Advanced React.js

by Samer Buna

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hello everyone, my name is Samer Buna, I am a software engineer at jsComplete. com and I am excited to present to you this advanced course about React. js. This course is about creating full-stack JavaScript applications with React. js and Node. js. I cover many advanced topics and best practices that I've learned over the years about using React. js and Node. js. Some of the topics that I will cover include: Configuring and customizing a full-stack JavaScript Environment, Working with Async API data, Managing Components State Internally and Externally, Components PropTypes validation, the Context API, Presentational and container components, Pure components, Higher Order Components, Searching data, Working with Timers, Components Lifecycle methods, Performance Optimization with the Chrome timeline and the React Perf addons, the practical value of Immutable Data Structures, and Production builds for both React. js and Node. js. By the end of this course, you should be comfortable with many advanced topics about creating and using presentational and stateful React components in production. I hope you'll join me on this journey to learn React. js beyond the basics with the Advanced React. js course at Pluralsight.

Introduction

Is This Course for You?

Welcome to the Advanced React. js course from Pluralsight. My name is Samer Buna and I started working with React. js in 2013, shortly after it was opened sourced and I've been doing that since. I've created many full-stack JavaScript applications with React. js and Node. js, and to do so efficiently, I challenged myself to learn everything I could about them. React. js and Node. js help me be fast and productive, and in this course I demonstrate many of the advanced topics and best practices that I learned over the past few years about using them. This is not a beginner course, but it's often hard to draw the line between what's basic and what's beyond basic. And while I intend to not cover the basics in this course, the chance that you do not know everything that's covered in this course is slim. There will be things that you probably know already and others that you don't. Feel free to skip what you already feel comfortable about. To help you keep up with the progress of the code in each clip of this course, I used a Github repository with tags representing the starting point of each and every clip, so you can skip clips if you want and just sync your code with the git tag for the next clip. If you're just starting to learn React and JavaScript, this course is probably not for you, at least, not yet. You do need a solid understanding of JavaScript, including modern JavaScript. I've written a blog post about learning React. js and in there I listed the relevant JavaScript features that you need to understand to be a productive React developer. If you don't feel comfortable with anything on this list, you should take a JavaScript course first. Pluralsight has many excellent courses about JavaScript. I also recommend taking at least the following two Pluralsight courses before this advanced one. The Getting Started course, by yours truly, and the Building Applications with React and Redux by Cory House. This Advanced JS course would make much more sense to you after these two other courses. This course will have a complete focus on React. js itself. We will not be using the popular libraries that are often used with React like Redux and React-Router. Focusing on React itself will help you to learn the advanced skills that are usually abstracted with these libraries. However, understanding how to use these libraries first will help you connect the dots in this course. To work through the examples presented here, you need, of course, a recent version of Node. js and React. js. Although older versions might work as well, you might run into different problems with them. It took about three months total to research and create this course. I've actually had to upgrade npm packages a few times during the development of this course, but I think you should be okay using the latest of everything. If something did not work as expected, please leave me a

note about it. I've exclusively used the Yarn package manager in this course. It's a replacement of NPM and it's much faster and smoother to work with. NPM 5 is trying to catch up with Yarn, but I think it still has ways to go. We're going to go over the task of configuring a full-stack JavaScript environment from scratch not using any templates or generators. We'll just use Babel and Webpack, and I picked those because they are the most popular libraries for the tasks they do. On the server side, we're going to use the Express. js Node framework to deliver assets and host the API. We'll also use the excellent PM2 process manager to run the node process both in development and production. Throughout this course, we'll cover a little bit of testing using the excellent Jest framework. We'll cover concepts like shallow rendering, snapshot testing, and the coverage report. And finally, I'll be actively using ESlint in this course and you should do, too. Eslint is really a must in any JavaScript project. I tried to be as practical as possible with all the examples in this course. I always try to avoid abstract examples when I can. There will be no foo or bar in this course, and I approach all module features to be covered with a practical need in mind. However, we're not really building a super useful application in this course, as I needed to come up with different examples to cover many different topics. This course was mostly not scripted. Just me, my microphone, and my screen. I ran into many unexpected errors and I tried to explain how to find and solve them. However, I tried to tightly edit the course so that I don't waste your time watching me slowly type something or talk long about something that can be summed up in short. You'll see me fast forward typing sometimes and paste code sections other times. I am hopeful that you're going to like the pace and content of this course, but if for any reason, you don't, I really want to hear from you. Tell me what worked for you and what didn't. Come chat with me at this Slack channel, which you can invite yourself to. Feel free to also ping me on Twitter as well if you do not use Slack. A tightly edited course might mean I could be going too fast in some places, or maybe I'll still be slow for your pace of learning, so don't forget that you can control the speed and pause the video where needed. You should pause the video often and actually try all the code examples that you're going to see in this course. If you just watch the course, chances are you'll forget the knowledge you gain here eventually, but if you challenge yourself to redo the examples and even try different examples with every concept you learn, that knowledge has a much better chance to stick in your brain.

Full-stack JavaScript with React.js

Introduction

Hello, in this module, we'll get things started and hash out all the setup and configurations needed to work with a Full-stack React environment. And we'll do so with a simple example React application that we'll start from scratch and have it work on both the client-side and the server-side. Other than the setup and configuration, which you can skip if you're comfortable with that, we'll talk about working with data and how the structure and shape of the data influence the way to create React components. We'll cover a little bit of testing in this module, including snapshot testing, and we'll also see an example of separating components responsibilities into what makes more sense in terms of data ownership. We'll conclude this module by rendering the React application on the server and shipping an initial HTML string to the browser, and talk about the benefits of doing so.

Configuring a Full-stack JavaScript Environment: Server Side

Let's dive right in and start a Full-Stack Javascript application from scratch. Make new directory advanced-react. The reason I'm going to start from scratch here is that I want you to understand all the elements that play any role in a data-driven production ready React application. This means understanding every configuration we're going to add and every command we're going to run. So we're not going to use a preconfigured environment with tools like Create-React-App, for example. This is a great option to begin a React project and prototype something really quick, but it abstracts a lot of things that you should learn how to do and learn about why they're needed, and learn about what other options you have. If you're comfortable configuring a full stack React application, feel free to skip this clip and sync with the Github repo for the next clip. The directory that we just created will host our React application. We'll put all the code related to that under a lib directory. This lib directory is going to be shared between the server and the client, as we're going to do server side rendering for this app. Eventually, we're going to create multiple nodes for our production application and balance the load on them somehow, but every node will be a web server to begin with. Every server node will serve static content and public assets. The public assets will contain the React application, so we'll make a directory for public assets as well. We'll start things out with a server. js file under the lib directory. We'll configure an express application in there. To start depending on packages like express, we need to initialize this project as a Node project and have a package. json file to document all the dependencies like express. Yarn init will do that task for us. Answer the few questions here. Let's change the entry point to lib/server. js. Also add the Github repo for this project. You can skip that step on your end. Author info. License. And we now have a starting point for the package. json file. Launch your favorite editor on this

directory. And let's have a really quick word about editors. If you're using vim, emax, atom, or sublime, you're good to go. Forward about one minute from this point. If, on the other hand, you're using something like webstorm, or vscode, or any other fancy IDE with fancy features, you might want to consider downgrading for this project. I don't have anything against those editors, but I think they add unnecessary and distracting noise around your workflow. This is just my personal opinion. However, no matter what editor you use, I want you to make sure you can integrate eslint with it. I can't say enough about the importance of having eslint integrated in your editor. I like to see any eslint errors on save. Instead of me discovering those errors a few seconds later in server logs or a browser console, eslint will save you that time. So, it's just that simple. If you don't have eslint, you're simply wasting some time. Eslint is so important it is going to be the first thing I configure in this project. Bring in a local eslint with yarn add --dev eslint. Eslint is a development dependency. Then, you need to bring in an eslint configuration file. Now, you have a lot of options there. You can use the eslint command itself with --init, and eslint is going to ask you a few questions there so you can configure your eslint with simple questions or using a popular style guide like Google's or Airbnb's. However, we need a style guide that understands modern Javascript syntax, like, for example, class properties. That's why I just use the recommended settings with the babel-eslint parser. And I also add the React plug-in and a few other personal customizations. If you want to use my exact configuration, it's part of my. file repo on GitHub. Alright, so we're going to go ahead and take this configuration and we're going to add it under our project as. eslintrc. js. Since we're using a React plug-in for eslint and babel-eslint, we need to bring in these dependencies, so yarn add --dev eslint-plugin-react. This is the React plug-in. And also babel-eslint itself. So go ahead and install these two dev dependencies, and after you do that, make sure you test your eslint configurations. So let's place a console log statement in here and make sure that eslint is configured correctly. Let's drop a semi-colon and here you go. Eslint is giving me hints that I have some problems. So it's extremely important that you get this instant feedback when you have problems in your code. Okay, we're now ready to start our express server web application. So we're going to bring in express as a dependency. We're also going to use ejs for minimal template work on the server side when we start rendering our React application on the server. Those are the two main production dependencies. And we'll start a very simple express application right away. In server. js, we need to bring in express and we need to initialize an application and we can do that by just invoking the express function. And then we need to use express static middleware to serve the public directory. And we'll go ahead and listen to a port here to run the express application. However, instead of hardcoding a port here, we should use a configuration variable here. I'm going to call it config. port. I'll go ahead and import this configuration from, say, require. We'll go ahead and place it on the same level, so config. js,

and we'll create config. js, and config. js is simply going to export those configurations. So, the first configuration we just use is the port. We'll try to read it from the process first, and if there is no variable in the process, we'll go ahead and use something hard-coded, just like that. And we'll pass in a function here for the listen handler. So we'll call it listenHandler. This is a function, and in here we can just console. info that we are running on this port. All right, we'll go ahead and test that just by running node lib/server. js, and I'm listening on port 8080. So now let's go ahead and prepare our / endpoint. So when I go to the /, what do you want me to do here? So we'll do the request response callback here, and we'll just render the index template. Now, this is a template that needs to exist in the ejs templates, so we'll create a new folder here top level views. This is the default folder where ejs is looking for template. So under views we'll create a new file and call it index. ejs. And in here let's just put a standard html template. We also need to configure express to use ejs as its template language. And we do that using this line app. set view engine is ejs. And I just realized there is a typo here, so it's public. So we've got the public directory that is statically served, so anything we put inside the public directory will be served directly. And we have ejs template, so we can render certain templates, like the index template in this example. We can also pass variables here to the template, so let's go ahead and test that. We'll pass a test variable here and make sure that you can use it from index. ejs. So, if everything worked correctly, I should see this h2 with a value of answer. Let's go ahead and test that. Restart the node command and go to localhost 8080, and things are working as expected. Okay, since I need to restart my node server every time I make a change, I'm going to go ahead and bring in a library to help me with that. This library is pm2. Now, when we take this project to production, pm2 will also help us a lot there in, for example, rendering a cluster instead of a single node and for zero downtime restarts and other things. So we'll prepare a script in package. json under the scripts section, and in here we'll do dev pm2 start lib/server. js --watch. And now we can start the server with yarn dev. This will start the pm2 process in the background. And I can check the logs for the process if I want to using yarn pm2 logs. So we'll go ahead and test that the watcher is working by changing server. js, and if we do that you'll see change detected here, and the server was restarted. The green lines here are the logs coming from the app while the blue lines here are the logs coming from the pm2 process itself. So, usually we're going to focus on the green lines when we are monitoring the logs for our application. Okay, we have one last configuration task to do server side. We are going to use Babel. The reason we are going to introduce Babel here is because our server side will eventually need to render a React application. Now we're going to use jsx in our React application, so we need the server side to understand the jsx syntax as well. So, if we have Babel, we can write our require syntax like these with the import syntax instead. So I'll go ahead and make sure that we can do that by converting these two lines into the import

syntax. When I do that now, my server is not going to work, because Node does not understand the import syntax yet. So we need Babel to make Node understand the import syntax. But we'll also configure Babel to understand the jsx syntax as well, so that when it's time, our server will be ready to render jsx content. So to configure Babel, we can add a Babel key in package. json, and in that babel key we need the presets key, and we need three presets. We need the React preset, we need the env preset, and I'm going to go ahead and add stage-2 so that we get to use features like class properties, which are on their way to become part of the language, but with Babel we can use them today. To make these presets work, we need to bring in a few dependencies, yarn add babel-cli. This is the first dependency that gives me babel node. And by the way, I need to change my pm2 configuration to tell pm2 to use babel node instead of regular node, and we can do that with the --interpreter option, just telling it to babel node instead of node. The other dependencies that I need are the presets. So we need babel-preset-react, babel-preset-env, and babel-preset-stage-2. Now, notice how I made these dependencies production dependencies and not development dependencies, because eventually I want to push my source code into a production server, and I want to compile the production ready code on that server itself instead of generating that code locally and pushing compiled code somehow through source control. Now, this totally depends on your production flow, the way you deploy your production ready application. By having Babel as a production dependency here and just pushing the source code as is to production and then invoking a Babel command on production to compile source code into something production ready is probably the easiest way, so we're going to go with that here. Okay, so now we can test what we did so far, because we have import statements, so things should work with Babel now. So restart the pm2 process and start it again with yarn dev, and go ahead and yarn pm2 logs as well to make sure I don't see any errors, and I don't see any errors. The server is running, and I now have the capabilities of Babel in my server node. In the next clip, we'll take this further and configure the client side with a simple React application.

