Using React Hooks

by Peter Kellner

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi. I'm Peter Kellner, and welcome to my course, Using React Hooks. I'm a part-time docs writer for Microsoft, part-time author, part-time conference organizer, and full-time learner. React Hooks represents a huge step forward towards building a fully functional React app with minimal use of class components. Let me convince you that React Hooks, based off of functional components, are the best way to build React apps. Managing state and lifecycle events with Hooks is much easier, requires less code, and is much more readable than doing the same with React class components. Hooks are how JavaScript was always meant to be programmed. The major topics that we will cover include built-in React Hooks, including useState, useRef, and useEffect, advanced Hooks like useReducer and useContext, we'll compare how React Hooks in functional components replace more complex code in class components, and finally, we'll build a real-world React app using Hooks that includes a custom React Hook designed to handle CRUD operations running against a REST server. By the end of this course, you'll completely understand how to, using React Hooks, make your React functional components work with state and component lifecycle events in your apps. You'll be able to build an entire React app with just functional components and React Hooks, no class components needed. This is not a beginner course on React itself. If you are not comfortable building simple React apps that use state, I suggest you

start out with the React: Getting Started or React Fundamentals course, and then take this one after. Join me and learn how you can use React Hooks in all your apps by watching this course, Using React Hooks, at Pluralsight.

Start Using React Hooks with useState, useEffect, and useRef

Why You Should Care About React Hooks

React Hooks adds the ability to manage React state and interface with real lifecycle events in React functional components. It does it in a clear, composable, declarative way. When I first read about React Hooks, I immediately loved the concept. I thought about how they could make developing React apps more straightforward and also with less complexity. I could expand more on React Hooks and tell you about how they improve the development experience by using composition instead of inheritance, how they bring to the forefront the best concepts for mixins, composition that is, and also how they significantly improve the experience of programming React. React Hooks makes the need for the patterns, prop drilling, render props, and HOCs, higher-order components, unnecessary, all of which can lead to complex code that's hard to write, hard to improve, and hard to maintain. Bottom line, React Hooks are a big plus to the React programming ecosystem. In this module, you'll learn how to use the three most-commonly used Hooks, useState, userRef, and useEffect. First up, useState. Over the past couple years, I can say confidently that the frontend programming world, regardless of what framework you are using, React, Angular, View, and others, have moved to using state as a way to build highly performant, easy to reason about web apps. The idea is basically every time the user sees something new on the screen, it's tracked as a new state. The simplest example of this is what happens when a browser user types into an input field. Each character the user types fires an event, and that event replaces the current value of what is currently stored in state with a new value. Both the old state and the new state values are maintained, making development and debugging easy. Throughout this course on React Hooks, I'll be using a conference website to show you React Hooks in a real-world production environment. The use case I'll first talk about is what happens when you land on a conference site home page and you want to sign up with your email to be notified of

upcoming events. Here we have a simplified home page that includes an input text field to enter an email address, and a button to press once the browser user has typed it in. As you can see here, I'm typing into the input field, and as I'm typing, you can see below my input, my typed input being echoed. That's because in our React app, we have our JavaScript listening to the onChange event of the input field, and every time it fires, we update our internal React state to the current value of the input field. As that state changes, the React app rerenders the field below the email address that we are typing into with the current state. I know I'm getting a little ahead of myself here, but basically all this is done in less than 10 lines of code. You can think of this as simply two-way data binding. Whatever we type into the text field gets rendered out in the field directly below it. Next up, we'll build this functionality from the ground up. I invite you to code along, as it's very simple and straightforward. I'll briefly run through setting up a project using my favorite tool chain for React, Next.js. Then, we'll dive into coding post haste, I promise. You are also welcome to review and grab the code from the GitHub repo Pluralsight course, Using React Hooks. This module's code is in the folder 02 Basic React Hooks. The code for each clip is in its own folder, making it easy to pick up and start coding along with me.

Setting up Our React Tool Chain

Before diving into the useState React hook as we just talked about, we first need to create a React app that will host our demo page. Like other spot frameworks, there is some setup necessary before we can do any coding of our conference app homepage. The workflow necessary is often referred to as an integrated tool chain. On Facebook's React home page, a few of these tool chains are listed. Create React App is the first one and is an awesome way to start, but in my opinion has a fatal flaw. It specifically does not support server-side rendering and, as such, leaves you with a first page download problem. The second toolchain listed on the React homepage is Next.js, often referred to as just Next. It does support server-side rendering and is the one that we will use for all our examples in this course. Just a side note, if you want to learn a lot more about Next, check out my recently released course here on Pluralsight titled Building Server-side Rendered React Apps for Beginners. Let's get started with Next and build ourselves a simple React app. The only prerequisite is you have to have the latest version of Node installed on your computer. If you don't have it, head over to the Node.js site, choose whatever is the appropriate download for your computer, Mac, Windows, or whatever, and install that. Creating our app is simple. I'm going to follow exactly the steps outlined here on the nextjs.org site. I'm going to open a command prompt in our empty directory, then enter the last three commands shown here as the first step toward creating the Next React app. That basically sets up our entire

React toolchain. To build and run our app, we need to add three script commands to our Node package.json file. Once done, we are ready to build our app for both development and production. Opening this directory in WebStorm, my IDE of choice, you can explore all the directories and files that were created. Of course, you can also use whatever IDE you want. There are great ones out there, including Visual Studio Code, Sublime, and others. Though we've set up our toolchain and web app, we still need to create our first React app. We do that by adding a new JavaScript file in the empty pages directory we previously created. The name of the page will also act as the route to that page. If the name of that file is index.js, that becomes our home or root page of our app, so let's just do that. For index.js, let's paste in the simplest functional component imaginable and include just one rendered element. And that will be the input element with a placeholder attribute so we can see it clearly when it renders. To launch our app, we run the dev script. That is at the command line, we enter npm run dev, and our site is launched at port 3000. Browsing there, we've got our single input element rendering. Next up, I'll deliver on my promise to show you how we created a demo where the input element listened to its onchange event, and we rendered the full value of the input field that was typed in.

Our First React Hook setState, and Tracking Input Field Value

Let's start out by hooking into the onChange event of our input field. This gets fired on every keystroke, which will allow us to capture the text field value as the user types into the field. We're going to want to update the react state that tracks this input fields value on every keystroke. The useState hook is going to help us here. We first have to declare it by referencing it from our import React statement. It's named export from the core React library, so we just add it in curly braces. We also want to put our hook definitions in the main body of the declared function and before the return that becomes the output of the page. More details about that later. We first have to decide what the type of the state object we want to track is. We could track a JavaScript object with multiple attributes, but typically with useState, we track a single JavaScript object or value. We want to track a simple string value here, which is what gets typed into the input field. The single parameter we passed to useState is we want that value, or string in this case, to initialize to. We will start out with an empty string. What's returned is an array whose first two properties are prescribed for us. Let's use the JavaScript destructuring syntax and name the first value inputText and the second value setInputText. Naming these is up to us, but what they do is not. InputText becomes a read-only variable whose value can only be set with the second value, which is always a function, and in our case, named setInputText. Do you see the next step coming? We've got a text input field that we've added an onChange event to that can capture are

typed in text as it changes. We have a setInputText function that, when called, updates state and that state value gets reflected in the read-only variable we just defined, inputText. Yep, what's coming is very straightforward. We just call setInputText from inside our onChange event and pass the current value of our input field to it. Internally that updates some state that then gets reflected in our inputText variable, also defined as part of the useState hook. Now all we have to do is add the input text variable to our output, with curly braces of course, and we've completed our two-way data binding example. When we run the page, you can see we've achieved our goal, echoing what the user types in. Every keystroke is causing a call to setInputText that updates the internal state of our useState call. That internal state is reflected in our variable, inputText, and then that inputText is just output to the screen. To drive home what we've done here, let's create a history of the text input changes and render that history to the page. Just like we created an ever changing inputText value, let's do the same, but this time let's create a new state value that's an array and initialize it with an empty array. Now, on every input field change event, let's add to the history list array the current value of the text field. Keep in mind we're doing this in addition to updating the inputText value. It's okay to have lots of useState calls in a single method and it's the way useState is almost always used. We could just have one useState command and pass a JavaScript object with lots of attributes, in this case it would just be the inputText and the inputText array. But that just makes for more complex programming with no real gain. The React Team recommends doing multiple useState calls. Next step, let's output the history array on every keystroke. A simple history list with a map in our JSX standard output does the trick. Now running the page, as I type into the input field, you can see not only the rendered text but also the history of the state changes. Make sense? If you understand this, you are well on your way to understanding how to use React hooks. They all work basically something like this. That is, they're available to run just in functional components. They all begin with a three letter u-s-e, or use, and they contribute in one way or another to tracking React state and also help interfacing with React lifecycle events.

