but it will do for now. Later you will get your IDs set automatically via a database.

Next get a slug for your post by taking the title and replacing any spaces with a hyphen, and finally, URL encoding it. Again, this may not be 100% bullet proof, but it does a solid job getting what you need for your app here.

Finally, set the state for posts as equal to the current array of posts plus our new one. As you can see, you are using array destructuring to accomplish this.

Now that you have your `addNewPost` function, you can modify your new post form route to look like the following:

```
<Route
  exact
  path="/new"
  render={() => <PostForm addNewPost={this.addNe
wPost} />}
/>
```

With your `addNewPost` function passed into your `PostForm` component as a prop, you can come back into `PostForm.js` and call this function from your `handlePostForm` function.

So your handlePostForm should now look like this:

```
handlePostForm  = e => {
  e.preventDefault();
  if (this.state.title) {
    const post = {
      title: this.state.title,
      content: this.state.content
    };
    this.props.addNewPost(post);
  } else {
    alert("Title required");
  }
};
```

Now, if you fill out the form and press submit, nothing on the page changes. But, if after submitting the form, you click on "My Site" to show all of the posts, you will see that the post has been added.

REDIRECTING FORM AFTER SUBMIT

With the app you are building, when someone submits the form, you want to redirect back to the home page so they can see the new post listed.

You will accomplish this using a React Router `<Redirect />` component and a value in state called “saved” that you set to false by default and then switch to true after the form has been submitted.

To start, add a value for saved to state and set it to false by default:

```
state = {
  title: "",
  content: "",
  saved: false
};
```

Now, after you call `this.props.addNewPost(post)` in your `handlePostForm` handler, you want to also set the state of saved to true:

```
this.setState({ saved: true });
```

The final step in this approach is to set a conditional statement inside your render method. If saved is true, redirect back home, and if not, return your form.

```
render() {
  if (this.state.saved === true) {
    return <Redirect to="/" />;
  }
  return (
  // Leave return unchanged
  )
}
```

This gives you a pattern where you have a value in state set to false until the form gets submitted. Then, once the form is submitted, the value switches to false and the `<Redirect />` component automatically gets called sending the user back to the homepage.

This completes your new post form, but you still need to get the Deltas from the `Quill` editor to display in the single `Post` component.

DISPLAYING DELTAS IN SINGLE POST COMPONENT

For help displaying deltas, go ahead and stop your production server and import the following library:

```
npm install quill-delta-to-html
```

Then start your server back up again with `npm  start`.

Next go into your `Post.js` file and import the following right below importing React:

```
import { QuillDeltaToHtmlConverter } from "quill-
delta-to-html";
```

Then modify the Post component to the following:

```
const Post = ({ post }) => {
  const converter = new QuillDeltaToHtmlConverte
r(post.content.ops, {});
  const contentHTML = converter.convert();

return (
  <article  className="post container">
    <h1>{post.title}</h1>
    <div
      className="content"
      dangerouslySetInnerHTML={{nbsp;__html:nbsp
;contentHTMLnbsp;}}
    />
    </article>
  );
};
```

Here you are passing your Delta post content into `QuillDeltaToHtmlConverter()`, then you need to call `.convert()` on this content. This will give you HTML in `contentHTML`.

When displaying HTML from a variable into a React app, you have to use a special property called `dangerouslySetInnerHTML`.

So rather than having something like this:

```
<div>
  {contentHTML}
</div>
```

You actually have to do something like this:

```
<div
  dangerouslySetInnerHTML={(() => ({ __html:
contentHTML }))()}
/>
```

This is to remind you that you are doing a potentially dangerous action of letting raw HTML run on the page.

However, with these modifications to your Post component, you can now display Deltas from your Quill editor.

Since this is complete, your last step will be to set your initial state of posts in `App.js` to an empty array and add all of your posts manually via your form.

WHAT'S NEXT?

The next major step for your project is to add the ability to edit and delete your posts. However, first you are going to add the ability to display messages when you have saved, updated, or deleted your content.

So let's take a look at how to add flash messages to your project next.

CHAPTER 20.

REACT PROJECT STEP #4. ADDING FLASH MESSAGES

Flash Messages in an app let a user know an action has taken place. In traditional apps with page refreshes, flash messages would be available after an action was submitted even if it caused a page refresh.

In your single page app design, your flash messages will appear for a few seconds and then disappear. You will use updates to state and `setTimeout` calls in order to do so.

GETTING STARTED

If you have successfully followed along with the previous steps, you can continue with the same code.

If you would like to start fresh, you can use the completed files from the last step, run `npm install`, and then `npm start`.

THE MESSAGE COMPONENT

Your message component will contain a few predetermined messages that you can display by updating the state in your main `App` component. You will also use a simple timer and some CSS to hide the message after it has displayed a few moments.

To start, create a new `Message.js` file in the components directory. Import React and then set up a functional component called `Message` that destructures `type` from props.

You also want a `messages` object in your component that has the various messages you can display. In the render method, display the appropriate message based on the prop passed.

Here is what the completed component will look like:

```
import React from "react";

const Message = ({ type }) => {
  const messages = {
  saved: "Post has been saved!",
  updated: "Post has been updated!",
  deleted: "Post has been deleted."
};
  return (
    <div className={`App-message ${type}`}>
      <p className="container">
        <strong>{messages[type]}</strong>
      </p>
    </div>
  );
};
export default Message;
```

You can see here that you pass in a prop called `type` with a value of “saved,” “updated,” or “deleted.” This will determine what message displays. You can later add additional messages using the same pattern as above.

In the `render()` method, display the `type` prop as a class in the wrapper div for styling purposes. Then display the specific message you want using the bracket format of calling object properties.

If you have not seen the `messages[type]` pattern before , it is how you can call an object property when the name of the property is a variable rather than something we can hard code.

Now that you have your message component created, let's look at how to integrate it into your app.

CONDITIONALLY RENDERING MESSAGES

At the top of your `App.js` file, import the Message component.

Then, in the App component, add `message` to state with a default value of `null`.

```
class App extends Component {
  state = {
    posts: [],
    message: null
  };
  // The rest stays the same for now
}
```

Now you can write some conditional code to only render the Message component if `this.state.message` is not equal to `null`.

Come down into the App `render()` method and add the following code right under the `<Header />`:

```
{this.state.message && <Message type={this.state
.message} />}
```

This will cause the `Message` component to render and receive the message type as props.

Next you will display a saved message when you add a new post.

Displaying the Saved Message

To display a message when a post has saved, you will come into your `addNewPost()` function in your `App` component.

Look for where you call `this.setState` in `addNewPost`. Update it to not just update the posts, but also set message in state to "saved."

```
this.setState({
  posts: [...this.state.posts, post],
  message: "saved"
});
```

This will cause your saved message to display. However, it will not disappear unless you set message in state back to null. To accomplish this, call `this.setState` inside of a `setTimeout` function.

This will go right at the end of your `addNewPost` function:

```
this.setState({
  posts: [...this.state.posts, post],
  message: "saved"
});
setTimeout(() => {
  this.setState({ message: null });
}, 1600);
```

Now, when you save a post, it will set the message state to "saved" to display your saved message. Then, after 1600 milliseconds, it will change the message in state back to null, making the `Message` component unmount.

POSSIBLE FURTHER MODIFICATIONS

You can use this same pattern when displaying other messages and then hiding them. It would also be possible to build a

`Message` component that includes the `setTimeout` call inside of it, but it would be a little more complex.

There are also several Flash Message packages that you can easily add to use in your app so that you don't have to build them yourself.

THE FINAL CODE

If you get stuck along the way, or would like to see the completed code, you can access it via the course repo here:

https://github.com/zgordon/react-book/project/blog-step-04

WHAT'S NEXT?

Now that you have messages set up and working in your app, let's move on to looking at how to edit and delete posts.

CHAPTER 21.

REACT PROJECT STEP #5. UPDATING CONTENT

In the last chapter, you looked at how to add flash messages into your app. Now let's turn our attention back to working with updating posts once they have been added.

This will involve a few steps. First, you want to add an "Edit" link to each of your posts. Then, when you click on the Edit link, it should redirect to the form and load the post.

Finally, clicking save in the edit form will save the updated post back to your App state and then redirect you back to view all posts. An "updated" message should also display.

GETTING STARTED

If you have successfully followed along with the previous steps, you can continue with the same code.

If you would like to start fresh, you can use the completed files from the last step, run `npm install`, and then `npm start`.

ADDING EDIT LINK TO POSTS

First, open your Posts component in `"src/components/Posts.js"` and add an edit link.

At the top of the file, import the following after you import React.

```
import { Link } from "react-router-dom";
```

Then, after the <h2> with a link to the post, add the following code:

```
<p>
  <Link to={`/edit/${post.slug}`}>Edit</Link>
</p>
```

This will give you a link very similar to the link to view a single post, except it will go to a new route: edit/post-slug.

Now you need to create this new route in your main `App` component.

ADDING EDIT ROUTE

Inside of the `App.js` file, come down to the `<Route>` for the new post form. After that route, add a new route as follows:

```
<Route
  path="/edit/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    if (post) {
      return <PostForm post={post} />;
    } else {
      return <Redirect to="/" />;
    }
  }}
/>
```

Let's break down what is happening here.

First, you set the path to "/edit/:postSlug." This will load when someone clicks on a post edit link.

Then, on your Route render method, you want to get the slug from the URL using `props.match.params.postSlug`, which React Router provides.