Configuring a Full-stack JavaScript Environment: Client Side

Let's review what we've done in the previous clip real quick. This point in source control is tagged as 2. 2. Here's what this commit is telling us. We configured Eslint, we created an Express server, this Express server is serving both static files under the public directory and ejs templates from the views directory. We have a single endpoint /. It's rendering the index template and passing along a test variable, and we're making the server listen on a configurable port. We have a package. json, where we configured our start script using pm2. We also configured babel to work

with pm2, and we documented all the dependencies in this package. json file. And we have a single index. ejs template under the views directory, which is responding to our / endpoints. So when we run the server now, this is what we see on the first page. The rest of this commit is for the yarn lock file. It's now time to render a React application on the client side. So we're going to create the whole React application under the lib directory. We'll create a new folder, name it components, and inside components we'll start with an index. js file. This file is going to be the starting point for our React application. So we're going to need to bring in React. We also need ReactDOM. We need ReactDOM here, because we are interfacing with the browser. We'll create a very simple React component, so function that renders an h2 hello React, and we'll use ReactDOM to render this component, so app, into the DOM somewhere, so we'll do a getElementById and we'll name this ID root. That's it. So now since we imported React and ReactDOM, we need a way to package this application into a single file that we can put in our index. ejs. So in index. ejs we're not going to do that anymore. We're going to have a root div, and this root div will start with static content and then React is going to take over this div and render the React application in there. So, we need to bring in a script. Now the best practice today is to put a single file in here, like bundle. js for example, and use something like Webpack to bundle the whole application into that single file. That's exactly what we're going to do. First, let's make sure to serve bundle. js from the root directory. We're going to place bundle. js inside the public directory. And let's see what we have. We have a dependency on react, react-dom, and we need webpack. So let's bring those in. So yarn add react, react-dom, and webpack. Once that is done, we need to tell webpack where to start and where to place the bundle. js, and we do that with a webpack. config. js file on the root level of this application. We'll bring in a configuration template from the webpack js documentation site. If we go to the concepts page, there are simple examples here about the entry output and loaders. Take the example from the loaders section. This is exactly what we need. Paste that in webpack. config. js. Let's go ahead and modify it. Our entry point is lib/components/Index. js. We want the output to go to our public directory, and we want the file name to be just bundle. js. That's it. Now, the rules section here is exactly what we need. We want webpack to use Babel for every file that ends in. js. I'm not actually using jsx, so we can simplify that by just keeping js. Now, this is actually a dependency, so we need to yarn add babel loader. Cool, so now we can test. We'll add a script in package. json. Let's call this script webpack. This is going to run webpack. It will read the webpack config. js by default, but we'll go ahead and run it with - wd flags, w so that webpack is going to watch our files, and d is for development mode, so that we get things like source map and make it easier for us to debug or code. We can now do yarn webpack. So if everything works, webpack should generate bundle. js. And it did. Bundle. js is about 2MB. It includes all of React, all of ReactDOM, plus our very

simple application. So now we can test our application. Refresh, and you'll see Hello React. Notice how I have this icon here that changed color. This is React dev tools telling me that this page is using the development build of React. So if we open the console, first make sure that you're not seeing any errors here, and the React dev tools, right here the last tab here, is going to connect to my React application and give me a way to debug it. Now I have a very simple application here, but once the React application starts growing, this tool is going to be super helpful in looking at what's going on and debugging our problems. It took some time to actually generate bundle. js, so now let's actually measure this slowness in webpack. I'm going to run the exact same command with time, but before I do that, let's remove the -w flag. Okay so about 7-8 seconds. That's actually really slow. Why is that happening? Well, because we said any file that ends with js. This will actually include everything under node_modules, so if we don't exclude node_modules here, the webpack process is going to be slow. I don't need babble to run on node_modules, I just need babble to run on my own code. So we can either use include or exclude here. The easiest way is to use an exclude property with a regular expression pattern that matches node_modules. So let's test the time webpack takes now after excluding the node_modules folder. So now it's just 2 seconds, which is a big difference, so do not forget this exclude directive. Now, we didn't have to do anything about babel configuration here, because we already configured the babel when we made our node server understands babel. In fact, we already have stage two features, and I'm going to test it. Let's put the w flag back in here, make it official, because now we know that it is faster, and I'll actually split this iTerm so that I see the logs from both server and webpack together, and run yarn webpack in the second terminal here. So now I have eyes on both the server and the client. You do want to keep an eye on both of these, because sometimes they fail, and you don't want to go waste time figuring out what's going on in the browser. If these guys are failing, you want to restart those first. Back to our index. js. So instead of a function component, we're going to use a class component. Now here's the thing. We're going to be defining a lot of class components and function components, so it is a very good time investment to have those configured in your editor as snippets. So you'll see me using a lot of those snippets from now on, and if you're using atom and you're interested in using the exact same snippets that I use, they're also part of my. files repo. So in here, we need a React class component, and this class components' name is App, and in here we'll just return a test string. And my snippet duplicated the import React, so we'll remove that. And here's what I want to test, I want to test that I can use class properties, so we'll define a state object here as a class property, and in here we'll put a test value. Make sure that we can read it. Hello Class Components, and let's try and read this. state. answer. So if our babel configuration is good to go, then this should just work. There is one small configuration that I think we need to do. Let's

assume that we have an async function here. We'll make it really simple. So this function is going to return promise. resolve with a value, and I want to use this function, say, in componentDidMount. I want to basically set the state to whatever value this function is going to give me. So we're going to come here and say something like answer is, so basically the value of the async function here. But since this is an async function, I can't do that directly. I have to resolve the promise. But the new syntax in JavaScript that's going to allow me to do that is to just await on this async function and label the host function as async. This async function is on the instance, so this. asyncfunction, and this should actually work if you have async/await configured, but if we test now, we're actually getting an error that regenerator runtime is not defined. So this is because we don't have babel polyfill configured here on the client side, and it is required for this feature to work. So let's configure that. In the webpack configuration we need a new entry, so we're going to convert this entry into an array instead of just a single value, and we need a new entry here for babel-polyfill. And we need to bring that dependency, so yarn add babel-polyfill. Excellent. Let's test now. Run web back one more time, and you see how the 37 is happening now. So with polyfill in place, the test is now working. The components start with the 42 value in the answer state and then on componentDidMount, which is now an async function. I am awaiting on this async function that returns the value and I'm putting the result in the answer. Now, I'm not handling any errors here. If you want to handle the error of an async await operation, you want to wrap this in a try-catch statement. But this is all for testing just to make sure the configuration is working correctly for you. We're going to remove all this test code next and start working with our actual example.

Working with Data

Let's review what we've done in the previous clip. This point in source control is tagged as 2. 3. We created a simple component for a simple React application and we made sure we can use some modern Javascript features like class properties and the async await syntax. To render the React application, we used webpack, and we generated a bundle. js file that we included in our index template. In the webpack configuration, we first used the babel polyfill library. We needed that so that the async await feature worked on the client side. And then we're starting from the index component and bundling everything into bundle. js. During that process, for every file that ends in js we are using the babel loader on it, and that would make webpack understand the jsx syntax. That's it. If you check out this point in the Github history, run both webpack and the server, you should see this output in the browser. We're now going to make a simple React application that works with a data collection and see how to present this data efficiently. Let's

assume that we have this list of articles coming from an API somewhere and we want to create a React application to display them. I prepared some test data for this exercise which you can access with this link. I'm going to download this data locally into the project. I'll use wget -O to save the file under lib/testData. json, and the link is bit. ly/react-blog-test-data. Take a moment to understand the simple structure of this data object and think about the components tree that we need to represent it. Here's an important question for you to think about first. Do you think we should work with this data as is or should we transform it into a different structure first? Pause the video here and think about that. This data object represents a list of articles as an array and a list of authors as another array. Every article and author object has a unique id attribute to identify it, and every article has an authorId attribute to map that book to its author. Very simple structure. I want you to understand an important point first. If you are receiving such structure from an API, that does not mean that your React application has to work with this data structure as is. You can always have a data transform layer between your API and your React application. But the question is, why would you do that? Well, depends on what you want to do in this app, how much data there is, and how much you want to optimize the app. For example, arrays are easier to work with to list records, but when you need to find an element in a collection, arrays are not the best structure, while objects (or maps), for example, are much better in that case. Since we want to look up an author for every article, we should probably convert the authors array into an object with every author indexed by their id. But what about the articles array? If you think about it, you'll eventually need to look up articles as well. For example, when you edit an article or delete an article. Those operations are a lot faster on objects than arrays, so we should probably convert the articles array into an object as well. Please note that this only matters if you have a big list. For very small lists, the performance difference is not significant. Another question about the structure here: should we make every article object nest its own author object instead of having just an authorId and needing to look up an author object every time we render an article? Certainly nesting the complete author object under every article would be easier to deal with in the React app, but think about articles with the same author, for example. If we nest the author object, we will be dealing with duplicate data there, and things might get complicated. So, I think, leaving the authors list as a separate entity is a lot better overall, although it would introduce a few complexities in the app. It's a compromise. Okay, since we want to work with objects instead of arrays, this means every time we receive an array from the API, we need a function to convert it into an object. This is the kind of function that we need to test and make sure it works exactly as we want, and make sure future code is not going to break this function. So, in the next clip, we'll get a testing framework up and running in this application, and we'll create tests alongside this new function to make sure it's going to work as we expect it.

Testing JavaScript Code with Jest

In the previous clip, we downloaded some test data to work with and decided that we want to transform the arrays coming from the API into objects first. To work through the function needed to do that task, we're going to introduce a testing framework, Jest. Jest is a one-stop shop when it comes to testing. It comes ready with expectation syntax, mocks, and a powerful runner. Yarn add --dev jest, and in package. json let's add a script for the test runner, so it's jest, and we'll do --watch to start jest in watch mode and have it run the relevant tests every time we make changes. So now we can start the runner with yarn test. So we have our test data and now we want an interface on this test data. Let's create a file under lib. Call that file DataApi. js, and in here we'll design our DataApi interface. We'll make this DataApi into a class, DataApi. And we'll just export this class. In this class, we'll have two APIs, getArticles and getAuthors. Before we write any code, let's write some tests. Under the same lib directory, let's create a folder __tests__. This is one of the special locations where jest looks for tests by default. Under tests, we'll create a new file, and in here, let's match the name that we're trying to test: DataApi. js. Jest will detect this file, but I don't have any tests yet, so let's write some. The syntax here is simple. We're going to describe DataApi, and inside DataApi we're going to expect it to expose articles as an object. We're also expecting it to expose authors as an object as well. We need to import the DataApi class from up one level DataApi. We need to also import the test data, which is a property here on the file that we downloaded. So we're importing this form up one levelest data. And now we'll do the interface step. So we'll do an API constant and in here we will instantiate an object from the DataApi. And let's just pass it the test data. And let's assume the API internally is going to handle the conversion of data from arrays into objects. So, inside the exposed articles as an object test, we'll bring in the articles from the API, api. getArticles. We'll also read two things from the rawData. From the first element in data. articles, we're reading the ID and the title for that article. Now, we can do some expectations here based on that data. I'll do two expectations. The first expectation I'm expecting articles, which I'm reading from the API, to have a property of articleId, because I'm expecting the articles to be an object now instead of an array. The other expectation is to try and read the title out of this object and match it to the original title that we are seeing in the rawData. We'll do the exact same expectations for the author object, but with author data as well. So we're expecting the authors to be an object that has a property of one of the author Ids, and we're expecting this object to have the exact same properties for every author. If we save now, jest will fail, because we have not implemented anything yet. And it looks like eslint is not happy about describe and it and expect, so jest actually defined everything underlined here as global variables. So to make eslint happy about those, we need to tell it that we are using jest, and

we can do that by just adding jest true here in the configuration. All these errors should go away. So now that we have failing tests for the expectations that we just designed here, we can now write our converging interface with confidence. So first I know that I'm going to get the data in the constructor here. We'll call that rawData. And in here let's just put it as an instance property rawData is rawData. Now since rawData. articles is going to be an array and rawData. authors is going to be an array, let's write a mapIntoObject function and assume that this function is going to receive an array, and this function will return the array mapped into an object and we'll go ahead and use this function, in getArticles, we're going to return this. mapIntoObject and here pass this. rawData. articles and similarly for the getAuthors we're going to map into object this. rawData. authors. We're still failing, because we have not implemented mapIntoObject. mapIntoObject is actually very simple. We can reduce the array using the reduce syntax, which gives us two things. It gives us the accumulator and the current element. And what we want to do here is we want to change the accumulator using the current element dot id and assign this ID the current element itself, and we need to return the accumulator, and the starting point for this is just an empty object. So let's test, and the tests are passing now.