Using useRef to Enable Mouseover to Colorize an Image

We now have the first and most commonly used React hook under our belt, useState. So let's move on to useRef, another common hook. Don't worry if you don't quite get useState completely. Through the course, we will be including useState in practically every example one way or another. So I promise, if you keep watching, you will absolutely be one with useState by the end of the course. What is the useRef hook? Basically, it gives our React app a way to directly work with an element in the DOM. Typically, we use React state and, of course, now the useState

hook to change what the user sees. But sometimes what we want to get to is elusive, and we need a way to directly access an HTML element. It's always best to avoid direct access to elements, but sometimes it's just necessary. Let's implement a very useful scenario that takes advantage of being able to access an HTML element directly. Let's program a hover-over effect that turns a black and white image to color when the user hovers over it or mouses over it and, of course, back to black and white when the user hovers out or mouses out of that image. To start, let's create a new page for our app by creating a new JavaScript functional component in our pages directory. Remember that with the Next.js framework, all we have to do is create a new JavaScript file in our pages directory, and the name of that file becomes our URL route when we browse to the site. Let's name the page ImageChangeOnMouseOver.js. Scaffold the simplest possible React functional component we can. In it, let's just put two image elements that each point to their own black and white picture. The way Next.js handles static resources like pictures is, by default, anything we put in a folder named static will be available as a resource to our app with the /static reference. Let's put some speaker images, both color and black and weight, into that static subfolder. Now setting our two source attributes to two separate black and white images, we've got our base page. Running Node to launch our website by typing npm run dev and then browsing to our new page /ImageChangeOnMouseOver, you can see we have two black and white images. Mousing over those images does nothing because we've not programmed our cool mouseover and mouseout effect yet. So what's the plan to make the mouseover work? Let's refactor our image or IMG element with our new element, a functional React component that I'll call ImageToggleOnMouseOver. Following the test-driven development mentality, let's first rename the IMG element ImageToggleOnMouseOver. Since we want our new element to swap the black and white picture for the color picture when the user mouses over it, let's create two attributes, one for primary image and one for secondary image, and assign them to the actual pictures in the static, speakers folder. With Next.js, I like to just put my files in the pages directory that are the actual URL routes. So let's create a new src folder and put our new functional React component there and name it ImageToggleOnMouseOver.js. Again, let's make the simplest possible functional component we can and have it just render the primary image passed in. We have to add a reference to the new component in our pages/ImageChangeOnMouseOver.js file, so let's do that. Then, browsing again to the same page, we've got exactly the same result, but now we have a React functional component we can work with to implement our mouseover functionality. Back in our new component, we know we want to handle the mouseover and mouseout events from our image element. So we add those event attributes to our rendered image element. Somehow, when these events get triggered, we want to swap the native image source attribute. That's where the React hook, useRef, comes in. The way it works is we create a

declared constant, in this case imageRer, and before we render our JSX, declare that constant by calling the hook useRef. What happened is when the component renders, it assigns imageRef to our constant, and now we can access imageRef.current to get to our image attributes. I'll add the lines of code that assign or swap the images in the change events. Looking at this code, notice that our JavaScript is using our reference, gotten from useRef, to effectively assign a new image to the element by reassigning the source attribute. Clear? Let's see if it works. Browsing again to the page, mousing in and out of the images, sure enough, the image swaps in and out, black and white to color, color to black and white, just as expected. So that's useRef in a nutshell. Next up, let's talk about the useEffect hook and how we can combine that with useRef. Let's look at doing the same image swap, but only colorize the image when it scrolls into view. This time, let's add and remove the listeners for scrolling when the image element mounts and unmounts. We will also need to decide if the image is in view when the page first comes up so we can know whether it should be colorized or not even before a scroll event happens. First, though, a quick intro to what useEffect does and then coding the scroll example after that.

The useEffect Hook

Now that we have useState and useRef on our list of React Hooks we know about, let's move on to useEffect, which maybe should be named useComponentDidAndWillUnmount, but since the React team named it useEffect, let's start out by talking about why. We often talk about side effects of functional components as a bad thing that is, in general, one thing we like about functional components is that they can be free of side effects. This means that if we call one with the same parameters over and over again, we'll get exactly the same results returned. Incidentally, when this is the case, we call that a pure component. useEffect, by definition, introduces side effects to React functional components. Specifically, when the component first is called, a function associated with useEffect is executed, and when the same functional component completes, a different function associated with useEffect is also executed. So is this a bad thing? Not necessarily. As we will see next, if we want to add listeners to DOM elements rendered in our functional components, useEffect is the perfect place to add them. Then of course, when the functional component goes away, we want to remove these listeners, avoiding any potential resource leaks in our apps. Maybe now you're wondering why would this not be a pure functional component still? That is, we added listeners, we took listeners away, what was the side effect? Well, I did say that a pure component guarantees the same result based on the same incoming parameters. That doesn't mean your component with side effects might be able to deliver the same thing, it's just not guaranteed. What is the syntax for the useEffect React Hook? It's as

follows. The first parameter must be a function, using the simple arrow syntax, we just create a function reference here, and let's just have it do a console.log for now, so we can see the output. The way we specify what gets run when this functional component exits is we return from this function another function, let's just do that now, and return another console.log. The second parameter of useEffect is just an array that contains a list of dependencies for the component. If this array is empty, then useEffect is only called once when the component is first mounted. If you want it called again before this component is unmounted, you need to have all the values in this array that change. In other words, the values that the rendered output is dependent on. Those values could be things like the true or false value of a checkbox field on the screen. If that changes, then that causes a re-render of the component, and the function in useEffect is called again. That's the theory behind useEffect and this syntax. Let's now put that to use in the next clip by updating our image toggler to handle scrolling, such that in-view image is colorized, and when it starts to scroll out of view, either from the top or the bottom, it's changed back to black and white. Stay tuned.

Using useEffect, useRef and useState for Colorizing on Scroll

Let's take a look at our previous example where we changed the image from black and white to color, based on mousing over and mousing out. We use the onMouseOver and onMouseOut attributes to assign events to the image tag. Essentially, what that did for us was to add the mouse listeners when the image component mounted, and remove those listeners when the image component dismounted. Instead of colorizing based on the mouseOver and mouseOut events, let's colorize based on whether the image is in full view on the browser and allow for scrolling to trigger the calculation to determine that. Here we have the previous code for our custom image control named ImageToggleOnMouseOver. Let's first make a copy of this and rename it ImageToggleOnScroll and of course, change the name inside the file that we export. First thing to do is remove the MouseOver events since we don't care about the mouse over and mouse out anymore. Then let's add our scroll event listener. This time, let's add it using the React hook useEffect when the component first mounts. Remember, the first parameter of useEffect is a function that gets executed when the component mounts. So let's create that now and paste in a function that adds the scroll listener. We have to remove this listener before the component unmounts so let's return from this function another function that removes the listener. I've now got a scroll handler defined, but not programmed. Thinking ahead, let's create with useState a variable named inView and of course, the setter for it, setInView, and we will use that to track whether this current component or image is completely in the browser showing window. By

default, let's assume we have a huge list of images and we'll just say the image is not in view, or false for the initial setting. What's nice is that because we used imageRef to get a handle on our image for colorizing, we can use that ref to figure out where on the page the image is. I'm pasting in a five-line function, isInView, that returns whether the image is showing in the scrollable area. GetBoundingClientRect combined with the size of the window does the trick for us. Next we code our scroll handler to set the inView state based on whether the image is actually in view. Each time the page scrolls, it's calculated again. To finish up our component, we just need to set the source attribute of the image to show the color image when this image component is in view and the black and white one when it's not, That's it for the ImageToggleOnScroll functional component. Let's now make a copy of our ImageChangeOnMouseOver base page in our pages directory and modify it to use the ImageToggleOnScroll instead of the ImageChangeOnMouseOver component. First, let's rename it inside the component and the default export. Then let's import our ImageToggleOnScroll component we just built. Now, instead of returning two speaker images side by side, let's replace that with simple mapping or a list of images that will get displayed vertically to make scrolling somewhat interesting. Just a reminder, any hook we use must be imported. Here on top, we now have these three hooks, useState, useRef, and useEffect imported as named imports. Now, when we launch our node server, npm run dev, and browse to the localhost/ImageChangeOnScroll, you can see our speakers. As we scroll down, you can see them colorizing, and back to black and white as they scroll out of the view. Did you notice something odd? Let me go back to the top and refresh the browser. See it now? My favorite speaker, Douglas Crawford, is black and white. As soon as I touch the scrollbar, he turns into color. Let's look at the code and see if we can figure out what happened. Here we have the image component. This gets run when the page first loads. Do you see it now? I'm guessing you do. We start with the image black and white, and since the page loads, the scroll handler does not get called, and even though Douglas is in view, he stays black and white. To fix this, I'll add to our useEffect a call to the state setter setInView that calls isInView when the component first mounts. Now, going back to the browser, refreshing the page, there is Douglas in color. Are we done? Maybe not. Let's refresh the page again and watch closely. Did you notice the top picture came up in black and white first and then immediately switched to color? That's because useEffect runs after the first render is completed, and if you remember in our ImageToggleOnScroll component, we set the default image to black and white. To fix this, we need to not show the image until our useEffect is called, and then show it. Let's create a new state called isLoading using the hook useState, and set it to true. Then, after our setInView method is called in useEffect, we call setIsLoading to set that state to false. We then add a simple check in our return, which renders the component, and if isLoading is true, don't render anything.

Because now we're checking on the image size before it's pulled from the Internet, we also need to hard code the image within height. One last thing that really important is when isLoading changes from true to false, we want this component to re-render. We make that happen by adding isLoading to the dependency array of useEffect; it's the second parameter. Now back to the browser, and finally our scrolling, colorizing example runs perfectly. Ship it.

Takeaways

You've now seen the React hooks that we will use over and over again, working in real-world scenarios throughout the course. UseState lets us track state really easily with very little ceremony. UseEffect gives us a clean way to set things, typically state, when components start and finish. And useRef gives us the control we need to get to DOM elements when other means are not quite so straightforward. Coming up in the next module, be prepared to see our trivial examples with no formatting morph into parts of a production quality conference website. A much more real-world sign up field will be programmed that includes email validation, as well as configuring a global option for whether or not to show it. Also, we'll learn about how to use the useReducer hook, and if you've ever used Redux before, you learn how you can get all the benefits of using Redux with practically no ceremony, like previously existed. Before diving into all that, we'll take a step backwards and explain more about React hooks so you'll know not only how to use them, but understand more of the basics around them, including a few simple rules to keep you out of trouble. Stay tuned.

