React: Getting Started

by Samer Buna

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hello everyone. My name is Samer Buna. I work at jsComplete where we create interactive educational content to help people learn coding. Welcome to this React.js: Getting Started course, from Pluralsight, covering the latest stateful function components with the all new React Hooks. React is a simple and powerful library that is used in many big web and mobile applications today. I personally use React in most of the projects I'm involved in. It helps me be fast and productive. In this course, I cover the new, simple way using function components with Hooks, and the other more verbose way, using class components. This is a fully hands-on course where I'll be building small, practical applications with React. There will be no foo or bar in this course, and I try to avoid contrived examples as much as possible. Learning to code is a practical experience. Try to do and redo the examples I'll present in this course and try to expand their scope and challenge yourself with every piece of knowledge that you learn. Some of the topics that we will cover in this course include React's design concepts, function and class components and their benefits, JSX and event handlers, working with data and APIs, React Hooks for state and side effects, taking input from the user, communicating between components, React one-way flow of data, and creating your own local development environment for React. By the end of this course, you should be comfortable working with function and class components in React, manage

an application state, and be able to build simple React applications from scratch. This course is a beginner-level course on the React.js library. No previous knowledge about React itself is needed, but you do need to be comfortable with the JavaScript language to get the most out of this course. I hope you'll join me on this journey to learn the basics of the excellent React.js library in this Getting Started course, at Pluralsight.

The Basics

Introduction

Hello, and welcome to the React.js: Getting Started course, from Pluralsight. This course was first published shortly after the React.js library was open sourced, but I've rerecorded the entire course to work with the latest version of React today. All the examples we do in this course should work on any React version greater than 16.8. In this course, we're covering the fundamentals of the React.js library. No previous React.js experience is required to take this course. We will be starting from level 0. But you do need some JavaScript experience first. To be specific about that requirement, you need to know how to define and use JavaScript variables, holding scalar values, objects, and the rays. You need to know how to define and use functions and classes. You need to know the basics of working with loops and conditionals. I have some resources here for you if you don't feel completely comfortable with the basics of the JavaScript language. I have a book and a bunch of interactive labs that you can try as well. The book is for the complete beginner and the labs cover some important fundamental concepts in JavaScript like function, scopes, and closures. I recommend that you check them out first. If you're coming to React with some previous knowledge of JavaScript, but you have not used the modern features of the language that were added in the past few years, that is not a problem. This course has a module to introduce these features. In that course module, I'll cover things like arrow functions, destructuring, REST, spread operators, classes, and more. We will not be using advanced JavaScript. A basic knowledge of the language will be enough for you to survive this course. However, you're likely to run into problems that are related to the language syntax rather than the React.js API. I've written an article on jsComplete about the common problems learners usually face when working with the React.js library. Scan through this article quick and keep it for your reference when you run into a problem while taking this course. Also remember that you can always ask for help in the Discussions tab available in this course page. Pluralsight mentors

actively monitor these discussion forums, but please be as descriptive as you can when you ask a question. Share your code, list any errors you're getting, and share a screenshot if you can. And when sharing your code, don't paste it directly here. This discussion tool is not good for that. Use a GitHub gist or something similar. Quick side note about the very important pause button in your video player. You're going to need to use it a lot in this course. I often get complaints that my Pluralsight courses are a bit fast and it's hard for people to keep up. This is not because I talk fast. This is because the courses are tightly edited with no breaks. A lot of content is intentionally jammed into a short course. You should use the pause and rewind buttons and control the play speed if you need to. Every time I ask you a question, pause the video and think about it. Every time I use something that you've never seen before, pause the video and google it. Rewind and watch things many times if you need to. If you're used to the pace of on-site workshops, you'll find the pace here much faster. The pause and rewind buttons are your best friends. Also, in some of the modules of this course, I'll be presenting you with challenges. Pause videos and do these challenges. The best way to learn is really to do. I'll also ask a lot of questions in this course, and I'll answer these questions right after, but I want you to imagine yourself in an interview for a job about React.js and treat these questions as if they were your interview questions. Try to answer them first before you listen to me answering them. Here's your first interview question. Why do you like React? Whether you have formed an opinion about React or not, as a React developer, you should know the strengths and weaknesses of the library. Let's talk about that next.

Why React?

I'm a big fan of starting with why? So before you dive in and write your first React component, let me make sure that you know why committing to learning React is a very good thing that you're doing. If you've already formed your opinion about React, you can skip this video. Let's start with React's official definition. It states that it is a JavaScript library for building user interfaces. It is important to understand the two different parts of this definition. React is a JavaScript library and not a framework. The words library and framework mean different things in different contexts, and this could be a point for or against React. What's important to remember here is that React is small, and it's not a complete solution. You will need to use other libraries with React to form solutions. React does not assume anything about the other parts in any full solution. It focuses on just one thing, which is the second part of the definition, building user interfaces. A user interface is anything we put in front of users to have them interact with a machine. User interfaces are everywhere, from the simple buttons on a microwave to the dashboard of a space shuttle. If the device we're trying to interface can understand JavaScript, we can use React to describe a user

interface for it. Since web browsers understand JavaScript, we can use React to describe web user interfaces. I like to use the word describe here because that is what we basically do with React. We just tell React what we want, and it will build the actual user interfaces on our behalf in the web browser. Without React or similar libraries, we would need to manually build user interfaces with native web APIs in JavaScript, and that is not as easy. When you hear the statement that React is declarative, this is exactly what it means. We describe user interfaces with React and tell it what we want, not how to do it. React will take care of the how and translate our declarative descriptions, which we write in the React language to actual user interfaces in the browser. Of course, HTML itself is a declarative language, but with React, we get to be declarative for HTML interfaces that represent dynamic data, not just static content. Let's go back to React being a library and not a framework, and let me answer this question. How exactly is not being a framework a good thing? Frameworks serve a great purpose, especially for young teams and startups. When working with a framework, many smart design decisions are already made for you, which gives you a clear path to focus on writing good application-level logic. However, frameworks come with some disadvantages as well. For experienced developers working on large codebases, these disadvantages are sometimes a deal breaker. Let me name two of the important disadvantages about using a framework. Frameworks are not flexible, although some claim to be. Frameworks want you to code everything a certain way. If you try to deviate from that way, the framework usually ends up fighting you about it. Frameworks are also large and full of features, and that makes them hard to customize for specialized cases. If you need to use only a small piece of a framework, you usually have to include the whole thing. This is changing today, but it is still not ideal. Some frameworks are going modular, which I think is great, but I am a big fan of the pure UNIX philosophy to write programs that do one thing and do it well. And React follows this philosophy because it is a small library that focuses on just one thing, enabling developers to declaratively describe their user interfaces and model the state of these interfaces. Here's a summary of the reasons why I think React gained this massive popularity. Working with the DOM API is hard. React basically gives developers the ability to work with a virtual browser that is friendlier than the real browser. When React was first released, I remember there was a lot of buzz around the performance of its virtual DOM, which we will talk about shortly. The virtual DOM performance is certainly a nice plus, but I think what developers like more about this is the fact that they wouldn't need to deal with the DOM API. Some people don't like this second point, but you'll often hear that React is just JavaScript. What that means is that there is a very small React API to learn, and after that, your JavaScript skills are what make you a better React developer. This is a big advantage over libraries with bigger APIs. Learning React pays off big time for iOS and Android mobile applications as well. React Native allows you to use your same React skills to

build native mobile applications. You can even share some logic between your web iOS and Android applications. The React team at Facebook tests all improvements and new features that get introduced to React right there on facebook.com, which increases the trust in the library among the community. It is rare to see big and serious bugs in React releases because they only get released after thorough production testing at Facebook. Most importantly, remember the one thing that React does exceptionally well. React established a new language between developers and browsers that allowed developers to declaratively describe stateful user interfaces. This means instead of coming up with steps for the transactions on their interfaces, developers just describe the interfaces in terms of a final state, like a function. When transactions happen to that state, React takes care of updating the user interfaces based on that. If someone asked you to give one reason why React is worth learning, this last one is the one. If you need to convince someone about React, you can send them to this article, which summarizes what we covered in this video. However, to keep the course short, I'll stop babbling about the why and get you started on the what and the how next.

React's Basic Concepts

React has three simple and fundamental concepts that you need to understand. The first concept is its components. With React, we describe user interfaces using components. If the term components sounds scary to you, you can really think of components as just functions. In fact, simple React components are actually just vanilla JavaScript functions, as we'll see in the next video. In any programming language, we invoke functions with some input, and they give us some output in return. We can reuse functions as needed and compose bigger functions from smaller ones. React components are exactly the same. They receive certain input objects, and they output a description of a user interface. We can reuse a single component in multiple user interfaces, and components can contain other components. However, you don't really invoke a React component. You just use it in your HTML as if it was just a regular HTML element. Also, unlike pure functions, a React component can have a private state to hold any data that may change over the lifecycle of the component. This ties to the second concept about React, it's nature of reactive updates. React's name is a simple explanation for this concept. When the state of a React component, the input, changes, the user interface it represents, the output, changes as well. This change in the description of the user interface has to be reflected in the device we are working with. In a browser, we need to regenerate the HTML views in the DOM tree. With React, we don't need to worry about how to reflect these changes or even manage when to take changes to the browser. React will simply react to the changes in a component's state and automatically update the parts

of the DOM that need to be updated. The third concept about React is its virtual representation of views in memory. Okay, this might sound a bit weird, but to build HTML web applications with React, we don't write HTML at all. We use JavaScript to generate HTML. Let me tell you why. When your web application receives just the data from the server in the background with AJAX, you need something more than HTML to work with that data, and you have two options. You can use an enhanced HTML template that has loops and conditionals, or you can rely on the power of JavaScript itself to generate the HTML from the data. Both approaches have advantages and disadvantages. React embraces the latter one and eliminates the extra step needed to parse and enhance the HTML template. One big advantage for this HTML in JavaScript approach is how it enables React to keep and use a virtual representation of HTML views in memory, which is commonly known as the virtual DOM, or the tree reconciliation algorithm. React uses the virtual DOM to compare versions of the UI in memory before it acts on them. I'll show you the practical value of that once we're comfortable with the basic syntax of React. Let's go back to the concept of the component. A React component can be one of two types. It can be either a function component or a class component. Both types can be stateful and have side effects, or they can be purely presentational. You should learn them both, but prefer to use function components over class components because they're really much simpler. Class components, however, are a bit more powerful. In this course, I'll use a mix of these two types so that we can learn them both. Both types can be compared to regular functions when it comes to their contract. They use a set of props and state as their input, and they output what looks like HTML, but is really a special JavaScript syntax called JSX. The props input is an explicit one. It is similar to the list of attributes an HTML element can have. The state input is an internal one, but is really the more interesting one because of how React uses it to auto-reflect changes in the browser. These two input objects have one important difference. Within a component, the state object can be changed while the props object represents fixed values. Props are immutable. Components can only change their internal state, not their properties. This is a core idea to understand in React, and I'll have many examples for you to bring it home. Okay, let's talk about this syntax that looks like HTML, but is not really HTML. Here's a full example. This is a simple React class component without any input, so no props or state, and with a simple h1 in a div output. On the left side, the component is written in the special JSX syntax. Remember how I said we don't write HTML at all in React, and instead we use the React API to generate our views? In a way, this JSX syntax that you see on the left is the compromise to that rule. It makes our task of using the React API as close as possible to writing HTML, but it is not HTML. It simply gets compiled to the pure JavaScript calls that you see here on the right. These React.createElement calls that you see on the right are what we ship to browsers. They are the JavaScript representation of this component's DOM, and they are what

React can efficiently translate into actual DOM operations to be performed by the browser. This is important to understand, but how about we stop with the theory and slides and write some code to understand all these concepts better.

Your First React Component

Let's implement this simple React component, a button with a numeric label, and clicking that button increments it's numeric label to count how many times the button was clicked. Ready to do that? Don't worry, we'll take it one step at a time. First, let me tell you about this interface that you see here. Before you can create a React application, you'll need to do some environment configurations, and that might be a bit overwhelming for someone who is just starting to explore. Before we go that route, we'll use this interface that I especially created for this course. It will help you learn the React way without the complexities of configuring a ton of tools to have a React environment. This playground is a good first step for quickly exploring React, but it stops being useful once the size of your application grows. This is why the last module of this course will be about configuring your own React environment. But until we're ready for that, we'll use this playground. This is the point in the course where you need to start doing things with me and not just watch me. Go to jscomplete.com/playground and follow along with what I do. I kept this interface as simple as possible. You have an editor and a display. You type in some JavaScript code in the editor, for example, math.random, and you press Ctrl+Enter and your could will be executed and its results displayed on the right. Just like a simple REPL, run, eval, print, loop, that you can use to test quick JavaScript expressions. To show more things in the display, you can use the HTML element with the ID of mountNode. You can grab a reference to this element using document.getElementById and specify mountNode as the ID. This is a DOM API call to select the display element. And once you do that, anything you put inside this mountNode HTML element will be displayed to the right. The more important thing about this playground is that the latest version of both React and ReactDOM are already preloaded here. In addition, the playground understands the JSX syntax. And it will also work with the modern JavaScript features, which we will be using with React. Guess what I used to build this playground? React, of course. You can actually tell that this playground is written in React, looking at this little icon here in my browser. This icon is coming from the React DevTools extension, which is something you should install right away, by the way. So if you don't have this icon in your browser, pause the video right here and go Google for React DevTools extension and add the extension to your browser. Once the extension is active, relaunch the jscomplete playground and make sure the icon is showing up here. The React DevTools extension allows us to inspect and interact with any React application

on any website. Let's inspect the jscomplete playground while we're here. Open your browser's DevTools. There should now be a new tab for React. This has been added by the React DevTools extension. You'll see that the extension is showing up two nodes here. The weird one is the main component for the jscomplete playground React app. You don't see normal component names here because the code is minified, but you can still see the structure of the app and you can see the properties and state of every component in the DOM tree, and you can interact with everything here. For example, this editor area here can be resized by dragging the separator. And this UI state is managed with React. If you can find the component responsible for that, you'll see how there is a state element that gets changed when the separator is dragged. You can even change this state directly from the DevTools and trigger the resize based on the new value. How cool is that? Take a moment and familiarize yourself with this extension. It is going to be a great asset for you when you start writing React applications. Every video in this course will start with a URL at jsdrops.com. That will take you to the starting point of that video. This way you can start from the exact point where I am. And if you're coding along in the video and you code did not work as expected at the end of the video, you can simply compare your code to the code available when you go to the next jsdrops URL in sequence. Here is the first URL. If you go to jsdrops.com/rgs1.1, rgs is for React Getting Started, you should see the exact code that I now have on the screen. Make sure that works for you. When this code is executed, you should see a Hello React line in the display. Let me go over what is going on here. We have a simple React function component named Hello. It returns a div. This component has no input. It's also a peer component, no state here. To display a React component in a browser, we need to instruct the ReactDOM library on how to do that. The function designed to do that is ReactDOM.render, which takes in two arguments. The first is the component to render, in our case it is the Hello component. Look how I used it here as if it was just another DOM element with a self closing tag. The second argument to the ReactDOM.render function is the DOM element in the browser where we wish the React component to show up. In this playground, we're using the mountNode display element. If we were to do this in a regular React environment, this element has to exist in the already rendered HTML before this code. You can think of this element as the entry point for React. We're telling React to take over this element and render all of the content within it. Time for another interview question. Explain what exactly is going on here on line 2. How are we writing HTML inside a JavaScript function? And why is this working and not throwing an error? This is not valid JavaScript. If you copy this code and try to execute it anywhere JavaScript can be executed, it will throw on error for sure. However, in the playground, the code works fine because the playground is equipped with a special JavaScript extension named JSX. That should be the gist of your answer. This line is not HTML, it is JSX. It will not be executed by the browser. It will

be executed by the JSX extension and compiled to something else, something the browser can understand. This absolutely means that what this browser is currently executing is not what you see on the screen. The playground is using a special compiler named Babel to convert JSX into React API calls. You can see exactly what Babel is doing to our JSX, if you go to the Babel REPL here under try it out. Make sure this React preset is selected, and paste in here the JSX value that we're testing in the playground. Babel compiles that into a call to the createElement function in the React top level API. That's it. That's the magic of JSX. You write what looks like HTML and Babel converts into React API calls. So in the good example we have here, the browser is not really executing this. Instead, it's executing the following, React.createElement, and this takes many arguments. The first argument is the element to be created, in this case, a div. The second argument is any attributes this element will have, the div element has no attributes. And the third argument is the child of that div element, in our case, it's just the string Hello React! Let me add a few more here to make sure this is working, and go ahead and execute this. And check it out, it is working. You see this line in React applications, but the browser is really executing this line instead. And this is true here as well, this line is also JSX. In this case, the conversion would be React.createElement, and it is a hello element here, this is not an HTML element, this is a React element. And it also doesn't have any attributes and it doesn't have any children. So this is the equivalent line in this case, and this will work as well. While you can totally use React this way, without JSX, it would be a lot harder to code and maintain. So let's stick with JSX. Let's now update this code to make it into the button counter example that I showed you at the beginning of this video. Instead of a Hello component, let's name our component Button, uppercase B button. You'll have to do this change to the function name and where we used it here in the JSX line, Button. Instead of returning a div, let's make it return a button HTML element, lowercase button here. Start that buttons label as TEST. Ctrl+Enter to execute, and you should see a button with a label of TEST. Now I intentionally rendered a button component and the button element in this example. This is to make you aware that the capitalization here is not optional. React has some rules here. A component name has to start with an uppercase letter, because if you don't do that, React will think you meant an HTML element. For example, if we named our React component lowercase b button and tried to render that here, React will just render an empty button element here, and it will really not use this function up here at all. This is a beginner mistake. Always name your components with an uppercase first letter. This rendered button here is not interactive yet. To make it so, we'll need to introduce some state to this component. We'll use the simple and powerful React hooks to do that in the next video.