Next you do a simple check to see if that post exists. If that post does exist, you will load the `<PostForm />`. If the post does not exist, someone likely tried to visit an edit link for a post that does not exist and you will just redirect back to the homepage to prevent any errors.

However, if you click on your edit link now, you will have a few problems.

First, you need to pass an event handler into the form responsible for updating the post in the `App` state. Then you need to modify your `PostForm` component to load the post object for editing.

Let's start with writing the function to update your modified post.

THE UPDATEPOST FUNCTION

Inside your App component, you already have an `addNewPost()` function. Right after that, you want to add an updatePost function that looks something like this:

```
updatePost = post => {
  post.slug =  this.getNewSlugFromTitle(post.tit
le);
  const index = this.state.posts.findIndex(
    p => p.id === post.id
  );
  const posts = this.state.posts
```

```
    .slice(0, index)
    .concat(this.state.posts.slice(index + 1));
  const newPosts = [...posts, post].sort((a, b)
=>
    a.id - b.id
  );
  this.setState({
    posts: newPosts,
    message: "updated"
  });
  setTimeout(() => {
    this.setState({ message: null  });
  }, 1600);
};
```

There is nothing complicated React wise going on here, but there is a bit of vanilla JavaScript that is worth unpacking.

First, you need to take the following code from your `addNewPost()` function and break it out into its own function called `getNewSlugFromTitle()`.

So find the following code in your `addNewPost` form:

```
post.slug = encodeURIComponent(
  post.title
  .toLowerCase()
  .split(" ")
  .join("-")
);
```

Replace that code with the following line:

```
post.slug                                              =
this.getNewSlugFromTitle(post.title);
```

And right before the `addNewPost()` function, create the following function:

```
getNewSlugFromTitle = title =>
  encodeURIComponent(
    title
    .toLowerCase()
    .split(" ")
    .join("-")
);
```

Now you can call this function in your `addNewPost()` and `updatePost()` functions.

The next code you see in your `updatePost()` function is some code that will find the index for the first post that has the same slug that you have passed in the URL.

```
const index = this.state.posts.findIndex(p => p.
id === post.id);
```

You then use that index to remove the post you just edited from the list of posts in state. Then add the new post back into the array.

```
const posts = this.state.posts
  .slice(0, index)
  .concat(this.state.posts.slice(index + 1));
const newPosts = [...posts, post].sort((a, b) =>
  a.id - b.id);
```

You also take the chance to sort the posts based on their IDs so that the edited post will still display in the same order as before. Otherwise, edited posts would be added to the end of the array of posts in state whenever you edited one.

Finally, set the posts in state equal to your new updated lists of posts. (You are also setting an updated message to display).

```
this.setState({
  posts: newPosts ,
  message: "update",
});
```

This pattern of finding an object within an array and making edits to it is not uncommon in JavaScript, so you may need to write code like this in the future. Just remember, most of this code is not React specific, but rather plain old vanilla JavaScript.

The last lines of code should look familiar from the addNewPost function. They simply set a timer to set message in state to null after 1600 milliseconds to remove the message component.

```
setTimeout(() => {
  this.setState({ message: null });
}, 1600);
```

Now that your updatePost function is complete, let's pass it as a prop into your `PostForm` component in your edit post route.

Come down into your Route for your edit post form. Then add the following prop:

```
<Route
  path="/edit/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    if (post) {
      return <PostForm updatePost={this.updatePo
st} post={post} />;
```

```
    } else {
      return <Redirect to="/" />;
    }
  }}
/>
```

Now, when you click to edit a post, the `PostForm` will receive the post to update as well as the function to call to make the update.

PASSING EMPTY POST INTO NEW POST FORM

If your `PostForm` receives a post object when you are editing but no post when you are adding a new post, it will add some unnecessary complexity to the component.

Instead, what you can do is pass in an empty post object into the `PostForm` when you are adding a new post. Then the `PostForm` component can always expect to receive a post to edit.

Find the `Route` for your new post form and modify it to pass in an empty post object like so:

```
<Route
  exact
  path="/new"
  render={() => (
    <PostForm
      addNewPost={this.addNewPost}
    post={{ id: 0, slug: "", title: "", content
: "" }}
  />
  )}
/>
```

This addition brings up an interesting React related point. If a component does not know what props it will receive, it requires additional conditional logic. If a component can always rely on receiving specific props, less conditional logic is necessary within the component. Neither approach is necessarily correct, but in general it is nice to have simpler components when possible.

LOADING THE POST INTO POSTFORM

Now your `PostForm` component will always receive a post as a prop. When acting as a new post form, **it** will receive an empty post object. When acting as an edit form, it will receive the post to edit.

In both cases, you can use the same form component to add new posts or edit existing ones.

Open the file for the `PostForm`.

First, make the following changes to your state:

```
>state = {
  post: {
    id: this.props.post.id,
    slug: this.props.post.slug,
    title: this.props.post.title,
    content: this.props.post.content
  },
  saved: false
};
```

Next set default values on your form elements and modify the `onChange` handles, since you modified the format of your state.

Find input field for the post title and change it to the following:

```
<input
  defaultValue={this.props.title}
  id="form-title"
  value={this.state.post.title}
  onChange={e =>
    this.setState({
      post: {
        ...this.state.post,
        title: e.target.value
      }
    })
  }
/>
```

Now get the `title` from props and set it as the `defaultValue` for your input field. Then set the value of the form based on the value of the `post.title` in state.

In the `onChange` method, set the state of post equal to the past value of the post in state. Then override the `post.title` with value from your form.

You have to do this because there is no way to set just one property of an object in state. Using destructuring though will give you exactly what you need.

The modifications to the Quill editor component are similar, but do not require a value property to be set.

Make the following changes to the Quill component `onChange` method:

```
<Quill
  defaultValue={this.state.post.content}
  onChange={(content, delta, source, editor) =>
{
    this.setState({
      post: {
        ...this.state.post,
        content: editor.getContents()
      }
    });
  }}
/>
```

You should now have an empty post loaded into the `PostForm` component when it is used as a new post form. Likewise, you should have the post to edit loaded into the form when the component is being used as an edit form.

While this seems to work, there is one potential problem here that has to deal with setting state from props in React.

SETTING STATE FROM PROPS

Setting state from props in React is a handy method to get an initial value of state from props, but it has a major problem.

Setting state from props will only happen the first time the component is loaded. If the props change in the future, those new prop values will not be set to state.

To demonstrate this problem, do the following:

1. Add a new post
2. Click to edit the post
3. Click on "New Post"

You will see that when you click "New Post", it still displays the values of the post you wanted to edit.

To resolve this problem, you are going to run very similar code on `componentDidMount()`. Then you can test to see if the value of props has changed, and if it has, you will update the state from props once again.

Add the following `componentDidMoutnt()` method to your `PostForm` component:

```
componentDidUpdate(prevProps, prevState) {
  if (prevProps.post.id !== this.props.post.id)
{
    this.setState({
      post: {
        id: this.props.post.id,
        slug: this.props.post.slug,
        title: this.props.post.title,
        content: this.props.post.content
      }
    });
  }
}
```

Now, if you repeat the same process as you did before, when you click on “New Post”, it will trigger the `props.post.id` value to change. It will change from the edit post object ID to zero, which is the value of the ID you are hard coding when you pass an empty post object into your new post form.

This code is a little repetitive, and demonstrates some of the problems you can have when setting the initial value of state from props. The code also serves as a work around to avoid the problem if props might ever need to change and re-update state.

MODIFYING THE HANDLEPOSTFORM HANDLER

Now that you have your form loading properly, let’s write your last little bit of code that will determine whether to call `addNewPost()` or `updatePost()`.

Inside of your `handlePostForm()`, add a simple conditional check to see if `updatePost()` is passed down in props. If it is, then call `updatePost()`, and if not, call `addNewPost()`.

Here is what your modified `handlePostForm()` should look like now:

```
handlePostForm = e => {
  e.preventDefault();
  if (this.state.post.title) {
    if (this.props.updatePost) {
      this.props.updatePost(this.state.post);
    } else {
      this.props.addNewPost(this.state.post);
    }
    this.setState({ saved: true });
  } else {
    alert("Title required");
  }
};
```

And there you have it. You have modified a single form component to work for both adding new posts and editing existing ones.

THE FINAL CODE

If you got stuck along the way, or would just like to look over the final, completed code, you can access the completed code here:

https://github.com/zgordon/react-book/tree/master/project/step-05-update

WHAT'S NEXT?

Now that you have the ability to add and edit posts, let's look at how to delete posts next. This will complete your core CRUD functionality involving posts in state.

Then you can look at how to make your state persistent with local storage and a remote database.

CHAPTER 22.

REACT PROJECT STEP #6. DELETING CONTENT

You've now come to the end of your CRUD setup for your app. In this step, you will look at how to add a delete post link to your list of posts. This link will handle removing a post from state.

Unlike adding and editing posts, you will not actually need a route to handle deleting posts. Instead, you can use a simple `onClick` event handler.

However, similar to the add and edit functionality, you will place the function for actually deleting a post from state in the App component where your main state is handled.

GETTING STARTED

If you have successfully followed along with the previous steps, you can continue with the same code.

If you would like to start fresh, you can use the completed files from the last step, run `npm install`, and then `npm start`.

CREATING THE DELETEPOST FUNCTION

Start off in the main `App` component. Scroll down to right after the `updatePost` function.