Using More Hooks: useContext, useReducer, useCallback, and useMemo

Why Should You Use React Hooks and Does It Matter?

In the previous module, you learned about the three React hooks you'll use most often, useState, useRef, and useEffect. Now that you've seen React hooks in action, let's take a short step backwards and talk about how hooks are used in a more generic sense. Then, after that, you'll learn about four more built-in React hooks provided by the Facebook team, useContext,

useReducer, useCallback, and useMemo. What's the big deal about React hooks? That depends. If you are like me, it's a huge deal. Using hooks means that now I can declaratively create functionality, sorry for the pun there, that before would have taken a lot of what I call ceremony without much added value. That is, prehooks, we had to use the React class component to do what we can now do in functional components very simply. For our echoing input field example, you would have needed to create a class constructor to initialize your state. You'd have had to do some fancy complex programming around calling getState and setState instead of just using a simple React hook that returns a variable and a setter for that variable. It's just trivial now. Do React hooks at anything new to React? Actually, no. They really don't. Everything that React hooks brings to the table, you could do before with React classes, just with a lot more work. So why do I care? The answer is, it depends. I can 100% promise that if you are working on a React team of any size, you will see a lot of React hooks being written and used, so you'll need to understand them at the least. Will you be cast aside as old school if you continue to use class components to manage your state? I don't think so. Should I upgrade all my existing class components to functional components so I can use hooks? I can weigh in on that. No, you should not. There is nothing wrong with JavaScript classes. There's likely not a performance difference of significance. And in any case, if you're still not convinced, when hooks were introduced by the React team, they said exactly the same thing, no need to rewrite. I do plan on using React hooks on all my new components because to me they're just a better programming paradigm. Finally, you may be wanting to ask, can you show me how hooks are better than React class components? Yes, I can, but you'll need to wait until the next module where I go back through several hooks we've created to show you the comparable functionality in class components. I feel strongly that the best way for you to learn React hooks is not to start with what I consider not a great model, React class components that is, and then convert that to React hooks, but to instead just teach you how to use React hooks from the ground up. They make so much sense. In the next clip, we'll talk about some basic usage rules for React hooks, how to use hooks, and some basic dos and don'ts. Stay tuned.

React Hooks Basic Usage and Rules

In the previous module, we dove right into the useState Hook with little formal introduction. Of course, we did use Hooks correctly, but didn't talk about what that means, so let's do that now. There are basically just two rules, React Hooks can only be called from functional React components, and in those functional React components, they must be called just from the top level. No nesting, no calling them inside other functions. The reason is that the React library

counts on Hooks always being called in the same order, every time. That is, if we put our Hooks in conditionals or functions, that could not be guaranteed. What's the best way to ensure that your React Hooks are being used correctly? The React team has provided us with some special ESLint rules that warn us if we use them incorrectly. Does that mean if we don't use them correctly, we will always get an ESLint warning? No, it does not. For example, if you include a Hook in a class component, you'd get no ESLint warning, and nonetheless your hooks would fail at runtime. Setting up lint rules for your projects is easy. The integration with your IDE, whether it's WebStorm like I'm using, VS Code, or others, is straightforward. In the next clip in this module, I'll explain much more about the starting point for our code examples here. For now, though, just ignore the details and let's just talk about adding ESLint to the project. If you want to follow along, you can pull the code from the GitHub repo pkellner/pluralsight-course-using-react-hooks folder named 03-More-Hooks-useContext-useReducer-useCallback-useMemo, and it will be clip02. First thing, you need to create an ESLint file. There are multiple ways to do that. My choice is to create in the base of your project in .eslintrc.json file. It references the React Hooks plugin and associated React Hook rules. These are mine. You're welcome to use them or set your own. You need to make sure you install the npm packages eslint, eslint-plugin-react, and react-hooks. And, it's only necessary to have them installed for development. Finally, in WebStorm, you need to go to the ESLint settings, use search eslint to get there, and choose Automatic ESLint configuration. That's it. Now, ESLint rules will help you while you code.

Updating Our Examples to a Conference Website including Bootstrap

Now that we have the three most commonly used Hooks under our belt, useState, useRef, and useEffect, let's move on to the next four built-in Hooks, that's useContext, useReducer, useCallback, and useMemo. As I mentioned earlier throughout the course, we will be using all the basic Hooks in our examples. That means that as we move forward into these four new Hooks, we'll be using the old ones as part of the demonstration. This means the demo code I show you is going to be more complex. Let's move on to a more real-world web application and leave behind our very simplistic app we built just for demonstration purposes in our previous clips. What you're looking at here is a full-blown styled conference website with a Home and Speaker page. It allows for navigation, routing, that is between Home and Speaker page. Notice the favor speaker functionality that's programmed into the Speaker page. I can click on the heart symbol on any speaker, and it toggles from on to off, bright red to dark red. Notice also that as I scroll the speakers, they colorize as they come into view. Assuming you watched the first module, you'll

recognize that functionality, as we developed the control for that that handles colorization with the interception of the onScroll event and using our useRef React Hook. Let's talk about what's in this app now. What I've done is I've taken just the three Hooks we learned in the previous module, useState, useRef, and useEffect, and using only functional components and these three basic Hooks, I've completely coded this site. This is going to be our starting point to learn the next four React Hooks, useContext, useReducer, useCallback, and useMemo. You may be thinking, why bother, everything works. Are those Hooks just wasting my time? No, there are two primary things we're going after with these new Hooks. One is performance, that is now, even though you don't see it, each time we favor a speaker, all the other speakers get re-rendered with JavaScript, meaning that if we had hundreds of speakers, your users of lower-power laptops might find their fans coming on and some ugly unresponsiveness in the browser that we will fix. Also, though you don't see it, we will add some configuration options to our app.js, such that we can enable program features, including showing the Saturday/Sunday selection check boxes, as well as optionally displaying the Sign me up for more information text box. Not only that, using these new Hooks gives you the opportunity to just do better, more reliable programming because they give a richer set of features. That is, we always do our job through the tools we are given, but it's nice to have better tools, and these Hooks are better tools. How have I changed our previous example code from the last module that just did colorizing based on scrolling or based on mousing over? Let's now look at WebStorm, and I'll do a quick code review of what we've got.

Reviewing Code of Our New Three-hooks-only Conference Website

Before running through the JavaScript that makes up our React app, first some small required changes. We need to install a few packages, specifically bootstrap, which gives us our layout, react-toastify, which gives us a nice pop-up you'll see later, and zeit/next-css, which allows us to import CSS directly into our JavaScript. Speaking of that, we also need to create the file next.config.js to our route, which essentially is our webpack configuration update that allows for the CSS import to work. Also in the static directory, I've added a few more images and a new site.css file to enhance our theming. Let's now take a more detailed run through of our new JavaScript for our conference website. Starting at the top, we now have two pages, index.js, which is our home, and speakers.js, which is our speakers. There are barely 10 lines of code, and all they do is call the common app functional component, which handles the menu routing to go between the pages. Notice that depending on which page was executed, pageToShow returns the component home or speakers. Looking at home.js, it's just a header, which is our nice logo, and it includes our signup component. The menu, of course, does the routing between the two

pages. SignMeUp is a rewrite with some added features of the simple input field and button we programmed in the previous module. Notice now we've added an emailValid state, and on every keystroke, the validateEmail function is called. And if the email is valid, the emailValid state gets set to true. Our button or submit is disabled when the email is not valid, meaning you can only submit a valid email. Makes sense. I didn't mention yet that this app is completely written using bootstrap CSS, which, incidentally, is included at the top of our home and our speakers pages. Notice also that we've added a sendEmailToBackend function that runs for 1 second, then pops up a toast message with a completion status. Again, very real-world like. Switching back to the running browser, you can see as I type and ultimately put in a valid email, the button goes from disable to enable. And when I click on it, a second later up comes way toast message with a nice, successful completion. Okay, that's the home page. Let's take a look at what we have for the speakers page. Just a little more complex, but still all functional components and all just using our first three commonly used hooks. Instead of using JSON, we've hardcoded our speaker data in an array in the file SpeakerData.js. Later in the course, we'll move this data to the REST server, JSON Server that is, and we'll remove this static speakerData class. Looking at our speakers.js page, now in the src directory, you can see we've got four useState calls that track our loading status, whether the Saturday and/or the Sunday checkboxes are selected, and, of course, the speaker array itself. We've got our useEffect here that simulates calling an outside service with a 1-second delay and, on completion, filters and sorts our data ready to be rendered. When it does, each speaker is rendered with another component, SpeakerDetail, where we pass in our HeartFavoriteHandler so when, in the SpeakerDetail component, the user clicks on the heart, the event handler in Speakers gets fired. That's about it. Feel free to play with the app and get yourself comfortable with it. It's all in the GitHub repo. Same as usual, pkellner / pluralsight-course-using-react-hooks This is our starting point for all our examples going forward in the rest of the course. Now for the fun part, refactoring for the four new hooks. Stay tuned.