Your First React Hook

Continuing with what we started in the previous video, the current code is available under this URL, and our component currently renders a stateless button. We now need to make the button increment a counter every time it's clicked. We need a state object. To use a state object, react has a special function named useState. This is also globally available here in the playground. So we're going to be invoking this function. This useState function is going to return two items for us. The first item is a state object, and the second item is a function to change that state object. The state object can be of any time you wish it to be. It could be a string, a number, or an array, or anything else. For this use case, we need it to be a number. I'm going to name this state object counter and name its updater function setCounter. The syntax you need here might look a bit weird, but since JavaScript functions can only return a single value, the useState function returns an array with exactly the two elements here. To make this work, we need a variable-defining keyword before this syntax. I'm going to use const. This special syntax here is using JavaScript destructuring feature to capture these two variables in one line. It's not magic or anything; useState returns an array with two elements, and in here we're destructuring these two elements into variables in our Button component. The argument to useState is the initial value for the state element, counter in our case. Since we want that counter to be a number, we'll set that to 0. To use the two new variables that we introduced, let me tell you a nice little thing about JSX. It supports displaying dynamic expressions if you place them within curly braces anywhere inside JSX. So if I make the button's label into curly braces, and inside these curly braces put any JavaScript expression I want, I'll use Math.random, and execute the code, the button will have random value every time I render this component. So to use this new state element, all we have to do is put the counter variable within curly braces and make that the label of the button element. The button will now be rendered with a label of 0. This is the same 0 that's coming from the initial value we specified for useState. So any value initialized here will show up as the button's label, but we'll keep it as 0. Now to use the setCounter updater function, every time we click on this button, we need to introduce an event handler. This is going to look similar to the way we can do this with the DOM API. We define an onClick attribute on the button element. This onClick attribute is also case sensitive. It has to be camelCase like this. And unlike the DOM version of the onClick, which receives a string, this react onClick attribute receives a function reference. So you always specify that inside curly braces as well. In here, we're going to specify a function reference. Let me create a dummy function here, function. I'm going to name it logRandom, and we'll make this function console.log(Math.random). Very simple. To use this function as the onClick event handler, we pass in here the functions reference, its name, just like this. To see the

console.log messages, we need to bring up the browser's console here. And after executing this code, every time I now click on that button, the console will print a random value. Note that when we passed in the logRandom function here, we did not invoke the function. This will not work. You just need to pass in the pointer to the function. You can also inline the function definition right here inside the curly braces. So, basically, you paste in the function definition, and this will work as well. We can make this more concise by using the new arrow function syntax, which can fit in a single line here, () =>, and then the body of the function directly after that. This highlighted part is an inline arrow function definition. We're not invoking the function here. We're defining it and passing this new reference to the onClick prop. This will work as well. So now that we saw how to wire an onClick event, what we need to do to make the counter count is to use the setCounter updater function that we got from useState. So instead of console logging a random value here, I'm going to use setCounter, invoke that function, and the argument to setCounter will become the new value of counter here. So if we pass in counter+1 as the argument just like this and execute this code, then every time the onClick event is triggered now, the counter will be incremented, and you'll see the button behaving as we wanted it to. This useState function is called a hook in the react world. It is similar to a mix-in or a module, but it is a stateful one that hooks this simple component into a state. What I need you to notice here is that to reflect the new value of the counter variable in the browser here, we did nothing. We just managed the state of the counter value. React is automatically updating the browser for us thanks to its reactive nature. We did not implement any transactions on this UI. We implemented the transactions on a JavaScript counter object in memory. Our UI implementation was basically telling React that we want the label of the button to always reflect the value of that counter object. We didn't do any DOM updates. React did. You're going to need a few more examples to appreciate this power. So let's add some more features to this basic example. Let's make our UI show many buttons and make all the buttons increment a single counter. But before we do that, let me give you a quick challenge. Instead of a simple counter, change this component to start the button with a label of 5, then double that value on each click. Go ahead and try to make this simple change, and you can see my solution under this URL.

Your First One-way Data Flow

In the previous video, we made a simple stateful Button component that renders an HTML button element and increments its numeric label when that button element is clicked. We introduced the useState React hook to manage a counter state. Let's improve this component. First, don't use long lines like this. They're hard to read. So let me format this return value to make it more

readable. There we go. Note how I used parentheses here, not curly braces. We're not returning an object here. We're returning a function call, remember, a React.createElement function call. Second Improvement. Instead of an inline function here, let's create an official click handler function. This new function has to be defined inside the Button component because it needs access to the counter and setCounter variables. You can use any name here, but it's commonly named handleClick. We can just paste the code we had inline before here as the body of this function. And in the onClick attribute here, we pass a reference to handleClick. Make sure this is all good and the button is still incriminating. So far, we've only worked with one component. Let's add more. Let's split our one Button component into two. We'll keep the Button component to be just the incrementer, and let's add a new Display component, which is going to just display the value of the counter. This new Display component is going to be a pure presentational one. It will not have a state of its own. That's normal. Not every React component has to have a stateful hook. So to create a new component, we define a function named Display, and we'll make it return some HTML element. For now, let me just put a placeholder div in here and execute. Notice that this new Display component did not show up because I have not included it in the rendered element yet. I just defined it. Let's include it. So remember to include a component, we need to use it like that. However, you can't just add it here as an adjacent sibling to the Button element. This will not work. Question, why does that not work? And the answer is because each one of these elements get translated into a function call. You have few options here to fix this problem. First, you can render an array of elements here and insert into that array as many React elements as you wish. This will work. This is usually a good solution when all the elements you're rendering are coming from the same component in a dynamic way. It's not ideal for the case we're doing here. The other option is to make these two React elements the children of another React element. For example, we can just enclose them in a div, create a div element, then render the Button and the Display inside this div element. The React API supports this. In fact, React has a special object. If you need to enclose multiple elements like this without introducing a new div parent, you can use React.Fragment. This will do exactly the same thing that the div did, but no new DOM parent will be introduced. This case is so common in React that the JSX extension has a shortcut for it. Instead of typing React.Fragment, you can just have an empty tag. This empty tag, if supported in the environment, will get compiled to the React.Fragment version. For the case that we're doing here, I think a div here is okay, so I'm going to keep that. Question, what can we do to make this better? And the answer is we should really extract this code into its own component. This new component can have any name, but you can just use App here. Go ahead and try to create this app component on your own. Make it return this DOM and use it in the ReactDOM.render call instead of what we have. We take the section, create a new function, name

it App, make this function return the exact DOM tree that we have down under. And then in here, instead of all that, we can just render the App component just like that. Since we're going to display the counter's value in the new Display component, we no longer need to show the counter's value as the label of this button. Instead, I'm going to change the label to just +1. Now we need to display the counter value as the message in the Display component. But we have a problem. We actually have two problems. The first problem is that the counter is currently a state element in the Button component, and we need to access it in the Display component, which is a sibling of the Button component in the current tree. So this is not going to work. The state in a React component can be accessed only by that component itself and no one else. To make this counter state accessible to both sibling components, we need to lift it one level up and put it in their parent component, which is the App component that we just introduced. We just move this useState line down to the App component right here. I'll initialize the counter with a different value here to make sure things are working. The logic of this handleClick function will need to change. We will come back to that in a minute. Let's just comment it out for now. Now that we have the counter state element in the App compartment, which is the parent of the Display component, we can flow some data from the parent to the child. In this case, we need to flow the value of the counter state into the Display component, which brings us to the mighty props object. We haven't really used it yet, so let me tell you about it. To pass a prop to a component, you specify an attribute here, just like in HTML. You can name the props of the component anything you want. For example, I'll make the Display component to receive a prop named message, and the value of that message is the counter variable that's coming from the useState hook. The Display component can now use its props object, which is the argument to the function here, and it's usually named props. You don't really have to name it props, but that's the convention. All function components receive this object even when they have no attributes. So the Button component is currently receiving its props object, and that object so far has been empty. Because a component can receive many attributes, this props object will have a key value pair for each attribute. This means to access the message prop and place its value within the display div, we do curly braces and use props.message. Let me test that real quick, and we have an error, handleClick is not defined because we've used it here and commented it out here. So let me just put an empty function here to get things working, and here we go. A counter value of 42 is now getting displayed. This is coming from the Display component. And what we did here is called the one-way flow of data. Parent components can flow their data down to children components. Parent components can also flow down behavior to their children, which is what we need to do next. In the App component, since the counter state is here now, we need a function on this level to handle updating this state. Let's name this function incrementCounter. The logic

for this function is actually the exact same logic that we had before in the handleClick function in the Button component. So we can just move it in here. This new function is going to update the App component's counter state to increment the counter value using the previous counter value. The onClick handler in the Button component now has to change. We want it to execute the incrementCounter function that's in the App component, but a component can only access its own functions. So to make the Button component able to invoke the incrementCounter function in the App component, we can pass a reference to incrementCounter to the Button component as a prop. Yes, props can hold functions as well, not just data. Functions are just objects in JavaScript, and you can pass any object value as a prop. We can name this new prop anything. I'll name it onClickFunction and pass it a value of incrementCounter, which is the reference to the function we defined in the App component. We can use this new pass down behavior directly in the onClick value. It will be a prop on this component, so we can access it with props.onClickFunction. Testing, all is good. Something very powerful is happening here. The onClickFunction property allowed the button to invoke the App component's incrementCounter function. It's like when we click that button, the Button component reaches out to the App component and says hey parent, go ahead and invoke that incrementCounter behavior now. In reality, the App component is the one in control here, and the Button component is just following generic rules. If you analyze the code as it is now, you'll realize how the Button component has no clue what happens when it gets clicked. It just follows the rules defined by the parent and invokes a generic onClick function. The parent controls what goes into that generic behavior. That's basically the concept of responsibility isolation. Each component here has certain responsibilities, and they get to focus on that. Look at the Display component too. From its point of view, the message value is not a state. It's just a value that the App component is passing to it. The Display component will always display that message. This is also a separation of responsibilities. As the designer of these components, you get to choose the level of responsibilities. For example, if we want to, we can make the responsibility of displaying the counter value part of the App component itself and not use a new Display component for that, but I like it this way. This App component has the responsibility of managing the counter state. That's an important design decision that we made, and it is one you're going to have to make a lot in a React application, where to define the state. And the answer is usually simple, down in a tree as close as possible to the children who need to access that value on the state.

Components Reusability

One of the selling points about components is reusability, making a component generic enough so that we can reuse it in different cases. Let's do that. Let's make the Button component more generic. Let's assume that we can pass in an increment value here, and the Button component will use that value instead of the hard coded one. So we'll use the more generic Button component to create a +1 button, a +5 button, and so on. We need to upgrade this +1 here to something dynamic. It will be a property of the component, the amount to increment, and we'll pass it down from the parent. Let's name this new attributes increment, and the value we need to pass here is a number. Since the value here is not a string, we'll need to use a set of curly braces here as well. So, for example, we can pass in 5. One quick note about this syntax here. We could pass the number like this, but the Button component in this case will see and work with that value as a string. So don't do that. Keep it this way so the Button component receives it as a numeric value, not a string. To use this new increment prop, we can just access it here as props.increment. There we go. We've got a +5. So let me go ahead and add a few more buttons here and use increment values of 1, 5, 10, and 100. Let's initialize this back to 0. And there we go. The UI rendered four buttons now with different labels, but they're not going to work yet. In fact, all of them would still increment with a +1 at this point because we did not change their handlers. So the challenge now is to make each button increment the globally displayed counter by its increment value, not just 1. Do you think you can do that on your own? Give it a try. I saved the code as of now for you under 1.5 here. Here's how I'd solve this challenge. This incrementCounter function here will now need to receive an increment value, and we can use its argument to do that. So receive incrementValue in here, and instead of using +1 we'll use the new incrementValue. Now we can invoke the incrementCounter function with different values. The incrementCounter function is aliased as onClickFunction for the button. So the Button component needs to invoke the onClick function with an argument now, which is its own increment value, and that can be accessed using props.increment. But we can't just pass the argument here like this, because remember, we need a function reference for the onClickEvent handler, and what I have on the screen right now is not a reference. It's an invocation of a function. This will not work. But we can simply wrap this invocation in an inline function to make it into a reference, and through the magic of JavaScript closures, this will work just fine. But again, it's a bit cleaner to do this logic inside the handleClick local function here. So let me go back to handleClick and put this new logic inside handleClick, just like that. Make sure this is still working, and all is good. This is the final code of this example, and I saved it here under 1.6 for you. Through this example, we've touched on the many basic React concepts, but this is still a relatively simple example. Let's build something a bit more useful next. But first, I need you to absolutely understand the value you're getting from using React's tree reconciliation under the hood That's the next video.

Tree Reconciliation in Action

You know the basics of React, and you've probably formed your initial impression about it. My initial impression was why go for all this trouble? Why use JavaScript to generate HTML? What exactly is the value I am getting from doing all that? Let me show you that value in action. In this example, which starts at 1.7 here, we have two simple DOM nodes, one being controlled with the native DOM web API directly using innerHTML and another being controlled with the React API, which in turn uses the DOM API. The only major difference between the ways we are building these two nodes in the browser is that in the HTML version, we used a string to represent the content, while in the React version, we used pure JavaScript calls and represented the content with an object instead of a string. There's no JSX, and I will also use a regular JavaScript interval timer here to create some UI state so that we can do a stateful change for both versions. No matter how complicated the HTML user interface is going to get, when using React, every HTML element will be represented with a JavaScript object using a React.createElement call. Let's now add some more features to this simple UI. Let's add an input box that we can type into. To nest elements in our HTML template, it is straightforward in the HTML version. We can just add an input element like this, and it will render. We can do the same in React by adding more arguments after the third argument for React.createElement. To match what we did in the native HTML node, we can add a fourth argument here that is another React.createElement call that renders an input element. This input element has no attributes and no children. There we go. Okay, not too bad so far. Let's add another node to both trees. This time, let's render the current time. I have a JavaScript line here that you can use to display the current time. Just include this in both versions. Since the first version is a template string here, if you notice these are backticks and not single quotes, so this makes the innerHTML a template string, and I can use a dollar sign syntax here and include the JavaScript line to render the time. Let me put this value inside a pre tag just to give it some monospaced font. Test that. The time is showing up. To add the timestamp in the React version, we add a fifth element to the top-level div element. This new fifth element is another React.createElement call. This time it's a pre tag as we used in the HTML version, no attributes as well, and the new date String for content. Both the HTML and React versions are still rendering the exact same HTML in the browser. But as you can see, the React way is a lot more complicated than the native DOM way. So let me ask you the question again. Why go for all this trouble? What is it that React does so well that is worth giving up the familiar HTML and having to learn a new API to write what can be simply written in HTML? The answer is not about rendering the first HTML view. It's about what we need to do to update any existing views in the DOM. So let's do an update operation on the DOM that we have so far. Let's simply make the timestamp tick every

second. We can easily repeat a JavaScript call in the browser using the set interval web timer API. So let's put all of our DOM manipulations for both the HTML and the React versions into a function. I'm going to name this function render. This can be a simple error function just like that. And I'll copy all of this code and put it inside the render function. We can then use this setInterval call to invoke the render function every second. So this should work now. When we refresh the display area, the timestamp string should be ticking every second in both versions. We are now updating our user interface in the DOM, and this is the moment when React will potentially blow your mind. If you try to type something in the text box of the HTML version, you will not be able to. This is very much expected because we are basically throwing away the whole DOM node on every tick and regenerating it. However, if you try to type something in the text box that is rendered with React, you can certainly do that. Although the whole React rendering code is within our ticking timer, React is changing only the timestamp text and not the whole DOM node. This is why the text input was not regenerated and we were able to type in it. You can see the different ways we are updating the DOM visually if you inspect the two DOM nodes in the Chrome DevTools elements panel. The Chrome DevTools highlight any DOM elements that get updated. You will see how we are regenerating the entire first mountNode element with every tick, while React is smartly only regenerating the content of the pre element in the second mountNode. This is React's smart diffing algorithm in action. It only regenerates in its DOM node what actually needs to be regenerated, while it keeps everything else the same. This diffing process is possible because of React's virtual DOM and the fact that we have a representation of our user interface in memory because we wrote it in JavaScript. For every tick in this example, React keeps the last UI version in memory, and when it has a new one to take to the browser, that new UI version will also be in memory. So React can compute the difference between the new and the old versions. In this example, the difference was the content of the pre element. React will then instruct the browser to update only the computed diff and not the whole DOM node. No matter how many times we regenerate our interface, React will take to the browser only the new partial updates. Note that the HTML version can be easily changed with a few more lines to make it update only the content of the pre element as well instead of the whole thing. But that requires some imperative programming. We'll first have to find the element that needs changing in the DOM tree and add some more imperative logic to change its content. We are not doing that in React. We're being declarative in React. We just told React that we'd like a pre element with the date string. No imperative logic is here, and yet we're still getting the efficiency of a tuned-up imperative alternative. This is the subtle power here. The React way is not only a lot more efficient, but it also removes a big layer of complexity about the way we think about updating user interfaces. Having React do all the computations about whether we should or should not

update the DOM enables us to focus on thinking about our data and state and the way to model that state. We then only manage the updates that's needed on the state without worrying about the steps needed to reflect these updates in the actual user interface in the browser because we know React will do exactly that for us, and it will do it in an efficient way.

Wrap Up

A React application is a set of reusable components. Components are just like functions. They take input and they output a description of a user interface in the form of a React element. The ReactDOM library enables us to render those React elements in the browser, and it will rerender them for us automatically when their in-memory state changes. To accomplish this, we write the component's markup using the React JavaScript API. Writing HTML in JavaScript is a lot different than what we're used to. But luckily, React has a way to write the virtual DOM in a syntax very close to the HTML syntax we're used to. This special React syntax is called JSX. Once we have the virtual DOM description in JSX, we can pre-transform it to valid React API calls before shipping it to the browser. Browsers do not have to deal with JSX. The input for a component is a set of properties you can access inside the component with its first argument object, which is usually named props, and also a set of state elements that a component can hook into with the special useState function. A component state can be changed inside that component, and every time a component changes its state, React rerenders it. The props of a component cannot be changed by the component, but the whole component can be rerendered with different props by the component's parent. The syntax to mount a React component in the browser is ReactDOM.render, and that takes two arguments, the component to render and the HTML element to hold the React-rendered markup. React also comes with normalized events that work across all browsers in a standard way. We've seen the onClickEvent handler in this course module, and there are other onSomething events like onChange and onSubmit, and many others. React actually has two types of components, function and class components. We've only seen function components, but I'll be covering class components in future videos as well. However, before we start another application on React, let's get you comfortable with all the modern JavaScript syntax that's usually used in the React ecosystem. That's our next course module.