Matching Data to React Components

In the previous clip tagged as 2. 5, we created a DataApi class and used it to expose data collections as objects instead of arrays. We're now ready to describe a UI for that data. We'll start with the top-level App component. We'll import it here for the ReactDOM call to render App from. /App, create a new file here App. js. This is a React class component. This App component will hold the state of our application. To keep things simple, let's put both the articles and the authors object right here on the top level state of the App component, and we'll import the DataApi from up one level DataApi, and we'll import the data just like we did for the test. So data from up one level test data, and we'll do a global variable here for the API, so new DataApi from the data. In here, we get the articles with api. getArticles, and we'll get the authors with api. getAuthors. Let's just render a div here to get things to run. So I'm going to take a look at the application now. So I see the application is rendering, and the reason I'm looking at the application here is because I want to see my data. Right here, articles and authors, and every article has that and every author has that. Let me actually console. log this. state here so that we see it in the console directly. The reason I did this is because from this point on, I'm going to describe a React application to interface this data. So the shape of this data, which now I'm seeing right here in the console, is going to be the main driver for how the components are going to be structured. The interface that we want to build here is a list of articles. Whenever you're describing a list of things, you

usually need at least two components, one to represent the list itself, and another to represent every item in that list. In our case, we need an ArticleList component and we also need an articleItem component inside this article list. The App component will render this article list component. To keep things simple at first, let's pass here both the articles list, which we can read from this. state. articles, and let's also pass the authors list from this. state. authors. We need to import this component, ArticleList, under components. This can be a functional component, so we'll do article list, and render a div here to hold the list of articles. This component needs to loop over the list of articles, but since we have the list as an object now, we need to loop over the object entities. We can use object. values function to get back the list of articles from props. articles. I will simply map that into an array of Article components. So in here we'll pass the Article component. The article object itself will also pass an author object, so the author here we have props. authors in this component, which is the object that holds all the authors, so we can look up the author for this article using article. authorId here. So now the Article component will receive the article object and the author object. Also don't forget to pass a key here, because this is a dynamic child, so we can use article. id here for the key. This component now depends on an Article component, so we'll go with article. /article, and we'll create article, article. js. Again, article can also be a function component here. This Article component is going to receive both the article object and the author object as part of its props, and it will render this information. We'll render the article title, so we have article. title. We'll also render date and body for the article, so date and body, and let's render the author information in here, author. first name space author. lastName, and we'll wrap the author name with an a href attribute, and the href here is going to be author. website, and inside that we'll paste the author name. Okay, if we've done everything correctly, our simple React application will actually render. I've been basing everything I do here on the shape of the data as I see it in the console log. So, let's go ahead and test. Fresh cannot find module ArticleList. Let's try to figure out that error. I totally misspelled this, so let's fix that. Test now, and all the data is coming in here.

Styling React Components

Let's do some very minimal styling. For styling, we have a lot of options. The first option is to give every element a class, right? And then we can globally style that class. And the second option is to pass here style object. So, for example, style is we'll create a styles object with named keys. So in here we'll do styles. article and we'll do the same for the title, the date, the author, and the body. Styles. title, styles. date, styles. author, styles. body. Now, we can create our styles object, so styles object, just can have it here. This will be the JavaScript API for working with CSS Styles.

We'll have a style for article. Let's test all of that, and this is starting to look more decent. Of course, the point of all these styles is not really to make things look better, but to show you the possibilities of inline styles here using JavaScript, not using strings, which is very powerful. So we'll soon come back to this styles object and try to analyze what we just did. But for now, know that you can pass this style special object, and you can use JavaScript API to create any styles in here. Note how when I created the style object, I made the style object a global object here on this file. I did not put the style object inside the Article component like that. This is actually important, because having the style object inside the Article component means that React is going to create a new style object every time it renders the Article component and every time it re-renders the Article component. So this is actually not very good. Keep the style object on the top level here or put it in its own module and import that module. Let's talk about one more example for the importance of where to place things. We have the date string here as a timestamp, so let's assume that we want to just display the date, right? So instead of this timestamp, I want to invoke a function, say for example, dateDisplay for this string, and you might be tempted to place the dateDisplay here. Display equal, say a function, this function takes a date string. It can simply just invoke new Date() for that date string and call the. toDateString on it. Right? I think this should actually work. Let's test it. And it works! Perfect! However, this means every time React renders and re-renders an Article component it will create a function here. That's wasteful. We should just make this function a global thing here, and the reason it's okay to make it global is because it does not depend on anything particular to an Article component. It just takes a string and convert it into an actual date string. Here's the rule of thumb: if you're thinking about creating a function here, you probably don't need to. And if you're absolutely sure that you need a function here, because, say, for example, this function is going to depend on the props of an Article component, then you're probably better off with a class component instead, and we'll hopefully see an example of that.

Components Responsibilities

What we did so far is simple, and it works. However, let's think about components' responsibilities for a little bit. Our code makes the ArticleList component smarter than it should be. For example, the ArticleList component knows that we have an authors list. It should not. It also knows that every article object has an authorId, and that information does not really belong there. This is not about readability, it's about dependency. The way we have our code now, we depend on what this ArticleList component does with that information, and that can be avoided. The rule of thumb here is simple. Try to make the children components less smart and more of presentational

components and try to have the top-level container component (like App in our case) control the logic of things like how to look up an author. An author is a property of an article, so the only component that should deal with this author property is the Article component. We violated that rule when we used the authorId attribute inside the ArticleList component. Now, take a moment and think about what can we do to improve this code into a better separation of responsibilities. Looking up one author information is an action related to a single article. So it makes sense to have the Article component do that, because it is the one who knows its authorId value. However, the App component is the one that has information about all the authors, so it is the one that can actually carry the operation of looking up an author. This is where we introduce a bridge. We write a function in the App component, and we let the Article component invoke that function. So we have an action. Let's name it lookupAuthor. We need to define this action in the App component and invoke it inside the Article component. In the future, we might have more actions, like, for example, updateRating on an article. So, let's introduce an object here in the App component to hold all of the actions. We'll call it articleActions. So this is an object, and inside that object we'll define lookupAuthor. This lookupAuthor is a function that is going to receive the authorId, and it will simply do this. state. authors, so normal lookup for that authorId. Very simple function. Now, we need to make this articleActions object available inside the Article component, so we'll need to pass it down the tree. We don't need to pass the authors list anymore, so instead, we pass articleActions, which is an object on the instance of this App component. And we'll need to do the same here in ArticleList. No more passing down the author object. Instead, we'll pass actions = props. articleActions. So, in the context of a single Article component, the property can be just actions. Now, this is slightly better for dependency, because this ArticleList component does not really know or care about what's in the articleActions object. It just relays that to its child, the Article component. If, for example, we need to change the logic of articleActions, we don't need to change the ArticleList component anymore, which is a win. Inside the Article component, instead of reading the author object from the props here, we now have actions, and we can look up the author of this article from the actions. So, actions. LookupAuthor using article. authorId. And the UI will behave exactly the same, but the components' responsibilities now are a little bit better.

Jest Snapshot Testing

So far, we created a simple React application to work with some test data. We created three components, App, ArticleList, and Article, and we talked a little bit about the components' responsibilities and how to decide on what data elements and behavior elements to pass down

the components tree. However, we have not written any tests for our components. We've been testing things manually, and in the next few clips we're going to make some changes here. So without having any tests to confirm that the changes are not going to break things, we will be in the dark. So we already have jest here configured for this application, so we can simply add some tests for the components. Now there are two major types of tests we can do here; we can add normal tests, tests that will test certain logic in our code. We can also use jest snapshots testing feature. So let me show you what that means. Under the components folder, let's add the jest tests folder. We're going to start by testing the article list component. This means that we have to understand the dependency of this ArticleList component. So this article list component depends on having an article object, and inside that object it expects an object that has an ID attribute. It also expect another property, articleActions, which is another object. So we're going to fake this dependency just to smoke test this ArticleList component. So inside the test component, let's create a new file, call it articleListTest. js. I'm going to start by bringing in some dependencies. So we need React, because we're going to be rendering components. We also need the component that we're testing, so we're going to bring in the ArticleList component from up one level article list, because now we're in the tests directory. We going to also bring in the react test renderer, so import renderer from the react-test-renderer library. This is the library we're going to use to create the React components' snapshots. So bring in this as a dependency, so -- div. Now we can test. So in here again we need to bring in the test syntax, so we're going to describe the ArticleList component, and in here we want to make sure that it renders correctly. Inside the test, we use the renderer, so we do renderer. create the ArticleList component. Let me actually show you how that looks like, so let's just render a div and let's put this in a variable, call this variable element, and I'm just going to console. log that variable. Let's run jest. Now we're seeing this variable right here. It's a react test instance, so this is the actual instance of the element and it has a lot of information that I don't really need for the test. There's one thing we can do is we can convert this to JSON. So if we do toJSON it would give me the actual object representation of that element. So this is much better. This is a lot easier to test. And when we start rendering nested elements, they're all going to be inside this children property, so this is practically a tree. So now let's go back to rendering our article list, and again, I need to pass in all the dependencies of this articleList. So we figured out that this articleList depends on an articles object, and it also expects an article actions object as well. So we need to fake those for the article List, and doing that is really simple. Let's have an object here, call this testProps, and it will have articles, and in here we will just have some test data, so I'm going to do two articles, one article a and another article b. And I'll task these articles as test props to the articleList. Now every article object is required to have an ID attribute, so let's make sure that we have an ID attribute here as well. And

then the other thing that the article list expects is an article action object, which has many things. Now this is going to be a full render, so it will actually render the Article components and invoke everything inside the Article components. So I need to pass in an actual lookup author here for the other component to look it up. Now we can just use a mock instead of an actual lookup author function here. We can just do jest. fn. This will give me a mock up function, and since the author component expects the lookup author function to return an object, we will return a fake object. So in here the default implementation is going to be just a fake object, just like that. And now we can pass these test properties to the article list component. So we'll just spread them here. And this will render the article list component. In fact, let's take a look. Fix this type of and it's also this here. Now we're seeing the tree, so we've got type, props, children, and I have two children, because I passed in two articles. So now we can write our expectations. The first expectation we can do is a snapshot expectation, so we're expecting the tree to match snapshot. The first time you run it, it will create a snapshot of the article list component, and every time you run it, it will just compare the new snapshot to the old snapshot. So if anything changes in the article list component or its children, then the snapshot is going to fail. So let me show you what that means. If you look under snapshot, and the snapshots, by the way, belongs to your source control, so we're going to commit that, if you look under shapshot, you'll see an actual representation of the article list component. Let's see how valuable that is. Let's say that after having this snapshot here, we went to this article list component and we want to try start refactoring things, so instead of passing an article as an object we're going to go ahead and spread this article, and the minute I save I have a failing test. So this one line added a lot of value to my tests, because I now have confirmation that my components are not rendering. So let's go back and fix that, go back to successful here. Now sometimes you'll change your component and the change will be valid. For example, let's go to article and go to the styles here and maybe change the date to be 0. 8 instead of 0. 85. This is a valid change. I am actually changing the mark-up of my components, so this will cause the snapchat to fail, but it will give you the exact difference of what's going on. So the font size was 8. 5 and now it's 8. Do you want this to be the new snapshot? Do you want to consider this as a valid change and update the snapshot that we have on file? And if you do, you can just hit the u button, and the u button is going to update the snapshot. So now the snapshot has the 0. 8 value instead of the 0. 85 value. So this snapshot testing is a quick thing you can do for every component really and have a validation that the component is rendering as the previous time you rendered it. So I encourage you to always do snapshots for all the components, but snapshot is not the only thing we can test, right? We have the tree itself and we have our code that we need to test. Now, it's very important that you pick the right things to test. For example, this part here is my own code. I'm here mapping an array of

articles into an array of Article components, so that's the thing I should test. I should test that the children of this article list component corresponds to whatever object I pass in. So since I passed in an object with two records, I should test that the component outputs two children, and that's really easy since I have this children attribute on the tree. So I can do something like expect the tree. children. length to be 2 and that's it. Now I have an extra test that confirms the mapping operation that I just did in the article list component.