Preparing React Context for Use with useContext Hook

In early 2018, with the release of React version 16.3, React Hooks is 16.8, the React team shipped a new context API whose purpose is to make accessing data and functions any place in your app very simple and straightforward. Before that, you either had to hack something yourself in JavaScript, or more likely, you would have passed properties around your components up and down complex component trees. It was ugly. It led to the design pattern commonly used known as prop drilling and also to HOC, higher-order components, to basically use class inheritance as a way to pass data around, equally ugly and troublesome. The context API was a gift. That gift just

got easier to program. That is, the new React Hook, useContext, makes it trivial to access context in any of your functional components without any unnatural acts. That is, creating wrapping tags in your render events that literally have nothing to do with the UI become obsolete. For those of you that have used mixins in the past, useContext feels kind of like a mixin in then it lets you add information to your class through a side channel. A perfect use case for the React Hook useContext and of course the context API involve storing configuration information in your app and having it available in any functional component that you want. Say, for example, in our conference web app, we've not assigned our speakers to either Saturday or Sunday. This is real in that at Code Camp, until we get feedback from attendees about what they are interested in, we don't create a schedule, that way we have some hope of spreading the people out over both days somewhat evenly. If you're thinking why not just use environmental variables for this, like process.env in Node, that's a reasonable question. In general, I'd agree with that thinking, and if we really only stored our config as a static JavaScript object, like we will here, that would be completely reasonable. However, in real life, config-type data often comes from a database or other external source, and being able to dynamically load it at runtime is very valuable. I'm assuming you're familiar with a context API, which is what useContext works off of. The idea is that we create a shared context at some level, where all the data below that level in the React component tree, that is, can share data, that is share data without having to pass props into components. When we created our speaker and home pages, we had them both call the app component with a property to differentiate them, that is, pageName = Speakers or pageName = Home. That app component is where we want to create our shared context for the entire app because all the routes we ever plan to make will share this app component. I realized that creating a writing component like this feels awkward. We do it because of how Next.js routes to individual JavaScript files in the pages directory, and this allows for a single root in our source directory that all our pages can share from. To app, let's create a context and export it. That way, other components can just import the context to use it. So here we have export const ConfigContex = React.createContext. That's literally all we need to create the context for our entire app. This is literally no different than what we had before Hooks. Hooks will come in when we consume this context. What's next is in our return of our app component, we wrap our pageToShow with ConfigContext.Provider. Then we pass the attribute value to the provider, which can be any JavaScript object. Let me just create one called configValue, and add an attribute to it, showSpeakerSpeakingDays, and set that attribute to true for now. We'll assign that to the value attribute of the ConfigContext provider. Reviewing, we've now wrapped our app with a context that includes our configValue object. Theoretically, we plan on being able to access that

value from any component that's below this component app. Let's test that by seeing how to look up our ShowSpeakerSpeakingDays Boolean flag in our speakers.js functional component.

Using useContext to Access Global Configuration from React Context API

Here we are in Speakers.js. First, let's import the React Hook useContext. Then let's pull in context by importing ConfigContext from app.js, where we just exported it. Then in our functional component, let's get a reference to our context with our useContext Hook. All we have to do now is, just before our Saturday and Sunday check boxes are displayed, we check our ShowSpeakerSpeakingDays config value from context, close the return in the JavaScript block, and we've got it. Launching the page with npm run dev, we see our Saturday/Sunday check boxes. Go back to app.js, and change the ShowSpeakerSpeakingDays to false, the browser updates, and the check boxes are gone. Mission accomplished. We've got a global config we can use with no passing properties through all our components to get them where we want, and a very clean use of the useContext React Hook. Because having one global config value seems wrong, let me add another titled showSignMeUp, and set that to false for now. I need not make any changes to our app component here because the value is passed, which includes all configure attributes. I simply go to the SignMeUp page, add useContext to the name imports from the React library, that's the Hook, import our configContext from the app.js like before, get a reference to context, then in the render method or the return, because it's a functional component, check the flag, and if the flag is false, don't render our sign up. Back to the browser, you can see our sign me up component is gone on both the Home and the Speakers page. Setting the value back to true, the page automatically refreshes, and the sign me component is back. Now that's a nice way to implement config options in a React app. Next up, we'll take a look at useReducer. Let me just warn you though, if in the past you stayed away from redux and reducers because of all the complexity around dispatch, actions, reducers, high-order components, you may find that using the reducer pattern is quite nice now with how they are implemented with React Hooks. Just a heads up to what I'll first talk about, useState, I'm sure now your favorite Hook, is built on top of useReducer with a very thin layer between them. Stay tuned, and let me convince you to start using useReducer in all your apps.

useReducer Is What useState Is Built On

You've probably noticed that I've been really hyping up useReducer so far in this course without saying anything more about it. The fact that it has reducer in the name makes one think of

reducts, actions, reducers, thunk, dispatch, higher order components, and basically complicated programming scenarios that a lot of engineers, me included, tend to shy away from. You've also probably sensed how excited I am about the useState hook and how powerful that is for simplifying managing state in your app. Let's bring useState and useReducer together. What do I mean by that? UseState is built on useReducer. Let me prove that to you, but first, a quick definition of what reducer means in the context of a react app. A reducer is simply a function that takes in a previousState as the first parameter, an action as the second parameter, and returns a newState. That's it. We're back to the Speakers.js file representing the speaker's page with the pictures. Let's just replace our speakersList useState declaration, that is, instead of const speakerList, setSpeakerList equals useState, here is pasted in the exact same functionality, but with useReducer. Notice, the first parameter I pass to the useReducer React Hook is just our boilerplate reducer, and the second parameter is what to initialize our state to. Of course, we have to include useReducer as a named import from React. And now, to prove it's the same, rerunning npm run dev and switching to the browser again, refreshing our Speakers page still brings up speakers. I admit, useState is a little shorter syntactically, but you've got to admit it's pretty close. Now let's make our reducer more practical, and most importantly, more extensible. Right now, our reducer has only one trick. It sets state to one value. Let's fix that. Here comes the dispatch and action keywords. Don't be scared, it's not a big deal. First, let's pull out the reducer function and name it speakersReducer, just a simple JavaScript refactor. Then let's create a switch statement based on the passed-in action type, and if the action type is setSpeakerList, then let's have a reducer return action data as our new state. By default, let's just have our reducer return whatever the current state that was passed in. Of course, we can't call setSpeakerList anymore because we've removed that declaration from what got returned from useReducer and replaced it with a dispatch name. Remember, it's just a name. We could have called dispatch anything, but by convention, everyone thinks of it as dispatch, so we'll go with that. Moving down in the code to our useEffect hook, we need to update the function call to setSpeakerList with the call to dispatch with the first parameter being an object with the attribute type set to setSpeakerList and the data set to our array that contains all the speakers. This matches with our reducer, so when the reducer gets called by the dispatch method, the new state is returned, it's just the data passed in as the data attribute to the action object. You can think of this as useState is just useReducer with only a default action type. Now we can create more action types as we need them, and we've got a very clear place to do our state reductions. Refreshing the browser, we've done it. This page is loading speaker data just like it did before, but now with the useReducer, dispatch, and inaction. At the moment, our favorite functionality for speakers is broken because

we have not updated that to use the reducer. In the next clip, we'll add a couple more actions to our reducer so we can handle the favoring and unfavoring of speakers.

Using useReducer with Multiple Dispatched Actions

The case for useReducer that we made in the last clip is a great example for how to use a reducer in a real world React app like our conference site with the Speakers page. However, having a reducer with only one action hardly seems worth it. The use state I replaced with useReducer was pretty much as good. In this clip, that will change. We will add two more actions to our reducer. Then, the power of the useReducer React hook starts to show through. The reducers we will add will be for favoriting and unfavoriting a particular speaker. Let's pick up where we left off in the previous clip, that is, with our speaker favorite button broken because we replaced our setSpeakerList function, which we got from the use state hook. Let's replace that setSpeakerList call with the dispatch that requires a valid action type. Our only action type we've defined so far is setSpeakerList. Now let's add two more action types, favorite and unfavorite, which means updating the speaker attribute to favorite or updating it to not favorite, true or false as the value. Since both favorite and unfavorite do about the same things, let's make a common function called updateFavorite in our speakerReducer and then call that from our different cases. Let's fill in the function with a simple map that goes through all the speakers, find the one to favor or unfavor, and then just toggle that. In either case, the full array is returned. All that's left is to add a dispatch invocation to the heartFavoriteHandler instead of updating with the setSpeakerList directly. That's how we would have done it before without the reducer. Testing the page now, we can select favorite and unfavorite on each speaker, and you can tell it's working because the change sticks the color of the heart changes. One final step, though really not necessary, is let's pull the reducer out to its own file, speakersReduce.js. It really just helps for organizing our project and making clear where we update our state from. Next up, useCallback and useMemo. Both will help with performance of your React apps.

Using useCallback to Improve React App Performance

So far in this module we've covered in detail the useContext and useReducer React hooks. We will now cover useCallback and useMemo. They are both used to memoize. With useCallback, memoizing a function and useMemo doing the same for a value. Before going too far, let me first define what memoization is. Wikipedia defines it as an optimization technique for returning cached results, but with a lot more words. That's it in a nutshell. The hook, useCallback, caches a

function and useMemo caches a value. That's it. How can we use these in React and specifically in our React app? Let me tell you, we've got the perfect place. Remember moments ago, we showed the favorite button being exercised and the heart changing from bright red to dark red when the heart is clicked. Well, turns out our React program behind the scenes is not just updating the heart on our speaker image, but turns out it's rerendering all speaker images, including all hearts, light and dark. I can prove it. I'll put a console log at the top of the render function of the SpeakerDetail component. This will let us know every time the SpeakerDetail rerenders. Bringing up the console window, browsing to our speaker page, you see one console log for every speaker. That makes sense. We just brought up the full page. Now, though, when I click on any speaker favorite button, look, they all rerender again. What's with that? So here's the problem. If we look how we call the heartFavoriteHandler, notice that every time the page renders, we pass the handler to the SpeakerDetail page. React doesn't know that that function is not changing, so it rerenders that component again, just in case. The good news is we can add the useCallback hook to the React import, and then the heartFavoriteHandler we call will be a memoized one and React won't feel the need to rerender it. Good enough? Well, not quite. We also need to memoize what the speaker detail page is returning, and we do that with the React.memo call, not to be confused with the React hook, useMemo, this is different. When we define our SpeakerDetail for export, we wrap our functional component in React.memo, and now that returns a cache, or memoized version of our speaker detail component, to the calling component. Now browsing to the page again, favoring a speaker, look at that, only one speaker detail is rendered. Another mission accomplished. Next up, useMemo to optimize retrieving and sorting data. Stay tuned.