Modern JavaScript Crash Course

ECMAScript and TC39

React developers love modern JavaScript and use most of its new features. We'll review these features in this course module, which you can totally skip if you think you're comfortable with the topic. JavaScript is a very different language than it used to be just a few years ago. ECMAScript, which is the official specification that JavaScript conforms to, has improved a lot in the past few years after a rather long period of no updates to the language at all. Today, the ECMAScript Technical Committee, which is known as TC39, makes yearly releases of ECMAScript, and modern browsers shortly follow by implementing the new features introduced in each year. This has started with ECMAScript 2015, or its other commonly known name, ES6. Since then, we've had yearly releases named ES plus the current year. Some of these releases were big, and others were very small, but the language now has a continuous update cycle that drives more innovative features and phases out the famous problems JavaScript had over the years. In this module, we'll go over some of the features that are usually used in React applications. This will not be a complete coverage of all the modern ECMAScript features. If you want to expand on this topic, check out The Complete Introduction to Modern JavaScript in the beginner collection at jsComplete.

Variables and Block Scopes

First up, let's talk about variables and block scopes. Here's one of my favorite JavaScript trick questions. Is this a valid JavaScript line of code? Testing it in the playground seems to work fine. There are no errors, which means it's valid. So now the question is, what did it do? These are nested block scopes. We could write code in here like var v = 42 and then access the v variable right after, and it would work fine. JavaScript does not really care about the spacing or new lines in here. A block scope is created with a pair of curly braces, just like this example here. This also applies to if statements and for statements as well. These also get their own block scopes. Block scopes are a bit different than function scopes, which are created for each function. You can see one difference when using the var keyword to define variables. Variables defined with var inside a function scope are okay. They don't leak out of that scope. If you try to access them outside of that scope, you can't. As you can see here, we cannot access the result variable that was defined inside the sum function's scope. However, when you define variables with var in a block scope, you can totally access them outside that scope afterward, which is a bit problematic. And here's one practical example of that. This for loop has an index variable that ticks from 1 to 10. You can access that variable inside the loop normally, but you can also access the same variable outside the loop. After all the iterations are done, the value of i here will be reported as 11, which is a bit

weird. This is why the more recommended way to declare variables in modern JavaScript is by using the let keyword instead of the var keyword. When defining variables with let, we won't have this weird out of scope access problem. If we replace this var here with let and redo the same test and try to access i after that, you'll get the error that i is not defined. This makes sense because we're outside of the scope where it was defined, so this is much better. Block scopes, like function scopes, can also be nested, like the trick question we started with. This is a nested block scoop. Within each level, the scope will protect the variables defined in it as long as we use the let keyword or the const keyword, which behaves in a good way like the let keyword, We use const when the reference assigned to a variable is meant to be a constant one. References assigned with const cannot be changed. Note how I'm saying references here and not values, because defining an object with const does not make it an immutable object. It just means constant reference to it. We can still change that object just like we can do within functions that receive objects as arguments. If the variable defined with const is a scaler one, like a string or an integer, you can think of it as an immutable object. Because these scaler values in JavaScript are immutable, we can't mutate the value of a string or an integer in JavaScript. And when we use const with these scalar values, we can't change the references either. However, placing an array or object in a const is a different story. The const will guarantee that the variable is pointing to the same array or object, but the content of the array or object can still be mutated. So be careful here, and keep that in mind. Variables defined with const are much better than those defined with let for scaler values and functions because you get a guarantee that the value did not accidentally change. Looking at this code example here and assuming that between the first and last line there is a big program, on the last line, if the program runs without any errors, we can confidently say that the answer variable still holds the 42 value. For the same example with let, we would have to parse through the code to figure out if the answer variable still holds the 42 value. If you need a variable to hold a changing scaler value, like a counter, for example, then using let is okay. However, for most other cases, it's probably much better for you to stick with using const for all your variables.

Arrow Functions

There are many ways to define a function in JavaScript, and the modern specification introduced a new way, arrow functions. It is a way to define a function without typing the keyword function, but rather by using an arrow symbol like this. This shorter syntax is preferable not only because it's shorter, but also because it behaves more predictably with closures. Let me tell you about that. An arrow function does not care who calls it, while a regular function cares very much about

that. A regular function, like X here, always binds the value for its this keyword for its caller. If it didn't have an explicit caller, the value of the this keyword will be determined by the calling environment. In the playground here, it's the global window object. An arrow function, on the other hand, like Y here, not caring about who called it, will close over the value of the this keyword for its scope at the time it was defined. Let me say that one more time. An arrow function will close over the value of the this keyword for its scope at the time it was defined. This makes it great for delayed execution cases like events and listeners because it gives easy access to the defining environment, not the calling environment. This is important, so let's take a look at an example. In this playground, the top-level this keyword is associated with a special object, which has the ID of REPL. Now, I've prepared this tester object that defines two similar functions. In both functions, I am logging the value of the this keyword to the display. Func1 is defined with the regular syntax, while func2 is defined with the arrrow syntax. When func1 is called, it's this keyword will be associated with its caller, which in this case, is the tester object itself. This is why you see the printed value for the this keyword in func1 representing the tester object itself. However, when func2 is called, it's this keyword will be associated with the same this keyword that was available in the function's scope when it was defined. It was this playground's REPL object that we've seen on the top level. This is a big benefit when working with listeners and event handlers, and it's why you'll see me using arrow functions often. One other cool thing about arrow functions is that if the function only has a single line that returns something, you can make it even more concise by removing the curly braces and the return keyword altogether. You can also remove the parentheses around the argument if the function receives a single argument, making it really short. This syntax is usually popular for functions that get passed to array methods, like map, reduce, and filter, and functional programming in general.

Object Literals

You can create a JavaScript object in a few different ways, but the most common way is to use an object literal. Here's an example of that. This is a lot easier than doing something like new object, which you can do if you want to. Literal initiation is very common in JavaScript. We use it for objects, arrays, strings, numbers, and even things like regular expressions. The object literal syntax supports a few modern goodies. Here's a simple example where this object defined two regular properties. If you need to define a property that holds a function, you can use this short syntax with object liberals. Of course, if you need an arrow function, you can still use this regular property syntax. Modern object literals also support dynamic properties using this syntax. It looks like an array literal, but don't confuse it with that. JavaScript will evaluate what's within the square

brackets and make the result of that the new property name. So, assuming we have a variable named mystery defined before this X object, here's a JavaScript interview question. What is the value of obj.mystery? It's undefined because this mystery property was defined with a dynamic property syntax. This means JavaScript will evaluate the mystery expression first, and whatever that expression evaluates to will become the object's property. For this case, the object will have a property named answer with the value of 42. Another widely popular feature about object liberals is available to you when you need to define an object with property names to map values that exist in the current scope with the exact same name. Here's an example. If we have a variable named InverseOfPI, we would like obj here to have a property named InverseOfPI holding the same value as the variable InverseOfPI. Instead of typing that name twice, you can use the shorter syntax by removing the second part. This shorter syntax is equivalent to what I had before. Objects are very popular in JavaScript. They are used to manage and communicate data, and using these features will make the code a bit shorter and easier to read.

Destructuring and Rest/Spread

The destructuring syntax is really simple, but I've seen it confuse many people before. Let me make sure that does not happen to you. Destructuring works for both arrays and objects. Here's an example for objects using the built-in Math object in JavaScript. When you have an object like Math, and you want to extract values out of this object into the enclosing scope, for example, instead of using Math.PI, you'd like to have a constant named PI to hold the value of Math.PI, which is easy because you can have a line like this for PI and another one for E if you need the same for E and so on. With the destructuring syntax, these three lines are equivalent to this single line. It destructures the three properties out of its right-hand object and into the current scope. This is useful when you need to use a few properties out of a bigger object. For example, here's a line to destructure Component, Fragment, and useState out of the React API. After this line, I can use the useState method directly like this. If I don't use destructuring here, I'll have to type a lot more characters and lines. Destructuring also works inside function arguments. If the argument passed to a function is an object, instead of using the name of the object every time you want to access its properties, you can use the destructuring syntax within the function parentheses to destructure just the properties that you're interested in and make them local to that function. This generally improves the readability of functions, and you'll see it used a lot for the props argument in React function components. So here we have a circleArea function, which expects an object as its argument, and it expects that object to have a radius property. We're destructuring the radius property out of that object and using it locally in the function. If we call this circleArea function

with an object like circle here, it will use its radius property inside of its calculation. Let's go ahead and test that. You'll see the circleArea calculation working as expected. Destructured arguments can also be defined with defaults, like regular arguments. Say I'd like to use a default value of 2 for a precision property here. Let's define a second options argument for this circleArea function here, and lets destructure our precisions out of that argument to use it in the function's body. If I'd like to use a default value of 2 for the precision property, I can just use the equal sign here after destructuring precision, and that means the default for precision, if not specified, will be 2. I can also make this whole second argument optional using an equal sign after the destructuring syntax. The same call here will use an empty object for the second argument of the function, and then it will use a default value of 2 for the precision property that is now used in the function. Of course, if we invoke the circleArea function with a second argument that has the precision property, that value will be used inside the function. As you can see, this destructuring feature offers a good alternative to using named arguments in functions, which is much better than relying on positional arguments. Destructuring, whether you do it in function arguments or directly with variables, also works for arrays. If you have an array of values and you want to extract these values into local variables, you can use the item's positions to destructure their values into local variables just like this. Note how I used double commas here to skip destructuring the third item in the array. The destructured variable fourth here will hold the value of 40. In React, the useState function returns an array of two items. We use array destructuring with useState to capture these two items into local variables. Array destructuring is useful when combined with the rest operator, which has an example here. By using these three dots, we are asking JavaScript to destructure only one item out of this array and then create a new array under the name restOfItems to hold the rest of the items after removing the first one. Let's test that. So first, here will be 10, and rest of items will be an array of 20, 30, and 40. This is powerful for splitting the array, and it's also even more powerful when working with objects to filter out certain properties from an object. Here's an example of that. Say that we have this data object that has a few temp properties, and we'd like to create a new object that has the same data except for temp1 and temp2. We can destructure temp1 and temp2 and then use the rest operator to get the remaining properties into a new object called person. Just like the three dots of rest, you can use the three dots to spread one array or object into a new array or object. This is useful for copying arrays and objects. You can spread the items in an array into a new array, like in this example. NewArray here will be a copy of the restOfItems array that we destructured above. Similarly, you can also spread the key-value pairs of an object into a new object, like this example. The newObject here will be a copy of the person object. Note that these copies are shallow copies. Any nested objects or arrays will be shared between these copies. Don't forget that.

Template Strings

Template strings are one of my favorite new features that were introduced to the JavaScript language a few years ago. Let me tell you about them. You can define strings in JavaScript using either single quotes or double quotes. These two ways to define string literals in JavaScript are equivalent. Modern JavaScript has a third way to define strings, and that's using the backtick character. Strings defined with the backtick character are called template strings because they can be used as a template with dynamic values. They support what we call interpolation. You can inject any dynamic expression in JavaScript within these dollar sign curly braces holders. For example, we can use Math.random here, and the final string will have the value of the expression included exactly where it was injected in the string. With template strings, you can also have multiple lines in the string, something that was not possible with the regular quoted strings. Backticks look very similar to single quotes, so make sure to train your eyes to spot template strings when they are used in examples.

Classes

JavaScript offers many programming paradigms, and object-oriented programming is one of them. Everything in JavaScript is an object, including functions. Modern JavaScript also added support for the class syntax. A class is a template or blueprint for you to define shared structure and behavior between similar objects. You can define new classes, make them extend other classes, and instantiate objects out of them using the new keyword, You can customize the construction of every object and define shared functions between these objects. Here's a standard class example that demonstrate all these features. We have a Person class and a Student class that extends the Person class. Every student is also a person. Both classes define a constructor function. The constructor function is a special one that gets called every time we instantiate an object out of the class, which we do using the new keyword, as you can see here. We are instantiating one object from the Person class and two other objects from the Student class. The arguments we pass here when we instantiate these objects are accessible in the constructor function of the class. The Person class expects a name argument, and it stores that value on the instance using the this keyword here. The Student class expects a name argument and the level argument. It stores the level value on its instance, and since it extends the Person class, it will call the super method with the name argument, which will invoke the Person class constructor function and store the name as well. Both classes defined a greet function that uses the values they store on each instance. In the third object, which we instantiated from the Student class here, I also defined a greet function directly on the object. When we test this script, o1 will

ease the greet method from its class, the Person class, o2 will use the great method from the Student class, and o3 will use its own directly defined greet method.

Promises and Async/Await

When you need to work with asynchronous operations, usually have to deal with promise objects. A promise is an object that might deliver data at a later point in the program. An example of an async function that returns a promise is the web fetch API that's natively available in some browsers. Here we're fetching information from the top-level GitHub API. Since fetch returns a promise, to consume that promise, we do at .then call on the result of fetch and supply a callback function in here. This callback function will receive the data from the API. The fetch API has a raw response. If you need to parse the data as JSON, you need to call the json method on the response object, and that json method is also on asynchronous one, so it returns a promise as well. To get the data, we need another .then call on the result of the json method, and in the callback of that, we can access the data. As you can see, this syntax might get complicated with more nesting of asynchronous operations or when we need to combine this with any looping logic. You can simplify the nesting here by making each promise callback return the promise object, but the whole .then syntax is a bit less readable than the modern way to consume promises in JavaScript, which is using async await. Let me show you that. You just await on the asynchronous call that returns a promise, and that will give you back the response object. Then you can await on the json method to access the JSON data, just like this. And to make these await calls, you need to label the function as async, and this will work exactly the same. The async await syntax is just a way for us to consume promises without having to nest .then calls. It's a bit simpler to read, but keep in mind that once you await on anything in a function like fetchData here, this function itself becomes asynchronous, and it will return a promise object.

The GitHub Cards App

What We Are Building

We've seen simple components, we worked with multiple components, and we've seen how and when to use state and props of a component. However, we haven't really worked with any real data yet. We're going to be doing exactly that in this course module, and we'll use the GitHub public REST API for it. We're going to build a simple GitHub profile card component that displays

information about lists of GitHub profiles. There's a form here where the user can type in another GitHub handle and use the Add card to add a new profile to the list of displayed profiles. So we will be learning how to take input from the user here and how to use that input to make calls to an API, the GitHub API for this example. And the goal of this app is to get you comfortable working with data objects. But the other thing about this app is that we're going to use React class components in here. While the React team might eventually decide to phase out class components in favor of stateful functions, although no plans for that yet, but class components have been the norm in React for a long time, and many React projects will continue to use them. As a React developer, you're expected to understand and be comfortable with both function and class components. This is going to be the only example we'll write using the class components in this course. After that, the bigger game application will be entirely written with function components and hooks, and I am certain that you're going to like working with function components a lot more than class ones, but you still need to learn them both.

React Class Components

Let's get started. At jsdrops.com/rgs2.1 I prepared a tiny App function component and rendered it to the DOM. We'll use this one to contain all of the other components in this app. I've also included some styles for the app here under the CSS editor just to space things out a bit. The first decision you need to make in a React application is the component structure. You need to decide how many components to use and what each component should describe. This is often easy if you have the full picture of the application you're building, but practically you don't. I usually start with what makes sense for me at the beginning and keep an open mind about it while making progress. I rename components a lot, and sometimes I remove them if I find no reason for them to stick around. There isn't a right or wrong answer here, but there are good and better answers, and you'll only get better with experience. So just start with something, and see where that takes you. Our application will eventually be a list of GitHub cards. That's your first clue that you need a component to represent a single card and another component to represent the list itself. You can start coding either one of these components. I like to start from the bottom up or from the deepest point in the tree, but it's sometimes useful to start with a top to bottom approach, especially when you know most of the components in the tree. Before we introduce new components, let's learn how to use the class syntax for components. Let's start by converting this App function component into a class one. This is simple. Instead of a function component, you define a class with the same name and make it extend a special class in React, the React.Component class. Uppercase component here. This extending of React.Component makes

your App JavaScript class an official React component. And just like function components, class components also map data to view. They do that using a few concepts related to class components. We're going to learn about two main concepts in this example, the concept of the constructor and the concept of the this keyword in classes. I'll explain them as we need them, but for now, you need to know that each React component must have a render function. A class can have as many functions as needed, but the render function in a React component is the only function that's required. You make the render function return the virtual DOM description of your component. This is the same as what the original function component returned, so we can return it here as is. However, instead of receiving props as arguments, in class components, both the props and the state are managed on an instance of the class. Now that we're using a class, we are creating instances of them, and each instance gets its props and state. So this title prop here needs to become this.props.title, and this will just work. Let me do some cleanups, and now we have an App class component. Let's now create the card component, so class Card extends React.Component, render function, and a return, and let's make it return a div placeholder. I'm going to give it a class of github-profile. Let me put it on multi lines and put a placeholder. And to render this card component, we need to include it in the App component that gets rendered in the DOM. So do the same here, multi line, top-level div container. I'll put the header that we had before, and we'll render a Card component. Using a class component is exactly the same as using a function component. There's no difference here. There we go. The Card component is showing up on the screen. I've prepared some markup for this Card, so we're just going to use it. This uses an img placeholder and simple div for information that contains the name and the company. Note how I've given many elements class names here. These class names, for me personally, make the code a little bit more readable. But it's also handy for us to quickly reference them in CSS style sheets and give them some styles. And this is exactly what I did here in the CSS style sheet. I've added some styling using the class names that I used in the markup. I saved the code we have so far here under rgs2.2.

Styling React Components

Before we get into working with data, let me tell you a bit about an alternative way to style your React components without using a global CSS style sheet. Let me actually scratch the global CSS sheet and lose all the styling for what we have so far. Now with this alternatively to style React components, you don't really need to have a className in here, so you can actually get rid of all these classNames if you want to. Instead, you pass a style property. Now, this is a special React property. It is not like the HTML style property that's usually frowned upon. Just like events,

instead of passing a string, we pass this special React property, a JavaScript object. So we use curly braces for a dynamic value, and then we use another set of curly braces to start an object literal. This is not a special double curly brace syntax. This is just an object literal inside the normal JSX dynamic expression syntax that we do with curly braces. Inside the style object we can specify any styles we want this element to have using the JavaScript API for styles. For example, if I want to give this GitHub profile card some margin, I can use margin and put the value in here in a string. So this would be 1rem, for example. Similarly, here is an example style to give the info class a display of inline-block and a marginLeft of 10 pixels. Let's also give the name here a bigger fontSize and test all that. Note the syntax here. It's all JavaScript. It's camel case for property names and it's strings for values. It is not the same as what you use in the regular CSS. Now, there is a lot of debate around this method. It feels like using inline styles, and some people call it that, but it's quite different, the difference being this is JavaScript, not strings. We can generate it and reuse it, and we have the complete power of JavaScript to do that. We can use conditional styles here without having to deal with conditional class names, which is usually a better deal. But this method also has its disadvantages. I often use a mix of JavaScript styles and global styles in my projects. JavaScript styles are excellent for conditional styling. Instead of using different class names based on a certain condition, with JavaScript styles, you can just output different objects, which, I think it's a bit cleaner. Here is a non-related example about that specific use case. This component will output its text in either green or red colors randomly about half the time. The logic for that is right here in the component. I like that. This is easier to work with than conditionally using a class name and then go track what that class name is doing in the global CSS style sheet. I'm pretty sure we're going to need to use this in some styles in the upcoming examples, so keep this in mind. The styling methods we talked about here are really the most basic ones. They sure have a lot of limits and challenges. To mention one example, it would be a big challenge to implement media queries with JavaScript style objects. The good news is there are a lot more solutions that offer more powerful ways. You can see a good list of the available options in this GitHub repository here. Some of these options are deprecated, so don't let this long list scare you. You can find the popular options here, judging by the number of stars, but you should really take a look at CSS modules, which has a few options here. But this babel-plugin-css-in-js one is probably a good starting point. There's also React Native for Web, which would make sense if you're planning to use React Native as well.