Server-side Rendering of React Components

In the previous clip, we did some testing for the ArticleList component, highlighting the capabilities of jest to do snapshot testing, which is useful. In this clip, let's think about the server-side rendering of the React component. Particularly, what happens if we don't have JavaScript enabled here? If we disable JavaScript, what's going to happen? Our application is going to be that, and search engines when indexing this application are basically going to see just that. So we can fix this problem by rendering this exact same React application on the server as well. When we do that, not only do we get search engine optimization here, but we also get a little bit of performance benefit, because when mounting React on the client side, if the browser already has a copy of the application, then React will just do nothing. Server-rendering this application is really straightforward, because we don't have any async data yet. So let's start with the views here. Index. js. Instead of what we have here, we are going to insert an HTML string that represents our application as rendered by React. We'll do a variable and in here we'll render some initial content. Now, this initial content will be prepared by the endpoint. However, before we go do that, when we render the application on the client side, React is going to compute the difference between what we render server side and what we're about to render client side. And it's actually sensitive to spaces. So, don't leave these spaces here. To make sure when the div happens it's the exact same string. Okay, so now we go back to server. js here, and instead of this test here, we're going to have initialContent, and this is something that we need to compute. So let's assume that we have a serverRender function, and this function is going to return the initialContent. We will import this function from serverRender in here, so on the same level. And then, we'll just use whatever this function returns as the initial content. So this file is ready here. It just invokes server render and server render is responsible for rendering the content. The reason I'm making this into a function, because eventually you might need to render different things based on parameters, like what endpoint you're rendering for, for example. But for now we'll just keep it simple. So now, we create serverRender under lib serverRender. js. So serverRender is going to need to import React, because we will be rendering React components. We also need

ReactDOMServer, and this is something that we can import from the react-dom/server. This object has an API to render, for example, a React application into a string, so we also need to import the application that we want to render. So it's app from. /components/App. This is the component that we can start with to render our application into a string. So now let's create a function, call this function serverRender. This function is going to simply return the string that represents the App component. So we can do that using ReactDOMServer. renderToString the App component, just like that. Let's export serverRender and we can test. We basically refresh. And here's the thing, this is happening without JavaScript. JavaScript is completely disabled now, so this is the actual content that I'm seeing as a search engine and also, if we enable JavaScript here and refresh, what happens is that React is going to do nothing on the client side, because the browser at this point already has a version of the application that exactly matches what React is attempting to render. So the first React operation here is going to be no operation at all. And that's definitely a performance win, especially on very slow clients. So let me actually show you what performance means here. Let's go back to not shipping the initial content. So no initial HTML. React has to mount the application in the DOM. And in here in Chrome, let's simulate a slow CPU. Let's actually also disable cache while the dev tools is open. In the performance tab setting, CPU throttling is 20 slow down, and then I don't need the performance tab. Let's monitor the network tab instead. And remember, we don't have HTML shipped initially by the server. So now, if I refresh this page, you'll see how the CPU is taking some time to render the application and during that time I'm not seeing anything, just empty application. While if we ship initial HTML, and do the exact same thing with the exact same throttling of performance, if we refresh now, the CPU will still try to do something, but while doing its thing I'm actually seeing the actual application here. Not only I'm seeing the actual application, I can also interact with the actual application, because it's all pre-rendered HTML. So this is much better performance on this very slow processor. The initial content was just rendered and I didn't have to wait for the slow processor to compile the application in JavaScript.

Summary

You've made it through this module. Well done. We've covered a lot here, but we're still just starting really. We configured a full-stack JavaScript environment in this module. We used Express as a web server, and used Babel for both server-side and client-side code, and to package the client-side code into a single bundled file, we used Webpack. We've also used Jest for testing, which helped us test vanilla JavaScript code and React components as well. We also explored the snapshot testing feature in Jest. We created an example React application that

works with mock data and analyzed briefly our choice of components design, especially around data flow. And finally, we used ReactDOMServer to make the server node respond with an initial HTML string that represents the exact same React application on the client side, and that improved the performance of the application. In the next module, we'll start changing the state of the React application and see how to work with official API data.

Working with an Asynchronous API

Introduction

Hello, in the last module, I've used a simple React example application to explain important introductory concepts about building and testing a full-stack React application that works with data. But we so far only worked with initial data coming from memory, which you'd probably never do in an actual production React application. So let's get real. In this module, we'll drop our in-memory data and replace it with an official API, and see what needs to be done on both the client and the server to deal with that. If you've skipped sections in the last module or if you want to simply sync with the current state of the code we wrote so far, you can checkout tag module-3-start, and run the following commands: yarn, to install all the dependencies. This will take a few moments on a fresh install. Then, in one terminal, start the pm2 watcher process with yarn dev. Then, yarn pm2 logs to watch the logs. In another terminal, start the webpack watcher process with yarn webpack, and then in your browser, you should see the initial data displayed in a React application. This is so far server-side ready as well. The application will render initial HTML without the need to execute any JavaScript in the browser, and initially, ReactDOM in the browser will do nothing.

Red/Green/Refactor

Before we dive into dealing with asynchronous data, I'd like to refactor a few things. We're currently in what we call a green state. We have a few tests to indicate that, green tests. But of course, in our case we don't really have very good test coverage. The more green tests we have, the greater would be our confidence to make changes to the source code, but we also have other signals indicating a green state. For example, this application rendering correctly in front of our

eyes without any errors in the console, that's also another valid signal that we are in a green state. Even the fact that webpack is rendering correctly and pm-2 is not giving me any errors, these are all valid signal for my green state, so I feel confident about doing a little bit of refactoring at this point, and all of these signals combined will give me the confidence that my refactoring didn't really break things. So the first thing I'd like to refactor here is the server render file. I think this file shares a lot of similarities with the index file under the components folder. Server render renders the React application to a string while the index file here and their components render the React application to the DOM. So I think these two files belong under the same category, and that's exactly what I'm going to do. I'm going to create a new folder here, call this folder renderers, and I'll move index. js under renderers. Since index. js is the renderer for the DOM, I'm going to rename it to dom. js, and I'll also move serverRender under renderer. I'll just call it server in there. So rename it to just server. Now you'll see right away that my signals are starting to give me indicators that things are broken now, so we need to fix them. So we can actually simply use a git grep command to see where we used the server under file before. So it was only used in lib server js, so let's go there, and previously we imported the function on the same level here, so now we need to import it from lib renderers server. The index file we can grep for index and see where we used it, and we only used that in the webpack configuration file, so we'll go to webpack config. We previously indicated that the entry point for the Webpack configuration is components/Index. js, and now it's renderers/dom. js. So this should actually make the webpack renderer and the signal here happy. It doesn't look like pm2 is happy, so it cannot find that, and the reason is we are actually on the same level, so we don't need lib. Now here's the thing. This is starting to become confusing, right? I have to keep track of where I'm rendering a file from. Am I on the same level as renderer? Do I need to go up level and then close renderer? So I think the relative require in this case is not a good option. I think it's better if we can do something like start from lib, or better actually, start from renderers, because all of our code is under the lib directory. Now, when you're requiring a file from the same level, like we're requiring config here, I think relative require in that case is ok. It's a good signal that the file is on the same level, but when we start requiring files from different level, like going up one or two levels and then going down another level, this relative require becomes confusing. I think in that case an absolute level require is much cleaner, and there's actually a really easy way to do that in node. All we need to do is indicate the NODE_PATH variable before our command. So if I go to package. json here, I start my command with pm2 start, I can say something like NODE_PATH equal. /lib directory, which means all of my requires should start from under the lib directory. So let's actually test that. Let's check the status of the PM2 process. So I have a PM2 process here. I actually need to delete this process, so I'm going to do PM2 delete server. It will start a new process using our dev script that understands

the NODE_PATH, so yarn dev, and let's check the logs one more time. Yarn PM2 logs, and looks like we have another error, and this error is because in server. js we are referencing a wrong path, so let's fix that under server. js, so the components folder here is no longer in the same level. So now, with the NODE_PATH configuration that I just did, I have two options: I can either go up one level, and this should work, or I can just say components/app, because the components folder is directly under the lib folder, which is my NODE_PATH. So let's check that out. PM2 is happy. It's running on 8080. What about webpack? Webpack is still not happy. So in dom. js, let's go to dom. js, cannot resolve app. Again, we are not on the same level anymore, so we'll do components app. But here's the thing. This NODE_PATH trick that we did for the PM2 process is not going to work for Webpack, because Webpack actually uses its own logic for resolving the files, but the fix is easy. In the Webpack configuration, all we need to do is add a resolve property. This resolve property is an object, and in that resolve property we define the modules property, and that's an array of all the folders that we want Webpack to use for resolving its dependencies. So now we have a new folder, and we can use path. resolve here, and my new folder that I want Webpack to resolve at is the. /lib folder, but I also need to include the node_modules folder in this case, because I am overriding the defaults. So now Webpack understands that it should look inside the lib folder and inside the node_modules folder when it encounters a require line. Let's test. Restart Webpack and there you go, things are rendering. Perfect. Let's see if the tests are passing, and they are. Let's now test that the application is rendering and it is. So what I just did is called green refactor. Basically I am refactoring a lot, but I am in the green state. I know things are working. I have plenty of signals to indicate that, so I have more confidence in the refactoring that I just did. This is important to remember: never refactor code in the red state. If you have failing tests, if you have an application that's not rendering, it's not the time for refactoring. Get the tests to pass first, no matter how bad the code that you write is to get those tests to pass. Get them to pass first. Get the application to render it. Get all the signals to give you indicators that things are working, commit that state, then refactor.

Working with a Separate State Manager

In the previous clip, which I tagged in the repo as 3. 2, we've done some refactoring. One change allowed us to use absolute import paths instead of relative ones. I think this makes understanding where the files are an easier process, but it also comes with the added cost of maintaining that NODE_PATH environment variable and a webpack resolve config. However, there is one other benefit for this approach that I'd like to mention here, because I think it's relevant. It's how easy this approach makes developing your own npm packages locally. So let me show you what I

mean. Let's assume we decided to maintain this DataApi. js file as its own package, knowing that we're going to be adding a lot of code here, and simply because we know we're going to need the exact same code in a different project. So we need to start a new git repo for this package. For simplicity in this course, I'm just going to manage this new package under the lib directory. Let's give it a more generic name, call it state API. And let's go in there. Now, of course I'm going to manage this package locally here just to keep things simple for this course and to keep all the code in one repo, but you can imagine that this folder can be a git submodule, for example, pointing to a separate package. Ok, so we'll take the previous code and we'll place it under state api. I'm going to create a new file. Let's actually put it under lib directory inside the state api. And we'll just place it in index. js. Exact same code. To make this state-api into its own package, we can yarn init, answer the few questions, and I'm going to make the entry point lib index. js. And we have package. json. So, the plan is to eventually publish this package. But I'm not going to do that now. We'll publish when we're ready. But here's the thing. I want my application to use this new package, and in development mode I just want my application to read directly from this state api. But when I take this project to production, I'm not going to have this state api directory. I'm just going to use the published package. All right, so I'm going to keep an eye on the tests, and I'm going to remove the previous data api, and I think we have tests here that should start failing. So let's go ahead and fix them. Everywhere we previously used data api, we now need to change it to use the new package. All right, so it looks like I'm using it in App. js. So instead of the previous up one level data api, all I need to do now is state api. And this will work locally, because of the NODE_PATH that we just did, and it will work in Production by reading state api from node_modules when we install state api as a dependency in production. This is actually really cool, because I didn't need to do any npm linking or stuff like that, and I get to grow both projects locally here in development. We need to also change the other test where we used data api, so instead of data api here, we are using state api. Just like that. And this should actually make the tests pass. So now, when we're ready, we publish this package to npm, and we push our host project to deploy without this local state api code, and things will just work because this is a normal import statement that will read from node_modules as well. Of course, before we publish this package, we should really transpile it with Babel, because it's probably a bad idea to publish an npm package that uses features that are not yet in node. So, what we do in this case, is we publish the Babel transpiled code instead of the code that we write.

Dealing with an Asynchronous API on the Client

It's time for us to get real and start working with async data. So far, our application worked within memory test data. I cannot do that anymore. We need to work with async data. And to do that, instead of faking an asynchronous API on the client, we're going to fake an asynchronous API on the server, because we're doing full stack and we want to make sure that our application is going to a server render even with async data. So this involves coding a little bit in Express, but it's really easy. In server. js we're going to import data from test data, just like we did on the client, but now we're doing this on the server side, and in here we are going to expose data as an API with the /data endpoint. The code is actually really simple. We just send this data, and Express is going to figure out that this is a JSON object. Let's actually test that. We'll go to localhost /data, and you should see the data now in an API endpoint. Very good. So now back to the client code. Now we have an API endpoint to fetch the data from, which means we need an Ajax library to do the fetching for us. My go to library is axios, and we can't read the data directly in the constructor here, because now the data is asynchronous, which means I need something like componentDidMount to start fetching the data. So that's exactly what we're going to do next. We're going to use axios. get call for /data, which is the API that we just prepared. Now this is an asynchronous call, so we can simply await on it just like that, and I'll go ahead and put that in a constant. I'm going to call this rawData here as well. Now since this is an await call, that means that we need to label this function as async. Then I can have my API. So I can do the same API call we did before. So new DataApi for rawData that I just got from the API, and once I have the API I can use a setState call. So I'm going to set the state to articles is api. getArticles and authors is api. getAuthors. Very simple. Previous state is not used here, so we can actually simply just use this object directly in the setState argument, but I like to always use a function argument just to make that into a habit. But we can actually just simply make that into the direct return here to simplify the code. So to test, we first need to bring in axios, so yarn add axios, and when that is done we'll restart Webpack, make sure it compiles correctly, and go test. Okay, so cannot read property articles of null. Why is that happening? Well, our application assume the existence of an articles object at the initial render. Before we go here, before the async, the application assumes that there is an articles object on the state. So we can fix this in one of many ways. We can, for example, render something else here if we don't have an articles or an authors object on the state, or we can simply just have an empty articles and authors objects on the state when the App component is getting mounted initially, just like that. So authors and articles are both empty objects. Let's test. Getting another error, cannot read property reduce of undefined. And this is actually because the axios response here is not the actual data that I'm interested in. It's a response object that wraps the actual data, so I'm going to call this response and what I'm interested in here is response. data. Okay, so I think that should work, hopefully, and it did. Okay,

great. What's happening server side? Are we good server side? And we can test that by disabling JavaScript and trying now. And we are not. Empty response. Why? Well, because this whole code here was not executed server-side. The server executed the initial state, which rendered an empty list of articles, and that was it. There is no componentDidMount. There is no setting of the state and the code server-side remained representing an empty articles collection. So now think about what we need to do in the server renderer to make the React application pre-render with the exact same async data that we fetched client-side, and don't assume that I can just import data here. I can't. The server side of this application will also need to fetch this data through Ajax, so just assume that this data is coming from a different node, not this exact node where the server is rendering.