Add a `deletePost` function that takes post as a parameter and then checks a `window.confirm` modal to verify the user wants to delete the post. It will look something like this:

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
  }
};
```

Inside of this, filter through the posts in state to get all of the posts that do not have the id of the post you want to delete:

```
const
 posts = this.state.posts.filter( p => p.id !==
post.id);
```

Then update the state with these filtered posts and set a deleted message to appear.

The final code will look something like this:

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
    const posts = this.state.posts.filter(p => p
.id !== post.id);
    this.setState({ posts, message: "deleted" })
;
    setTimeout(() => {
      this.setState({ message: null });
    }, 1600);
  }
};
```

Now pass down `deletePost` into your `Posts` component and call it directly from there, where you have access to the post you want to delete.

Come down into the `Route` that calls the `<Posts />` component and pass in `deletePost` like this:

```
<Posts posts={this.state.posts} deletePost={this
.deletePost} />
```

Now you can call this function from within your post listing component.

ADDING THE DELETE POST LINK

As mentioned, you will create a link in your `Posts` component. This will go next to the Edit link that will call `deletePost`.

However, you will not actually use an anchor tag `<a>` element for this. Instead, you will use a button styled as a link.

React suggests that you do not use links unless they can actually link somewhere. In your app, you are using the React Router `<Link>` components when you want to create links somewhere.

Your delete link does not actually go anywhere. You could create a route just to handle deleting your post, but this is not necessary.

What you really want is an action to take place, but you want to have the interface for calling the action look like a link. However, in these cases in React, you will actually use buttons styled as links.

In the CSS you received, buttons will receive similar styling to links when they have a "linkLink" class added:

```
button.linkLike {
  background:inherit;
  border: none;
  color: #26738D;
  font-size: inherit;
  text-decoration: underline;
}
```

```
button.linkLike:hover {
  cursor: pointer;
}
```

Open the `"src/components/Posts.js"` file and add the following inside the `<p>` tag containing the Edit link:

```
<p>
  <Link to={`/edit/${post.slug}`}>Edit</Link>
  {" | "}
  <button className="linkLike" onClick={() => de
letePost(post)}>
    Delete
  </button>
</p>
```

The first thing you see here is an interesting oddity about adding extra spaces inside JSX. You need to put them inside of a JavaScript string, which in turn gets wrapped in curly braces `{" | "}`. If you just used the pipe character `|` on its own, you would not see any spaces appear before and after it, even if you left spaces in your source code.

Next you see your button with the `linkLike` class added and an `onClick` event handler that directly calls `deletePost`. Since you already have access to the post via props, you can pass it directly.

Remember that `deletePost` is passed down through props. So, to prevent from having to call it like `props.deletePost`, you can destructure it when you receive the props.

```
const Posts = ({ posts, deletePost }) => (
  // Component code
);
```

Now you should see a button styled as a link that appears next to the Edit link.

When you click on the Delete button, a confirm window pops up in the browser. This is an easy safeguard to make sure that if someone did not mean to delete that post, when they click cancel or no, the post will not be deleted. If they click confirm, the conditional check you wrote earlier will return true, and the post should be deleted. You should also see a deleted message appear.

FINAL CODE

As with all the steps, if you got stuck along the way, or would like to just look over the final code for this step, you can access it here:

https://github.com/zgordon/react-book/tree/master/project/step-06-delete

WHAT'S NEXT?

You have finally completed your basic CRUD operations! However, saving everything in state like you have done is fragile, and you have probably grown tired of having to keep adding new posts each time the page gets refreshed.

So, over the course of the remaining steps, we will explore ways to make your state persistent. You'll accomplish this using local storage and a database. First, let's look at how to easily add local storage support for your state. You have already done this in a practice exercise, so it should seem familiar.

CHAPTER 23.

REACT PROJECT STEP #7. MAINTAINING PERSISTENT STATE WITH LOCAL STORAGE

As you have noticed, the state for your app resets every time the page refreshes. In this chapter, you will look at a simple solution to help make your state stay persistent between page loads (and even closing and re-opening the browser).

Rather than manually writing the code for saving your state in local storage, you will use the React Simple Storage package.

GETTING SETUP

If you have been successfully coding along with the past project steps, you can simply continue with the same code base.

Or, if you would like to start fresh and follow along, you can grab the completed files from the last step, run `npm install`, and then `npm start`.

SETTING UP LOCAL STORAGE

To start, you want to install the React Simple Storage package. Make sure your React development server is stopped and then run the following:

```
npm install react-simple-storage
```

This will get you the package. Then, in your `App.js` file, add the following import right after you import from React Router:

```
import SimpleStorage from "react-simple-
storage";
```

Completing the configuration is quite simple, especially since the only state you need to make persistent is the main App component state.

To complete the set up, come down right between the `<div className="App">` and the `<Header />` in your App `render()`. Add the following:

```
<div className="App">
  <SimpleStorage parent={this} />
  <Header />
```

This is all the Simple Storage package needs to track your App component state and save it to local storage.

Now, when you add a new post and refresh the page, or even close the tab and then access the page again, you will see the post still present.

FINAL CODE

If you have any problems with this step, or would just like to look over the final code, you can find it here:

https://github.com/zgordon/react-book/tree/master/project/step-07-localstorage

WHAT'S NEXT?

Saving your data to local storage is handy, but your app still has a few major limitations that make it not quite ready for launch.

First, anyone can add a post. It would be a good idea to add some authentication to your app so that you only let authorized users edit content.

Also, every new visitor to the site will see no posts loaded, since they are all saved in state. You will need a way to save your posts so that anyone accessing your sites will see a live view of what posts are published.

To solve both of these problems, you will use the Firebase platform from Google, which includes both authentication and real time databases.

CHAPTER 24.

REACT PROJECT STEP #8. AUTHENTICATING WITH A FIREBASE DATABASE

This will be a major step in your application. You are going to set up user authentication for your app. The user authentication will accept an email and password. Then it will check to see if it matches any users you have set up.

This way you can hide the New Post link from the header, and the Edit and Delete options from your Posts, if a user has not logged in to the site yet.

Rather than build all of this, you will use the Google Firebase platform. Along with a lot of other features, Firebase includes an interface to manage users. In addition, it has ready to use functions to handle each step of the authentication process.

GETTING STARTED

To get started, you can either use your code from the previous step, or you can start with the completed files from the last step available in the course repo here:

https://github.com/zgordon/react-book/tree/master/project/step-07-localstorage

SETTING UP A FIREBASE PROJECT

To begin, you will need a free Firebase account. Head over to https://firebase.google.com and sign up for a free account.

Once you have logged in, you will see an option to start a new project. Give your project a name like "React Blog Demo." You may also have to accept some terms and conditions. Once you do, click to create the new project.

After Firebase has created your new project, you can continue to the project dashboard.

Under Develop, go to Authentication and click to Set up sign-in method. Edit the Email/Password Sign-in method to enable it, and then click save.

Finally, click on the Users tab under Authentication and Add a user. Enter in your own email and a secure password since you will ultimately launch your site live to a production server.

CONNECTING FIREBASE TO REACT

As you can see, setting up Firebase for authentication is quite simple. Connecting your React app to Firebase includes a few more steps.

First, you want to create a file in your app that will save the basic information you need to connect to Firebase. None of this information is particularly private or secure, so you do not have to worry about it being bundled with your client side JavaScript.

Note that in general, you want to make sure any authentication methods you use with React in the browser do not accidentally make any secure information public.

To get started, make sure your development server is stopped and import the firebase package from NPM.

```
npm install firebase
```

Restart your server, and then in the `"src"` folder of your project, add a new file `firebase.js`. Import the following at the top of the file:

```
import * as firebase from "firebase/app";
import "firebase/auth";
import "firebase/database";
```

This will import a few things. First, you get the main firebase library needed to initialize anything working with Firebase. Then you import the authentication and database libraries specifically as well.

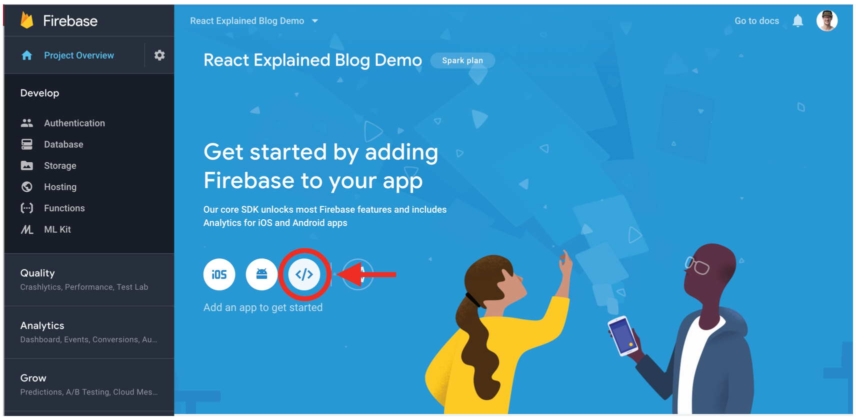
Below the imports, add the following empty configuration object, firebase initialization, and exporting of firebase as well:

```
const config = {
};
firebase.initializeApp(config);
export default firebase;
```

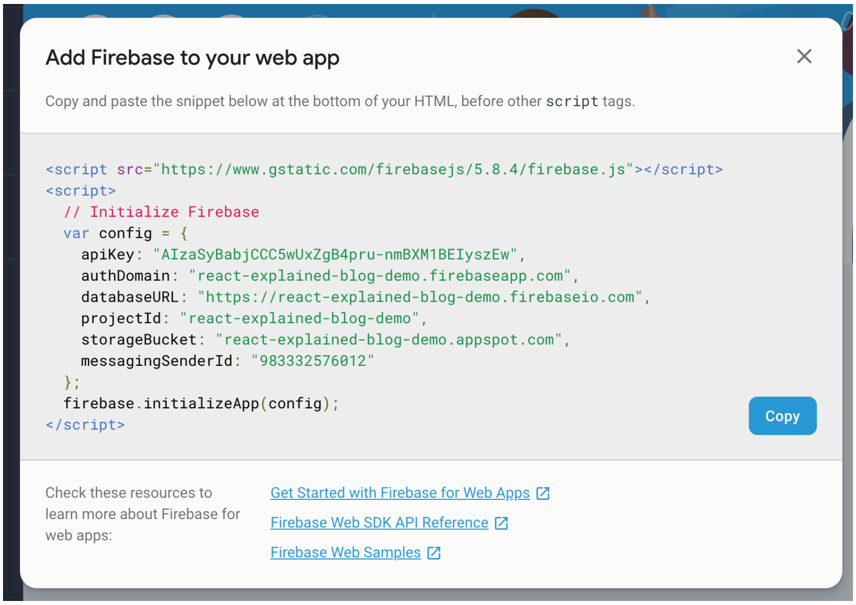
Before you can go any further, you need to get the specific configurations for the Firebase project you just created.