Using useMemo to Optimize Filtering and Sorting Speakers

UseMemo is the final React Hook you'll learn about. As I said earlier, it's strictly about memoizing values. Looking at our Speaker.js component, that's the one that renders all the speaker data to the page, notice that just before the return of this functional component, remember, return is the render of the functional component world, we both filter our speakers based on whether they are speaking Saturday or Sunday, and we also sort them by their first names. Granted, that's not a big calculation, but if we did happen to have a lot of speakers and a lot of complex filtering that might even involve external web service calls, it could be a big deal. Our plan is to use memoization to capture the speakers to show, along with the dependencies that will cause a recalculation and then rerender either the calculated or the memoized data. So let's do it. First, we update our React import statement to include useMemo. Then we go back to our calculation

of speakerListFiltered, and let's replace that speaker list with a memoized version of it. Uh-oh, we've got an error. React Hook "useMemo" is called conditionally. Remember in the beginning of this module we talked about React rules, and we installed ESLint to help us not break those rules? Well, that's paying off now. It's obvious why we are breaking this rule. It's because speakerListFiltered is loading as a conditional based on the isLoading flag. What's the fix? In this case, it's easy. Just filter the list first into a constant then pass that constant to the conditional. Nice, the error's gone. Now, when we browse to our speakers page, all works. Let's test the checkboxes for Saturday and Sunday. Great, they both work. Now, how do we know it's working? Back to the code. Notice the second parameter of useMemo is a dependency array. This means that if any of these change, then the memoized version is dumped and the filter and sort are run again. Let's remove speakingSunday. Now, when we browse again back to the page, uncheck Saturday. Yep, that works. Uncheck Sunday, nothing changes. Well, there you go. By removing the Sunday checkbox dependency, the memoized version of the data is kept, and no refilter or resort happens. Mission Accomplished. UseMemo does what we expect.

Takeaways

In this module, we started with useContext that gave us the ability to pass data, config data in our example, down the component tree without prop drilling. UseReducer gave us a nice way to organize our state management, useCallback gave us a nice performance gain by not having all our speaker detail pages have to rerender on every button click of any speaker, and finally, we used useMemo to cache some data on our client, saving some compute time on our app and hopefully making it more responsive in the process. We've now covered all the basic React Hooks we plan on. For the rest of this course, we'll look at patterns in React usage of Hooks. Stay with me, and together we'll look at several useful scenarios.

Migrating Your Existing Apps to React Hooks

What's Different and What's the Same between Hooks and No Hooks

In this module, you'll learn how using React Hooks in functional components differs from using state and lifecycle events in React class components. Just to be clear, React Hooks are only available in React functional components. This means now you have a great way to build React apps 100% with JavaScript functions. You don't have to use JavaScript classes at all to manage your React state and component lifecycle methods. It can all be done with functions and the new React Hooks API. In the previous two modules, you learned about React Hooks and specifically seven Hooks built in to React. This is not a beginner course on React, which means that I assume you've already been using React in both managing state and working with lifecycle component events before using React Hooks, which pretty much means before December 2018. In this module, we'll take a look at the Hooks useState, useEffect, and useRef, and how the example code written in previous modules could have been written using class components. That is, we'll start out with a simple input text field that only requires setting state to work. That's equivalent to using the useState Hook. Then we'll move on to an example that handles mousing in and out of an image to effect a black-and-white-to-color transition. That example brings in directly referencing the image element and is equivalent to what the Hook useRef does for us. Lastly, we'll look at the scrolling example that colorizes the image only when it's showing in the displayed browser window. That example uses the three lifecycle events, componentDidMount, componentDidUpdate, and componentWillUnmount, which the useEffect Hook handles for us in the React Hooks example previously built. I'm guessing you'll see after this like I did, how much simpler and more clean Hooks are at doing the same thing we would have to use React class components for previously. Just a side note. When I talk about Hooks or React Hooks, you can always assume I mean React Hooks and functional components. When I talk about classes or class components, I really mean React class components. By the way, this is the only module that I'll talk about React class components in their way of managing state and lifecycle events in React.

Tracking State Changes in React Hooks Functional Components vs. React Class Components

Before diving into code samples, comparing React functional components that include React hooks to class components, let me first go through our three most used hooks and make the simplest possible comparison with pseudo code of how they look with React hooks and how they look with class components. First up, the state. In React hooks, setting state is easy, and the state itself is practically invisible. In the body of the function, we declare the constant that holds the state and then the function that sets it, const inputText, setInputText useState. Then in the input

element, we program the event onChange, and when called it updates the state by calling setInputText. That's it. With the class component, it's a little more involved. We need to declare and initialize a class variable, state.inputText. That's basically the same is putting the initialization in a class constructor, but just a little nicer with the ES6 syntax. The input element event registration is the same as the hook and then it just has an unchanged listener attached to it. The listener is a little more complex because you have to reference this and then call setState and then pass in the value there. Overall not drastically different, but as things grow, it's harder to manage state with class components. Next up, ref. As you know, ref helps get us a reference to a DOM element in React without going through a state change operation. This one is not that different. With a hook, you get a reference to the image, calling use ref, and with a class component, you just call the React method, createRef, basically they do the same thing. How you include the ref itself in the return render is identical. The final hook we'll compare is useEffect. This is the one that's the most different and, frankly, combined with setState is where the real power of hooks comes in. On this side, the simplicity of useEffect compared to the equivalent in a class component really is highlighted. UseEffect is just one method that gets executed either once when a class is mounted or multiple times based on the dependency array. The return of the first function is the function that gets executed when the function is unmounted. Class components, on the other hand, have three separate methods, componentDidMount, componentDidUpdate, and componentWillUnmount. Programming these three methods and keeping them in sync can be a real headache. Coming up, when we do our colorizing scrolling example, you'll see the difference, and you will understand why hooks are really just easier and cleaner. Let's move on now and show some concrete example of these three hooks in action and write their identical counterparts as class components.

What useState Represents in a Class Component

Let's go back to our previous example where we echoed input from an input field and then tracked that input history. Here we have the final code developed from our module on the three basic hooks. Specifically, this is the example that's stored and echoed the value typed into an input field and echoed it back on the line below. Notice useState sets up our state with the constant inputText and the setter function, setInputText. The onChange event simply updates the state and thereby changes the value of our constant inputText. Looking at the class component counterpart, one big difference you see right off is that we actually have to track the state itself. That is, we declare the state, as well as a property associated with the state called inputText. Then in the onChange event, we have to call the inherited class method, setState, in order to create a

new state, not terrible, but just a little bit harder, and as more state needs to be changed, it just gets harder and harder. Comparing the two states side by side, you get an idea of how things stack up between the two, specifically, the initialization of the state, the change event itself, and how the state is echoed back. I would say the biggest difference is the state is more or less hidden from us when using Hooks, making for less what I refer to as ceremony and working with Hooks, then in working with classes. To drive home the point of how much simpler using hooks is than using classes, let me show you the difference between our example we built earlier where we had both the inputText value to maintain state for the input field, as well as the array to maintain a history of the input text changes. Remember, you're looking at the code here exactly as we had it at the end of the earlier module. Running this code, you can see just like before as we type each keystroke, an onChange event fires and that updates the inputText value, as well as appends the current value to the inputText history array. Again, looking at this code using Hooks, it's just so simple. One more useState called and one more line in our onChange event to update the history array. Looking at the equivalent class component, you can see our handleChange event has a nested function with a setState in order to track previousState, and, of course, when we reference inputText in the history list, we always have to remember to reference this.state as its prefix. I do get that I could destructure state at the top of the render method to eliminate some of the complexity, but still, we have to deal with that complexity. With Hooks, no complex reference or the need to destructure our state.

What useRef Represents in a Class Component

UseRef is very similar to its class component counterpart. The example code that I've converted back to a class component is just the mouseover to colorize an image example. What you are looking at here is the functional component with the useRef Hook that does that. Running it now, so you remember, you can see mousing in and out colorizes. Mousing in turns it to color; mousing out turns it back to black and white. Of course, it's the React reference to the DOM element img that helps us with that. Here we have the React Hook useRef doing that for us. Switching over to the class component, the non-Hooks version that is, it just calls React.createRef to create the reference. In the call that establishes the value in the img element is identical. Looking at the code side by side with the differences highlighted really shows how little difference there actually is. In the next section, we'll look at the useEffect Hook and how very different the call in the class component is to the call when using Hooks. Specifically in the call using the React component, it's one call, literally useEffect. In the class component version, it's three, componentDidMount,

componentDidUpdate, and componentWillUnmount. Much simpler with a Hook, and you'll see that coming up.