Dealing with an Asynchronous API on the Server

Dealing with the async API from the server is also easy. We basically need to use axios here as well, so we'll import axios from axios, and we'll do the axios call here. However, before we do that, we need to make this App component server-side render friendly. Right now it's not. Let's take a look at the App component. So the App component starts with empty values for both articles and authors, and then asynchronously load the data with componentDidMount. So to make this App component server-side render friendly, we need to have it understand initial data. So the server-side code needs to render the App component with some initial data. So let's actually call it initialData, and in here we're going to pass some variable coming from the axios fetch call. This means that the App component needs to start understanding this initialData prop. So, for example, the initial articles will come from this. props. initialData. articles, and similarly, the initial authors will come from this. props. initialData. authors. So now if we pass an initial data to the App component, it will render correctly, but what we just did actually broke the client side, because the client side starts with empty props here. If we go to the client renderer, the dom. js, it renders the App without any props. So to make this code work both ways, we also need to pass initial data structure here. We can simply just pass an empty initial data object here, so let's do initialData, and in here we'll do articles is an empty object and authors is an empty object as well. This is the same initial data that I just removed from the state, and we'll pass this initialData in here. So with this change in the DOM renderer, the application will again render correctly on the client side, but now I have a way for the server side to also specify the initialData here as well. So the initialData in this case will come from the axios fetch call, which is actually exactly the same thing we did here. So we need the axios call, we need the DataApi, so we'll import that from state-api. Note that I didn't need to change any paths, right? It's all absolute under lib, and I'm

using await here, which means I need to label the server render as async as well. So now my initial data is this exact object, but on the server side. So let's do that. The initialData here is exactly the same object that I used on the state, and I just pass initialData. So with this change, both the server and the client will render the exact same thing. However, since we made the server render async, we now have to deal with this asynchronous function when we render the content as well. So inside server. js, the actual file where we're actually using server render, server render is now an async call, which means I can await on it and just label the function as async as well. Before we can test, this axios call worked for us on the client, because this / data was a relative URL on the client. On the server, we don't have that option, so instead of data let's do a variable here, and this will need to be HTTP host and port and then /data. And we can read the host and port because they are configurable. We'll just read them from the configuration directly. Let's take a look at configuration. We only have port actually in configuration. Let's add host. So I'll try to read that from the environment if it exists. If it doesn't, we'll just use localhost, and then back to the renderer, so import config. Instead of host and port, we'll do config. host and config. port. Let's make sure the server is happy, and it is. Let's test. Perfect, we're seeing the content with JavaScript disabled. Very cool. Now what happens when we enable JavaScript? Well, the application will just render. It will render server-side and then it will render client-side, but you'll notice that React is going to give you a warning. Now, this warning is important. So let me demonstrate why React is giving me a warning here. To do so, I'm going to the App component, and let's put a debugger statement right here in the render call. Here's the first stop of the debugger line. As you can see, the browser has a copy of the HTML, and that's because we rendered it server side. However, at this point, my state has empty articles objects and empty author objects, because the DOM renderer is trying to do exactly that, render the App component with empty articles object and empty authors object. So, in the second stop here, React has rendered my App component without any data, and then after this render call, when the empty App component is mounted, the code that we have in componentDidMount will fetch the data again and render it back in the browser. So this is actually wasteful on so many ways. First, we are fetching the exact same data twice, and second, we're throwing away a good DOM and replacing it with the exact same DOM. So that's wasteful. React should continue to do nothing in that case. So take a moment now and think about how we can solve this problem.

Delivering the Initial Data

We need to fix two problems. We need to avoid re-fetching the data client-side, so we can't actually do all of this. We also need to avoid throwing away the initial DOM that gets mounted

through the server render, and that's happening in the DOM renderer, because we're passing initialData as empty collection. So, to render the React application client-side, and have it sync with the DOM coming from the server side, we need to render the React application here with actual data. So we need access to the data here. So we can't pass an empty initialData object. Now, this initialData is something that we already fetched server-side, so we need to somehow communicate this initialData that exists server-side to the DOM renderer as well. So we can go simply fetch it from the server, but that requires a network operation. Instead of doing that, there is a simple trick that we can do. In the server renderer, instead of just exposing the initial markup, we can also expose the initial data. Let's do that. So in here I'm going to return two things. I'm going to return the initial markup, which is what I had before, but I'm also going to return the initial data, just like that. So now if I go to the server code in here, the result of waiting on the server renderer is going to be an initial content, but that's going to be an actual object. So I'm actually going to just spread this object here to have both initial markup and initial data available in the EJS template. Now I have initial markup, but I also have initial data, which I can expose with a script tag just like that. So we'll put this directly on the window object, so initial data is the initial data coming from the server variable in here. But to get this to work, we need to JSON. stringify it. This is just a requirement here in ejs, but the idea is to have the actual initial data object exposed to the client-side as a global variable. So now the client-side in here can use this initial data directly from the window objects, so window. initialData. And we can test our theory now by re-rendering the application. I see the content and I don't see any warnings from React. Why? Because now we're fetching the data once. In fact, we're delivering the data using the window object right here. You can see the data globally available on the window object and that's the exact object that the DOM renderer used to pre-render the React application and figured out that it needs to do nothing because it's the exact same markup.

Reading State from the State Manager

Now that we preload the initial data through the initial server request, we can clean up the async code that we have here in the App component. We don't need the componentDidMount, we don't need axios, and we don't need DataApi. Much cleaner. In fact, we can actually not use a state object at all. We're not changing the state, so we can totally get rid of the state object here if we want at this point, but I'm going to keep it because soon we'll introduce an action that changes this state, and we will do that through the external state manager that we started under the state-api package. So far, we've been using individual methods in the state-api package to structure the App state, but it would be ideal if the state-api package was responsible for the whole state

object and we just always read it from there. So, instead of specifying articles and authors here, we just specify a data store and always read the state of the application from that store. So that is exactly what I'm going to do right now. I'm going to get rid of this initialData object. And this DataApi variable, I'm going to rename it to StateApi just for consistency. And I'm going to call this store, not just an API. So, it's exactly the same thing, but I'm going to add responsibilities to this store. So, every time we create a new object from the state-api, we get back a store that manages everything about the data. And in here, instead of passing initialData, I'm going to pass the store object, just like that. So my app only cares about the store object. I don't have initialData here anymore, but I'm going to keep an initialData concept and just pass the rawData in here. Okay, so things will not work yet, because I need to do some changes on the client side as well. So let's take a look at the client side. So the client side will now have an initialData, but it's actually the rawData, so the client side will need to do exactly the same steps. We need to initialize the store object, so in this case it will be store equal new StateApi, but this time we're reading the rawData from the window object, window. initialData, and I need to pass this store object just like I did server side. Store is store. And I also need the state-api package. So now, let's go back to the App component. Instead of manually structuring the state of the application here, we can now read it from the store object, which we're passing here as a prop. So the store object is available here as this. props. store, and I'll just assume that this store object will have a getState method that returns the whole state. So that's all the changes that I need to do in the App component. Now, let's go to the state-api package. So under index here, keep in mind that we use that with getState. So in here I'm going to go ahead and expose a getState method, and inside getState I can return an object to represent both the articles, which I can read from this. getArticles, and the authors, which I can read from this. getAuthors here. And things will actually work at this point. In fact, let's test that and see where we are. Things are working just fine, both the server side and the client-side rendering correctly without any warnings from React. However, if I'm going to read this state multiple times, this means that I'm going to be returning this object and basically doing the conversion that I do inside getArticles and getAuthors every time I read the state. So that's actually not ideal. So let's fix that. Instead of managing rawData here, I'm going to go ahead and manage this. data, the actual data object that I'm going to expose. And this data object will have articles and I'll compute the value of these articles right here inside the constructor, so this. mapIntoObject from rawData. articles. I'll do exactly the same for the authors property, so authors to be computed from inside the map into object. This means I don't really need to expose getArticles and getAuthors object now, because I'm always exposing getState method, and the getState method can simply return this. data. So, what we have is equivalent, but now every time I read the state, I'm reading the exact same variable and I'm not doing the computation every time.

I think this is better. Make sure things are still rendering, and they are. Let me rename this into state-api again for consistency. And now since I have a state-api object that manages the state of the application, I can always read from there. So, for example, in the App component, I previously had a lookupAuthor object that I had to pass down to the Article component to look up an author. But now I can do this thing inside my state manager, so I'm going to kill this part here, I don't need it anymore, and inside the state-api, we can define lookupAuthor directly here. So this is going to take an authorId, and it will simply return this. data. authors for the passed-in authorId. So exactly the same logic, but now inside this state-api. Let me actually change those into class properties so that we don't have to deal with any binding. So getState is a Class Property here and it's using Arrow functions. So now if we go back to the App component, I don't need to pass articleActions down to the Article component. Instead, I can pass the store itself, because I have the look up author defined on the store itself. So look how much we're simplifying this App component now. Instead of articleActions, I'm going to pass store, which is part of the props of the component here. So this. props. store. And in articleList I need to do the same. Instead of passing actions, I'll just pass the store itself, so the store is now a property here, store, and inside the Article component, instead of reading the actions here, I'm going to go ahead and read the store. And the lookup author is part of the store API, just like that. And I think this should work as well, and it is working. It's rendering both server-side and client-side exactly the same, but now I have one manager of this state, this state API class from which I created the store object, and it manages the state. It manages the initial shape of the state, it manages any functions I need from that state. That made my React application much simpler. The React application depends on the store for reading the initial state. It also depends on the store for passing down some actions all the way to the Article component and looking up the author from this store here directly. However, to get this to work, I had to pass the store object to all the components, including this ArticleList component, which really did not care about the store. What do you think we can do to make that better? How can we make the Article component access the store, because it needs to, but without having the ArticleList component pass the store as a prop? Because again, the ArticleList component does not really need to be even aware of the existence of a store object. Think about that, and in the next module we'll talk about it. But before we go there, I think the changes that we made here probably broke the few tests that we have. Let's fix those. The first test is in the snapshot of the ArticleList component. Previously, we had article actions, now we have store, and it's actually the exact same signature. And the other failing tests is for the DataApi itself. So let's take a look at that and their tests DataApi. We are still reading getArticles here, so I'm going to go ahead and rename this as state-api for consistency, and I'll rename this as store. Now, instead of getArticles, I need to do getState, which is a function, and then read the articles

from there. And same thing for the authors. Instead of getAuthors I need to do to getState, and then read the authors from there. So that should make them pass.

Summary

In this module, we made the React application work with data coming from an asynchronous API. We started things out by refactoring the state manager into its own node package. We then saw how if we made the client fetch the initial data, we would need to make an unnecessary second fetch on the server. So instead of doing that, we made a single Ajax fetch on the server, made the server expose both initial markup and initial data, and used the state manager as a store to manage the state on the client side, feeding the state the initial data as exposed by the server on the initial request. But so far, we have not changed the state. We are ready to do that next. In the next module, we'll introduce a few concepts like the Context API, which everyone will tell you not to use, and the concept of higher order components, which are getting really popular among the React community.

The Context API and Higher Order Components

Type-checking with PropTypes

In this module, we'll explore React powerful context API and how to safely use it with higher-order components. This point in the project so far is tagged as module-4-start. Make sure that you have both PM2 and Webpack running, and you should see the simple application rendering here from asynchronous data, both client side and server side. So far, the application is really simple. We haven't done any type checking in our application at all, which generally means that when we have problems, we're not going to be able to discover those problems easily. So to protect ourselves from that, we can utilize the prop-types package from React. And it's really simple. We just import PropTypes and define a propTypes object with the properties that we need to define types for. But before we do that, let's introduce a problem. Let's take a look at the Article component. So the Article component assumes that it has an article object on the props, and then uses many properties of that object, including title and date and actually modify the date. So, if somehow the Article component was rendered with an article that does not have the

date property, we're in trouble. Let's simulate that. In the test data, let's just remove one of these date properties. Let's try to re-render. You'll notice that this article now has an invalid date, and React didn't really warn me about this at all, and that's because I don't have any type checking for my properties. So let's fix that. We'll import PropTypes from the prop-types package. This is a dependency that we need to bring in. Yarn add prop-types. Run Webpack again, and to use prop-types, under the definition of the Article component, we will define a propTypes object. This object has a property for every prop that we need to define types for. So we're interested in the article prop. Now we have many options for the prop types. In fact, they're all listed here, so you can do things like number object strings simple, but if you have an object with a certain shape, this shape prop type is really great, because with it we can define the prop types for the properties of an article. So let's do that. So article is PropTypes. shape. To solve the problem that we just introduced, we need to define the date property, which is PropTypes. string, and it's required, so I can't actually render my Article component without a date property on the article that I pass to the Article component. So how is that going to help me? Well, if I re-render the application now, I get a warning from the React. The date is marked required, but the value is undefined. So I have a clear signal here that I have a problem, which is really handy. This is the minimum type checking that you should do in a React application. If you want to have more features in type checking, there is flow type from Facebook as well, which is a static type checker for JavaScript. And you can use flow here for all JavaScript, not just React components. Okay, so now with this isRequired signal there, if we put the missing date back, that warning is going to go away. Refresh. It's rendering. No warnings. Of course, we should define all the properties of an article here, so we have title, we have date, and we have body, and actually all of them are strings and required. So title and body. Make sure things are rendering without any errors.