In your project Dashboard, you should see a gear icon next to Project Overview that will let you access your Project settings.

There you should see a web icon like </> , similar to this:


Firebase
Project Overview
Develop
Authentication
Database
Storage
Hosting
Functions
ML Kit
Quality
Crashlytics, Performance, Test Lab
Analytics
Dashboard, Events, Conversions, Au...
Grow
Predictions, A/B Testing, Cloud Mes...
React Explained Blog Demo
Go to docs
React Explained Blog Demo
Spark plan
Get started by adding Firebase to your app
Our core SDK unlocks most Firebase features and includes Analytics for iOS and Android apps
iOS
</>
Add an app to get started

This will give you a panel with your Firebase settings:



Add Firebase to your web app

Copy and paste the snippet below at the bottom of your HTML, before other `script` tags.

```
<script src="https://www.gstatic.com/firebasejs/5.8.4/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyBabjCCC5wUxZgB4pru-nmBXM1BEIyszEw",
    authDomain: "react-explained-blog-demo.firebaseapp.com",
    databaseURL: "https://react-explained-blog-demo.firebaseio.com",
    projectId: "react-explained-blog-demo",
    storageBucket: "react-explained-blog-demo.appspot.com",
    messagingSenderId: "983332576012"
  };
  firebase.initializeApp(config);
</script>
```

Check these resources to learn more about Firebase for web apps:

Get Started with Firebase for Web Apps
Firebase Web SDK API Reference
Firebase Web Samples

Now you want to take all of the properties from that var config that Firebase gives you and copy and paste them into your `firebase.js` file.

The final code for this example project will look different than yours:

```
import * as firebase from "firebase/app";
import "firebase/auth";
import "firebase/database";

const config = {
  apiKey: "AIzaSyBabjCCC5wUxZgB4pru-
nmBXM1BEIyszEw",
  authDomain: "react-explained-blog-
demo.firebaseapp.com",
  databaseURL: "https://react-explained-blog-
demo.firebaseio.com",
  projectId: "react-explained-blog-demo",
  storageBucket: "react-explained-blog-
demo.appspot.com",
  messagingSenderId: "983332576012"
};
firebase.initializeApp(config);
export default firebase;
```

You will have to set up your config to include the configurations specifically for your project. The configurations above will not work for you. This is just meant as an example of what your final `firebase.js` file should look like.

With your Firebase project set up, a user added, and your `firebase.js` configuration file complete, you can build a Login form and Logout button that you hook into Firebase and your own App state.

CREATING A LOGIN COMPONENT

Create a new file `"src/components/Login.js"` with a class based component called `Login`.

You will want to have state for the email and password, as well as an event handler to handle the form submission.

Your initial `Login.js` file should look like this:

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

export default class Login extends Component {
  state = {
    email: "",
    password: ""
  };
  handleLogin = e => {
    e.preventDefault();
    console.log(this.state.email, this.state.pas
sword);
  };
  render() {
    return (
      <formnbsp;className="container"nbsp;name="
login"nbsp;onSubmit={this.handleLogin}>
      </form>
    );
  }
}
```

You can see here the email and password set to empty strings by default. Then your `handleLogin` function is called when the login form submits.

Now let's add in labels and input fields to let the user enter in a username and password. There is nothing particularly special about this React code below, and it should make sense to you at this point.

```
<form className="container" name="login" onSubmi
t={this.handleLogin}>
  <p>
  <label htmlFor="email">Email:</label>
  <input
    type="email"
    onChange={e => this.setState({ email: e.targ
et.value })}
  />
  </p>
  <p>
    <label htmlFor="password">Password:</label>
    <input
      type="password"
      onChange={e
=> this.setState({ password: e.target.value })}
    />
  </p>
</form>
```

Finally, you can add a submit button to your form. You will do one special little thing with this and set the button to disabled until there is a value for both the email and password. You can do this using the disabled button property.

Place your button right before the closing form tag:

```
<p>
  <button
    type="submit"
    disabled={!this.state.email && !this.state.p
assword}
  >
      Login
    </button>
  </p>
</form>
```

This component should now log out the email and password to the console on form submission. But you have no way to access your Login form.

So let's go back into your App component and add a Route for

/login that loads your login form. You will also add a login link to your header to make it easy to access.

ADDING ROUTE AND LINK FOR LOGIN

Back in the `App.js` file, import your Login component at the top of the file.

Then add a `Route` with an exact path of `"/login"` right before your Route to the new post form.

You can start off for now with a very simple Route like this:

```
<Route
  exact
  path="/login"
  component={Login}
/>
```

You will come back later and add to this, but for now, let's just add a link to the login page in your Header component.

Inside of `"src/components/Header.js"`, add a login link to the list of links:

```
<li>
  <Link to="/login">Login</Link>
</li>
```

You may notice that now the New Post and Login links appear in the menu. Obviously this will have to change, but for the moment, let's just test that you can access the login page and log out the email and password from the form.

Click on the Login link in your browser, and then open the web inspector. Enter in an email and password, and then click Login. Then log the email and username to the console to test you have everything.

Now you can move on to making the actual firebase authentication call to check if the email and password match any that you set up in your Firebase project dashboard.

You will also add to your state whether a user is currently authenticated. This will allow you to add conditional checks throughout your app to determine whether to hide or show certain components.

AUTHENTICATING WITH FIREBASE VIA EMAIL AND PASSWORD

Authenticating with Firebase is actually a pretty simple process. In your `App.js` file, you want to first import the firebase file you set up, like so:

```
import firebase from "./firebase";
```

Then you want to create a new function inside of your App component called `onLogin()` that accepts two parameters: email and password. Inside of this, you will handle the firebase authentication and then pass `onLogin()` down to your Login component as a prop to be called there.

To authenticate with Firebase, first call `firebase.auth()`. This will give you access to a number of different methods for authentication, including `signInWithEmailAndPassword()`, which takes an email and password and returns a promise with the authenticated user or an error statement of what went wrong.

Here is what a simple `onLogin` function would look like:

```
onLogin = (email, password) => {
  firebase
    .auth()
    .signInWithEmailAndPassword(email, password)
    .then(user => console.log("Logged in")
    .catch(error => console.error(error));
};
```

Let's finish wiring this up by passing onLogin down into the Login component into your Routes.

```
<Route
  exact
  path="/login"
  render={() =>
  <Login onLogin={this.onLogin} />
  }
/>
```

Now, within your `Login` component, inside of the `handleLogin` function you wrote previously, you can call `onLogin` with the email and password from the form.

Here is what your updated `handleLogin` function will now look like:

```
handleLogin = e => {
  e.preventDefault();
  this.props.onLogin(this.state.email, this.stat
e.password);
};
```

This will take the email and password the user submits and pass them back up into `signInWithEmailAndPassword()`.

A note here is that the email and password you enter to authenticate must match the email and password you set up in Firebase when setting up the project.

Now logging in to the site should authenticate you and log out

"Logged in" to the console. But you do not have a way to remember that the user is authenticated.

Next you need to add to state a way to keep track of whether a user has logged in already or not.

ADDING ISAUTHENTICATED TO STATE

Let's add a new property to your App state called "`isAuthenticated`" with a default value of false.

```
state = {
  isAuthenticated: false,
  posts: [],
  message: null
};
```

Now, inside of your onLogin function, you can set the state of `isAuthenticated` to true when you are sure the user has authenticated.