What useEffect Represents in a Class Component

This is my favorite example for showing how React hooks really excel when compared to using state and lifecycle events in class components. UseEffect is the React hook entrance to state and lifecycle management. If you remember from our example that I'm showing here, when we first loaded this page, the speakers in the view not scrolled off are properly colored. As you scroll the images off the page, either up or down, the scrolled off images changed to black and white and then to color when scrolled back in. This is a little tricky because there are two separate events being tracked. First, before the page is scrolled, the images that are showing must be shown correctly, colorized or not. Then, as the page scrolls, the images need to recalculate themselves to determine if they should each be colorized. Looking at the code, the useEffect React hook handles this really gracefully. The first parameter passed to useEffect is a function that is guaranteed to be executed when the component first loads. It adds the scroll listener to the manage scroll changes, then sets the state of whether the component is in view by calling the function isInView. That way, when it renders, the component knows whether it renders to the black and white or color image. The return of this function is a function itself that gets called when the component dismounts. In our case, it just removes the scroll handler. The second parameter of useEffect is a dependency array. What this means is you give the useEffect an array of values, in our case just one state is loading. This determines whether the useEffect gets called again as state changes, which happens during a lifetime of this component. Let me just jump to a slide here and explain in more detail the impact of this parameter and how it works with your functional React component. If you leave the second parameter of useEffect out or pass it as null, then the function passed into useEffect is run on every state change for the life of the component. If the second parameter is an empty array, then the useEffect first parameter function is executed just once when the component is first mounted and never again. Finally, if the second parameter has values, then those values are the dependency values. That means if any of those values change between state changes, then the first parameter of useEffect function is executed. That is, in my example, when isLoading changes from false to true, the function is run. Again, back to our React hooks code that implemented our scrolling colorizing speaker viewer. It's really clean. When the component first loads, setInView is checked, and the appropriate state variable is set. Since isLoading is a dependency, and we are changing that inside useEffect with setIsLoading, then in our next render, it's run again. IsLoading started out as true, so the image did not render

on the first page load. But after that, the dependency isLoading changed, and it does render. Scrolling, of course, just works because of the unscroll event. But at this point, that's pretty trivial because useEffect never runs again as isLoading doesn't change again. Now for the ugliness. Let's take a look at the class component version of this colorizing scrolling component. It works the same, so no complaint. But as I explain it or try to explain it, try not to get too lost. First, of course, we have a constructor that initializes our state. We set inView to false, though that doesn't really matter since we have to figure it out anyway. And we set isLoading to true just like we did in the hooks version useState call. Now it gets tricky. Instead of useEffect, we have three independent class methods, componentDidMount, componentDidUpdate, and componentWillUnmount. Regarding componentDidMount, this only runs once ever for a component, kind of like if you put an empty array in the useEffect dependency parameter. ComponentDidUpdate is where it gets tricky. That method gets passed in all the old props and states on a state change, and then it's up to us to decide if a state change is significant enough for us to do something. And if so, what to do? Notice we are checking isLoading, and if it changed, then we check to see if the component is in view. If we forgot to call this function, then when the state changes from loading to not loading, isLoading is true to false that is, then isInView check would never happen, and our image in the browser would not be colorized properly. Finally, componentWillUnmount removes the listener. Nothing tricky there, just need to remember to do it.

Takeaways

If you can't tell already, I'm obviously a big fan of using React Hooks instead of React class components. I don't like invisible classes like componentDidUpdate you have to remember to include. What I mean by invisible is that since it involves a base class component, you have to know to mention it. Same goes for componentDidMount and componentWillUnmount, but the difference there is you use those two often enough that they're hard to forget. ComponentDidUpdate you don't use that often, or at least I don't use it that often in my experience, so it's easier to forget. I also don't like having to access variables with names like this.state.isLoading. I don't like having to refer to state as, well, state. I'd prefer to just reference variables I've defined with useState. Hopefully I've convinced you of the same in this module. We've looked at just three of the seven Hooks we've talked about so far in this course. These the three most obvious ones to talk about. The use cases around the others like useReducer are just as compelling, but the differences become more profound and are harder to do a compare like I've done here. Suffice it to say Hooks are just simpler. I won't hold it against you if you still love

classes and continue to use them. At least now, though, you have a good understanding of React Hooks and how you can use them and how they can do everything that you did previously with React class components.

Learn How to Combine Existing React Hooks into New Combined Hooks

Benefits of Combing React Hooks into Custom Hooks

This module is all about combining multiple React Hooks into a single custom Hook. When the React team brought us React Hooks, they did so primarily to allow us to use functional components to manage both state and component lifecycle events. A big motivation for that is that JavaScript functions are so incredibly simple that it seemed a shame to only be able to use them for essentially read-only static rendering. One huge benefit of the Hooks design is that they are easy to use and don't clutter up your code with complex usage patterns. That's the good news. The bad news is that as we try to solve more complex problems, we end up using more and more Hooks in our components, and we get back to complex, hard-to-understand, hard-to-reason-about functional components quickly. The good news is part of the design of Hooks is that they easily compose themselves into a custom Hook. To start this module, we'll talk about this email entry web form that is a slight modification to our sign up to be notified form we built earlier in the course that simply validates the entered email address and when valid, enabled the submit button. The wrinkle we are adding is a timeout counter similar to one used by companies like Eventbrite to sign up for conferences that give you a certain amount of time after you arrive on the forum to sign up. Notice that it shows the time left, and when it finally expires, both the text input and the submit are disabled. We'll first look at the code that creates this form using the Hooks we've learned so far. Then we'll refactor it to extract a custom Hook, drastically simplifying what it takes to build this functionality, as well is giving us a reusable React Hook we can use in other projects with similar requirements. Let's get to it.

Timing out and Email Validation on the Signup Form

To start, we need a routing page in our pages directory that launches our form. All this page does is it renders the EmailValidatingForm React component from our source directory. Heading over there, we've got our code that's behind the app we were just running, fairly straightforward. First thing is we have a validateEmail function that uses a regex expression to validate an email address. Then we declare with useState a Boolean that defaults our emailValid state to false. That's because we will start with an empty text string, which is not a valid email. Then we create our email state, but instead of using useState, we create a simple reducer using useReducer. Remember, useState is exactly this reducer, but without the on-the-fly updating of validating email. A side note, I find the more I use Hooks, the more I use useReducer instead of useState. In this example where instead of creating state with useState, we use useReducer. This is a perfect example of how convenient it is to combine the email validation with the setting of an mail into one useReducer call. If we didn't do this, we would need to in line, in the onChanged event of our input element, a validation check for email. It's much better to encapsulate the check in the setEmail method, which is what our useReducer call gives us. A quick side note, you may have noticed that I'm using reducers, but no actions. This is completely valid, and what I'm doing is making my action pass to the reducer the content itself. More complex reducers are better served by using action types. There's a good example of this on the React website at this URL. Next thing is to add the form timer. We create a count integer to count down until our form times out and is no longer valid. And then I'm going to cheat a little and include a custom React Hook, useInterval, created by the React team to replace setInterval for timer-type events like this. Looking briefly at the useInterval code, what it does in a nutshell is it sets and clears a timer on every component mount and dismount. We shouldn't use the setInterval call directly in our component because of its dynamic nature. It can interfere with React's lifecycle processing. Dan Abramov has a lengthy explanation about this, and you can read about it, but for now, let's just use it. Notice that includes an invocation of useEffect to solve this, which means in our timing-out form component, when we call useInterval, we are getting both of these useEffect Hooks in our form, very clean. We've got our React save timer now with no more code than we would've have had normally if we had used setInterval. We do need to import setInterval, just like importing any other React component. A rendered form is exactly what we had before, that is onChanged calls setEmail, the emailValid state drives whether we disable the Submit button. And now we have a div outputting at the bottom of the form that shows us either the time left to sign up or a message saying time's up, Of course, our input element and Submit button also check for time's up. That said, the top of our functional component, the JavaScript logic, that is, is about 20 lines of code. Let's see if we can make that into one line of code in the next clip. Spoiler alert, we will. Stay tuned.

Combining Multiple React Hooks into One Custom Hook

If we consider all the code, starting with the valid email function through the useInterval hook, we need to think about how much of that we can refactor into a separate function. Remember, a React hook is just a function, and the strategy for refactoring out a function is to look at a chunk of code, see what it depends on, figure out what it produces, and then do the extraction. Looking at this code, we can see that we want to pass in the constant secondsFormValidFor, and if we look in our rendered output, what we want to get out is count, email, setEmail, and emailValid. Let's do the refactor. Copy the code. Name a new custom hook file. Let's name it useEmailValidation.js, and generate a skeleton JavaScript functional component. Add the secondsFormValidFor parameter to pass in, paste the rest of the code into the body, then, here comes the not-so-obvious part, we return a JavaScript object that has attributes the four values we want the hook to return. These return values include the returns from other hook calls, like email, setEmail. In other words, this new custom hook returns the results of hook calls made inside it. Now, back to the original file, EmailValidatingForm. We call our new custom hook, const, count, email, setEmail, emailValid, equals useEmailValidation, passing in 10 seconds. Add the import statement on top, and we're done. All those lines of code, including hooks calls, reduced to a one line custom React hook call. Mission accomplished. You've now seen how easy it is to make our own reusable consolidated custom hook. Next up, we'll create a really easy-to-use custom hook that makes calling REST GET calls practically trivial while maintaining the tricky error handling and state changes as the REST call asynchronously fetches data.