React's Context API

In the previous module, we introduced the store object to manage the state of this application, and we added a lookupAuthor method on that object. We needed to use this lookupAuthor method in the Article component, so we had to pass down the store object from the App component to the Article component, passing it through the ArticleList component as well. However, the ArticleList component didn't really need this store object. The question was, how can we make the ArticleComponent use the store object without having the ArticleList component deal with it? The answer is simply make it into a global variable, but since this store object is associated with an instance of the App component, we need a global object associated with that instance. React has a solution for that. It's the powerful Context API. The React

documentation warns about using the Context API, for good reasons. The context object is after all a global concept, and anything global should be avoided if possible. However, I think it's important to understand the context API, especially if you're using a library that uses it, like Redux or React Router. So I'll show you how to use the Context API, and I'll then tell you what libraries like Redux do to make you avoid the direct use of the Context API, which the React documentation warns that it's not stable and might change. So, the goal is, we want the application to continue to work, without this line here in the ArticleList component. Removing this line will break the application. Here's how we can use the context API to fix that. In the App component, we need to define the context object. We do that with a getChildContext function. Whatever we return from this function will be our context object, so we're going to return the store, which is a property on this App component. So this. props. store. Just like that. Now, to make the contexts API work, we need to define the context type. We do that with a static property here in the App component, childContextTypes. For every element of this new context that we're introducing, we have to specify a type. So the store, for example, is an object. So we can use the prop-types library to do that. PropTypes. object. Of course, we need to import prop-types, and that's it. That's all we need to do to make the store part of the context API, which can be globally available to any component within the React application. Now, for every component that needs to use this context, like, for example, the Article component, we also need to define the context types here. So article. contextTypes, very similar to propTypes. And in here we define that the store is an object using the prop-types library as well. And just by doing so, we declare that this Article component is allowed to use this context object, which has the store object globally defined on it. So a function component can access the context object using its second argument here. So the first argument is the props and the second argument is the context. And now, instead of reading the store from the props, we can read it directly from the context. So context. store. Let's make sure that things are rendering, and looks like they are. So let's review this really powerful context API. We define a getChildContext function. In that function, we return exactly the context object that we want to be globally available for every component. In our case, we just defined the store. We also have to define the context types. In here, we do childContextTypes and we define the type of every element on that context. Then, inside the component we also define what types from the context that we want to access, and then in the second argument for a function component, we can access the context. We can actually de-structure the story here directly if you want to, which would allow us to just use it directly here as well.

Shallow Rendering with Enzyme

The changes that we made to how to access the store broke the tests. Let's run them and press a to run all the tests. Now we have one failing test, which is the article list render. You'll see in the output here that we're now getting warnings about the proptypes. Remember that we added proptypes validation. So this is actually just a warning, but the actual error is that the lookup author method is no longer defined inside the Article component, and the reason is previously we passed the store down to the Article component from the article list component, and now we are reading the store globally. So this part here is no longer valid, and to get this test to work, since I'm not testing an App component here, I have to fake this now global context object to make the Article component render correctly. So that powerful context API that we just used is making testing of components much harder, because now we have a global dependency. And this is happening because the renderer here is a tree renderer. It renders the full tree, so when I say render the article list, it will start from the article list component and render that, and then it will actually render every child component inside that article list component. Now we can do a few things here. We can either adopt an integration testing approach, which means pass actual data to the component and try to fake the context of the store. This is probably a good strategy to test the top level component, but for components below that, I think a unit testing strategy is much better. So let me tell you what that means. With a unit testing mentality, the point of this test here is to test the article list component, not the Article component. And React has a concept to do that. Instead of tree rendering, we can use shallow rendering. Shallow rendering will only render the article list component, and then for every article inside the article list component, it will just have a stub that in here there will be an article. Now, the react-test-renderer actually support shallow rendering, but there is a much better library to work with shallow rendering. It's called Enzyme from Airbnb. This tool actually enriches the React built-in testing utilities with much better syntax. For example, it adopts a syntax similar to jQuery for finding elements in the DOM, which is really cool, but we're going to use it here for its shallow rendering capabilities. So this is going to be a dependency. We need to yarn add --div enzyme, and instead of the react test renderer, we're going to import the shallow function from enzyme. Ok, so the article list component still expects an articles object like that. There is no longer store, so I can remove this from the test props, and instead of rendering a tree, we're going to render what's commonly named as wrapper. So instead of renderer. create, we can just call this shallow function. We don't need to convert things to JSON. So I'm going to comment out this part and take a look at what this wrapper look like. It's a shallow wrapper object, so for example, we can do something like wrapper. node. props. children. Take a look at that, and you see that this is the array of the

rendered articles. So the syntax is a little bit different now. So instead of the previous tree. children now we can do something like wrapper. node. props. children. length. Let's actually make sure that this is going to pass, and it looks like it is passing. So I'm going to remove this console log, and I can actually just expect the wrapper itself to match the snapshot. Enzyme support that. However our snapshot is actually different now. So previously we were rendering actual articles, and now with the shallow wrapper we are not rendering the articles. So you'll see inside the shallow render output that this is where we're going to render an article, but we're not really actually rendering an article. So this is perfect. I can update the snapshot now, and it's going to pass, and if you take a look at the snapshot now, you'll see how it is not rendering every article and just putting an Article component to be rendered here. However, I still see the prop type validation firing, which actually surprises me, I wouldn't expect Enzyme to invoke the prop type validation, because it's only shallow rendering the article list component, not the article. So there might be a bug here somewhere that's causing Enzyme to invoke the prop types of an article and it shouldn't. But if I want to work around it, I can simply import the Article component here, so import article from up one level article, and I can fake the prop types of the Article components. So article. proptypes equal an empty object, and that should remove the warning. So let's keep this syntax for now. So enzyme really has a lot of features. For example, instead of wrapper. node. props. children, I can, for example, do something like wrapper. find Article, and that should also be two. So this find syntax really is a convenient way of locating elements rendered within a component instead of depending on the shape of the tree, which is really cool.

Presentational Components and Container Components

Let's take another look at the Article component. It now uses the context object to access the store. We've seen how introducing this global context object made testing things harder. If, for example, we want to test this Article component now, we have to fake a context object. We can't even shallow render this component without that. This is one other reason why we should consider a different approach. One thing we can do is to split this Article component into two, one component is responsible for extracting the store out of the context API, and another responsible for rendering an article object. A component that does anything beside presenting UI is popularly known as a container component. So basically, we'll consider this Article component here the presentational component, and then we'll create another component, call it article container. This is also a function component. And this component is going to be responsible for extracting the store out of the context. So the signature is the same, props and store out of context. And now, from within the article container component, we simply return the Article

component, and we'll copy the props. The exact same props. Just like that. So this is a pure container component. Now we have the option of passing the, for example, store as a property to the Article component directly from the context here. Because this here will be context object, so I am destructuring the store out of that and now I am passing the store as a property, which means in the Article component I don't need to access the context anymore. I can read the store from the props. We can read it from here. This also means that I don't need the context types on the Article component anymore. I just need it on the article container component, just like that. So let me move this part next to this part, and I need to export the article container in this case, because that's what I need to use, and now I have two separate components. A presentational component, which is easier to test now that it doesn't depend on a context object, and the container component, with the only purpose of being responsible for extracting the store out of the context object. So, this is much better already, but we can go one step further. We can actually make this article container component a generic one. Say, for example, that we want to access a store from other components. Instead of duplicating this container logic per component, we can create a generic function that generates a container component responsible for extracting the store out of the context object. That function is what's popularly known as higher order component.

Higher Order Components

We're about to create a generic function that generates a container component to provide any component with the store object without having it deal directly with the context API. Since we're providing a store here, I'm going to call it the storeProvider and I'll just place it here on the same level as the Article component. So we're going to remove this article container component logic and instead assume that the article container is going to be generated using the storeProvider function. Now, this function will need to know which component it's providing for, so we'll pass the Article component as its only argument here. In fact, we can simply just have this function call here as the default export here directly. Let's now create storeProvider, so under components, storeProvider. So this is going to be a simple function, storeProvider. This function is going to receive a certain component, and it will basically create a container component and return that. So let's do that. Let's create a new component. I'm just going to call it WithStore. It's a function component, so it will have props and it will have the store object destructured out of the context argument, and this WithStore component is going to render our original component with the props copied out and the store passed as a property directly to the original component. And of course, since this WithStore component accesses the context API, we need to define the context

types here, which are basically what we had before, so store is a PropTypes. object, and I'm already using both React, so import React, and prop-types as well. And this storeProvider is going to simply return with store. And I need to export this storeProvider function. That's basically it. This is the simplest higher-order component function. It's a function that would receive a regular React component. It will wrap it with a new component. We call this component WithStore, and this wrapper component will have some other logic like, for example in our case, it is fetching the store out of the context API and providing the store to the original component as a prop. So at this point, we should test things out. So make sure the application is rendering, and it's still rendering. So now if you inspect the tree of the application, you'll see that the ArticleList component actually renders the WithStore component, and every WithStore component has the store context API, and it renders an Article component and that Article component only gets props. So our contained Article component is now a pure presentational component and the WithStore component is the container component. And it's the only one that is accessing the context API, and it's a generic one. So say, for example, I needed the store in the ArticleList component, I can also have it wrapped with this storeProvider function that generates a component. It's usually a good practice to have the name of this container component related to the component that it contains, and we can easily do that using the React displayName property, so we can do something like WithStore. displayName is, for example, we can do the original component name and Container. So we'll be back to, for example, ArticleContainer or any other component container. Let's test that. So I've got App, ArticleList and then ArticleContainer. And now every time I provide the store, I'll get a container component that wraps an original component and pass down the store as a prop. Now, usually when we create higher-order components, we don't use function components, we use class components. Because usually a higher-order component is going to manage some kind of state and maybe need some component lifecycle method. So instead of function component, we can simply use a class component, so I'm going to return class extends React. Component, and with a class component we do things a bit differently. So we can use static properties for the display name, and also for the context types, and after that we need to define the render function, which is going to return the exact same thing that we have here. So component passing in the props in the store. However, in a class component we need to do this. props, and we need to read the context from this. context. store. So this would be exactly equivalent, but now using a class component instead of a function component, and now we can manage the state if we need to, and we can use lifecycle methods as well. I forgot a return statement here. Let's test and make sure things are still working, and I'm seeing the article container component wrapping Article components. There is one other small fix that we need to do. The tests are now failing, and they're failing basically on

this line, because previously we were trying to find the Article component in the shallow rendered output, and right now a shallow rendered article list component will only have article container components. It will not see the Article components at all. So we need to simply change this into article container, and at this point I can actually remove our previous hack for the prop types on an article, because the Article component is not included at all in the shallow rendered output of the article list component. So if we run the tests now, this part should pass. However, we still need to update our snapshots, because it's now rendering article containers and not articles, so we'll just do that.

Mapping Extra Props

The Article component is now a contained one, which takes it further into the presentational direction, but one can argue that it still depends on the store API here by calling the lookupAuthor method directly from the store object. It would be ideal if we can extract this logic as well and have it maintained by a different entity. Since we now have a storeProvider function that basically renders this article with the same props, we can have this function also pass extra props. Just like we did with the store, we can have it pass the author directly. So I want to basically remove this line and also remove the store constant here, and just assume that the author object is coming from the props as well. Now, this author logic needs two things to be invoked, a store and an article, which is part of the original props. So I can basically have a function that applies to exactly these two things, the store, and the originalProps of the wrapped component, and I'll make this function return a simple object with the new props that I want this Article component to have. So, that'll be the author, which is coming from a call to store. lookupAuthor for the originalProps. article. authorId. So, a good name for this function would be extraProps. Now, since the storeProvider has access to both the originalProps and the store object, all we need to do is have the storeProvider invoke this extraProps function and make its returned object part of the props that it uses on the wrapped Article component. We'll use a functional programming trick to pass the extraProps function to the storeProvider. We'll make storeProvider a function that returns another function, and pass extraProps to the first function, and keep the wrapped component as the argument for the second function. So in storeProvider, it'll now be a function that returns another function. Arrow functions make this really easy to write, and now this first function will have access to an extraProps function as an argument. And in here, when we render the original component, we'll also pass a new set of props. We'll spread the extraProps function called using the store object, which is part of the context here, so this. context. store, and the original props, which in here would be just this. props. Just like that. And

that's it. Things should actually work at this point, exactly as they do. But if you take a look at the tree now, you'll see that the article container component passed down both article and author. It also passed down the store, which I didn't really use here, but I'll leave it in case some other component needs extra control over the store object. So this storeProvider function is now a customizable higher order component, because it can accept a generic function, extraProps, that we can use in any component to isolate any props logic that depends on the store, which leaves the wrapped component as a pure presentational one that only receives normal props and present a UI for them. If you've used the React Redux bindings before, this should look familiar, because it's similar to what the connect function in that library does. This is now a recognized, good pattern to do when you need components to access things out of the Context API because it isolates that access into just one function instead of different individual components accessing the Context API on their own.