```
onLogin = (email, password) => {
  firebase
    .auth()
    .signInWithEmailAndPassword(email, password)
    .then(user => {
      this.setState({ isAuthenticated: true });
    })
  .catch(error => console.error(error));
};
```

With this information in state, you can add some conditional statements throughout your app to hide certain functionality if a user is not authenticated.

Then you will also go back and create a Logout link in the header as well.

CHECKING FOR AUTHENTICATION THROUGHOUT APP

In this section, you are going to go back through your app and make the following changes:

- Hide the Login link if authenticated
- Show the New Post link only if authenticated
- Show the Edit and Delete links only if authenticated
- Redirect protected Routes to "/login" if not authenticated
- Redirect login Route to "/" if authenticated

This will give your app a better flow, based on whether a user has logged in or not. This will also protect actions that you only want authenticated users to perform.

UPDATING THE HEADER

To start, let's pass `isAuthenticated` as a prop into your Header component from `App.js`:

```
<Router>
  <div className="App">
    <SimpleStorage parent={this} />
    <Header
isAuthenticated={this.state.isAuthenticated}
    />
    {this.state.message && <Message
type={this.state.message} />}
```

Now, in the `Header` component itself, write a conditional check to see if the user has authenticated:

```
const Header = props => (
  <header className="App-header">
    <ul className="container">
```

```
      <li>
        <Link to="/">My Site</Link>
      </li>
      {props.authenticated ? (
      <li>
        <Link to="/new">New Post</Link>
      </li>
    ) : (
      <li>
        <Link to="/login">Login</Link>
      </li>
      )}
    </ul>
  </header>
);
```

When you view the site now, you should see the Login link if not authenticated, and the New Post link if you are.

UPDATING THE EDIT AND DELETE LINKS

Now that you have updated your header navigation, let's turn our attention to the Edit and Delete links in your Posts component.

First, pass `isAuthenticated` down into Posts in your `App.js` Route:

```
<Route
  exact
  path="/"
  render={() => (
    <Posts
      isAuthenticated={this.state.isAuthenticate
d}
      posts={this.state.posts}
    />
  )}
/>
```

This will let you update your `Posts` component with the following conditional:

```
{posts.map(post => (
  <li key={post.id}>
    <h2>
      <Link to={`/post/${post.slug}`}>{post.titl
e}</Link>
    </h2>
    {isAuthenticated && (
      <p>
    <Link to={`/edit/${post.slug}`}>Edit</Link>
    {" | "}
    <button
      className="linkLike"
      onClick={e => {
        e.preventDefault();
        deletePost(post);
      }}
      >
      Delete
    </button>
    </p>
    )}
  </li>
))}
```

Don't forget that, in order to call `isAuthenticated` directly like this, you will need to destructure it from props like so:

```
const Posts = ({ posts, deletePost, isAuthentica
ted }) => (
```

When you test your app now, you should see the Edit and Button elements only visible if the user is authenticated.

REDIRECT ROUTES BASED ON ISAUTHENTICATED

The next thing you want to do is add some conditional code to protect certain routes and make sure that others, like the "/login" route are only accessibly under the correct conditions.

Here is what you are going to do:

1. Redirect "/login" to "/" if the user is authenticated
2. Redirect "/new" to "/login" if *not* authenticated
3. Redirect "/edit/post-slug" to "/login" if *not* authenticated

There are several ways you can handle this. You can pass `isAuthenticated` into the components called via these routes and add conditional logic there telling a user to log in to access that page.

Or you can add conditional logic in the Route `render()` method and prevent the protected component from ever loading. For this project, you are going to take this approach, but each could be valid in certain instances based on the design of your app.

Let's start off coming down into the "/login" route in your app and only make it accessible if the user has not authenticated, which makes sense. You add this extra layer of protection because even though the Login link is not accessible when authenticated, the path itself still is.

```
<Route
  exact
  path="/login"
  render={() =>
    !this.state.isAuthenticated ? (
      <Login onLogin={this.onLogin} />
    ) : (
      <Redirect to="/" />
    )
  }
/>
```

You can see here, in the updated code above, that if the user is not authenticated, you will load the Login component. You also see that if a user has already authenticated, you will redirect them back to the homepage.

In your next update, you want to come down into your "/new" route, which should only be accessible if the user is authenticated. Your update in the "/new" route will actually look like the opposite of the example above.

```
<Route
  exact
  path="/new"
  render={() =>
    this.state.isAuthenticated ? (
      <PostForm
        addNewPost={this.addNewPost}
        post={{ id: 0, slug: "", title: "", cont
ent: "" }}
      />
    ) : (
    <Redirect to="/login" />
    )
  }
/>
```

Here you are only loading the `<PostForm />` component if the user *is* authenticated.

The update for your edit form route will look very similar to the new post. The exception is that you already had a conditional in the edit post route checking to make sure the post you were trying to edit actually existed.

So, in the update to your `"/edit/:postSlug"` route, include checking to see if the post exists. Also include the state of `isAuthenticated`.

```
<Route
  path="/edit/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    if (post && this.state.isAuthenticated) {
      return
<PostForm updatePost={this.updatePost} post={pos
t} />;
    } else if (post
&& !this.state.isAuthenticated) {
      return <Redirect to="/login" />;
    } else {
      return <Redirect to="/" />;
    }
  }}
/>
```

You can see that you first check to see if a post exists. Then you load the `<PostForm />` if the post exists and the user is authenticated. If the post exists and the user is *not* authenticated, you redirect them to the Login page. Finally, if no post exists, then you redirect them back to the homepage.

Now you have your protected path working. If someone is on the add new post or edit a post page and clicks Logout, they will be redirected to the appropriate place.

WRITING AN ONLOGOUT FUNCTION

You now have a clear difference in your app between being logged in and logged out. Let's complete the process now with a Logout link to terminate your authentication with Firebase and update the state of isAuthenticated back to false.

You will start off in your `App` component, adding an `onLogout` function right after your `onLogin` function.

The `onLogout` function will call a special `firebase.auth()` method called `signOut()`, which will automatically stop the current user's authentication with Firebase.

It looks like this in action:

```
onLogout = () => {
  firebase
    .auth()
    .signOut()
    .then(() => {
      this.setState({ isAuthenticated: false });
    })
  .catch(error => console.error(error));
};
```

You can see here that after Firebase signs the user out, you are able to set `isAuthenticated` back to `false`.

With this set up, you can now pass `onLogout` down into your `<Header />` component and call the function directly from within your `Header`.

```
<div className="App">
  <SimpleStorage parent={this} />
  <Header
    isAuthenticated={this.state.isAuthenticated}
    onLogout={this.onLogout}
  />
{this.state.message
&& <Message type={this.state.message} />}
```

With this function written and passed as your props, let's look at how to create the Logout link.

CREATING A LOGOUT LINK

Actually, you are going to create a Logout button and style it as a link. The reason is similar to the Delete button you created.

Semantically, you want an action to occur (logging out). However, you don't actually have a page you need to send the user to in order for you to execute this action and update your app accordingly.

So, in your `Header.js` file, let's update your navigation with the following button:

```
{isAuthenticated ? (
  <>
    <li>
      <Link to="/new">New Post</Link>
    </li>
    <li>
    <button
      className="linkLike"
      onClick={e => {
        e.preventDefault();
        onLogout();
    }}
      >
        Logout
      </button>
    </li>
  </>
) : (
  <li>
    <Link to="/login">Login</Link>
  </li>
)}
```

Note that you had to add a `Fragment` wrapper around the New Post and Logout list item because this expression must return a single React element, not two.

Also remember that in order to call onLogout directly like this, you must destructure it from the props when you set up your component:

```
const Header = ({ isAuthenticated, onLogout }) =
> (
```

You now have authentication set up in your app. You accomplished this using Firebase and component state.

While this is a pretty solid workflow, and Firebase is certainly a great choice for production React projects, there still are a few things you can do to further lock down your App.

Disabling React Developer Tools

The React Developer Tools browser extension allows you to explore your React apps in the browser. It also allows you to even make changes to the state of your app.

This can present a bit of a problem since a savy user could manually change the state of isAuthenticated to true in an attempt to spoof your app. Now, luckily they will not be able to add, edit, or delete anything since just changing the state of your app does not authenticate the user.

However, it is possible to prevent React Developer Tools from working on your site with the following snippet of code:

```
<script>
  if (typeof
window.__REACT_DEVTOOLS_GLOBAL_HOOK__ ===
 'object') {
    __REACT_DEVTOOLS_GLOBAL_HOOK__.inject = func
tion() {};
  }
</script>
```

This should be placed in the `"/public/index.html"` file before the link to the `manifest.json` file.

If adding extra layers of security to prevent a user from messing with your React app too much appeals to you, remember, you only want to add this code when you go to build your production for shipping to production. React Developer Tools is a handy tool for React app, and one you should explore a bit more as you're building your React Apps, if you haven't already.

However, ultimately, your add, edit, and delete functions require true authentication with firebase, so your app is not necessarily at risk with React Developer Tools enabled in production.

USING SSL IN DEVELOPMENT

Another good general security practice is that you should try to use SSL across not just your production and staging environments, but also for your local development environment.

This is especially true if your local development needs to make a request to third party APIs (as does your app).

Create React App will easily let you start your development server using https rather than http by setting an HTTPS setting to true.

If you are running Linux, you will call what you see below:

```
HTTPS=true npm start
```

Notice that this is the same as your `npm start` command, but you just added `HTTPS=true` before it.

It is also important to note that Create React App will use a self signed certificate, so most modern browsers will display a warning asking if you want to trust this site. It is fine to trust this. The process for trusting self signed certificates is a bit more complex, and not always worth the extra work, but you can now trust that your development server will run on `https`, which is what you want.

You do not necessarily need to use https with every single project, but it is not a bad idea to do when dealing with live APIs.

FINAL CODE

You did a lot in this chapter, and it might not be a bad idea to look over the final code if you got stuck along the way, or if you want to just see everything up and running.

You can access the completed code for this chapter here:

https://github.com/zgordon/react-book/tree/master/project/step-08-authentication

WHAT'S NEXT?

Although you have authenticated with Firebase, all of your posts are still only saving to state.

So let's take a look at how to go about connecting our state to the Firebase database to keep it persistent from session to session.

CHAPTER 25.

REACT PROJECT STEP #9. CRUD AND LIVE SYNCING WITH FIREBASE

You are now at the final step of building your project. In this chapter, you are going to connect your posts to the Firebase database.

Since authentication is already set up, you do not have to worry about connecting to Firebase.

However, you will need to modify a few things in your app in order to get everything working:

1. Update `addNewPost()` to push to Firebase
2. Update `updatePost()` to work with Firebase
3. Update `deletePost()` to remove from Firebase
4. Add `componentDidMount()` API call to get posts from Firebase
5. Change `post.id` to reference Firebase keys

Let's go ahead and look at each one of these steps, and learn some things about the Firebase database API along the way.

GETTING STARTED

To get started, you can either use your code from the previous step, or you can start with the completed files from the last step available in the course repo here:

https://github.com/zgordon/react-book/tree/master/project/step-08-authentication

SWITCHING FROM IDS TO FIREBASE KEYS

When you work with Firebase, it can automatically provide you unique identifiers for your posts.

Previously, your app used post IDs based on the length of the posts array. Once you start deleting and adding new posts, it is possible with this setup to have posts with duplicate IDs, which defeats the purpose of having IDs.

The first change you want to make to your app is to remove reference to IDs and replace them with reference to Firebase keys.

To start, come down into the "/new" route in your App component and modify this line:

```
post={{ id: 0, slug: "", title: "", content: ""
}}
```

And change it to reference a null key instead:

```
post={{ key: null, slug: "", title: "", content
: "" }}
```

Next come into the `"src/components/PostForm.js"` file, and in the state and componentDidUpdate, change the reference from id to key:

```
class PostForm extends Component {
  state = {
    post: {
      key: this.props.post.key,
      slug: this.props.post.slug,
      title: this.props.post.title,
      content: this.props.post.content
    },
    saved: false
  };
  componentDidUpdate(prevProps, prevState) {
    if (prevProps.post.key !==
this.props.post.key) {
      this.setState({
        post: {
          key: this.props.post.key,
          slug: this.props.post.slug,
          title: this.props.post.title,
          content: this.props.post.content
        }
  });
  }
}
```

Technically, within your App, you could refer to key as ID, but I think it makes more sense to use key as a reference name in your app since that is what Firebase calls it.

What these changes so far have done is prepare you for when you modify your add, edit, and delete functions. Each of these functions will need to reference the Firebase post key. Now you will have either the correct post key when you need it, or you will have a null post key for new posts – and Firebase can generate one for us.

Next let's look at modifying your `addNewPost()` function to save your posts to Firebase.

ADDING NEW POSTS TO FIREBASE DATABASE

In this section, you will tackle modifying your `addNewPost()` function in your App component to save posts to Firebase rather than state.

You will still be able to sync your posts from the Firebase database to your App state, but you do not need to do it inside your `addNewPost()` anymore. You will come back to syncing Firebase posts to state shortly. For now you can focus on just saving to Firebase.

The first thing you need to know is how to reference your posts in the Firebase database.

You can do that with the following code:

```
const postsRef = firebase.database().ref("posts"
);
```

This will give you access to your main project database, and within that, a subset of data stored under "posts". A single Firebase database can actually store different subsets of data. For example, you could also have something like the following in an imagined project:

```
const pagesRef = firebase.database().ref("pages"
);
const eventsRef = firebase.database().ref("event
s");
const commentsRef = firebase.database().ref("com
ments");
```

So, once you have reference to your posts in Firebase, which is currently empty, you have access to several built in methods to do things with this data.

To add content to the posts database, you can call `.push()`, which will pushyour post object to the "posts" array in Firebase.

Here is what your updated `addNewPost()` function will look like:

```
addNewPost = post => {
  const postsRef = firebase.database().ref("post
s");
  post.slug = this.getNewSlugFromTitle(post.titl
e);
  delete post.key;
  postsRef.push(post);
  this.setState({
    message: "saved"
  });
    setTimeout(() => {
      this.setState({ message: null });
  }, 1600);
};
```

You can see you set up your posts ref in Firebase, create a slug for your post, delete the `null` post key you had set up for new posts, and then finally, call `.push(post)`.

You also remove the updating of posts in state. Since Firebase is a real time database, when you load your posts from Firebase later in this chapter, your posts will automatically update on your site whenever a change is made to the data in Firebase.

So now, if you log in to your site and add a new post, you should see that post show up in your Firebase database console.

Log in to Firebase and click into your project. Then find Database in the navigation, and you should see your new post there.

However, your new post is not yet showing up in your app itself. So, now that you have posts saved in Firebase, let's look at how to display them in your app.

DISPLAYING REAL TIME POSTS FROM FIREBASE

Firebase is a real time database. That means that it has the functionality built in to display a live version of posts in your app. If a post is added, edited, or deleted via the Firebase API or console, then your app will automatically be updated as well.

You will set up this live link between Firebase and your app in a `componentDidMount()` hook in your App component.

To start, you will add `componentDidMount()` and set up your posts reference.

```
componentDidMount() {
  const postsRef = firebase.database().ref("post
s");
}
```

Now that you have access to your posts ref, you can call the `.on()` method Firebase gives you for a real time sync with your data. The `.on()` method gives you what Firebase calls a snapshot of your data that contains an updated reference to your posts.

```
componentDidMount() {
  const postsRef = firebase.database().ref("post
s");
  postsRef.on("value", snapshot => {
    const posts = snapshot.val();
  });
}
```

Now you have access to your posts in Firebase. You can add them to your state and they will keep their live sync with Firebase.

What you will do is create a new empty posts array and then loop through the Firebase posts and add them to your new array. Finally, you will add that new array to state.

```
componentDidMount() {
  const postsRef = firebase.database().ref("post
s");
  postsRef.on("value", snapshot => {
    const posts = snapshot.val();
    const newStatePosts = [];
    for (let post in posts) {
      newStatePosts.push({
        key: post,
        slug: posts[post].slug,
        title: posts[post].title,
        content: posts[post].content
      });
    }
    this.setState({ posts: newStatePosts });
  });
}
```

One important thing to point out here is that when you first get each post from the snapshot, what you actually have is the post key.

So let's look at this line found above:

```
key: post
```

You are assigning post from Firebase, which is actually the key, to post.key in your state. Then, to access a specific post object, you use the following bracket object reference:

```
posts[post]
```

If you were to just do a direct mapping of posts from the snapshot into your state, you would not actually get what you were expecting and your app would break.

The great thing about these updates you have made is that if you make a change to any of the posts in Firebase, those changes will automatically be reflected on the site.

To test this, log into your app and add a new post. The post should display on the homepage.

Now go into Firebase > Your Project > Database, and you should see that post. Modify the title of the post within Firebase. Then come back into your app and you should see those changes reflected.

The same would be true if you were to delete this post in Firebase or add a new post. It should now make sense why you don't need to update state when you add a new post to Firebase. Firebase will automatically update your snapshot of posts in your app whenever changes are made.

To continue with the process, let's look next at how to modify your `updatePost()` function so that you can edit Firebase posts within your app.

UPDATE THE UPDATEPOST() FUNCTION

Now that you have new posts saving into Firebase, let's look at how to edit these Firebase posts within your `updatePost()` function in your App component.

To do this, you need to know that you can get a specific post in your Firebase database by adding the post key to your post reference lookup.

```
const postRef = firebase.database().ref("posts/"
+ post.key);
```

Then you can call the `.update()` method that Firebase gives you and identify what you want to update from that post.

Here is what your modified `updatePost()` function will look like once it is integrated with Firebase:

```
updatePost = post => {
  const postRef = firebase.database().ref("posts
/" + post.key);
  postRef.update({
    slug: this.getNewSlugFromTitle(post.title),
    title: post.title,
    content: post.content
  });
  this.setState({ message: "updated" });
  setTimeout(() => {
    this.setState({ message: null });
  }, 1600);
};
```

Once again, you do not need to update your posts in state since they will automatically receive these changes automatically.

There should not really be anything confusing about what you have done here. Go ahead and test this out in your app.

Log in, create a new post, edit that post and see that the changes are reflected both in the app and in the Firebase database dashboard.

Now you just have a final step of updating your `deletePost()` function to delete posts from Firebase rather than directly from state.

UPDATING DELETEPOSTS() FUNCTION

Find the `deletePost()` function in the App component.

Inside of the window.confirm check, you will first find the post you want to delete with the following:

```
const postRef = firebase.database().ref("posts/
" + post.key);
```

Then you can simply call the `.remove()` method Firebase gives you to remove items based on their key.

Here is what your updated `deletePost()` method will look like:

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
    const
postRef = firebase.database().ref("posts/" + pos
t.key);
    postRef.remove();
    this.setState({ message: "deleted" });
    setTimeout(() => {
      this.setState({ message: null });
    }, 1600);
  }
};
```

Once again, because of your live database sync, you do not need to remove the post from state here. You can simply tell Firebase to delete the post and your app and state will automatically receive the update that a post has been deleted.

FINAL CODE

You have now fully integrated your app with Firebase. The authentication works, and rather than adding, updating and deleting posts in state, you are updating your Firebase database and letting the changes be live updated automatically in your app.

If you got stuck along the way, or just want to look over the final code, you can find it here:

WHAT'S NEXT?

You could continue to build out your app with additional features, like pages, comments, or the ability to register a user from your site. But for now, let's call your project done and focus next on how to push a completed app to production.

CHAPTER 26.

REACT PROJECT STEP #10. DEPLOYING THE PROJECT

In the final step here, you will look at how to deploy your React app live with a hosting account.

There are many hosting options for React apps like yours. Technically there aren't many server requirements. This is because your final code to deploy will all be HTML, CSS, and client side JavaScript.