Introducing the Custom Hook useAxiosfetch for async REST

Let's jump into a fully coded custom React hook I've put together that handles retrieving and updating data with the REST protocol, specifically using the get and put action methods. Remember, this is just a JavaScript function. Starting at the top, we're going to use the hooks useState, useEffect, and useReducer, so those get imported. Axios is our choice for dealing with REST. And of course, we have to install it with npm install axios --save. And right off the bat, we have a reducer function. Let's skip that for now and come back to it when we reference it in useReducer. Recalling how REST data fetches work, they take a URL to call and then asynchronously they return data. Since our hook is going to survive state transitions, when we first call it, we need it to return something. So we need to pass in the initial data that will be returned immediately, that is before the REST service returns data. So our two passed-in parameters to our custom React took are initialUrl and initialData. Now for my favorite hook, useReducer. If we think of it like useState for a moment and look at what we have here, it returns

the state values isLoading, hasErrored, errorMessage, and the data. This gives us everything our client needs. That is when the Axios fetch hook is first called, isLoading is true, hasErrored is false, errorMessage is empty, and the return data array is empty. If there's an error after the initial fetch, isLoading becomes false, hasErrored true, and errorMessage contains the error, and the data returned stays empty. Finally, if successful, isLoading is false, hasErrored is false, and the return data is our data back from the REST service. So how did this happen? First thing is we initialize our state and set up the reducer with the useReducer we skipped over in the beginning. We've got our initial value set, and the reducer is set. More on that in a minute. We then have a useEffect hook, which executes when the component first is called. To satisfy how useEffect works, we need to execute an async function that does our work. What happens here is we follow our state diagram. First thing is we dispatch to our reducer, FETCH_INIT. Then in a try-catch, we have Axios issue in asynchronous call to fetch data from our URL and await the results. If success, we dispatch to our reducer a FETCH_SUCCESS. And if it fails, we dispatch a FETH_FAILURE. Now let's look at the reducer we initially skipped over. As you would expect, it's simply a case statement that returns a new state that includes our new updated state values, depending on init, success, or failure. Notice we have one extra reducer action state we've not talked about, and that's REPLACE_DATA. It's just a one-shotter. We use it to replace one record in our data that is already loaded. More on that later. When we get around to actually selecting a favorite speaker and need to change just one speaker record, we'll use this REPLACE_DATA action. One thing nice here in this reducer function is that the state updates are grouped logically. If we had separated our state with separate useState hooks, it would be possible to create states that don't make sense, like if we returned hasErrored true and an empty string as errorMessage. The reducer constrain state to be consistent in that way. Finally, at the bottom of our custom hook, we return all the attributes of state and also a function that we can use later to update one data record. Let's now go into our conference website developed in an earlier module and replace the code that fetches speakers with a single custom React hook.

Integrating Our Custom Hook useAxiosfetch in Our Conference App

Previously, we completed building a conference speakers page. In our speakers component, we simulated loading speakers from an external source by creating a promise that timed out in one second. Then we use static speaker data we imported from a local JavaScript file, speakerdata.js, that when the promise resolved, it loaded. Time to replace that with our custom React hook we just developed, useAxiosFetch. The changes are very straightforward and result in our replacing our useEffect in speaker.js with a call to our new useAxiosFetch custom React hook. Let's

comment out our original useReducer to create the speaker list, our useState that took care of isLoading and the entire useEffect that loaded the data from our speakerdata.js file with the promise. We import our useAxiosFetch hook, JavaScript file that is, then invoke it at the top of our component here. Since useAxiosFetch is hard-coded to export the name data array as data, we need to change our reference from speaker list to data in these two places. Notice the URL localhost:4000/speakers. This is just a local server that returns JSON data for development purposes only. We'll be using the Node package JSON server for this. It's basically just a simple Node Express server designed to serve REST data from a local JSON file. Let's first install json-server. We then add a script line to launch it from package.json, and we enter npm run json-server at our terminal prompt to actually launch it. Notice the script essentially launches the file db.json and hosted a port 4000. I've copied in a db.json file that has just the speakers. We'll be doing all our REST calls against this JSON server to simulate a production environment. You can read about this and a lot more that JSON server supports at this URL. Since now we are loading from a real REST server and not a static JavaScript file, the remote server might not be available. Let's handle the error that Axios could return to us directly. We do have the properties hasErrored and errorMessage so let's add some code to our return to check for the error, and if there is one, that is hasErrored equals true, then don't return an empty array, but instead render the error message. For the sake of this course, I'm adding a reminder here to suggest to you that maybe the cause of the error is that you didn't start your local JSON server with the command npm run json-server. I'll talk about that next. Now, when we run our revised conference website with our useAxiosFetch custom hook, we get speakers loaded just like before. When we look at our network traffic with Chrome debug tools, we have the network traffic showing from our JSON server we just configured. Done and done. We've now got our custom hook useAxiosFetch integrated into our conference website, and we've built a simple REST server that lets us test our REST get as if it were in production. Next up, let's update our speaker favorite feature to also work with our custom react hook, useAxiosFetch, and the REST put verb.

Updating Speaker Favorite with Our useAxiosfetch Custom Hook

To complete our example and update our code to not only fetch the data from a REST service, but to also update or PUT to our REST service when the user favorites a speaker, we need to update the heartFavoriteHandler. Currently in SpeakerDetail, our heartFavoriteHandler is passing just a new and toggled favorite Boolean value. The button was pressed is on the SpeakerDetail component. That button element has an attribute named data-sessionid, which we have access to. Therefore, our handler in Speakers.js knows what the sessionId is, and it can simply toggle the

Boolean attribute favorite in our state data. Now, though, we want to update not only what is displayed on the page, but also the full speaker record by posting that full speaker record to a REST server, which means we need the full speaker record passed up from the SpeakerDetail component. Previously, we passed all the attributes into the speaker record in a render section of Speakers.js, except for the Boolean sat and sun, which are the attributes that indicate whether the particular speaker is speaking on Saturday, Sunday, or both. We just need to add these here so the speaker detail page has them and can return them back from the click event on the speaker favorite button. Moving to our SpeakerDetail component, we make just a couple of small changes. We add to our passed in record two more attributes, sat and sun, to make it so we can pass back to our Speakers.js component a full speaker record. We then replace our not favorite parameter we are passing up to our handler in Speakers.js with the full speaker record and we'll let Speakers.js do the toggle for us. Back to Speakers.js, we update our favorite value, the second parameter, that is, to speakerRec. Next, using a nice ECMAScript 6 feature, the spread operator, in one line we create a new speakerRec with a favorite Boolean value toggled. We then do an axios.put of that toggled record to a REST server, and on completion, we call the method updateDataRecord. Remember, we got that back from our useAxios Fetch custom hook that causes the UI to update and the heart to toggle to either favorite, light red, or not favorite, dark red. We, of course, have to import Axios because we just used it. We are now ready to go. Browsing again to our Speakers page, you can see favoring speakers again works. You may be able to see the slight delay after I click on the heart icon because our JSON server is set to include a slight delay in returning. Also, you can see the network traffic of the Axios PUT. That's it. We've completed our custom React hook. It fetches data and updates to a real REST server exactly as we would have expected. Along the way, we've created a custom hook that we can reuse on other REST calls as needed. The only assumption we made is that there is a primary key named ID so that our update record functionality works. Also, it would be really easy to take this code and extend it to include insert and postReducer methods. The completed source for this is all on GitHub at the URL pkellner/pluralsight-course-using-react-hooks, and the section is 05-Combining-React-Hooks. In our next module, we'll update this example to include basic authentication such that only logged in users will be able to favorite speakers, as would be the case in a production conference website. We'll use a basic passport authentication server to implement the login scenario and code this into our conference app. Stay tuned.

Integrating Authentication into Your App with React Hooks

What Is Authentication?

In this module, you'll learn about how to add authentication to your app. Authentication has a very specific meaning in a Web app. It means that once a browser has connected to a Web server, that same browser, when connected again, can be confirmed as, well, the same browser. Does it mean that the user connecting is safe and secure? Well, maybe secure if you're using HTTPS and all the endpoints of that connection are secure, but definitely it does not mean safe. An authenticated user could easily be same bad guy back again. Authentication makes no attempt to verify good guy connecting, just same guy connecting. How does a user at a browser typically authenticate with a server? Probably the most common way is by entering a user name and password. That's what we will be using in this module. However, users often authenticate using other services offered by third-party providers, like Google, Twitter, Microsoft, LinkedIn, and others. These service's offer their own authentication so that we, as Web developers, don't have to worry about maintaining user names and passwords for the user to connect or authenticate to our site. The way that typically works is that when the user comes to our site, we redirect them to an authentication provider, again, Google, Twitter, whatever. The user logs into a dialogue the provider supplies with their credentials, and then we, our server that is, get a unique identifier back that we associate with that user. Again, could be same bad guy back again, but at least you recognize the user as the same one as we met before. Okay, enough definitions. Let's move on to adding auth to our conference website. Let's also make it so that only logged in or authenticated users will be able to update who their favorite speakers are. Let's make it so when no user is authenticated, a Login button is shown, and when a user is authenticated their authenticated email and a Log out button is shown. Stay tuned.

Updating Our Conference Site as if Login Was Done

Let's assume that we've got all authentication working in our React web app and that somehow we've been passed an authenticated email to a React component that is a parent of all other components in our app. We are looking at App.js of our conference website, and this is such a

component. We currently only have two pages in our app. Both of these pages share App.js as their parent. This is also where we store our app configuration information that we programmed in a previous module. That is, like whether to show our sign up page or not. Let's add to the global config, a loggedInUserEmail. That is, let's assume at this point we know who the logged-in user is, and if there is no logged-in user or authenticated user, then this value will be undefined. I'm sure you're thinking, how can you just claim a logged-in user? That's the part where I want you to take a leap of faith for now. Later in this module, I'll teach you how to add login and logout screens to your app, and then you get the logged-in user's email on every invocation of this App.js component, which, as I said, is the parent component for all pages. Remember, the purpose of this course is for you to learn React Hooks, not adding authentication to your React app. Handling the UI side of authentication is such a great example of how Hooks can be used in real-life programming, that I just can't resist showing you how easy it is to integrate this into a real-world web app. When we learned about the useContext, React Hook, we learned that we first had to create a context, as we are doing here, then wrap every page with that context and a passed-in value that we can access in every child component. Now because of that, we have our logged-in user's email available everywhere. Heading over to our SignMeUp page, we have what we show a user we don't know yet, also meaning not logged in. If the user is logged in, then they already know about our site, so instead of showing them the sign-up, we show them their logged-in email and give them a Logout button to press if they want to log out. I know we called this the SignMeUp page, but it's also a good place to have a logout if they already signed in. Because we already have the ability using the useContext React Hook to access our context on this page, we can use that same context now to get our logged-in user's email. Above our original render, let's paste in some code that checks to see if we have a logged-in user and, if so, output that email address and a Logout button, which we'll implement later in this module. While we are here, let's add a Login button to our SignMeUp page so that when not authenticated or logged in, the user has an easy way to log in and become authenticated. We'll put that button right next to the input field and, of course, implement it later in this module. Running node, like always, npm run dev and browsing to localhost 3000, you can see now, instead of our SignMeUp page, we show our logged-in user, as well as our new Logout button. Going back to our config in App.js, removing the logged-in user from the config, refreshing the browser, sure enough, there's our original SignMeUp, as well as our new Login button. For our React app, the only other thing we will do with our sign in is to only show the speaker favorite heart button When the user is logged in. That's really easy. We go to our speakerDetail.js page, add our context reference, and useContext Hook to access our global context. Then we just wrap our button with some JavaScript JSX logic that checks for a logged-in user email and only displays the heart button if there is a logged-in