Summary

In this module, we covered a few different topics to improve the React application that we're writing. We first talked about validating components props using the PropTypes package, and we used that to make sure every article object that we pass to an Article component has the right shape. We then explored React's powerful context API and how it can be used to pass React-application-scoped global variable to any component in the tree. We used the context object to give children components access to the store object. Using the context API made testing things a bit harder, so we talked about ways around that and one of them is shallow rendering and we used the Enzyme package to do that. One simple trick when working with the context API is to split a component into two. Leave one component purely a presentational one that maps data to UI, and have another component that contains the presentational one and perform any non-presentational logic like accessing the context API, for example. Using a higher order component function, we can make the presentational to container relation a generic one that allows us to wrap any presentational component with a context-fetching container component. We can also customize this container component to provide extra props, which allowed us to further extract any logic that depends on the global context out of the presentational components and into a separate function that is testable on its own. So far, we've been only reading static values from an initial state. It's time to start thinking about what happens when the state which is managed outside of the React application changes.

Subscribing to State

Upgrading Dependencies

In the previous module, we talked about the Context API and Higher order components, but so far we have not changed anything we put on the state. We've just initialized state with some data. So in this module, we'll start working with the state, and we have a few options to do so. We'll first talk about the React's built-in setState method and then we'll talk about how to manage the state externally. If you need to sync your code with the current code of the repo, the last commit is tagged as module-5-start. Before we dive into working with the state, let's do a quick upgrade of the application dependencies. It's been awhile since I first started this repo, because you know, putting together a Pluralsight course takes some time. We'll probably find a few packages that can be upgraded, and most importantly React itself released 15. 6 recently, so I thought now is a good point to upgrade and make sure everything still works as expected. We can use the yarn upgrade-interactive command, which will give us a list of all the packages that can be upgraded. So it looks like we have a few here, including React and ReactDOM. So we can partially select the packages that we need to upgrade or hit a, to select all of them. We'll just upgrade all packages. And once all the packages are updated, we'll restart our system, make sure everything still works. Restart Webpack. Let's also make sure the tests are running. It looks like tests are fine. Webpack is done. Let's test. Everything looks good. Let's also make sure that we're still server rendering, by disabling JavaScript here. Looks like we're still server rendering.

Using the setState Function

Let's put a few things on the state. We'll start with a search component that filters the displayed articles. Let's create a new component for that. I'm going to call it search bar, and it's a regular React component. And we'll make it return an input element. I'll give it a type search. And let's also put a placeholder, something like Enter search term, closing tag. And to render this component, we'll include it in the App in here. So we'll import search bar and we'll include it in the rendered output in here, search bar. Of course we can't just return two components like this. So we have two options. In React 16, actually we can return an array, just like that, instead of just one element. And of course if you want to return an array, we have to give each element a key attribute. So this is a good option, if the rendered output is generated with an array. But for a simple list of component like this case, I'd rather just return a div. So we'll make this component return a div. And we'll make that div include the two components. And we can test that, make

sure that search bar renders. And it does. So now we want to type something in here and filter the list as we type. This means after we're done typing, we want to put this search term on the state, so that the App component gets to filter the list of articles based on that search term. So in the search bar component, we'll do something like onChange, go ahead and handle search. We'll put that as a handle search function on the component instance, and we'll define that handle search function. And basically I want to first console. log the value of the input. Now we have a few options to extract the value of the input as a user type. The first option is to define a ref attribute in here. This ref attribute takes in a function, and this function exposes the input element itself as we type, so we can place something like this. searchInput is that input element that the function exposes. And in here we can access the value the user types with this. searchInput. value, pretty simple. Let's actually test that. So hello, and you'll see that we access the value as the user types. The other option, and the more recommended option, is to put the value the user types on the component state itself. So instead of doing a ref attribute in here, we'll define a state property here for the component, and on that state let's put a search term, which starts as empty string, and we'll control the value of this input element through this new state property, so this. state. searchterm. So this makes the component a controlled one, because the value is being controlled by React. But onChange we need to update the state of this search term, this handle search function receives the event itself, and we can do something like this. setState every time the user types. We'll go ahead and set the search term to be the event. target, which is the input element,. value, and to access this search term, we can simply read it from the state, so this. state. searchTerm. So let's go ahead and test that. Now when we type, not only do we get the value here, but React is also aware of this search term. So if we inspect this search bar component, we'll see that the search term is now managed through the state of the search bar component. So this is a more preferable option to get the input from the user, because this way the search bar component will be aware of the state change that happened in the UI. But now after we type a term, just like that, we need to make the App component aware of this search term, because the App component is the one that's going to filter the list of articles. However, instead of filtering the article on every character typed, we should filter it when the user is done typing. So instead of doing a filter on h, we'll do a filter on just hello, which means that we need to debounce the action of making the App component aware of that state. So we can actually test this debouncing feature before we go the App component. So after we set the state, we want a function that debounce the action of submitting the state to the component. If you want to do something after we set the state, in React we have to use the second argument for setState, which is a callback that gets invoked after the state operation is done. So the debounce operation should go here. So you can simply use the debounce function from the excellent lodash libraries. So we'll import

debounce from lodash. debounce, just like that. So when we import it this way, lodash. debounce, that means that we're not really importing the whole lodash library, we're just importing the debounce function from the lodash library. This is a dependency, however, so we want to yarn add lodash. debounce. So let's define a component function here. Let's call it doSearch, and this function is going to be debounced, so this function is a debounced version. And we'll go ahead and test it with just a console. log statement, so console. log. And I'm going to read this. state. searchTerm in here, and we'll debounce this after a delay of 300 milliseconds. And inside the setState, we'll just invoke doSearch, so this. doSearch. And we can go ahead and test that. My expectation would be, as I type, this handle search is going to set the state, and it's going to invoke a debounce function, which means I'm not going to see the console. log statement, until I pause for 300 milliseconds. So we can go ahead and test that. And I'm going to type in hello. And you can see that only hello was console logged, so not every character I typed. And this works if I hit backspace as well, you'll see an empty string. So we can assume that this component is going to receive the do search operation from the App component, because the App component needs to be aware of this search term. So we can simply assume that the function name would be also doSearch. So in here, instead of console logging the term, we'll do this. props. doSearch. So the component receives a doSearch function, and it will debounce it locally here, and call this locally debounce function inside the callback of the setState component. So now in the App component, we'll have to pass a doSearch operation here to the search bar component. And in here, we'll call it locally this. setSearchTerm, so set search term will be a function that receives the search term. So it receives search term, and we can simply just use this. setState and put the search term in here. So now when we type in the local search bar component, it will invoke this function and this function is going to set the search term on the state of the App component. And now the App component can go ahead and use this search term to filter the list of articles. So we can test that. Basically my expectation here, as I type a full word and pause for 300 milliseconds, I should see a search term on the state of the App component. So let's go ahead and verify that. And we see a search term on the state of the App component. And if we clear the search term, the search term is going to be cleared on the state of the App component. However, in here we were initially reading the states from the store, and now we're locally changing the state of the App component. So we're going to fix that next. But for the App component to actually use the search term, we should probably also initialize it as an empty string in the store. So we'll go to the store, and in here the initial data will also set search term to be an empty string. So now in the App component all we need to do is filter the list of articles according to the search term. So in here we'll do something like, let articles = this. state. articles. And if there is a search term, we need to filter the list of articles. Now the list of articles here is an object. So there is no built in filter

operation for an object. So if we want to filter it, we have to filter it through an array and convert the array back into object. But the lodash library has an excellent function that can help us with that. It's called pickBy. So we'll import that, so pickBy from lodash. pickBy. And this is a dependency, so yarn add lodash. pickBy. And using pickBy is really simple. We'll just do articles here = pickBy, and this pickby has two arguments. So the first argument is the object that you want to filter, and the second argument is a function. And this function receives both the key and value for every item in the object. And we want to return true if we want to pick that item. So since we want to search for the value title and description, we can do something like value. title. match, the searchterm. Or value. body. match, searchterm. Of course, I need to read the search term out of the states, so we'll go ahead and destructure both articles and search term from the state object. And now instead of using this. state. articles in here, we'll use the articles variable, which could be filtered based on the search term. I think we can test that. So we'll go ahead and do some search. Looks like we have an error, cannot read property match of undefined. So actually pickBy receives the value first and then the key. And we're not really using the key, so we can actually skip that. Let's test now and go ahead and search for better. And looks like it's working. So we now have something on the state. But again we set the state locally, so now the store is not really aware of this search term. So in the next clip we'll fix that problem. And always read the state from the store and have the store itself aware of any state changes.

Subscribing to an External State

We're now writing to the state through the React setState method. However, we previously read the state from the store. The React built-in setState method is usually good enough if you have a very small state and you don't have a lot of actions on this state object. However when the state gets bigger, it's usually a good idea to manage the state separately from the React application, so that you don't have a lot of logic in one React component. And that's exactly what we're going to do, because we previously read the states from the store. So the idea is we're going to make this store object responsible for everything state. And the App component will always just read from the store object. So my goal is to move these three lines into the store. So we'll comment them out here, and I'll put them in the state API index. js, set search term. And now this store doesn't have a setState method, because, well, this setState method is coming from React, so we'll come back to this part and implement it somehow in the store. But once we have a set search term, we can call it directly here. So basically instead of all this, we can actually call set search term directly from the store, so this. props. store. setSearchTerm, and this way the store object is going to be aware of this search term, and I'm not managing the state in the React component at all. So we'll

go ahead and implement set search term. So this store manages add data property, and it just returns that property. So I need to set the search term here. So this is super simple, we can just do this. data. searchterm = searchterm. However once we do that, we have a disconnection between the App component and the store, because we are changing the data in the store object; however, the App component reads the state out of the store object only when it gets constructed. So we basically need this React component to update when the store state changes. And we have a few options to do that, but the idea is that we need to subscribe to store changes, and by subscribe I mean provide the store object a callback to be executed every time that store object changes its data. We can implement this in a lot of ways. So for example, we can make this state API an event emitter, and every time we change the state, we can do something like this. emit a change event. And we can make the App component add a listener to this change event emitter and invoke a callback then. This is actually the core of the flux pattern in React applications if you're just using a vanilla flux implementation. A flux store isn't an event emitter, just like that. However the simpler choice, is how the Redux library does that, which is basically to manage subscription in the same object. And when it's time to notify subscribers, we just loop over them. So this is simple, let's do that. We'll manage subscribers in a variable on the instance here. Let's just call it subscriptions and we can manage that with an object, so every time anybody subscribes the store will just place it their subscription on that object. And we're going to need an ID to identify every subscription, so we'll do last subscription ID and we'll start that with one, so every time we subscribe a user we're going to increment that subscription ID. So we basically need two methods here on the store object. We first need a subscribe method, so every component that needs to subscribe to the state of this store it's going to call this method so this would be a function and it receives a callback. I'll talk about that in a little bit. But we also need to unsubscribe, so every component that subscribe should also be able to unsubscribe somehow. Right? When we unmount the component we should unsubscribe, because this is an active listener, so if the component gets unmounted, we don't want this listener to exist anymore. So subscription is easy. We basically want to add an element to this. subscription, so this. subscription, and we'll use this. lastSubscriptionId, and we just place the callback function on that. So the value for the subscription is just the callback function. So the callback function is what the subscriber is interested to be executed, so basically when I subscribe to something I provide a callback function, and that means that I'm interested in your data change and every time that data changes you should invoke this callback. So I'm going to go ahead and increment this. lastSubscriptionID for the next one. In fact, let me actually increment it before, so that we get to return the subscription ID. We can start that from zero so that we have one as the first subscription ID, and I'll just return this. lastSubscriptionID. And the reason I'm returning this.

lastSubscriptionID is so that I enable the unsubscribe to the subscriptionID, so the unsubscribe will have a subscription ID in here. And it will just simply delete this. subscriptions for that subscription ID. Very simple. So to subscribe to the store, you just call the subscribe method with a callback function to be executed when the data changes. And to unsubscribe, when you subscribe you get a subscription ID and you can unsubscribe by calling the unsubscribe method with that subscription ID. So let's go ahead and subscribe to this store from the App component. We can do that in componentDidMount. So basically in componentDidMount, once we have a component, we'll go ahead and do this. props. store. subscribe, and we want to pass a callback here. Right? And this is going to give us a subscription ID, right? So we can actually place that on the instance like that so that we access it in component will unmount. So now in component will unmount, we will just do this. props. store. unsubscribe using this. subscriptionID. So this is very simple, we just subscribe to the store, we get back a subscription ID and when we unmount the component we just unsubscribe. However, we need to know what to do here. Right? In here, what are we going to do? So the App component subscription basically needs to update the component State, the React internal state, every time the store state changes, because we want to keep both of these in sync. So we'll create a new method. We'll call this method onStoreChange. This method is local here, so we can do something like this. setState and we can read the state from the store. So again, this. props. store. getState, just like that. And we can pass this on store change to the subscribe as a callback, so we'll do this. onStoreChange. So basically what I'm doing here is that I'm expressing interest in the store state and I'd like the store to execute this callback every time its data changes. This way the App components state will be completely in sync with the store's State. So I get to manage the state in a store object externally to the App component, and I get to reflect those changes to the App component to get my React application to update when it needs to. Okay, so now all we need to do to finish this feature is basically to notify all subscribers. So every time a subscriber subscribes, we're going to put them on this subscription object. So when we do any data change, like this line, we need to notify all subscribers basically. So let's create a new method here and call this method notifySubscribers, and this is a very simple method. We'll basically need to loop over this. subscriptions, which is an object, so we'll simply do object. values, and we get all the callbacks we're going to loop over these callbacks. And for each callback, we'll get access to that callback here, so the value here is the callback. And we want to simply execute that callback, just like that. So loop over all the subscription values, which are callback functions. And for each one of them, just execute it. So now, every time we do a change like that, we want to call notify subscribers, just like that, this. notifySubscribers. So I think at this point we can actually test. Let's do that. So we'll re-render an application, make sure we don't see any errors, and try to search. And looks like it's working.