However, there are several hosting providers that allow you to easily push to your staging or production environments from the command line. These are particularly appealing to React developers.

For this project, you will use https://netlify.com for hosting. The main reason for this is that they offer free hosting plans. They also have a great number of other features that make them a popular hosting provider for React developers.

GETTING STARTED

To get started, you can either use your code from the previous step, or you can start with the completed files from the last step available in the course repo here:

https://github.com/zgordon/react-book/tree/master/project/step-09-database

If you are using the code from the last step in the link above, make sure to run `npm install` before proceeding with further steps.

BUILDING YOUR APP FOR PRODUCTION

The first thing to remember is that Create React App offers a `build` command that will bundle your app and get it ready for production.

Before you push your site to Netlify, let's create a build version of your app.

Open your project directory and run the following:

```
npm run build
```

This will create a build folder with all of your bundled JavaScript, CSS, and HTML.

If you have already installed the Serve package in the Create React App chapter and practice exercises, you should be able to run the following command to see a local version of what you are going to push to production.

If you have not already installed the Serve package, go ahead and run the following:

```
npm install -g serve
```

Once this is done, or if you have already installed Serve, go ahead and launch your build on a local server with the following command:

```
serve -s build
```

You now see a production ready version of your app running on a local server. Hopefully your app functions properly. There should not be any difference in functionality between this version of your app and the one you get when you run `npm start`.

Go ahead and stop your build server with `ctl + C`.

This build version is what you want to push to production, so next let's set up a Netlifly account.

SET UP A NETLIFY ACCOUNT

To set up a free Netlify account is quite simple. Head over to https://app.netlify.com/signup and create a free account. I like to log in via Github, but you can create an account using the format best for you.

INSTALLING THE NETLIFY COMMAND LINE UTILITY

Now you can install the Netlify command line tool. In your project directory, run the following:

```
npm install netlify-cli -g
```

This will give you a global install of the command line tools for pushing to Netlify servers.

A TEST DEPLOY OF YOUR SITE TO NETLIFY

Before you push directly to production, let's look at "staging," a testing feature netlify offers.

Run the following command:

```
netlify deploy
```

This will launch your browser to connect to your Netlify account. Once you have granted permission for your terminal to access your netlify account, you can come back into the terminal.

You should see a prompt like the following:

```
You are now logged into your Netlify account!

Run netlify status for account details

To see all available commands run: netlify help

This folder isn't linked to a site yet
? What would you like to do?
? ⇄ Link this directory to an existing site
   + Create & configure a new site Press the
down arrow to highlight and select "+ Create &
configure a new site".
```

Enter in a name for your site like `"react-explained-demo"` and press enter. This will serve as the site URL, so you want to make sure the name is URL friendly.

Next you will be asked what account you would like to tie this site to. Likely you just have your own personal account you just set up to select from. Select the appropriate account and press enter.

Finally, you need to tell it what to deploy from your project. You want to enter in `"build"` so that it will only deploy your build directory and not your entire project, including your uncompiled source code. Type `"build"` and hit enter.

It should now prompt you with a test Live Draft URL to check. Use this to test that everything looks correct.

You can now go back and make changes to your app if necessary and run `npm run build` and `netlify deploy` again to retest everything.

Or, if everything looks good, you can go ahead and push straight to production.

PUSHING TO PRODUCTION ON NETLIFY

Once you make sure everything looks correct, you can add the `--prod` flag to the `netlify deploy` command to skip the testing link.

Run the following command to push directly to production:

```
netlify deploy --prod
```

Once again, you will have to tell it the path to deploy. Type "`build`" again and hit enter.

You should now get the Live URL of your site, which you can visit to see your site live.

PUSHING FUTURE UPDATES

In the future, if you need to make changes to your app, you can do so as you have done throughout all the last steps.

Run `npm start` to enter your development environment, and when you are done, run `npm build` followed by `netlify deploy` or `netlify deploy -- prod`.

You can easily repeat this process whenever you need to make updates to your app.

WHAT'S NEXT?

There is a lot more to hosting with Netlify, and I encourage you to check out their site and some of their resources.

There is also a lot more you could do to your app to improve it. In fact, in the next chapter, you'll learn how to "Refactor" your app to clean up some of the code and pull some data logic out from your UI code.

CHAPTER 27.

REACT PROJECT STEP #11. REFACTORING YOUR CODE

Over the last ten chapters, you built a simple React app and deployed it to a live server.

In this chapter, you are going to go back and make a few modifications to your code base to make it a little cleaner. In programming, this generally is referred as “refactoring” your code. You aren’t changing any functionality to your app. You’re just changing how things are written and organized.

Often times, after you complete the functionality for a React project, you will want to go back and refactor it to work or look more efficient at a code level.

The first thing you want to refactor is your flash messages.

REFACTORING YOUR FLASH MESSAGES

In your current code base, you have some functions like these:

```
addNewPost = post => {
  const postsRef = firebase.database().ref("post
s");
    post.slug = this.getNewSlugFromTitle(post.ti
tle);
```

```
    delete post.key;
    postsRef.push(post);
    this.setState({
    message: "saved"
  });
  setTimeout(() => {
    this.setState({ message: null });
  }, 1600);
};
```

And you needed to repeat code like this every time you wanted to display a flash message to the user.

Here is another example of very similar code:

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
    const postRef = firebase.database().ref("pos
ts/ + post.key);
    postRef.remove();
    this.setState({ message: "deleted" });
    setTimeout(() => {
      this.setState({ message: null });
    }, 1600);
  }
};
```

Since this code is so similar, and it is being repeated, it breaks the common programming principal of DRY (Do not Repeat Yourself). Thus, you should rewrite it.

To improve this, create a method inside your `App` class called `displayMessage()`. This will take care of this duplicate code issue.Your function should look something like this:

```
displayMessage = type => {
  this.setState({ message: type });
  setTimeout(() => {
    this.setState({ message: null });
  }, 1600);
};
```

You can see here that the common code, including the timer, is now all in one place.

The next step is to find everywhere in your app you are displaying a message and replace it with a call to `displayMessage()`.

As it stands currently, that happens in the `addNewPost()`, `updatePost()`, and `deletePost()` functions.

Here is what they will look like once they are refactored:

```
addNewPost = post => {
  const postsRef = firebase.database().ref("post
s");
  post.slug
= this.getNewSlugFromTitle(post.title);
  delete post.key;
  postsRef.push(post);
  this.displayMessage("saved");
};

updatePost = post => {
  const postRef = firebase.database().ref("posts
/" + post.key);
  postRef.update({
    slug: this.getNewSlugFromTitle(post.title),
    title: post.title,
    content: post.content
  });
  this.displayMessage("updated");
};
```

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
    const postRef = firebase.database().ref("pos
ts/" + post.key);
    postRef.remove();
    this.displayMessage("deleted");
  }
};
```

In general, anytime you have repeated code, it is a good idea to refactor it into a simple function that can be reused without repeating code.

Also, if you ever want to change how long messages get displayed, you can easily do that in one place.

REFACTORING YOUR API FIREBASE CODE

One of the important things to remember about React is that it is meant to be a User Interface library. Therefore, code that has to do with the User Interface should go inside your React code. However, code that does not have to deal with the User Interface specifically should not go directly in your React code.

The code that you use to connect to, authenticate with, and CRUD with Firebase, *technically* does not really belong hardcoded into your React code. So, in this refactoring, you are going to pull out your code from your `App.js` file and store it in a separate `appService.js` file. Then, in the future, if you ever need to set up a different database, you can easily swap all of your code out in one file rather than finding it within your React code.

You can also do some creative tweaking of your Firebase configuration file to help with using a different Firebase database during development and production.

Creating an appServices.js File

The first thing you will do is create a new file in your `src` directory called `appService.js`. In it you will create functions for everything needed to authenticate and CRUD with Firebase.

Take a look at what the final file will look like, and then I will break it down bit by bit.

```
import firebase from './firebase'

export default new class AppService {

login(email, password) {
  return firebase.auth().signInWithEmailAndPassw
ord(email, password)
}

logout() {
  return firebase.auth.signOut()
}

subscribeToPosts(callback) {
  firebase.database().ref("posts").on("value",
snapshot => {
    const posts = snapshot.val();
    const newStatePosts = [];
    for (let post in posts) {
      newStatePosts.push({
        key: post,
        slug: posts[post].slug,
        title: posts[post].title,
        content: posts[post].content
      });
    }
    callback(newStatePosts)
  });
}

getNewSlugFromTitle = (title) => {
  encodeURIComponent(
    title
      .toLowerCase()
      .split(" ")
      .join("-")
  );
}

savePost = (post) => {
  return firebase.database().ref("posts").push({
    ...post,
    slug: this.getNewSlugFromTitle(post.title)
  })
}

deletePost = (post) => {
  return
firebase.database().ref(`posts/${post.key}`).rem
ove()
}

updatePost = (post) => {
  return
firebase.database().ref(`posts/${post.key}`).upd
ate({
    title: post.title,
    slug: this.getNewSlugFromTitle(post.title),
    content: post.content
    })
  }
}()
```

Notice that you have a number of new functions created:

- `login()` – This will take the the email and password, run `signInWithEmailAndPassword()`, and then return the value. So now you can call login() in your app and it has no specific ties to Firebase. This means you could swap out a different authentication method in the future.
- `logout()` – This will serve as a wrapper for Firebase's `signOut()` function. Again, it allows you to swap out your authentication library later on if needed.
- `subscribeToPosts()` – This one works a little differently. This function will help you create the live connection with the Firebase database. However, the parameter it accepts is another function that will be called once the posts are received from Firebase. In its final use, it will look something like this: `subscribeToPosts(posts => this.setState({ posts }))`. The function you pass it will actually take the posts from Firebase and update them in the state of your app.
- `getNewSlugFromTitle(title)` – This code does not actually deal with Firebase directly, but it is only needed when you are saving and updating content. Since it is not needed anywhere else in your app, you will place it in this file along with your other Firebase related functions.
- `savePost(post)` – This is pretty straightforward. It will take a post and then save it to Firebase.
- `deletePost(post)` – This is another simple one. This takes a post and deletes it in Firebase.
- `updatePost(post)` – Your final refactoring will include pulling out the functionality for taking an existing post in Firebase and editing it.