Working with Data

Under the rgs2.3 link here, I added a temporary testData array for us to work with. This data was copied from the GitHub API so that once we're done testing our app with this fake test data, it will just work with the real API data. The URL I used to copy the GitHub data is api.github.com/users , and then you put any GitHub user name, and you'll see their data as a JSON object. Since we want to render multiple cards, we need another component to hold the different cards. We'll name this new component CardList. We can potentially manage the state of the application in this CardList component, but until we make that decision, we'll keep this component simple. I am going to actually write this component as a function component, just for you to be comfortable mixing these two components styles together. Not all components in the same React application have to be one type or the other. The CardList function component will receive the standard props argument and let's make it return a div that will hold our list of cards. So inside this div we'll render a Card component, just like that, and we'll need to change the app component to render the CardList instead of the Card. Make sure things are still working. Now let's put some real data in the Card component. We'll use the testData array for that. So we need to uncomment this part to make the test data available in the scope. And inside the render function for the Card component, I'm going to create a local variable here, call it profile, and grab the first element of the test data array. Now we need to change the place holder data here with dynamic values from the profile variable. So we need curly braces, and for the image we're going to use profile.avatar_url. For the name, we need profile.name. And for the company name, we need profile.company. Let's test that. There we go. Dan's profile information is ready. Now that we have an idea how a card is going to look like, let's make it reusable. If we want to render multiple cards right now, they will all render Dan's information because we are fetching the same profile in the Card component itself. So instead of doing this, we'll make the Card component receive its data through its props. So, for example, it will receive name property, like this, and a company property, and so on. One way of doing that is to take the object that holds the props and spread it inside the Card component element, like this. When we use the spread operator with an object like this in the React component, all the properties of that object will become props for this component. So let me spread two objects here to test, we'll spread 0 and 1. And to make this work in the Card component, the profile variable becomes this.props. Capture all the props coming for the Card component as a profile object, and we're already using that for data. Let's test. There we go, progress. The this keyword in here refers to an instance of the Card component. That's something that you need to understand. A beginner question here might be, what exactly is an instance? I mean, you probably know that objects can be instantiated from classes, and we call every object an instance. But what is an instance in a React application? Well, every time we use a class component, like we're doing here twice, React internally creates an

instance from the component and uses it to render the element. That instance is something that React keeps in memory for each rendered element. It's not really in the browser. What's in the browser is the result of the DOM operation React came up with using that instance render method, which came from the component. Don't worry if you don't completely understand that. What you need to keep in mind for now is that each time we use a component, we get a different instance to work with. So the this keyword in the first card is different from the this keyword in the second card. And that's why React places the different props for each card on their different instance. And by using generic props in the Card component, we made it reusable. This means we can now use it to render any card, but we're still hard coding values in the CardList component. So let's fix that. And what we need here is to take an array of objects, as in the testData array in here, and map it into an array of card elements. So in here, instead of the hard coded cards, we can start with testData. This testData is an array, so we can map it into another array. This gives us access to a single profile object in the array of profiles. We can map that into a Card element just like this. And as props for this mapped Card element, we can spread the profile variable that was exposed from the map, and we don't need the hard coded values, and this should work. This line here is a mix of JavaScript and React. This map function here is a JavaScript function that you can invoke on arrays. It takes a function as an argument, and then it uses this function to convert one array into another array, using the return values in the function. So this map line is converting the testData object into something like this, an array of Cards elements because it's returning the Card element inside its function. And an array of Card elements is just an array of React.create element calls, and React is okay with that, React understands arrays. It will just join the array of Card components on automatically render them all. Now that we have a tested card, we can build the form for the user to type in a GitHub user name, query the GitHub API for that user name's data, and append it in the array of data that this app is using. We'll do that next.

Initializing and Reading the State Object

To eventually take input from the user, we can use a simple HTML form with an input and a button. Let's create a new React component. I'm going to call it Form, and this needs to extend React.Component. It needs a render function, and it needs to return some kind of DOM. I'll make it return a form HTML element. And in there, let's use an input element, give it a placeholder, something like GitHub username. And we'll render a button to Add new card. Very simple. To make this component show up in the browser, we need to include it somewhere in what we're rendering, and it can be included in many places. For example, I can put it inside the CardList component, render the Form component, and it will show up. But that would be completely

wrong to do. This Form component is not part of a card listing logic. Every component has its own separate responsibility. The CardList component renders a list of cards, and the Form component renders an input form. So we should not mix them together. It would be so much better to render the Form component in the top-level App component as a sibling to the CardList component. The App component will handle the connection between the CardList component and the Form component. Testing that, both components now show up, and we did not mix their logic. The data array driving the code we have so far is still a global variable, which is not a good thing. Your component should avoid reading global variables. This is a test data object though, so it is a temporary thing that we're going to get rid of. But instead of reading it directly from the CardList component, let's make it part of the App component and make the CardList component receive it as a prop. I'll name this prop profiles and initialize it from the testData. Now in the CardList component, instead of testData directly here, we need to use props.profiles because it's now a property that's coming from the parent, and this CardList component is no longer depending on anything global. Remember, always test your changes as soon as you can. Now, since we want React to re-render the CardList component every time a new record is added to this profile's array, the easiest way to trigger this re-render is to place the profiles array on the special state object. After that, all we need to do is just add a new record to the array and React will react and reflect the new change in the UI. The question is which component should hold this new state? So far, the profiles array is only used by the CardList component. So we can manage the state of the profiles in the CardList component itself. But remember that the Form component will need to append a record to this profiles array, and it can't do that if the owner of the array is its sibling, the CardList component. To allow both the CardList and the Form component to access the profiles array, we should put it on the state of the top-level App component itself. In a class component, the state is also managed on the in-memory instance that React associates with every mounted component. To initialize a state object for the App component, we need to tap into the native class constructor method, which gets called for every instantiated object. This special constructor method receives the instance props as well. It has to call the JavaScript super method. This is a JavaScript thing. This super method is needed to honor the link between the App class and the class that it extends from, React.Component. And React will complain if you don't do that. You should also pass the props object to the super method. Once inside the constructor, we have access to the special state object that React manages for each class component. We can initialize it here by using a call to this.state = and put an object here. Unlike useState in function components, this state instance property has to be an object in class components. It can't be a string or an integer, for example. Now we can place the profiles array directly within this object. So this is going to start as an empty array eventually. But we're

testing with our testData, so let's just place testData in here. To read this new state element, because the CardList component needs it, we can flow it down using this.state.profiles. State is an object on the instance, and the profiles array is a property on that object. Test and make sure we did not break anything. Before we move on, let me show you a simpler and shorter syntax for this extra code that we have to deal with. Instead of all this, we can simply use a class field like this without a constructor call. So this is definitely so much simpler than dealing with a constructor object. This is not yet part of the official JavaScript language, but it is going to be. The syntax works in here because the JS complete playground uses Babel to transpile things to normal JavaScript that the browser will understand. This class field syntax also has some power when combined with arrow functions as we'll shortly see. When you configure your own React application, you'll have to use something like Babel anyway to compile JSX into JavaScript. So it's a big, easy win to also include and use the JavaScript features that are well on their way to become an official part of the language. So I'm just going to keep this syntax. Let's implement the logic of the Form component next. We need to read the value of the text box every time we click on the ad button. Simple, right? Except when we type into this text box, we're changing the state of the UI as well, and it is currently being changed natively, not through React.

Taking Input from the User

To take input from the user, we need to define an event handler in the React UI. We can define an onClick event here on the Add card button, but I prefer to handle this with an onSubmit event for the HTML form element itself. By using onSubmit, you can utilize native form submission features. For example, you can make this input required, and the onSubmit event will honor that in modern browsers. Let's define an instance function here on the form component. I'm going to use the exact same class field syntax that we used for the state in the app component. Let's call this new function handleSubmit, and we'll make it an arrow function and wire it to be triggered onSubmit. So in here we do this.handleSubmit. Every React event function receives an event argument. You can name this anything; it doesn't have to be event. This is just the last argument that the function receives. And for React events, this argument is just a wrapper around the native JavaScript event object. All the methods available on the native event object are also available here. For example, since we want to take over the HTML submit logic, we should prevent the default form submission behavior here using event.preventDefault. This is important when you're working with forms because without preventing default, if you submit the form, your page is going to refresh. So just be aware of that. Now, to read the value that the user types in this box, we can simply use the DOM API. We can give the input here an ID attribute and use getElementById to read its value.

React has a special property named ref that we can use to get a reference to this element. This is kind of like a fancy ID that React keeps in-memory and associates with every rendered element. To use a ref, you need to instantiate an object here. You can name it anything. I'll name it userNameInput, and we do a React.createRef call in here. And to use this ref, since it is part of the instance now, we can just do this.userNameInput, just like that. And here it is on multiple lines. We added ref this.userNameInput, which is the result of calling the createRef method from the React API. Now, inside the handleSubmit, let's console.log the value. The value is we grab the reference using this.userNameInput, and this is a ref object. The current value is stored under .current, and this .current reference is the HTML input element itself, so we can do .value on it to read the value that the user types. Let's test. Type in something, submit, and we can read the value that I typed. So this is how we use refs. But React has another method to work with this input element, which is to control their values directly through React itself, rather than reading it from the DOM elements. This method is often labeled as controlled components, and it has some more advantages over the simple ref property. However, it does require a little bit more code, but it's still simple, so let's use it. So I'll remove this ref attribute. We will not be able to read it this way. We're going to have to find another way to read it. And we don't need the ref attribute on the input element at all. Instead of all that, we introduce a state object, and in the state object, we define an element to handle the input value of the userName field, so maybe something like userName. And we'll initialize it as an empty string. Then, we use this new state element as the value of the input element. So we do value= this.state.userName. And this immediately creates a controlled element. Now, we're controlling the value of the input. However, once you do that, you can't really type in this field anymore because React is now controlling the value. This is why you need an onChange event so that the DOM can tell React that something has changed in this input and you should reflect it in the UI as well. And I'm going to use an inline function here for the onChange event. This function receives the event object as its argument, and it needs to do one simple thing. It needs to change the value of the userName state element into what was typed in the text box. To do that, in the class component we use this.setState and pass it an object that has the new state. In this case, we need to pass it to userName is and in here grab the value that the user typed directly from the DOM. So we can do that using event.target.value. So this is the syntax that we need to use in order to change the state of a class component. It's a little bit different than function component hooks because this function is always named setState and it always receives an object, and it will merge this object with the current state of the component. So now, if we've done everything correctly, we can access the value that the user types using this.state.userName directly, like this. Let's test that. And there we go. So basically, the onChange event happened on every character that we typed here, and it made React aware of this element

state change and React reflected the change back to the element itself because it's a regular React state change. So what exactly is the benefit of doing all this instead of just a simple ref? Well, you can see that clearly if you open the dev tools and find this input element, right here it's on the form component, and observe as you type something here, you'll see how React is aware of this state change for every single character, while previously React was not aware of what was being typed in the input box. So this method is valuable if you need to provide some kind of feedback for the user as the user is typing. One example of that would be a password strength indicator or the count of characters as the user is typing, as in Twitter's Tweet form. Because now that this whole state is in React, we can add a UI description on it as well. And we don't need to read the value from the DOM, it's in React's memory. So that's simply the story of controlling a component to force its UI state through React rather than through the native DOM, and that keeps things in sync for React. There's nothing wrong about reading form data from the DOM if you don't need React to be aware of that data. This should just be a different tool in your box when you need it.

Working with Ajax Calls

Now that we can read the input that the user typed, we are ready to ask the GitHub API for a single profile data using this type input. We can use the native fetch call available in browsers to do an AJAX call. But this playground also has the axios library, and axios is a simple one to use here. So in the handleSubmit function, all we need to do is axios.get and then specify the API endpoint where the data is, and this is the API input that we need. So in here, if we're to fetch Dan's profile information again, we just use this syntax. But because we want this to be a dynamic part, we'll go ahead and make this whole thing into a template string, note backticks in here, and inside this, we can inject the value of the username that the user inputted in the box, which is this.state.userName. We put that in here. Okay, that's how you do an axios call. Now axios.get returns back a promise. So to consume this promise, we can await on it, and this will give us back a response object. To make this await call work, we need to label this handleSubmit as async. Now we can console.log the response object, let's go ahead and test that. So in here, type any valid GitHub username, click Add card, and the handleSubmit will go and fetch that particular username's profile data, and it will print back the response. Now this response has a data attribute that has the actual profile information from GitHub. So what we need here is response.data. This is just the way axios works. It returns a response object and then a data attribute on that response object that has the JSON data parsed and ready for us. Now all we need to do next is append this data object that we get from the API to the profiles array that we

have on the app state, and when we do that, react will re-render the App component and show the new GitHub profile. However, this Form component can't access the profiles state elements directly here because react components have a one-way flow of data, and a component can't change the state of its parent. But remember that the App component can pass properties to its Form child component, and those properties can be simple primitive values or function references. So if the App component passes a function reference to the Form component, we can change the state of the App component in that function, and the Form component will be able to invoke that function because it will be part of its props object. Let's define a new function here on the App component. I'm going to name it addNewProfile, make it an arrow function, and this function will simply receive the new profileData as an argument, so this will be the profileData, and it will do something with it. For now, let's just console.log this profile data to make sure things are working in the App component. Then we pass to the Form component a new prop. Let's name this prop here onSubmit and pass it this.addNewProfile. So the addNewProfile is a function that we defined on the instance of this component, and in here we are making it available as a new prop on the Form component. So within this handleSubmit function, instead of console.log here, we can access the new prop that became part of the Form component. And to do that, we do this.props.onSubmit. This is a new prop, and it holds a function reference. And that function reference is an alias to the addNewProfile function in the App component. And as an argument, we pass it the data attribute that's coming from the axios response. If we test now and type in a valid GitHub username, you will see the App component console log line, and you'll see the GitHub profile data ready for us here. So here, instead of console.log, we need to put this new profileData on the state of the App component. We need to append this object to the array of objects that we have stored as profiles here on the state. And to do that, we have a few options. We do need to invoke setState. Remember, this is the function that we need to change the state of a react class component, and in here we can pass either an object or a function. The updater function syntax is the same here, so I'm going to use a function. This function will give you access to the previous state. And what you return here from this function becomes the new state. So I'm going to return profiles:, and in here I will spread the existing profiles. That is something that we can read using prevState.profiles, and then append the new profileData. This is equivalent to doing a concat operation on the profiles array. This is the spread operator syntax. And with that, I think we're ready to test. So starting with the testData, we can now add a fourth card. There you go. So now we can remove all the testData and start the profiles array as an empty one like this and remove the testData from the scope. We don't need that anymore. I have included the GitHub username I've been testing with here. So now you can test from an empty card and add one card at a time. Notice how the username stayed here after we added the card. So one small

improvement that we can do on this is to reset the username field after we're done adding it to the profiles array. So we can simply just have another setState call in here and pass in a userName value of empty string, and that would reset the userName state value that react is using to track the username that gets typed into the box. So let's test one more time. And now, after adding the card, the username here should reset. There's another important improvement that we need. When you start adding data to this form, react will give you a warning about the key prop. This key prop is something react needs whenever you render a dynamic list of children like we're doing here in the CardList component. React will give you a warning that if you don't specify some kind of identity for this dynamic element, then react will just assume that the position of the element is its identity, and that might cause problems if you start re-ordering those elements. So to avoid issues like this, you just pass a key property here to react, and this key property has to be unique in this particular DOM node. So the GitHub data comes with an ID that's unique, so we can just use that. We can do profile.id, and if you do that, that warning should go away. And I think we have the GitHub cards app. You can use it to add as many cards as you need. It's all driven from a single array, and it fetches information directly from GitHub. Now there are a lot of improvements that we can do to this code, but there is one very important thing that we did not do. We did not handle errors. What should the UI do if the user types in an invalid GitHub handle? What should the UI do if the request of data fails over the network? What should the UI do if the request is taking too long? In fact, if you were in an interview doing an exercise like this, you should voice these concerns right away. These concerns are not really beginner level, so I'm going to keep them as an exercise for you. I cover examples of them in my book, React.js Beyond the Basics at jscomplete. There are other simple improvements you can make to this code. For example, here in handleSubmit, we have some logic to fetch data from an API. Then we have some application logic about what to do with that data that's coming from the API. This is a bad mix. A component should not have this much responsibility. Your whole app should not really depend directly on a library like axios for example. You should have a small agent-type module that has one responsibility to communicate with external APIs and make your code only depend on that agent module. Another thing you could do is extract this logic about managing the state of the username input into its own component as well. And this would simplify the Form component's code. I think this is a good stopping point for this example. Play around with the final code, which I saved here under jsdrops.com/rgs2.7. One quick exercise you can do, convert all the class components into function components. So instead of state here, you use a hook, and instead of this.setState, you use the updater function from the hook.

Wrap Up

We've built a few simple React components in this module, a Card component to render information about a GitHub profile, a CardList component to convert an array of records into an array of Card components, that form component to read input from the user, and an app component to manage the relation between all the other components. We've managed the records array as a state element on the top level App component, which allowed us to share data between multiple components, and it allowed us to append new GitHub profiles to the UI by simply appending the GitHub API response to that state element. In the Form component, we explored how to access an element in the DOM from React directly using the special ref attribute, and we've explored how to read from an input element using React's state itself, with the help of an unchanged event. Components written with this latter method are known as controlled components. In the next module, we'll build a simple game for kids. Here's a preview of it. The player gets a random number of stars between 1 and 9, and the set of numbers from 1 to 9. They need to select the numbers that sum up to the current random number of stars. When they pick the right sum, a new random number of stars will appear and they need to pick again, and keep doing that until all the numbers are used.