user. Now refreshing our browser because there is no logged-in user, the heart does not show. Putting the logged-in user's email back in our global config in App.js, refreshing the browser again and, as expected, the heart is back. That's it for how React Hooks help us with managing the UI of authenticated users on our site. Going forward now, I'll explain how to make all this work in our React conference app. That is, I'll add a login page and all the logic, both server and client side, to make this work, as well as log out. The rest of this module really has nothing to do with React Hooks and is not meant to be a thorough explanation of how to do authentication in a React app. But it will give you some understanding and flavor for what's involved. Here on Pluralsight there's another excellent course by Cory House that's all about authentication and specifically about using Auth0 as an auth provider to solve this problem. Cory does a great job of explaining in detail what you'd need to do for doing authentication in a real production React app.

Customizing Our Node Server for Authentication

Throughout this course, I've been very careful to not do anything specific with the tool chain I'm using. That is, I've used Next.js, though I could have just as easily used create React app and the course would have been 99% the same. That's over now. Authentication is heavily dependent on the server code, and for that the tool chain is just very important. To that end, we will continue to use Next.js, and I'll need to explain some more about how Next.js works for you to understand what is coming up. For the sake of this explanation, what's important to know about Next.js is that it's essentially behind the scenes launching and Node.js Web server with Webpack as the bundler builder. Let's take a little of the magic out of that by creating our own server.js file that we launch with Node itself. We just type at the command problem node server.js. Our app is now running exactly as it did before, but of course we still have no authentication. Since we are using Node as our server, let's use the most popular and configurable Node authentication system out there. It's called Passport, and the nice thing is there's tons of documentation and plugins for it. Passport refers to the different ways of logging in as strategies. The simplest is just called the local strategy. All it requires is a password validator and some way to look up details beyond the basic authentication key. That key typically being a user name or email. Let's get started by installing a bunch of Node packages, including passport. That is, we do npm install express path body-parser cookie-parser express-session passport passport-local connect-flash and local storage. Before we launch our express server in our server.js, we need to instantiate our local Passport strategy. We also include a serializeUser and deserializeUser method, which we've simplified to only track the user's email. Typically, this is where we would use a fast local server like Redis to store more details about the user, like first name, last name, occupation. and more.

Next, in our express server itself, we need to install the Passport middleware and include everything necessary for Passport to maintain the user's session between server postbacks, meaning page reloads. Just for your information, the session manager we are using here is not meant to be scalable. It's just for development. You can read more about how to make this scale on the Passport website. The last thing we need to do in our server is create entry points for log in and log out. That is, for log in our React app needs an endpoint where a form can post user name and a password. If password validation in this local strategy passes, the user's logged in. For our demonstration here, validateUser simply checks to see if the user name and password or the same. If so, then authentication succeeds. If not, it fails. For log out, we need an endpoint to log the user out, or essentially cancel the authentication and invalidate that session information. That's essentially it on the server side. When the user is logged in, the Passport local strategy will automatically tack onto the request object, pass to our client React app an attribute user that includes the authenticated user name. Next, let's look at the client side and see that happen. Then see how the property of the request object gets passed to our app.js file, which moments ago we saw integrating the logged in user into our React app itself.

Integrating the Server-generated Request Object

If you could say that the tool chain Next.js has secret sauce, it would be that it makes building React apps that automatically generate full page HTML on the server for immediately displaying that HTML on the client. It's a huge deal. Let me say that again. It's a huge deal. The Achilles heel of single page apps in general is that their performance is really bad on the first page load. That is the experience the user has when they go to the site URL the first time is just not good. Next.js totally solves this problem. Here on Pluralsight, my previous course to this one was all about Next.js and solving that first page download problem. It's titled Building Server-side Rendered React Apps for Beginners. In the intro, I graphically show how bad the problem is with all the back and forth between the client and the server before any part of the page is shown to a browser. The cleverness of Next.js server-side rendering is baked into the JavaScript file in the pages directory that represents the URL route. Here, we're looking at pages/index.js, which is our home page. How Next.js makes building server-side rendered apps easy is by prescribing that we are meant to add a static method to our JavaScript file in our pages directory named getInitialProps. The genius of Next is that this static method is run on the server-side, that is inside Node.js when the URL is first hit, then again, when the page renders on the client side. While running on the server-side, getInitialProps has access to the request object, which contains the authenticated email, assuming the user is authenticated. Then, whatever getInitialProps returns,

that is, the objects it returns, get fed into the client-side rendered component here in the same index.js file. Remember, what makes this work is that getInitialProps is a static method, and Next.js knows how to process it. Following that, the data is shuttled by way of static JavaScript objects embedded in the HTML to the browser for execution there. There's a little more to it, but I think you get the idea. If you are following along coding, you'll need to add a little more detail to getInitialProps and also at a file pages/_app.js. Both of these are involved with pulling the request attribute, user.email, out of the Node server and getting it to the browser where the client-side React is running. You may also notice that we have some browser local storage involved. This solves the problem of what happens when a user, for example, locally browses from the Home page to the Speaker's page. Since there's no post backed involved, this is a spy, after all, we need to store the authentication data someplace, browser local storage is a good place for that. I do realize that I've pasted in a lot of code here between pages/index.js and _app.js. Both file is very specific to the Next.js library. It's also dependent on the Passport authentication library we installed in the last clip. The details here are not really relevant to React Hooks, but what is important is that somehow we managed to get attached to the property passed into our index.js client site. executing JavaScript file, a user object. That user object contains the logged-in users email as an attribute. I've added the user object to our app component call, so that we can pass it to our React context provider. Switching over to app.js, let's add that as a passed-in attribute, in addition to the pageName attribute. Now, we simply add to our config value the loggedInUserEmail. And instead of hard coding that email in our config value object, it's pulled from our Passport authentication server based on the logged-in user. At this point, we are really done. We've got the authenticated user's email on the client side, where it's passed to our app.js and shared through context and the useContext React Hook, remember Hooks, through the rest of the app. Just like index.js, we need to have an identical getInitialProps in our pages/speakers.js file. One last thing, we've not actually created a way to post to our Node server/login with our username and password. Remember, that's how we authenticate our user. All we need to do is create a pages/login.js file in our pages directory. That file references a new login component in our source directory, which is just a simple HTML form with a post action that sends the typed-in user and password back to our Node server. And we add a reference to our login page here in app.js, and return that. Finally, let's run this. We type node server.js at our terminal prompt, browse to our Home page, press the Login button, that takes us to our new Login page. Enter an email address, then that email address again as the password, and you can see we're now logged in. The normal sign-up form is replaced by our login name and a Logout button. Navigating to our Speakers page, we've got all our speakers, and notice the heart is showing. If we log out, navigate back to the Speaker's page, the heart's gone. We can only favor speakers when we are logged in.

Remember, that was our requirement, and it makes perfect sense. I realize there's a lot to take in, and if you really want to learn more, I suggest digging into Next.js It's awesome, and you won't be sorry.

Module and Course Wrap Up

I realize that in the last 5 minutes or so, I did a whirlwind run through to create a working React web app that handles authentication. If you didn't follow the details, you shouldn't be disappointed. My goal was to simply get you to a point where you could usefully use React context and the useContext Hook to fully integrate authentication into your app. That is to say, by passing our authenticated user into our base component, apps.js, and then in app.js, wrapping all other child components in a context element whose value contains the authenticated email address. That allowed us to share our context with all components in our app. That means our useContext Hook is able to be used in any component to determine whether it should be treated as authenticated or not authenticated. I'm confident that from the top-level app.js component, you totally know what to do. How you get your user's email passed into that top-level component can be solved in a lot of different ways. I just showed you a simple one using the Node.js Passport local strategy in the Passport package. Regarding React Hooks in all the other modules, I hope you feel confident now to start using Hooks in your day-to-day programming of React apps. As I'm 100% sure you noticed, I think Hooks are the best thing that has ever happened to React, and before Hooks, I was already a big React fan. I just didn't realize how much better React Hooks could make things. The big takeaway is that with Hooks, you can do everything in React now the functional way and not have to use ECMAScript classes. Personally, I think that's a better way to work with JavaScript, but, as I've said many times, to each their own. Classes are good, functions are good, follow what you think is best and what makes you the most productive building React web apps.

Course author

Peter Kellner

Peter is an independent software consultant, specializing in .NET development from 1985 through 2001, as well as an MVP since 2006. He was founder and president of Tufden Inc, where he successfully...

Course info

Level	Beginner
Rating	★★★★★ (103)
My rating	★★★★★
Duration	1h 53m
Released	3 May 2019

Share course