Go ahead and create a new `appService.js` file, if you have not already, and copy over the code above into it.

Importing appService.js into our index.js File

Now that you have your Firebase code in its own file, you have to import that file into your app and make all of the functions available.

The way you will do that is to modify your `src/index.js` to import appService.js and pass it down as props into your `App` component.

This is what our new index.js file should look like:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

import "./index.css";
import appService from "./appService";
import * as serviceWorker from "./serviceWorker";

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

serviceWorker.unregister();
```

Now all of your `appService` methods are available in your `App` component. So now you can turn our attention to refactoring your `App.js` file.

Refactoring Your Main App Component

Now that you have this.props.appService available in your main App component, you can start going through each of your related

App methods and removing any hardcoded reference to Firebase.

To start with, you can remove this import at the top of your `App.js`:

```
import firebase from "./firebase";
```

The first method we want to modify after that is onLogin. Your updated onLogin should look like this:

```
onLogin = (email, password) => {
  console.log(email, password);
  this.props.appService
    .login(email, password)
    .then(user => {
      this.setState({ isAuthenticated: true });
    })
  .catch(error => console.error(error));
};
```

Notice that you have taken out specific reference to `firebase.auth().signInWithEmailAndPassword(email, password)` and now refer to your own generic login function.

Next up you have your `onLogout()` method, which should remove reference to firebase as well. It should end up looking like this:

```
onLogout = () => {
  this.props.appService
    .signOut()
    .then(() => {
      this.setState({ isAuthenticated: false });
    })
  .catch(error => console.error(error));
};
```

It is a good idea to test this along the way to make sure that everything still works!

Next, in your App.js to refactor, you want to remove the `getNewSlugFromTitle()` function completely. Now you can simplify your addNewPost() method quite a bit and leave it simply like this:

```
addNewPost = post => {
  this.props.appService.savePost(post);
  this.displayMessage("saved");
};
```

A similar thing will happen to your `updatePost()` function, which will look like this once it is simplified:

```
updatePost = post => {
  this.props.appService.updatePost(post);
  this.displayMessage("updated");
};
```

Finally, you can clean up your deletePost function to look like the following:

```
deletePost = post => {
  if (window.confirm("Delete this post?")) {
    this.props.appService.deletePost(post);
    this.displayMessage("deleted");
  }
};
```

In all of these examples, you simply replaced any reference to `firebase.someMethod()` with reference to your `appService.someMethod()`.

The last step of this part of refactoring will happen within your `componentDidMount()`. Here you want to take out all of the code referencing `firebase.database().ref()` and `.on()` and replace it with your `appService` code.

Your final componentDidMount() should look like this now:

```
componentDidMount() {
  this.props.appService
    .subscribeToPosts(posts => this.setState({ p
osts }));
}
```

This method is a little more confusing than the others because instead of passing in a string or object as a parameter, you are passing in an actual function to get called. How this works is `subscribeToPosts()` is called and sets up the subscription to the posts. Then, at the end of `subscribeToPosts()`, it calls `posts => this.setState({ posts })` and triggers the posts in state to match the same as those in Firebase.

There is a good chance you might break your app as you make these changes, so take your time and check everything along the way.

As mentioned, none of these changes are required. However, it is considered a best practice to pull anything relating directly to your data out of your React code dealing with the user interface.

RENDERING AUTHENTICATE ROUTES REFACTORING

The last bit of refactoring you will do in your app has to deal with some duplicated code you use in your Routes.

If you remember, each time you want to test if someone is authenticated in order to view a route, you have some code like this:

```
<Route
  path="/edit/:postSlug"
  render={props => {
    const post = this.state.posts.find(
      post => post.slug === props.match.params.p
ostSlug
    );
    if (post && this.state.isAuthenticated) {
      return
<PostForm updatePost={this.updatePost}
post={post} />;
    } else {
      return <Redirect to="/" />;
    }
  }}
/>
```

Although this is technically fine, it does lead to some duplication of code and could mean if you ever want to change how authenticated routes work, you'd have to change your code in multiple places.

So, to clean things up, you can introduce a new function called `renderAuthRoute()` that handles this logic for us.

Here is what your `renderAuthRoute()` method would look like:

```
renderAuthRoute = (Component, props) => (
  this.state.isAuthenticated ? (
    <Component {...props} />
): <Redirect to="/" />)
```

This function will take the Component you want to load if the user is authenticated. It will also take the props for that component as parameters.

Then it will run the conditional check to see if the user is Authenticated. If they are authenticated, you will load the desired component. If they are not authenticated, you will redirect them home. In the future, if you wanted to change this to redirect users to a /login route, it would be easy to do in one place.

Now you can update your authenticated Routes to look like the following:

```
<Route
  exact
  path="/login"
  render={() =>
    this.renderAuthRoute(Login, {
      onLogin: this.onLogin
    })
  }
/>
```

That takes your Login route and makes sure the user is redirected if they are already logged in to the app.

Next up you have your New Post route to modify:

```
<Route
  exact
  path="/new"
  render={() =>
    this.renderAuthRoute(PostForm, {
      addNewPost: this.addNewPost,
      post: {
        key: null,
        slug: "",
        title: "",
        content: ""
      }
    })
  }
/>
```

Then the final route you have to update is your Edit Post route:

```
<Route
  path="/edit/:postSlug"
    render={props => {
      const post = this.state.posts.find(
   post => post.slug === props.match.params.post
Slug
  );
    if (post) {
      return this.renderAuthRoute(PostForm, {
        updatePost: this.updatePost,
        post
      });
    } else {
      return <Redirect to="/" />;
    }
  }}
/>
```

Now you have refactored and simplified all of your authenticated routes.

WE CAN ALWAYS TAKE IT FURTHER

There is even more you could do to refactor your code. However, at a certain point in refactoring it is good to stop, make sure everything still works, and ship your refactored code.

So, now is a good time to stop refactoring. You can go back to the previous chapter to review how you can deploy the refactored code that you have made.

When you're building a React application of your own, it can be helpful to get a code review for your app to see if there is anything that needs to be refactored.

However, during your building process, I like the approach of building it so it works, then going back to refactor and clean up the code later.

Regardless of your process, remember that refactoring code is an important part of developing with React.

TAKING REACT FURTHER

A REVIEW OF WHAT WE'VE LEARNED

In this book, we have learned many of the foundations of the User Interface library.

We started off with a review of some helpful JavaScript and development tools.

We then looked at the how Elements and Components work in React. We looked at how to write JSX in our React apps. We also looked at how we can use Create React App for easily spinning up and maintaining React apps.

Then we got into some of the most important React concepts: props, state and the component lifecycle.

Finally, we built an entire app from scratch with React, learned how to integrate with the Firebase Real Time Database, and push our app live to production with Netlify hosting.

NEXT STEPS FOR LEARNING

From here, I would suggest you go out and start trying to build an app of your own.

It is less important what you build and more important that you build something. In trying to build projects with React on your own, you will hit walls that help you more deeply understand how to architecture code with React.

Remember, there is often more than one way to build something with React.

When you get stuck with how to do something, try searching around for answers. Sites like Stack Exchange, blogs on Medium, and just the web in general have tons of resources on how to solve specific problems and explain React concepts in different ways than we have here.

However, it is essential you now apply what you have learned to a project of your own.

MORE ADVANCED REACT TOPICS

This book should serve as a solid foundation for working with React. However, there is certainly more you could learn.

Here are a few more advanced React topics that would be worth exploring once you feel you have a good understanding of the basics we have covered so far in this book.

Styled Components is an alternative way to applying CSS in React that involves placing your styles in your JavaScript rather than in a seperate CSS file, like we did in our project.

Higher Order Components are special components that take a component in, populate it with data, and return a new data. A common place you see these used is when you want to pass data from an API into a component.

Redux is a global state management library for React. Although this library is usually not necessary until you have apps with more complex data structure and functionality, it is a helpful library to learn along with React.

Server Side React can also be very helpful to learn because it will let you to render server side content with React. This way you do not have blank pages loading that require API calls in the browser to run before any data is displayed on the page.

Although these next topics are important to learn in time, it is important to make sure you firmly understand and can implement the concepts we have already covered in this book.

APPLYING WHAT YOU LEARN

As you continue to learn React, try to apply what you learn into real world projects at work or into personal projects of your own. This will ensure that what you learn stays with you.

You may also want to access whether you are getting paid the right amount based on your skills set. It may be helpful to research what jobs or pay scales are available based on what you now know.

Also, remember, the technology landscape is always changing. Learn React well and apply it to your projects. But also keep in mind that eventually there will be new things to learn about React, and eventually something new to learn that replaces React.

However, at the time of publication (and for some time to come) React stands as one of the best choices for frontend JavaScript libraries. So, practice what you have learned and go out and create something great!