The Star Match Game

What We Are Building

In this course module, we're going to build an in-browser game was React. I named it Star Match, and it's a simple math skills game for kids. This app will teach us a few very important concepts about React function components, React hooks, and optimizing state and side effects in a React components tree. You can play the game at this URL here. Go ahead and explore it a little bit because that will help you understand the design decisions we will be making. The game is simple. When it starts, the player gets a random number of stars between 1 and 9 and also the fixed set of numbers from 1 to 9. The goal is to use all nine numbers. For each random number of stars, the player needs to pick one or more numbers that sum to the number of stars. So for 2, I can pick 2. For 4 stars, I have 2 options. I can either pick 4 or I can pick 3 plus 1. For 9, I have 2 options here as well. I can either pick 9 directly or 4 plus 5. Now while the player is picking numbers, they get marked as candidates because it's not a complete answer. And if they pick more than the count of stars, numbers get marked as wrong, and the player can unpick these

candidates or wrong numbers to be able to pick a correct sum. And the game will always draw a number of stars that is playable. So, for example, in this case, we only have 1, 2, and 6 available, so the only number of stars that get drawn are either 1, 2, 6 or the combination of them. And once we're done with the core UI of the game, we'll add a timer to make the game a little bit more interesting and harder to win. If the timer runs out and you're not done picking all the numbers, the game is over. And you can always click Play Again to reset everything about this game and play again. Let's try to win this time. We've got 9. We have 8, 4, 9. We could do 6 and 3, 3. We could do 2 and 1. And then we have 7 and 5. And that was a close one. Okay, enough play. Let's write some code.

Working with Static Markup

I kept the scope and logic of the game as simple as possible for you to focus on learning React concepts and not get distracted by any HTML, CSS, or even the math that we need for this game. In fact, in this jsdrops.com/rgs3.1 URL, I have given you all the distracting elements ready. This has a static HTML template for us to start with. It has all the CSS that I've used in the game, and also I've included some globals here that has the theme colors I chose for the PlayNumbers, and all the math science were going to need. All of this is not really React, don't get distracted by it, but certainly take a look and familiarize yourself with the elements were going to work with here. The static HTML markup here is already represented as a single React component. You should be comfortable with that by now. This is all JSX. Under the Colors object, I used four properties, these represent all the statuses of any PlayNumber. A number starts as available, you click it, and that click might change the number into one of the other three statuses based on the current number of stars and the running sum of picked numbers. In the utils object, we have a few handy functions to sum an array, create an array of numbers, pick a random number in the range, or pick a random sum in an array of numbers. I've implemented these with vanilla JavaScript to keep things simple, but you can opt to use different implementations if you wish. You could, for example, use Lodash here. My point is, all of this logic has nothing to do with React. React can just use it. When that's the case, it's a good abstraction to have on its own rather than mixing it with the React application logic. One quick note before we begin, don't get overwhelmed with the complexity of the game. Pick one small increment that's easy to test and focus on that. Don't, for example, start thinking about how to track candidate clicks right now. We're not ready for that. There are plenty of simpler things we can do. Pick one, focus 100% on it, test it, take a break maybe, then come pick another small testable increment. I think there are two beginner increments here. We could start extracting components and making a component tree. We need

to do that. However, I think getting rid of any static lists here is the better way to begin. Stars will be rendered from a certain value and that value will not always be nine. So we could get rid of all these static divs here and make a loop from a predefined value. Let me actually define a stars variable right away, const stars equal, I'm going to start with 5. So the current focus is to display five stars based on this value. Now, we could use a for loop and manually push elements in an array for this task. But the declarative way of doing this is to start with an array of IDs and map those IDs into an array of star divs. This makes the code shorter and easier to understand and maintain. Avoid for and while loops in React if you can. It's not a hard rule though. In the utils object, we have this range function that we can use to create an array from a number. We just give it the min and the max. So let's use that. So instead of all these stars, we now need a dynamic expression. We're going to start from utils.range, create a range from 1 to the current number of stars, and map this range, because that range is now an array from 1 to 5, we're going to map it into an array of stars. So let's give us an ID for the star, let's just call it starId. And we want to map this into an array of the original divs that we had in the markup. And because this is a list of dynamic children, we should pass a key attribute that is unique for this node. So we can just use the starId here. So test this increment on its own and we are rendering five stars, try to render a different number, and there we go. And similarly, we should also get rid of all these numbers, now that we know how to create a range, we can just do the same here. So let's just do that, dynamic expression, utils, the range is always going to be from 1 to 9, in this case. And we're going to map it into a button component, right? So this gives us access to a single number, and instead of all these numbers, get rid of them and just take one of them here and put it in the map, and this is going to render the number itself. The number is 1 through 9. And we can also use the same number as a key here. So test that. We are rendering nine numbers. And we got rid of all the static lists. Now here's the thing. You should separate this step from the extracting of components step, and I think by doing this first, you get a better idea into what components are good candidates to be extracted on their own. One other thing we could do here is start the number of stars with a random number each time the component renders. This is easy. We also have the random function here in utils. So instead of 6 here, we can do utils.random and 1 through 9. And test that. So now each time the component renders, a different random number of stars should appear. Now because the number of stars is going to change while we're playing, and we need React to reflect this change in the UI, we should probably make the star count a state element. And that's easy because we can take this initial value and do a useState call and pass this initial value to the useState call, and if you remember, the useState call returns an array of two elements, so we can have stars and we can have setStars. So this should work as well. But now the number of stars is a state element. We read it from the state and we could change this

state element, and React is going to rerender the number of stars. Now here's the thing, in the current increment we're focusing on, we did not really need to make the stars variable a state element. So this is a little bit of thinking ahead. We don't have enough clues that this state element is absolutely needed. And guess what? It is not really needed if we're to super optimize the code. But from experience, whenever you identify a data element that's used in the UI and is going to change value, you should make it into a state element, and only optimize the elements on the state when you have enough clues about how they fit with the other elements on the state. It is normal for the state elements of React components to grow and shrink as we design them. I've saved the code we have so far under the rgs3.2 link here.

Extracting Components for Reusability and Readability

Continuing from the previous progress, a good next increment is to think about what components we need in this app. Should we, for example, extract the help line into its own header component maybe? Or is that a not-needed abstraction? Should we make the stars section a component? We can also implement this whole app with a single component, but that would leave us with a really big component. We need to start thinking about splitting responsibilities. Here's the thing. Too few or too many components are both bad. You need a good balance here. Let me give you one important pointer to identify a good candidate for a component. Every time in the UI you have many items that share similar data or behavior, that's a candidate for an Item component. The play numbers here will have the exact same behavior. Each click on a number will invoke logic to determine if that click is good or bad. The numbers are going to invoke this behavior with different data, but the nature of the behavior is similar, so this is a good clue that we should extract a play number into a component. Let's do that. I'll take this part and get back here once I'm done designing the component. Okay, what's a good name for this component? Let's call it maybe Number. Each react component receives a set of props, and we'll make it return the exact same value that we had before. So now back in here, we do Number, and this Number will need to receive the property of which number to display. So let's just pass it here as a prop, number ={number}, and because it's now a prop in here, I can access it as props.number. I don't need the key here anymore. The key is needed on the immediate element in the loop, this thing in here. So let's cut this part and put it back in here and make sure things are still working. Cool! Here's an important question now. What is wrong with what I just did? Just because what I did is working doesn't make it right. One part of the change I just made is fundamentally wrong, and you should be able to identify it. And I'm talking about this Number name. By naming my component Number, I am actually overriding the native Number class that's available in

JavaScript. This Number constructor is used in JavaScript to convert a string into a number. For example, say that these edges of the random range are received for some reason as strings. If you just use them as strings here, things will not work as expected. So one solution is to call the Number constructor on this to change it into a number. But guess what? This will not work now because I've named my component Number. So in here I am not using the Number constructor anymore. I am using the component I just made. So this is bad. Be aware of the top-level JavaScript objects in your scope and do not override them. A good way to avoid this conflict is to always name your components with two words instead of one. Maybe we call it PlayNumber instead of Number, which means we need to change this into PlayNumber. And now this Number call here is using the native JavaScript Number. Okay, now that we have a shared component to represent a single number, we can define behavior on this number. Let's go ahead and define an onClick behavior. We need to identify when a number is clicked. So I'll inline an error function here and just use a console.log line that we have clicked on a number. So we need props.number here. And let's test this change. Click around and make sure the number values are being reported. Here's an important question for you. How exactly is this working? We have a function here, and this function gets invoked on each click. What is the JavaScript concept that's at play here that makes this click handler access the value of each number? The onClick Handler is a function within another function, the PlayNumber function. So focusing on just the play numbers, we created nine top functions and then nine click handler functions in them. So what is making all this work? And the answer is JavaScript closures. Each onClick function here closes over the scope of its owner number and gives us access to its props. We have nine onClick handlers, and they all have different closures closing over different scoops. This is important to understand and remember when working with stateful function components in react because they do depend on closures. And this could also be a source of bugs later on if we forget to refresh the scope of a closure. So you just need to start identifying when closures are being used in your function components. Okay, what other components should we extract? So the concept behind extracting a play number here is reusability. We want to make a play number reusable and define some shared behavior on it. The other guideline for extracting components is readability and making components smaller in general. An example of this would be here. This part here is displaying stars, but that information is not immediately visible. I could make this part a little bit more readable by rendering a component here. Maybe we call this one StarsDisplay, and StarsDisplay is a simple component that receives props and returns what we had before. But now we have a list of adjacent items here, so we need to make this into a single element because every react component needs to render a single element. Now, if we don't want to introduce any new element nesting in the tree, we could just use the react fragment concept and wrap this whole

thing with an empty GSX tag here that represents a react.fragment. The other challenge about this change is we previously had access to the number of stars when we were here, but now we added another level abstraction. So we need to flow down the number of stars to the StarsDisplay, and it doesn't have to be the exact same name. I could say this is just the count of how many stars to display, and this count can be assigned to stars, and now in here instead of stars, the range is props.count. I still need the key here because that is the dynamic child inside the map. But with that, let's test this change, and things seem to be working. Now you can definitely think about extracting more components out of this StarMatch component, but just don't overdo it. Let's just run with these three components for now and focus more on the interactions between them. So let's start thinking about the next react concept here, which is how to make the UI an exact description of all the state elements in this app. We've already made the UI describe how to render a number of stars, but that's only one state element. We have more. We need to think about picking a number next.

View Functions: State => UI

Let's now start thinking about what will actually happen when we click on these numbers. There's some math logic that we need to come up with, and that's mostly JavaScript. But here's the thing, for every behavior you're trying to implement in the React application, there are two aspects to it. There's the actual logic for the behavior, and there's the UI logic to describe what this behavior is going to be changing. So when we click on a number, we're going to have to change that number's color based on its new state. We're going to have to do some math computations to figure out this new state. But we also have to reflect this new state change to the UI. And it's often easier to start by doing the UI logic. First, however, to design your UI logic, you need to come up with what elements you need to place on the state. Because we want React to change the color of every number we click, we will have to maintain some containers of data on the state. If you remember the number statuses, we have candidateNumbers, we have wrongNumbers, we have usedNumbers, and we also have the list of availableNumbers. Which of these containers should we place on the state to enable us to describe all the possible states? And the general advice in React stateful components is that you should minimize the state. Don't place on the state anything that could be computed from the other things that you place on the state. For example, wrong numbers can be computed from candidate numbers because we have the count of stars and we could just sum the candidate numbers. And if the sum is greater than the count, then we have wrong numbers, so we shouldn't really place the wrong numbers on the state. Also, the list of usedNumbers and the list of availableNumbers are just the inverse of each other. If the

number is used, it's not available, so we could just use one or the other. We could start a usedNumbers container as an empty array, and then every time we need to mark a number as used, push a new element that array. Or we could start the list of available numbers as all the numbers and every time we need to mark a number as used, we remove number from the availableNumbers. I think having the availableNumbers on the state is a little bit more interesting than having the usedNumbers. So when you design your state elements, make sure they're enough to represent all the possible states and also make sure they are the minimum to represent all the possible states. So let's make them official. I'll start with the availableNumbers here, we'll put them in constants, availableNumbers, and you also get a setAvailableNumbers from Reacts useState. And in here you do useState and you pass in the initial value. Now the initial value of all availableNumbers is the full range. So we're going to come back to here and put the full range of the availableNumbers here. And similarly, the candidates in a constant and you get setCandidateNums. And in here we useState, and the initial value for the candidateNumbers could be just an empty array. But here's the interesting thing about the increment that we're doing. We don't really need to worry about the transactions on these candidateNumbers and availableNumbers yet. We just need to come up with a UI to describe them. So we can just use mock values in here. Let's, for example, say that 2 and 3 are candidates. And let's say that we only have 1, 2, 3 and 4 and 5 as available, which means 6, 7 and 8 and 9 are used already. We can't play them anymore. So by using mock data like this, we get to focus on the UI description and not the transactions on the state. Okay, so now 2 and 3 needs to show up as candidates. They will be wrong candidates rendered as red if the number of stars is less than 5 and they will be blue candidates if the number of random stars is more than 5. And we know that 6, 7, 8 and 9 needs to show up as green, used, not available anymore. So the number component needs access to this data, both the availableNumbers and the candidateNumbers, and actually the number of stars, to be able to render itself. So we need to pass some kind of data to the PlayNumber component. Let me put things on multiple lines. Let me also put the number component here on multiple lines. And now let's think about what kind of data we need to pass to the PlayNumber component to make it render itself. You could, for example, think about passing the state elements to each PlayNumber component. I could be passing the availableNumbers here and the candidateNumbers and the stars to make a PlayNumber decide its own style. However, this is too much information for a single PlayNumber. A single PlayNumber doesn't care about all the availableNumbers. It doesn't care about all the candidateNumbers. It only cares about whether itself is an availableNumber, a candidateNumber, a usedNumber, or a wrongNumber. So instead of availableNumbers, we could pass this PlayNumber component Boolean attributes, like, for example, is it used? Is it a candidate? This is a little bit better than passing the whole information

down to the PlayNumber component. You want the component props to be exact, only the data that's needed to render the component and nothing else, because by passing more information than needed, the PlayNumber component would be rerendering itself in a few cases unnecessarily. However, instead of passing multiple Booleans about what status the PlayNumber is, we could just pass the status itself directly and pass just one value for the number component. And to do that in a readable way, maybe we can introduce a number status function and pass the number ID here, the actual number, and this function is going to return what status the number is using all the data that's available here in StarMatch. So let's do that. We'll introduce a function, call it numberStatus. This function takes in a number and it will return one of four values. The order of the conditions that we're going to write in this function actually matter because if the number is used, it can't be a candidate or wrong. So let's start with used. A number is used if it's not in the availableNumbers. So we could just have an if statement here and do a check on this availableNumbers array and say something like if the number is not in the availableNumbers array, and in JavaScript we can just use the includes method on an array. So if availableNumber includes number is false, the number is used. So we could just return used in here. Now if the number is in the availableNumber array, now we can check if it's a candidate. So is the number in the candidateNums array? And we can use a similar check here. If number is in candidate, that means the number could be either a correct candidate or a wrong candidate. So we're going to return one of two values here. And finally, if the number is not used and it's not a candidate, it is an availableNumber. So let's return available here just for consistency and so that we can look up the style from the theme. By having consistent status for each number, we can just look up the style of the number from this theme, So I've got available, used. Now we need to figure out wrong and candidate. Candidate means we're still playing. And wrong means we picked candidates, but they are wrong. So we need to compute a value now. Remember how we didn't place the wrong numbers on the state because we determined that this is something that we can compute? We need to do this computation now. So let's call this candidatesAreWrong. So the candidates are wrong when the sum of all the numbers in the candidateNumbers array is greater than the count of stars. Now, in the utils object we have a way to summon array. So we're just going to use that. So candidatesAreWrong when utils.sum for the candidateNumbers, the whole array of candidateNumbers, is greater than the count of stars. This makes all the current candidates wrong. So in here, in this condition, if the number is a candidate, then we're going to say, are the candidates wrong? And if they are wrong, then this candidateNumber has a status of wrong. And if the candidates are not wrong, then this candidateNumber is an actual candidate that needs to show up as blue. Okay, let's test this logic. We are passing a new status property here, and this status property is going to be computed from all the mock data that we placed on the state. So,

in the PlayNumber component, we need to introduce conditional styles based on this new status property. And we can use React style object for this. Now the style we need here is really simple. We need to just change the background color of the number. So we do backgroundColor in here, and this backgroundColor is the value that we need to pick from the colors object. And since we have the number status here, this is just a direct look up of the colors object. So the background of a PlayNumber is the value from colors that is associated with props.status, the new property that we just computed and passed to each number, this one. And we can test. And check it out. The numbers that are not available are showing up as selected, and the numbers that are candidates are showing up as blue here because we have nine stars, and two plus three is still less than nine. If we render a case where two plus three is more than the number of stars, they are showing up as wrong. So what I just did here is complete UI logic. I didn't do any transactions on the state. I did not do any math to figure out if the number should go in the candidateNumbers or if it should be removed from the availableNumbers. That's what we're going to do next. But just notice how I focused on what state elements to introduce and what changes in the UI description to do to make it honor these new state elements. And now that we're done testing this UI description, we can go back to the actual initial values that we need for each state element. We need empty array for the candidateNums, and we need the full list of numbers, and it's the exact same thing that we're using here to represent all the available numbers. So I can take this one here and make it the initial value for the available numbers, and the UI is rendering all the numbers as available, no candidates regardless of the count of stars. The logic that we did here in the numberStatus function,, by the way, is not very important. This logic will depend on the application you're writing. The React concept that you need to appreciate here is how we've introduced elements on the state of this application, mocked the initial values of these elements, and changed our UI description, the function of our data, to honor these new elements. Now we're ready for the next step, which is to figure out how to transact on these new state elements, to complete the story of this app. We'll do that next.

Behavior Functions: State => New State

Now that we have state elements and we have a UI description to honor these state elements, we need to come up with the logic to figure out how to do transactions on these state elements. This is the logic that's going to happen on each number click, and it needs to be here in the top-level component where the state is owned. So let's define a function, call this function what to do on each number click, so onNumberClick. This function needs to receive the number that was clicked. Here are the decisions that we need to make in this function, based on the current status

of the number, what should be the new status of the number? Now because we have computed the current status for each number, we shouldn't do that again. Each number here is aware of its own current status, so when we click on it, it can communicate this status to the onNumberClick. So let's test this first increment. When we click on a number, let's report the status of the number that was clicked right here from the onNumberClick. We're going to have to pass this new behavior, the onNumberClick behavior, to the PlayNumber component. That's the component we're going to be clicking. So let's define a new prop here, call it onClick, and we pass it here as onNumberClick. This is how the parent component is telling the child component what behavior to invoke each time it's clicked. In the PlayNumber component, we will now be receiving this behavior as a prop. So instead of console logging here onClick, let's invoke this behavior. We can access it as props.onClick. This is what we're going to be doing onClick, we're going to invoke the behavior that was defined by the parent, and we can pass in what number is being clicked. We can also pass in what's the status of the number that's being clicked instead of computing the status again. So in here we can also pass props.status. Now, down in the onNumberClick, this function is receiving the number that was clicked and the current status of the number that was clicked, and the logic here needs to determine what is the new status of this number that was clicked. So now for these computations you have to take into consideration all the possible statuses for this number. We could be clicking on an available number. That's good. We could be clicking on a used number. That's not good. We should not be picking another used number. Once the number is used, it can't be used again, so that's actually a good condition to start with. If the current status of the number is used, we can't do anything here. We should just return do nothing. Now, if the current number is not used, then we need to make it into a candidate number, but also remember that we have an existing candidateNums array, so we're really appending a new number to the candidateNums array. And this should happen anyway. Let's come up with a new candidate numbers, and this is the existing candidateNums array concatenated with the number that was just clicked. From this point, we have two possibilities. The set of new candidate numbers could be just candidates, and for that case, we just need to place this new candidate numbers array on the state of candidate numbers. But the other possibility that now with this addition to candidate numbers we have a right pick, and in that case we need to update the game to mark these candidate numbers as used and redraw the stars. So there are two branches here. So let's introduce an if statement. The logic of this statement is, do we have an exact pick or not? So we need to sum the new candidate array. We can use utils.sum for that, sum newCandidateNums, and check this number against the count of star. If it doesn't equal the count of stars, then we don't have a correct answer, we just need to mark the number as candidates, and we have a setCandidateNumbers function to do that. We just do

setCandidateNumbers, the newCandidateNumbers. However, in the else branch here, that means the sum of the newCandidateNumbers equals the count of stars. We have a correct pick. All the newCandidateNumbers should be marked as used. They should be removed from the available numbers array, and we need to reset the candidate numbers to an empty array. And we also need to redraw the number of stars. So we actually need to invoke all three state_____ function, but we need to figure out the new available numbers. We need to remove the new candidate numbers from the set of available numbers. So let's come up with newAvailableNums and compute that. This starts from availableNums, but then we can filter out all the candidate numbers. The JavaScript filter function will give us access to each available number, so give it a name here, and then the condition that we need here is the available number included in the new candidate numbers. So we can do newCandidateNums.include, this n, and if this condition is true, we need to filter out the number from the available numbers, so we need to negate this for the JavaScript filter function. If the number is not included in the new candidate numbers, keep it in the new available numbers, otherwise, remove it. Now that we have new available numbers we can setAvailableNums. This is the state Hook _____ updater function, and the value is the newAvailableNums. We also need to reset the candidate numbers, so setCandidateNums to be an empty array again, because the player is going to be picking more candidate numbers. The other thing that we need here is to redraw the number of stars. We need to do that as well. Now that we have a correct pick, we need to redraw the number of stars. But here's the thing, we can only redraw numbers that are playable. We can't pick any numbers of stars. This is where the util function randomSumIn comes in handy. This takes in an array and the max sum, and it will compute all the sum permutations in that array and pick a random sum from that that's less than the max here. The max is 9 for this game, and this array that we need to pick random sums from is the new available numbers array. So back in the onNumberClick, to redraw the number of stars we're going to do setStars, and in here we'll do utils.randomSumIn the newAvailableNumbers, and our max here is 9. Redraw a number of star that is playable, and I think we can test. I see a typo here. This includes needs an e. This is the exact type of problem that a strongly typed wrapper around JavaScript would detect. A very good next step after this course is for you to explore something like TypeScript. Okay, let's test our progress so far. We have nine stars. Pick 9, 9 is selected, and the new number of stars is picked. Pick 8, and we have a new random stars, which is also 8, and now we have to pick two numbers. We have to pick either 7, 1 or 2, 6 or 5, 3. We can also pick 4, 3, and 1, and now three numbers are selected and we have a new number of stars, which is seven. So if I pick 7, I've got 2, and if I pick something more than the count of stars, it is now marked as wrong. Now we didn't really do anything about the wrong status here, but we don't have to, because, remember, the state of candidates being wrong is always computed. No

further logic is needed for marking a number as wrong. Now here's the thing, I marked the number as wrong, but now I have no way to unmark it. In the final game, you can click on a candidate number to unmark it so that you can win the game. So let's do that next. This is a case that we need to handle right here before we put the number as a new candidate. We need another condition here that says if the number is available, then put it in the new candidate numbers, but if the number is not available, what should we do with it? So I'm going to introduce a new condition right here because this could be a simple turn array. We always need a new candidate numbers array. So in here we could say, if the current status of the number is available, then we can mark it as candidate. If it's not available, that means it's part of the candidate numbers array, and we should remove it. And we can use the JavaScript filter function to do that declaratively. We can say candidateNums.filter the number that was just clicked. So this will give us access to existing candidate numbers, and we need to filter out the case when the candidate number does not equal the number that was clicked, so keep all the candidate numbers except the number that was clicked. And this is _____one-to-many equal. So there we go, let's test. Let's pick 1, and now we have four. I'm going to go ahead and pick 2 and 2 is selected, and I'm going to pick 8, now the set of candidate numbers is marked as wrong, and if I pick 2 again, it should be unmarked as candidate. Pick 8, not a candidate anymore, so this worked for both candidates and wrong numbers. And if I pick an available number, then the game should continue as normal. So once again, don't focus too much on this logic that we did to make this feature work. This logic will be different based on the application. Just appreciate the fact that we're just transacting on the state, and the UI of the description that we've already made is ready for us to reflect these transactions.

Resetting the State

The code we have so far is under rgs3.5. And guess what? We have a playable game. We can actually win this game now, with 7, we can pick 6 and 1. There's 4, another 7, 8 and 9, and done. But now the player has no way to play this game again. So when we reach this state, let's put a button here to play this game again. We need a condition here in the left section of the game, and that's a simple condition that should determine if the game is done. And rendering this starsDisplay component should only happen if the game is not done. If the game is done, we should render something else, the Play Again button. Now you might be tempted to put the computation about whether the game is done or not right here. In fact, it's really easy. The game is done when the available nums.length equal to 0. That's it. That's the condition that we need for when the game is done. And if this condition is true, we need to render the play again UI logic. If

this condition is false, we need to render the starsDisplay logic that we had before. So let's assume we're going to have a PlayAgain component, and we're going to render this here if the condition is true. The thing is, whenever you have some kind of computation like this, you shouldn't really do it right here within what the function returned. And that's for many reasons, but mostly for readability. And this is actually true for this other computation here, but I'm going to focus on this one. This is a little bit less readable than doing something like gameIsDone. And that's why you should prefer to do these computations in variables before the return statement. So let's do that. Maybe right where we did the candidatesAreWrong, we can also do gameIsDone. And in here we put the exact same condition that we've tested. This right here is a much better place to do the computations. In fact, the structure of a React component should always be similar. The first few lines should be any hooks into the state, any hooks into any side effects of this component, and then after that, you do any computations based on the state. This is core application logic. While the return statement here is a description of the UI based on all your states and all your computations out of that state. So I think that distinction is handy. Okay, let's write the logic for this PlayAgain component. And that's another thing, when you have conditional UI, it's usually a good idea to put the conditional logic in a component rather than in-lining it here, because that would make the top level component less readable. All right, let's do PlayAgain. PlayAgain is a component. Put it right here. A component that takes props. And it could just return a simple div. Give it a class of GameDone, this is the class that I've prepared for this section. And in here we can just have a simple button that says Play Again. And before you worry about how to play again, test that this logic is rendering correctly. Go over a complete game in here and make sure that the Play Again button is going to show up. Test that. And there you go, Play Again is showing up. Okay. How do we play again? You have two important ways to play again. We can simply reset the state, and it's really easy. You can reset the state to the same initial values that you've started with on the very first render of the game. So we could have a function here. Let's call this function resetGame, and this is a very simple function that will basically do three sets state calls. It will set the stars to the initial value, which is this thing in here. It will set the available numbers to the same initial value that we started with, which is a range from 1 to 9. And it will also set the candidate nums to an empty array, and this would reset the game because now we're back to the initial status of the game. And to use this resetGame, we need to pass it to PlayAgain. So in here we can give it any name. I'm going to just name it onClick, right, separation of responsibilities. This PlayAgain button doesn't really need to know what is going on onClick, it just needs instructions of what to do onClick. So onClick, this PlayAgain button receives it as a props, and on its own onClick event it can just invoke this props that came from the parent. That's it. Let's go ahead and test. We need to play a full game in here. And now when I click PlayAgain, the

whole game should be reset and I can play again. So this is method one, and it's usually enough. However, things get complicated when your components have side effects. By side effects, I mean things like subscribing to data or starting timers, which is something we're going to do next. Remember, we're going to have to implement this time limit on the game, so we're going to start a timer, and the timer in a component is basically side effect. So when you reset the game, you need to also reset any side effects, not just the state of the game. So just keep that in mind. Once we implement this timer, you'll see how the timer is actually a side effect, and you'll see how React handles, creating and removing side effects, and we'll come back to this resetGame function and write it in a better way that also takes into account the side effect of the game. We'll do that in the next video.

Using Side Effects Hooks

We are ready to implement this timer here. Now, this is the number of seconds left. You have 10 seconds to play this game, and we need the UI to reflect the countdown. So every time we have a countdown, we need the UI to trigger a render. This means that we need to place this logic here on the state of the component to enable React to retrigger a UI render for every second that ticks. So let's call this new concept secondsLeft, and let's give it a new state element. So I'll do secondsLeft, and I'll do setSecondsLeft in here. And this is a useState call, and the initial value of the secondsLeft is 10 seconds. That can be customizable later on. The secondsLeft is something that needs to be rendered here instead of the hard coded 10 that we had in this dev, and test that. Now in JavaScript, when you need to start a timer that ticks every second, you have the option to do a setInterval. This setInterval function can be a way for us to implement this countdown feature, but we can also implement it by taking advantage of the fact that React will invoke our StarMatch function here every time a state changes, and the state is going to change every 1 second at minimum, which means we can also use a setTimeout call to implement this exact same feature because we know that the StarMatch function is going to be invoked every second. Using setTimeout here is a bit more interesting for you to understand the nature of a React component's side effects. So let's implement the countdown feature with a simple setTimeout call. But before we do, we need to understand how to implement side effects in React. The only React Hook we used so far is the useState hook. React has a few other Hooks function, and one of them is useEffect. This, of course, is React.useEffect, but it's available here globally in the playground. This useEffect is exactly what the name suggests. It is a way for you to introduce some side effect for this component. The useEffect takes in a function, and it will run this function every time the owner component renders itself. So if we put a console.log line here, this line really

means that the component has done rendering. You'll see this line every time the component is done rendering. So when I click on a number here, the component has to rerender itself because the state has changed, and when it's done rendering again, it will console.log this line, and it will continue doing that every time the component renders, which means we can introduce our own side effect right here inside the useEffect in here. We need to setTimeout after exactly 1 second and give this setTimeout a function to basically change the seconds left and decrement the current value by 1. So we can use the setSecondsLeft here and the new seconds left would be the existing secondsLeft - 1. Very simple. And guess what? Because useEffect is going to set the state in here, and because that set state is going to cause React to rerender the StarMatch component, the useEffect is going to continue to run, and this will actually make a loop. You can test this right away. The secondsLeft are going to start ticking immediately and we don't have any exit condition from this loop. So we need an exit condition, because right now it's just going to continue going under 0. So in here, we could introduce an if statement to exit this looping mechanism. And the if statement is basically if the secondsLeft is still greater than 0, only in that case introduce a timer. But once the secondsLeft are 0, then don't do this timer again. So now the timer should tick all the way to 0, and then it will stop because this condition is going to stop it and we will not introduce a new timer when the seconds are 0. Cool. But here's the thing. This useEffect is also going to run when the other state changes. So when I click a number to mark it as candidate or used, the useEffect is going to trigger again, which means if we run this code and start clicking numbers, this timer is going to be created far more often than we need to, and that might introduce bugs if you're not careful because every click here is triggering a useEffect call and it's creating its own timer. The good practice here is that whenever you create a side effect, you have to clean that side effect when it's no longer needed. The useEffect hook has a mechanism for us to clean the side effect when it's no longer needed. So let me show you this mechanism. Let me comment out this timer code and we'll get back to it later. The mechanism to clean a side effect is in the returned value of this callback function within the side effect. You can return a function here, and React will invoke this function when it's about to unmount the component, when it's about to rerender the component. So in here, the console.log line could be something like this Component is changing, and React is going to invoke this function every time it sees a change in the component and the component needs to rerender, or even when the component needs to be unmounted. remember, the effect itself runs every time the component is done rendering, done rendering. In here, the component is changing and it's going to rerender, so the component is going to rerender, basically. These are the two mechanisms for introducing a side effect and cleaning up that side effect. So if you render this code in here, you'll see that the component is done rendering. And when you click any number, you'll see how the component is

going to rerender, and it rendered, and then it's done rendering. This happens on each click. So what we need to do for our timer is to basically clean up any timer that we introduce in one render cycle if the component is going to rerender. To clean up a timer, the setTimeout call gives us back a timerId, so we can capture this timerId in here. And to clean the timer, to remove a timer, all you need to do is call the clearTimeout call on this timerId. So in here, in the same if statement, we could return the cleanup effect. The cleanup effect is to basically call the clearTimeout call on the same timerId. So now, every time the component rerenders itself, we will remove the previous timer and introduce a new timer, and this way we will be sure that there will be no timer conflicts. You should always get in the habit of cleaning up after your effects. If you introduce an effect, you should always return a function that cleans up after that effect. However, introducing this cleanup function actually introduced another issue with this timer, and you can see that one in action if you click on one of the numbers repeatedly. You'll notice that the timer stops. Basically, you can cheat on this game if you need more time by just repeatedly clicking on a number. This is because the useEffect runs on every state change. And now when it does, it resets the timer that's trying to decrement seconds left using the new cleanup function. So when we click on a number, we're also resetting the timer. Because this is just an enhancement, I'll leave this issue as is in the first version of the game. However, I wrote an article with more details on why this problem is happening and the few ways it can be solved. You can check that one out this URL. So, we have a timer, and that timer is ticking down all the way to 0, and we can play and win the game before the time runs out, but we should also change the status of the game to done when the time runs out and actually have a different message based on whether the time ran out or the player was able to pick all the nine numbers before the time runs out. So this gameIsDone condition can change now and account for the time. So we need to do one more condition about when the game is over. Let's actually create a new variable, call this variable gameIsLost. So the game is lost when the seconds left are exactly 0. And we should probably change gameIsDone to gameIsWon. If you're able to take all the available numbers down to 0, then you've won the game. But if you run out of time, you lost the game. And the game is done if either of these two variables are true. But here's the thing. Instead of managing two states about the game status, how about we come up with a single variable that represents the game status? So let's do gameStatus in here, and we can put the same condition to figure out the game status. If this is true, then the game is won, right? And if this condition is false, then we need to check the other condition. Are we out of time? If we're out of time, then the game is lost; otherwise, the game is still playing. So maybe we call that status active. So this one gameStatus variable captures these two computations. It's usually better to have a single variable rather than two variables that are falling in the same scope of computations. Now, if the nested ternary here is bothering you, that is

really not the point. You can have two if statements here to replace the nested ternary, or even have a function to compute the game status on every render. But I'm just going to move on. Now that we have gameStatus, we need to determine here whether to show the play again or the stars display, the condition is now if the gameStatus is not active, you need to show the Play Again. You show the Play Again in two cases, when the game is won or lost. Also, let's pass the gameStatus itself to the PlayAgain component in order to show a different message if the game is won or lost. So we pass gameStatus as gameStatus in here, and in the PlayAgain component, let's introduce a message. So we'll put that in a new dev here, give it a className of message, and the message here will need to be different if the game is won or lost. So we know that the game status is not active here, so we can just introduce a condition, and the gameStatus is part of the props of this component, so props.gameStatus, and we can simply have an inline condition here that says if the game is lost, render the message Game Over. If the game is won, render a message for the winners. We can also use the exact same condition here to style this message and show it in a different color if the game is won and a different color of the game is lost. And for that we can use React's style property, and in here pass in an object that is based on the same condition. We can, for example, change the color of the message based on the props.gameStatus condition equaling to lost or won. If it's lost, show the message as red. And if the game is won, show the message as green. React's style property is handy to do this inline conditional logic here, that is depending on the component props. And I think we can test. So now if we let the timer run all the way down to 0, the game should be marked as lost, and the Play Again should show up with a Game Over message. But here's the thing. This reset is not working now. Why? Well, because we also need to reset the secondsLeft to 10, remember? So when you need to reset the game, you need to reset all the state, and also you need to reset any side effects that you introduce in the game. Now in here, we really stopped the timer, so we don't have any more side effects. But in case you have a side effect, you need to call the cleanup as well when you reset the game. So in the next video, I'm going to show you how to reset the game by unmounting the component rather than by resetting the state because when you unmount the component, remember, React will also call all the cleanups that you have. And I think that's a cleaner way to implement the Play Again button. Before we do that, if you noticed, the timer continued running all the way to 0, even though the game was done. So we should actually stop the timer when the game is won. There's also another bug in this game. When the timer runs out and the game status is lost, the player can still play the game. Let me show you that. So Game Over, and I can still click those numbers. That does not make sense. There are a few conditions that we need to do, so let's go over them. The first one is right here. We should not introduce a new timer if the game has been won, if the game is done. And the game is done if the available numbers are down to 0. So in here, we should only

introduce a new timer if seconds left are greater than 0 and the available numbers are also greater than 0. These two conditions means that we're still playing, and we should continue counting down. But if the available numbers are 0, then the timer should stop. The other condition that we need is in the onNumberClick function. If we click a number, and the game is actually done, then we should also do nothing. So, we could just add another condition here to the same if statement. If the game is not active, right, if gameStatus is not active or the status of the current number that we're clicking is used, we really should do nothing and not change the state again. So to test that, let the timer run down all the way to 0 and click again and see if you can click these numbers. You should not be able to click these numbers.

Unmounting and Remounting Components

Under the rgs3.7 link, we've implemented the timer, and we've implemented the statuses of the game. The game can be active, won, or lost, and the only thing that stopped working for us is the Play Again button. So it is time for us to talk about that. Instead of this resetGame, I'm going to come up with a different mechanism to reset the game. And that mechanism is to simply unmount the component from the DOM and remount it again. However, to accomplish that, I'm going to make the StarMatch component a container of another component that I will introduce, which I'm going to call Game because now this application is not just a single game. This application will render many games as we go on. So let's keep this ReactDOM.render call to render the StarMatch, but let's rename the StarMatch here into Game instead of StarMatch and introduce a new component and call that component StarMatch. So we'll call this StarMatch, and this is a function that basically returns a game. So this is exactly the same and test that we didn't break anything. Game is rendering, the timer is running down, and now I get this error that resetGame is not defined because I've used it here. So we need to come up with a different resetGame. And to reset the game, you need to just unmount to this component and render a new component. And here is a trick to do that. If you specify a key for this component, remember the key attribute? The key attribute is the unique identity that React uses to identify a component element, which means if the key element changes, say from 1 to 2, React will see a completely different game because the game with the key 2 is a different element than the game with the key 1. So if we place a key here and change it when we click to play again, React will just unmount game 1 and mount a brand new component that is game 2. And when React unmounts game 1, it will reset everything about game 1, including side effects. And when it mounts a new game, it will introduce that with a brand new state for each game. So we can simply introduce a state here for the game ID. Let's call it gameId, and you also get a setGameId call. And you can use the state

hook for that and start it from any value really. So I'm going to start by 1. And when you return the game, we return it with a gameId. This will render the game, but we still didn't replace the onClick mechanism here. Now here's the thing. Because the resetGame is now part of the StarMatch logic, we're going to have to pass a behavior here down to every Game component to design the reset mechanism. So a good name for this function would be startNewGame. And this is what this function is going to do. I'm just going to inline it here. This function is going to set the gameId to a different value. We can just increment the existing gameId. So every time I need to reset the game, all I need to do is change the key attribute. Because the key attribute is hooked to a state in here, I can just change the state and React will unmount the previous game that had the current key and mount a new game that has the new key. So now the Game component receives a prop called startNewGame, and this prop is going to reset the game. So we need to pass it down here to the PlayAgain component. The onClick becomes props.startNewGame. And now when the timer runs down all the way to 0, it looks like we have an error. Oh, props is not defined. So a component here needs props. It's undefined. Let's test this again. When the timer runs out, we should see the Game Over. When we click on that Play Again button, here's what's going to happen. React is going to change the gameId from 1 to 2, and that will make the game unmount, clear all the side effects, and mount a new game with a new state. All the new state elements here are going to be brand new because we are mounting a new game, and there you go. This should happen for both winning and losing the games. So let's test the other case here. Play Again should reset the whole thing. It will reset the timer. It will clean up all the side effects. So this concludes the first playable version of this game. But before we conclude this course module, let me tell you about one other important concept about React components, and that is how to extract some logic into a different entity. If you look at the Game component now, it is big. There is a lot of things going on here in the Game component. The other components are fine. They're small, they receive props, and they render things based on those props. But the Game component is huge. It has states, the initial values of the state. It has effects. It has computations about the state, it has functions to transact on the state, and it also has its own render logic based on the state and the computations. Now the reason this component is huge is because it's the only state manager in this application. It has the responsibility of managing the state, and it also has the responsibility of rendering the game tree. And while this works, there is actually a better way to split these two responsibilities. We'll talk about that in the next video.

Using Custom Hooks

We've completed the first version of this game in the previous video, and we have the code under rgs3.5. In this last video, I want to show you how you can split some of the responsibilities that the big game component is currently handling, and I want to talk about two different things. There are actually three main things that the game component is doing. It is managing the state. This includes initializing values on the state, and it also includes setting values on the state. So let me comment out these two sections here, managing the state and managing side effects and setting values on the state. The other two things that the component is doing is making some computations based on values from the state and also rendering a UI based on these computations and based on the state. So let's focus on the first type. Let's focus on the type where the game component is managing the state, initializing the state, defining side effects, cleaning up side effects and also transacting on the state using set calls in here. This logic really belongs in the same place, and React has a way for us to extract all this logic into its own function. So to prepare for that, let me take all this logic. I'll keep it here in useGame for a reference. We're going to put it in a new function. Now this new function is called a Custom Hook, and it's a special function. It's a special function because it's going to be a stateful function. We're going to extract logic that is basically React hooks, useState and useEffect, and we're going to group these hooks in a single custom function. So this is actually a normal function. It's special in the sense that it is using special React hooks functions, and those functions manage state. This is why there's a very good practice to always name this function starting with the word use. Use something. By naming as use, this is just your manual label that the function is going to contain React hooks, and it should follow the rule of hooks, which we didn't really talk about, but it's a very simple rule. The rule of hooks is that you always use the React hooks function in the same order, so you can't define them conditionally. You can't have an if statement around one of the hooks, that would not work. React requires you to always have the exact same order. We can't, for example, have this if statement outside of the call of the hook. It's totally fine inside the hook, but you can't conditionally use the hooks. That's the rule of hook. So by naming this function with the word use, you can always enforce this same rule within the function. So linters and code formatters can use this hint that the function is a custom hook function. Now what to name the function depends on what the function is going to do, and this function is going to manage stars, available numbers, candidate numbers, seconds left, effects on the game. It is really the whole game state. So we're extracting the whole game state into this function, so I'm just going to call it useGameState. Very simple. Now, the arguments to this function and the return value to this function could be anything. This doesn't have to return an array, for example. We can return a single value, or we can return an object if we want to. We can also pass this function some arguments that are usually related to the game configuration. So, for example, we can make this

into an argument, the number 9, how many stars and how many numbers to render? But I'm just going to keep things simple and not introduce any arguments and move all this code up to the useState hook. The other thing that we need to move is when we set the state, when we do any transactions on the state, we can move that as well to the exact same useGameState hook. And let me uncomment everything in here and start thinking about what we need to do here for this section of the code. So we know that this section of the code needs to be invoked when we click on numbers. This is a good candidate to make it into its own function within the useGameState hook. We can call this function setGameState. So this is the function that you can invoke from any user of this custom hook to change the game state. And we'll just keep it on the same convention of set something, so setGameState. So now we need to analyze this logic that's within this function and figure out what it depends on. It depends on the newCandidateNumbers array. Now we can pass this newCandidateNumbers array to the function and just call the function with the newCandidateNumbers array. And we can also just include this whole computations of the newCandidateNumbers within the setGameState function. But I think it's okay to leave this here and just invoke the new setGameState function with the newCandidateNumbers array, as we've designed it here. So now think about what we need to expose from this custom hook. What does the game component need in order to render itself? Because we could also think about including more logic in the custom hook. For example, we could include some computation logic in the custom hook. And it really depends on what the return value from this component is. What does it depend on? It depends on the gameStatus, that's one. It also depends on the number of stars, and it needs the seconds left, and it also needs access to whatever is inside the numberStatus function, because this function is also part of the UI logic. So it needs access to the availableNumbers and it needs access to the candidateNumbers. So all of these state elements needs to be exposed to the game component. We definitely don't need access to the updaters anymore, because all the updaters are within that single setGameState function that we've designed. So let me return an object right here from the new custom hook that we're writing, the useGameState hook. Let's return an object, and in this object expose the things that the game component needs. It needs the number of stars, it needs the availableNums, it needs the candidateNums, and it needs the secondsLeft, and it also needs access to the setGameState function that we've designed in here. So now this custom hook is the state manager. It manages everything about state. It initializes the state, it initializes the side effect, and it gives us predefined behavior to transact on the state. So anywhere I am using state in the game component, I could just use the elements that this new hook is exposing for me. So in here we can just call this useGameStatefunction. And since it returns many things, let me destructure them here, will read the stars, the available numbers, the candidate numbers, the seconds left, and the

setGameState. And now everything will work the same because we've destructured these elements into the local scope of the game component. So no further changes needed to all the computations. The only other change that we need is here where we previously transacted on the state. We now just need to invoke the new setGameState function, and we've designed it to receive a list of new candidate numbers, so we can just pass the newCandidateNumbers in here to have the game state work with that. Okay, so quick review. We have created a custom hook. It's a function. This is not how you define a function. So here is an error function. This is the useGameState hook. This hook is going to manage the state for us. It will initialize the state, and it will give us a behavior to transact on the state, and then just gives us everything ready in a single place. We use these elements out of the custom hook and then continue to use them normally in the game component, and do the computations on these elements, and use them to render a component. And things will just work. Let's make sure that we don't have any problems. Looks like things are working fine and the game is playable as is. I'm going to let the timer run out and make sure the game over in that case. Play Again, and try to win the game this time. Seven, what we do for 7? Four and 3, then 8, then 6 and then 9. Timer stopped and the game is won. So if you look at the game component now, you'll notice that it is a much smaller component. It only reads values from a custom hook. It has a few computations, including a function to do some computations. It has a single behavior that determines if the number click should actually place things on the newCandidateNumbers or not, and then it just hooks into another entity that manages the state for us. The game component responsibility is mostly to describe the UI that's based on the current values on all the states. Managing the state can be extracted into its own manager. This is how you do custom hooks in React. The final good of the game, as it is now, is under rgs3.9, and it's a wrap on this game. I know this wasn't an easy exercise. This game has some complexities that we have to go through. But again, don't focus on the application specific complexities, focus on the new React concepts that you learned here. Most importantly, that's how we initialize state, how we initialize side effects, how we clean up side effects, and how we define custom hooks to group everything that's about managing the state and transacting on the state, and leave our components to just use these elements, use the state, compute anything out of the state, and then come up with the UI description that reflects the current value's of those states. Another important concept that we learned is how to use the key attribute to unmount a component and remount it again and clean up its side effects and give it a brand new state. Take a look at the final code and read it carefully See if you can find any problems in the game that I'm not aware of, because there might be a bug hiding here in plain sight. And if you find anything or if you have any questions in general, definitely voice them in the discussions forum on the course page. It is now time for us to graduate from this playground tool and for me to show you how to

start a local development environment in your machine. And I'm going to take this exact same game and make it work in a local development environment, and I will share all the code on GitHub.

Setting up a Development Environment

Introduction

Now that you've learned the basics of the React API, you need to set up a local development environment. You can't just keep using the playground tool. Setting up a development environment is not going to be fun. You need to make many different tools work together. The tools have different APIs and each tool will need to be configured, and all of these tools have different release cycles, so you might run into problems or versions might not be compatible and won't work together. The whole environment might suddenly stop working after a certain upgrade in the tools, and you'll have to spend some time debugging environment issues. Furthermore, a development environment is different from a production environment, which means what works for you in development might not work in production. So you'll have to set up a production environment locally on your machine as well to test your changes in a simulated environment. And don't forget that you need an official test environment as well. Tracking what needs to be configured for each environment is challenging. We've been rendering React applications to the DOM. That's just one renderer for many. Your React application might need to be server-side rendered as well. And there's also the test renderer, which treats your application a bit differently. These different renderers often need different configurations. Luckily, there are some high-level tools that you can use to escape some of the nightmares of dealing with environments, the debugging them, and keeping them up to date. The most popular tool in the React ecosystem is create-react-app. As a beginner, this is a very good first step, so let me show you how to use it.

Create React App

The Create React App project can help you have a local development environment for React using a single command, and you'll be done in a few minutes. You don't need to learn any new

APIs or maintain any new configurations. This is both good and bad. It's great to get an env up and running in minutes, but by not going through all the needed steps, you're also not learning what elements are in play. This is why I will also create a React development environment without using Create React App, in the next video. Now, to have a local development environment, you need Node. Create React App is just an npm package and npm is Node Package Manager. So the first step is to have a Node running on your system. You can install Node in many ways. On Windows, there's an installer you can download, and on Mac, I like to use Homebrew for this. If you don't have Node or if you have an old version of it, pause here and go grab the latest and install it. Once Node is available in your system, you should be able to run the command npm -v and see a version like this. Along with npm, you should also have the npx command. So from this point, you have two options. You can either npm install with a -g flag, the create-react-app package, and then when this command is done, you run the create-react-app command and give it an application name. This name could be anything. I'm going to use cra-test in here. This is the first option. This will download the Create React App and store it in your system and then run the create-react-app command to create a React application. However, the better way to do this is to use the npx command instead, and you do that with npx create-react-app, and just supply the name of your application right here. So I'm going to let this run and tell you what this npx command is doing. Instead of keeping a copy of the Create React App package locally and manually maintaining it when it gets new releases, for example, with npx, you don't really do that, you just use whatever is the recent release of Create React App directly from the npm registry. The x in npx is for execute. So we are executing an npm package here, and we don't have that package, so npx will download that package and cache it, but it will also download a new one if needed the next time you run the same command. So at this point, Create React App took over and Create React App is installing its own dependency, namely react, react-dom, and react-scripts, as you can see here. The create-react-app command will create a directory under your current working directory using the same app name that you've put as an argument. You can see here some of the commands that you can run under this directory. For example, we run npm start to start a Web server on your files and open up a browser on that Web server's host import. And there you go, you now have a fully configured local React development environment, and it's a good one. It is loaded with great features that will save you a lot of time. Explore the generated files. Let's open up an editor on this directory. You'll see a source directory in here. This is where the React components are to be hosted. The example app that got rendered in the browser is all here, starting with the index.js file. This is the main renderer and it renders a top level a[[component here. One important difference between what we used in the playground and here in the local environment is that nothing is defined globally for you. If you need to use ReactDOM in

your file, you need to import ReactDOM from ReactDOM, just like that. If you're using JSX in your code, you need to import React because JSX compiles into React API calls. This Create React App is also configured for you to import CSS file, which is yet another way to work with CSS in React. This is all possible thanks to the Webpack tool, which I'm going to tell you about in just a little bit. Now with the defaults in Create React App, you don't need to worry about how to compile JSX. This is all abstracted internally in the Create React App package. However, you can eject this application from that internal abstraction using the npm run eject command, and this action is permanent. This command will copy all the configurations and scripts used by this tool locally to your project, and you can modify them as you wish. This is great. Let's take a moment to explore all these configurations. They're mainly under the config directory and under the scripts directory in here. So if you take a look at these configurations, you'll see here what I was talking about in the introduction. This stuff is challenging. There is a lot to learn here, but it's also a bit extreme because the Create React App package has many more tools than what is absolutely necessary. These tools are great, but optional. So let me show you how to configure your own bare bones local environment and explain what elements need to be in play.

Installing Environment Dependencies

To create a JavaScript development environment for React from scratch, you need to install a few dependencies and wire them to work together. The tools that are usually used for development environments are constantly changing. So what I'm going to do in this video right now might be different from what you need to do when you watch the video. This is the reason I created this guide under jscomplete.com/reactful. I'm going to try to keep this guide up to date when things change. So when you're ready to create your own bare bone environment, you can just follow this guide. In this guide, I also wrote that details behind the libraries and tools that make up a Full Stack JavaScript environment for React. We don't have a lot of time in this course to go over every single element in an environment and every single configuration that's needed, but I'm going to explain the big pictures here, and you can get the more detailed version in this guide. So our first step is to install the dependencies, so I'm going to scan through this guide until we have an actionable step. This is your first step. You want to start with an empty directory. So go ahead and create a directory. We go into that directory and we run the npm init command and this command is going to ask us a few questions about the package that we're creating. Now we can just pass the y argument for yes, and just go with the default in here. So when this command is done, you should have a package.json file in your system. Okay, the next step is to install the dependencies. Now, this guide is designed for a Full Stack JavaScript environment. That means

that it will get you ready to do things like server-side rendering, not just have a frontend React application, and this is why the next dependency here is Express and Express is a framework to create a Node.js web server. This web server is your first step to do server-side rendering of your React application. So let's go ahead and install Express in here. The npm install command will download the Express package and it will place it under a node modules folder. Now, we didn't create a node modules folder, but the npm command is going to create this node modules folder for us and it will also write this new dependency to the package.json file, which we just created with npm init. So let me open up an editor here on this directory and take a look at things. We only have a package.json file, and as you can see, the npm install command immediately wrote this new dependency to our package.json and it also created the node_modules folder and installed a lot of packages under the node_modules folder. These packages came from the dependencies of Express itself, and by depending on the Express package, you're really depending on all these other dependencies and all of them got installed when you installed Express. Okay, let's keep going. The other dependencies that you need are React and ReactDOM so let me put them to install. These are the actual packages that contain the React core library and the ReactDOM library that we need to use to render a React application to the DOM or when we actually use server-side rendering as well. Both of these renderers are within the ReactDOM package. Keep going. Other dependencies that you need, Webpack. Now, what is Webpack? Webpack is a module bundler. So when we take our React application into the file system, we're going to put it into multiple modules. We're going to also depend on multiple external modules like React and ReactDOM. So when we ship things to the browser, we need to ship everything as single bundle because browsers don't know how to work with modules yet. This is going to change in the future. The normal practice to ship JavaScript applications is just to bundle all your application in a single file and ship that to the browser. So the Webpack package and you need the core library here on the Webpack and you also need the command to invoke the library, which is under the Webpack's CLI package. This Webpack package is going to do the bundling for us. We're going to tell it start from file X and then everything that file x require bring it under a single bundle file, and then we can serve that bundled file on our web server and use it to mount the React application to the DOM, so this is Webpack. All of these dependencies that we're installing are local dependencies. They go under the node_modules folder, and the npm install command is going to write them to our Dependencies section in package.json. Now, these are production dependencies, these are main dependencies, which means when you install things on a production server, you're going to get all that as well. And here's the thing, depending on your deployment strategy, you might not really need these to be on a production server. You might opt to package everything locally first, and then just deploy ready-bundled file to production. In

that case, you would not need Webpack and even React and ReactDOM in your production dependencies. But I'm going to assume the simplest deployment strategy here, which is just to push the code to production and bundle things for production right there in the production server rather than pushing bundled code in your source control repositories. Okay, we have Webpack. Other dependencies that we need are the Babel stuff. So you need this whole line here and let me talk about it while it installs. So Babel is the package that compiles JSX into regular React API calls. You need to hook Babel into the Webpack process because Webpack is going to be bundling your system. So we need to tell Webpack that while it's bundling the system, if it sees any JSX stuff, it should invoke Babel to convert this JSX stuff into React API calls and the package that we can use to do that is called Babel Loader, and the other packages are needed for configuring Babel. This is the core package for the Babel compiler. This node package is needed if you're doing server-side rendering. You also need node to understand JSX and the two presets here are really the minimum configuration that you need to tell Babel to compile React. So React preset is for the JSX and the env preset is if you want to use any modern JavaScript and you want to target older browsers like, for example, when you use an error function and you want to target an older browser that doesn't understand error function, this env preset can compile your code to not use error functions. Most modern browsers will understand error functions, and this is why this env preset is customizable. You can tell it which browsers to target. If you tell it to target more recent browsers, then the size of the bundle that it generates will be smaller. Okay, we have Babel. So now let's talk about the development dependencies. So these are definitely development dependencies. You don't need them in production and you can install them with the -D flag to mark them as development. One of them is nodemon. Nodemon is a package that lets us automatically restart node when we change things in node because node requires you to restart it every time you change something and it doesn't have a development mode by default. You have to keep restarting the main node process to test things out. So this nodemon is a watcher on top of the node command, and it will automatically restart the command when you save changes to disk, very handy. The other very important development dependency is ESLint. So let's go ahead and bring in ESLint. If you've noticed, I've made some mistakes while developing in the playground and I didn't have the eyes of ESLint to immediately tell me that I've made mistakes. If you have ESLint integrated in your editor, you don't need to go through these mistakes. You don't need to actually run your code because ESLint is going to immediately analyze your code and tell you that you have a problem or your code is lacking in certain qualities, for example. You can have consistent styling to your code by using ESLint and by sharing the ESLint configuration across team, so the whole team will, for example, use single quotes, instead of double quotes, and they will all do semicolons because ESLint can do these checks on the

code as you type. But beside the consistency in styling, you will also get other very valuable errors like, for example, you've used a variable that you didn't define, that's probably a mistake or a variable that you've defined is never used. That's also a mistake. Why keep unused variables around. Plenty of very valuable feedback as you type and as you save your files, so definitely have ESLint in your stack and in your editor. Now, in here we're changing ESLint to understand the Babel and also work with plugin for React. This plugin is going to make ESLint understand JSX and understand facts like if you're using JSX, then you have to have React in scope and other plenty of recommended settings that are specific to a React project, so definitely a good thing to have in a React project. Okay, let's take a look at the dependencies. We have the main dependencies in here, and we have the development dependencies here. So these are written to the package.json file under the devDependencies section. And if you install things on a production server, you do not get the devDependencies by default, which you don't. You don't need a nodemon process on your production server. You don't really need to run ESLint on your production server. You might need to run ESLint on your continuous integration server, but you can configure that server to install devDependencies. So these are all the dependencies that you need for a bare bone Full Stack JavaScript development environment for React in Node. In the next video, I'll tell you have to configure these tools to work together.

Configuring Your Own Environment

We've installed the dependencies, and now it's time for us to configure those dependencies. The first thing that you need to configure is ESLint itself, and I've included in this guide an example object that you can use to configure ESLint. So to use this configuration, you need to introduce a new file top level here in your project and call this file .eslintrc.js and then paste an object like this in here. This file is already configured to use the Babel ESLint plugin. It is extending the recommended settings, and you can use this part here to customize the recommended settings. For example, if you want to turn off prop validation, you can just uncomment this example here and use react/props off so you don't get warnings about prop validations. Prop-types validations is actually really cool, and it's definitely something that you should consider. Okay, let's keep going. We haven't done or even talked about any testing in this course really. So I'm going to skip over the dependencies for Jest. Jest is definitely one of the best options when it comes to testing React applications. And it's a good next step for you to look into. Another thing I'm recommending here is Prettier, and I think Prettier is in the same family as ESLint. This should really be part of your stack and actually have Prettier configured to work with ESLint because it does work very well with ESLint. And once you have the configuration, you just save the file to

automatically format the file. So take a look at prettier as well. Okay, so in here I am proposing an initial directory structure. You don't need to follow this directory structure. But if you follow this exact structure, then the configuration for Webpack is going to be a little bit easier because these are the defaults that Webpack is going to look for. So let's make these directories. Mkdir dist is for distribution. Mkdir src sources where we place the actual React application and all the other JavaScript files, including the server files as well. And when we're ready to build things for production, Webpack is going to take our source files and it's going to write them into the distribution directory. And under src, we'll create a components directory, and we'll create a server directory to put the web server code in there. Alright, let's configure Webpack and Babel. Babel is easy. We've already installed the presets and installed the core library. All we need to do is put this simple object in a babelconfig.js file. So on the top level as well, you create a new file, babel.config.js, and you put this object in that file. This object instructs Babel to use these presets when it's time for Babel to process any file and then another file on the top level as well for Webpack. So copy this and create another file in here, top level, this time it is webpack.config.js, and paste in the content of this file. And this final is simple. We didn't have to specify entry point or output point for Webpack because we're using the defaults. But we do have to tell Webpack to invoke Babel on any file that ends with JS because this is a React application, and I'm going to be using the .js extensions for all the React components, which means I'm going to be writing JSX in those React components, and I need Babel to run on those files before Webpack picks them up and includes them in the bundle. So that's the syntax that we use for Webpack. We just add a rule here. For every file that ends with a .js extension and is not under the node_modules directory, use the babel-loader on it. And the babel-loader is going to pick up the configuration that we just did for Babel and compile all the modern JavaScript into something that your targeted browsers are going to understand. Furthermore, after all these configurations, you need a way to run the webpack command and to run a web server as well. So that's where you put things in package.json under the scripts in here. You can have many tasks, for example a task to run all your tests or a task to run a web server. This is what I did here. This task here, and I named it dev-server, this task will run the nodemon command, and it will execute babel-node on a file that I've decided to place here under server/server.js, ignoring the dist directory because the dist directory is all generated files, so I don't really need to watch it. So we need to put out an example server.js here, and I have included one in this article. The other command that we need is to create the bundle using the webpack command. So for that, I named this one dev-bundle, and it will run the webpack command with -w and -d. W is for watch, so this would be a watch mode as well for Webpack. And d is for development, so this will generate a development-friendly bundle that we can, for example, debug through and it's readable. It's not minified. It's not

uglified. It's readable. All right, this is all the configuration that you need. Now to test all this configuration, I've included a sample React application in this guide. So let me quickly put these files in the project that we're testing with, so under components we create an app.js, and this is the app.js file. Under src/index.js, we create this file, and I'll talk a little bit about this file. So this is directly under src, and it's named index.js. This is the DOM renderer. This takes in the App component that I just defined, and it will use the ReactDOM.hydrate on that App component. Now we didn't really talk about hydrate previously. We've done render in the playground. But hydrate is related to server-side rendered content, and I'm going to show you what that means. So let's create that web server. This web server is also designed with server-side rendering capabilities in mind. So this is a good simple example for you to parse through and see how you can use a React application to serve or render content within a back-end web server. So under the server directory, we'll create a new file, and I call that server.js as well and paste the content of this file in here. And I want to show you one thing that immediately happened. Look at this. ESLint is immediately telling me that I've used the console.log line, and it's giving me a warning. Do you really need to use the console.log line because you're not supposed to leave console logs behind in your code. But in this particular case, this console log is fine, and there are some ways to work around these cases at ESLint. Alright, so let's keep going. This sample application is ready. Now you need two terminals to run two different commands. So I'm going to split this terminal horizontally here and invoke the command to run a development server. The server will run on port 4242 by default and then invoke another command to run the bundler to run Webpack to take the React application and generate a single main.js bundle file for us to use in the web server. And guess what this web server is already wired to use this main.js file, and the express server is already wired to serve static files from the dist directory. So if you've done everything correctly, your server should be up and running on port 4242 in here. And there you go. The server is running. We have the sample button increment example that we've started with in this course in here and things are working. And things are also server-side rendered. So what do I mean by that? I mean if you go to the settings here, and you completely disable JavaScript in here and by disabling JavaScript, you can't run React in your browser, right? You don't have JavaScript, you don't have React. But if you refresh this application with JavaScript disabled, you will still get the exact same application because it's server-side rendered. So take a look at this sample application and try to understand what's going on. This is really one other good next step for you to dive into. I just pushed the code that we did under this GitHub repo, so github.com/jscomplete/rgs-template. And I've also locked all the versions for their exact numbers as of now. This way, if you want to clone this repo and install the exact same versions, things should just work as is with the configuration that's stored in this repo. But if you want to live on

the edge and get the latest and greatest, in the next video, I'll show you another way to get a bare-bone repo with the latest configuration and a little bit extra power to help you with your React applications.

Reactful

I've labeled this guide as Reactful after a library that I've created to automatically go through all these steps and create a bare-bone repo that is fully configured for server-side render and that's ejected by default. So all the configurations are flat, and all the files that are there ready for you to customize them. That package is called Reactful. This is how you use it. You do npx reactful, and you specify the name of the app. I'm going to use this to recreate the Star Match game on a local development environment. So the usage here is exactly like the create-react-app repo. But the difference is this will be bare bone, no extra features, just the absolute minimum things that you need. It will also be ejected by default, so you get all the scripts and configuration flat and ready for you to configure. And this application will also be server-side ready by default. So this command will create a new directory under the current working directory, and right now it is installing the dependencies. And when the command is done, it will also give you a little bit of instructions on how to start your project. So this command is basically doing the exact same steps that we did here with a little bit extra configuration. For example, when it configures Webpack, it also configures some tasks for you to run Webpack in production and to configure things for production. It will also include a way for you to create a component using the reactful command. So you go into the created directory, and you just run npm start. This will run two concurrent commands. It will run Webpack and of the server in the back end. And the server runs on 4242 by default. So we're going to go in there. So here is the Reactful. Same button example is going here, and this is also server-side rendered by default. So let me run the editor here, and let me show you a few things about this Reactful project. So the directory structure is a little bit different here, and the configuration is a little bit more involved. There are some configurations about extracting CSS. There are some optimization about production. What we need to focus on right now is how to take our Star Match application that we all wrote in a single file really in the playground and see how things fit in a locally managed React development environment right now. So the sample React app that you see here is under components under App.js. So this is where we can start our component. In fact, I'm going to grab the final code that we have for the Star Match game under the rgs3.9 URL and take it to the Reactful project. So this App component is already configured to be rendered and also to be server-side rendered. So I'm going to take leverage of that. I'm going to keep the App component as is and make it return our

StarMatch component. And for that, we can just paste the code that we had before, all of it really, so all the components that we have in here in the App.js file. We've got one, two, three, four components and one custom hook and all the dependencies of these components in that exact same file, including the color theme and the math science object that we used. We don't need to render anything in here, but we need to make the App component return what we previously rendered in the playground. So we get rid of this line and render this line. And guess what? We could test that. Just refresh this page. It doesn't automatically refresh. And it looks like we have a server error, and we have a lot of ESLint errors in here that we need to go through. But let me take a look at the server error. So back in the server log, you should see the error. The error is useEffect is not defined. Now here's the thing. Because this project is server-side rendered, you will get the errors on the server first, not on the client. If we weren't doing server-side rendering stuff, you would get this error in your Chrome DevTools. But now because we're doing server-side rendering logic, the server is going to start using your React application before the browser. So this is something that you need to be aware of when you do server=side rendered content. So this area here is easy to fix. We previously had useEffect available globally in the playground. And in here, we don't have anything available globally. Everything you use, you have to import from somewhere. This App component was already importing React and the useStateFunction hook for us. So we also need to import the useEffect function hook as well. Okay, so let me take a look at all these ESLint problems and check it out. The playground seems to be inserting tabs instead of spaces and ESLint is immediately complaining about that. It says I am configured for two spaces, not tabs. So this formatting job is something that you should have Prettier do, and Prettier should do it on save. Every time you save the file, you should configure your editor to run Prettier. Now I turned off this setting in my editor here for you to see the value of Prettier. If you're using VS Code, you need to check the Format On Save setting. Now back in your file, check it out. When I save, all the problems that are fixable and any indentation that doesn't fit the current configuration of Prettier will be automatically fixed by Prettier when you save right there. So we're down to zero problems. Prettier actually fixed all the problems. All the indentations, any wrong formatting that I had, it fixed that. For example, it figured out that this nested ternary that we did for the game status can fit on a single line, and that's okay. So that is Prettier. Definitely use Prettier. Do not manually format your code anymore. Alright, let's test and check it out. The game is rendering. I have zero styles, but the game is rendering. It looks really ugly without that CSS. So let me bring that CSS and show you where you can include it. There is a styles directory in here, and there's an index.scss. This is for Sass, so you can use Sass in this environment as well. And I can just paste the CSS content that I grabbed from the same URL in the index.scss file as all these classes are in the markup. So let's test, check it out. The game is rendering.

Structuring Component Files

We have a single file that has many components and many functions, and the next step from this point is to extract one component per file. So this StarsDisplay component should really go into its own component. You create a file under components, and you call it to StarDisplay.js, and you paste the content of StarsDisplay in there. Now, because it's now its own component, it will need its own dependencies. You will need to import React. This component is not stateful, so it's not using any of the React Hooks. And you will also need to export something from this file. Usually we export default StarsDisplay, the name of the component, just like this. And the ESLint can help you figure out more dependencies. This component depends on utils, which means I should extract utils firsts, so let's do that. I'm going to extract this utilities object, cut it from App.js, create a new file. I'll just put this file under src directory because this is not really component, and call this file math-utils.js and paste the file in here, automatically format it with Prettier, and export to the only object in this file. so we can do export default utils. And now any file can start depending on this math-utils object, and we need the StarDisplay to depend on that. And to do that, we just import utils from the exact location where we put the math-utils object, which is up one level from here, because I am inside components in this part, and pick math-utils. So now, as you can see, ESLint is okay with this file. Okay, so I'll continue doing this. I'm going to extract all the components, so now we can get rid of this StarsDisplay. We don't need it here anymore, because now it has its own file, and instead import StarsDisplay from the exact location where I placed it, which is under this level StarDisplay. I actually named the file StarDisplay, but it doesn't matter because you can rename it when you import it. But I do like to have the file names match the component names because I think that makes them easy to manage. All right, let's do this for all the other components. We'll create one for PlayNumber. New file under components, PlayNumber.js, import to PlayNumber, import any dependencies PlayNumber is asking for, and export default PlayNumber. And check it out, PlayNumber is depending on colors, so we can make colors into its own module as well. But I think that PlayNumber component is the only one that depends on colors, so I'm just going to move the colors theme to the PlayNumber component just for simplicity here. All right, we've got PlayNumber. Now we can use PlayNumber in here. Import PlayNumber from where I placed it, which is PlayNumber. All right, we'll do the same for PlayAgain. New file under components for PlayAgain, PlayAgain.js. Paste in PlayAgain, format it, bring in the dependencies, which is react first and nothing else, and export default PlayAgain. And in here we will import PlayAgain from where we saved it, which is PlayAgain.js. Similarly, both the useGameState and the Game component probably belong in one file, so I'm going to do that. I'm going to take them both and place them in a new file and call that file

Game.js, paste all the content in here, and use all these ESLint hints to figure out what dependencies I need for this file. So I think we need all of these dependencies, really. And it will format, it needs utils as well, so let's give it utils now that we have it in the exact same place. And scroll through to see ESLint is saying this is defined but not used, so you want to do something with it? Yes, I do. What do we need to do with Game? We need to export it, so export default Game, and check it out. Now, this file is the one that depended on PlayNumber, StarsDisplay, and PlayAgain. This app component doesn't really depend on this anymore, and that's okay. We can take these statements and put them in Game.js instead of the app component here. And back in the app component, now we only need to import Game from ./Game, and I no longer useEffect in here. I only use the StarMatch component. So this is much simpler. This app component becomes my StarMatch component. In fact, I could just export default StarMatch itself, and things would still work. I don't need this abstraction of another app component that wraps over the StarMatch component. This is much simpler. You can do a little bit more. You can split this big Game component into more files, maybe put the useGameState into its own module instead of in the same file. But I think this is a good stopping point for this exercise. Make sure things are still working and the game is rendering. It looks like we have a problem. Cannot resolve StarDisplay, so we definitely need to change that. This says StarsDisplay. We renamed the file and didn't rename the import. Make sure the server is running. Sometimes you need to restart the server. In some cases, the crash is unrecoverable, really, so you need to restart the server, test things in here, and it looks like we have a working game. If you look up the game in the dev tools, you should see the components. Check it out. We have a game with the id of 1, and under the game there is all the PlayNumbers that we have and the PlayAgain component that already rendered. And if I hit PlayAgain, take a look at the dev tools. This Game id is going to increment to 2, right? That's the expectation. And it did. And inside the game, if I start playing, the state of the game is going to change. It's grouped under GameState here because of how we used a custom Hook. Now unfortunately, because there's a timer that ticks all the time, this GameState is going to refresh every time the timer ticks. But you can test things out by stopping the timer if you want to. I pushed what I did here to get StarMatch to work locally on the reactful template, all on GitHub for your reference under this repo, jscomplete/rgs-star-match.

What's Next

It is a wrap on this course. This course was just an introduction to the basics of React. There is a lot more to learn from here. Check out the official documentation next. It has some great content that I think you're ready for. If you want to take a much deeper dive into React, guess who has an

Advanced React course on Pluralsight? Queue this up for you to go through next, but before you do, I recommend that you build one simple React application all by yourself now, something similar to the Star Match game. In fact, I have a book where I go over building three more games, just like Star Match. You can play the games of this book at jscomplete.com. Pick one of these games and run with it on your own. You can also improve Star Match and add more features to it. Think about managing points based on how fast the player wins the game. Maybe store these points somewhere and create a leaderboard page. And finally, if you're planning to use React with REST APIs, you should really take a look at GraphQL. It is a much better solution to working with data. I have some reading resources about GraphQL here at jsComplete, and I also have a Pluralsight course about the topic. Thanks for taking this course. See you in the next one.

Course author

Samer Buna

Samer Buna is a polyglot coder with years of practical experience in designing, implementing, and testing software, including web and mobile applications development, API design, functional...

Course info

Level	Beginner
Rating	★★★★½ (2153)
My rating	★★★★★
Duration	4h 2m
Updated	20 Apr 2020

Share course