Here's what's happening, when I typed the letter here, the state of the store itself has changed. So I can highlight this and inspect it $r. props. store. getState, and you'll see that the search term is set on the state of the store. However my React application rendered because I subscribed to that store. So every time we set the search term, we're going to notify all subscribers. So possibly more than one component is going to subscribe to the same store. So when we do a data change, when we do a state change, we want to notify all subscribers. However we will probably change more than just the search term on the state. So every time we change the state of the store component here, we need to notify subscribers. So it's probably a good idea to have a method that changes the state and just call that method all the time. So we can, for example, instead of notify subscribers here, and instead of setting the data here, we can think about calling this method. We do something like this dot maybe, I'm going to call it merge with state. And the reason I called it merge with state is because I'm going to pass in this search term, just like that. And I want this search term object to be merged with the state that I have originally, and I don't need to call notify subscribers, because merge with state is going to do that. So merge with state is a function. This function receives a value, let's call this value state change. And what I'm going to do is I'm going to merge it with the state, so we'll do this. data = the original data, and then my states change, just like that. So basically copy the original data and then merge the state change objects that I pass. And after I do that I can notify subscribers. So now every time I want to do any change, I do it just through this merge with state function that's going to generically do that operation and notify all the subscribers. And this way I'll have just one call to notify subscribers, I don't have to do it every time I do a change to the state, as long as I do the change through the merge with state function. Let's go ahead and test this change as well. Refresh, search for something. And looks like things are working.

Passing State to Child Components

Now that we have a state that we can subscribe to, I'm going to go ahead and introduce another component that subscribes to this state. So how about we display the current timestamp and make it tick every second. So I'll go ahead and do that, let's add a new component here, we'll call this component timestamp. js, and it's a regular class component, and I'm going to call it timestamp. So we can actually subscribe to the state directly here or we can pass the time to the component as a prop. I'm going to go ahead and pass the time to this component as a prop, so basically in the App component we will import timestamp. And I'm going to include it here right before the search bar, and this component is going to receive the timestamp value, so basically it will have a timestamp value that it's going to read from the state, so this. state. timestamp. And

we'll just go ahead and render this timestamp in the timestamp component, so in here we'll just render it in a div, so render this. props. timestamp. Now I don't have a timestamp property on the state, so that's something that I need to initialize. And we'll initialize it in the store. So in the store in here, we need a timestamp property, and we can initialize it with new date. And I think we can test that, so refresh. And looks like I have an error. So what's going on here? So, objects are not valid as React child, that's because we rendered the timestamp here directly. We can tostring that, and I think that will work. So let's go ahead and test that. And I have the timestamp. Of course, the timestamp is not ticking, because I didn't change it on the state. So we'll do that next. So we can do that in the store itself. Basically in here we can setInterval, and inside the setInterval, we'll do something every one second, and in here it will just do this. merge with state a new timestamp object that is basically a new date, just like that. And I think that should work. So let's go ahead and test that. And it looks like the time is ticking right there. Now I'm getting a warning here, React attempted to reuse markup, which tells me that the server rendered output is different than the client rendered output. And let's think about that. So we have a new date and then the interval is going to kick and change that. So basically this is probably happening before the component mounts and the component mounts with a different value. So to solve this problem, instead of doing this in the constructor, we can do it in componentDidMount. So let's do an action here in the store and call this action something like startClock. And this is going to basically do exactly that. We format that, and we'll just call this startClock action from the App componentDidMount. So on componentDidMount, after we subscribe to the store, we're going to do this. prop. store. startClock. So let's go ahead and test that now. So now my App is ticking and I don't have the problem of the check sum here. Cool! So the reason I started a timeStamp timer here is that in the next module, we're going to talk about the performance a little bit. And we're going to see how putting things on the state is going to affect the performance of the React application.

Subscribing to State from Child Components

This Timestamp component is currently a simple presentational component; in fact, we don't need it to be a class component here at all, we can simply use a function component. One can argue, though, that this timestamp component should be a complete stateful one that manages its own state and tick the timer as well. In which case, we don't need the global state. However, I'm going to assume that this timestamp is needed elsewhere in the application and leave it as part of the global state. But instead of having the Timestamp component receive the value of the timestamp as a prop, maybe we can have the Timestamp component itself read that value

directly from the state. We are managing the state externally, so having child component read that state directly becomes an option. Please note that this is probably not the best option for this particular case, but the point here is to demonstrate that option, and in the next module we'll take a quick look at the performance of this application. So we can simply provide the global context store to this Timestamp component by using the storeProvider function. And to use that function, we just wrap the timestamp component with a storeProvider call. And remember that we need an extra props function. The extra props function actually is not optional, although we can easily make it optional here by checking for the value or provide the default argument. However I'm just going to leave it required for now. So we're going to need an extra props, I'm going to define this function here, and it takes in the store and the original props, in case I need those, and it returns an object of props. So let's think about that. I don't want to pass the timestamp directly here. I want to read the timestamp from the state. However, instead of reading it directly from the state like that, we can make it into an extra prop in here, so call this timestamp and read it from store. getState function dot timestamp, right? We have that option. We have access to the store here in the extra props. So this should work as expected. We don't need access to the original props, because actually I'm going to kill the original props, so in the timestamp here, we're no longer passing an original prop to the timestamp component. We are just rendering it and the timestamp component gets access to the store through the storeProvider and then it Maps an extra prop that is being read from the store state. And this will actually work. The Timestamp is still ticking. And we can validate that, we have a timestamp container, and we have a timestamp that takes the timestamp value as a prop. Now why exactly did this work? The timestamp component doesn't have a local state, it just gets property. Even the contained component doesn't have a local state. So this timestamp is actually working right now, because since the global state is changing, the App component is rendering, and the App component is rendering all the children components, including the timestamp container and the timestamp. Although if you think about it, the props and state of the timestamp container did not really change. So we might want to optimize our timestamp container, which is a valid case here. So I'm going to make it a pure component, which means if the props and state of the component didn't really change, then don't re-render. And if we do that, this timeStamp is going to stop ticking. So the question is why? Why did it stop ticking, when I changed the timestamp container to a pure one, although the timestamp value on the global State is still changing? We are reading that value directly from the timestamp component through the storeProvider. We're actually getting the initial value; however, the timestamp component is not ticking. So pause the video here and think about that. Why? The timestamp is a presentational component. It just gets properties, so it doesn't have any state, and now the container component is a pure component,

so it's not going to re-render unless it has a reason to, which means it has different props or it has different state. So although the global state is changing, this pure component doesn't really care about that. If we want to make the timestamp component a truly connected component, which means a component dependent on the global state, we basically have to fake a state change in the container component every time the store is changed. So we need to subscribe to the store. So we can actually do the exact same thing that we did in the App component. So I'm going to copy these lines and put them in the storeProvider in here, paste the exact same code with a few differences. We don't need to start the clock here, we don't need to do setState in here, we'll talk about that in a little bit. We need to read the store from context instead of from props, because this is the higher-order component that accesses the store directly from context. So we subscribe to the store, and every time the store is going to execute on store change. When the store changes, I don't really need to set the state here at all. I actually don't need to manage State on this component. All I need to do is give this component a signal that it needs to re-render, because the timestamp component is reading something from the store, which means I need to re-render this component to cause the timestamp component to re-render as well. So we can actually simply just use a force update call. So every time the state of the store changes, just force update this component, because we know that a contained component might read the state out of the store and we need that contained component to re-render. So we can test that. And now the timestamp is kicking again, because of that force update call. So the timestamp container is a pure component; however, it's being forced update every time the state of the store is changed, and this will cause the timestamp component to re-render. Let me say this one more time. This is probably not the best option for this particular case; however, we did that just to demonstrate the option that since the state is managed externally to the React application, children component can subscribe to that state as well. We don't have to pass the state from the App component. So if we have a component that is five or six levels deep, instead of passing the state through the tree, we can subscribe to it directly from the child component and we can actually do that through the same storeProvider function. So now that we have a change state event every second, we want to analyze what's going on the React application, what is the React application really rendering, what is the React application taking to the DOM? And we'll do that in the next module.

A Bit of Refactoring

Before we move on to the next module, there are a few things that we can refactor in this application. And we're in a green state, all the tests are running, and our use cases are working as

expected. The first thing we can do is right here. Previously we had the App component pass the function to set the search term to the search bar component, because we managed the state locally in the App component itself. But now that we managed the state externally in the store, there is no need for the App component to pass this action. We can directly call this action from the search bar component. So I'm going to remove this part and go to the search bar component, and in here, instead of doing do search, I can access the store directly through the storeProvider and invoke store. set search term directly here. All I need to do is import the store provider here and make this search bar component a connected component by calling the storeProvider, just like that. However, in this particular case, we don't need extra props in here, and the storeProvider function require an extra props. So I have the option of passing an empty function here, or alternatively, we can just make that extra props here an optional function and we can do that with a default argument in JavaScript. So extra props is an optional function that returns an empty object, so it'll merge an empty object with the props of the component. So now this search bar component does not specify an extra props, and that's okay, while the other components specify extra props that would be considered. So we should definitely go ahead and test that. Search again and things are still working. I've noticed while testing that our search is not case-sensitive, and also we're getting this force update can only update a mounted or mounting component, so let's fix these problems. To make the search case insensitive, we can go to the App component in here and do something like, make the search term a regular expression. So let's do search regular expression equal new regular expression for search term, and make it case insensitive, just like that. And in here instead of matching on search term, we'll match on search regular expression instead. So we can go ahead and test that. Now the search should be case insensitive. So now let's think about why this is happening, why force update is saying can only update a mounted component. This means that in our storeProvider here, when we do force update, we are force updating a component that was completely unmounted. And this is happening because of the order things are wired in our React application, so the order of the component mounting unmounting, and also the order of subscribing and unsubscribing, and the order of the store notifying subscribers. So there is a lot of optimization that can be done to prevent this particular problem. In fact, libraries like react-redux exist to fix these kind of situations. However, if you want to implement the very simple solution, that would simply be to treat this. subscriptionID as a flag so when you unmount we reset this flag to null. So this means that I don't have a subscription anymore, and in the store change here we can just check for this flag. So if there is a subscription ID, we'll go ahead and force update, but otherwise we won't, just like that. So this is a guard against updating the component that was unmounted, and this should actually work. I've just noticed that we have a typo here, so we can do a project find and replace to fix the typo, and we

do find all, and all of these look like they should be replaced, so replace all. Okay, so now we can go ahead and test that the force update is not firing. So basically you want to search, delete the search, search again, delete the search, and make sure no force update is firing for a component that was unmounted already. This is one of the many reasons why using a pre-optimized library connector, like, for example, react-redux, is such a good idea, if you don't want to deal with all these cases. And in most cases, you don't realize that you need these solutions. So in most cases, these library actually protect you from problems that you don't know you have. However, they also hide a learning opportunity, and this is why in this course we're not using these libraries. I'm trying to face these problems and learn from them. And there are by the way many other problems that we have with the code right now that we will hopefully uncover in the next module when we talk about performance optimization and see how to fix them. Okay let's take a look at see if there are other optimizations we can do. So I actually realized that we're still passing the store to the article list component. We actually don't need that anymore, because we can read the store directly from the context with a storeProvider function. So the article list component actually does not use the store at all, so that was an extra prop that's unneeded. Now sometimes it's not easy to manually spot these problems. However, in the next module we'll take a look at some techniques that will help us identify these problems as well. So I think this is good enough refactoring for now. There's a lot more that we can do. But in the interest of time, I'd like to move on to a different topic. And in the next module, we'll actually see some signals of things that should definitely be refactored. But instead of manually verifying any refactoring that we're doing right now, we will have actual numbers and actual measurements that will tell us if the refactoring was actually better than what we had before. So we'll do that in next module.

Summary

We started this module by upgrading the application package dependencies, and we used the upgrade-interactive yarn command to do that. I like this command, because it gives me a clear list of what will be upgraded, and I get to choose what to upgrade if I want to. We then started working with the state of the application. We first worked with React's built-in setState method, and we used that to manage a controlled SearchBar component and to place a searchTerm on the App top-level state, which the App component used to filter the displayed list of articles. We then moved the state of the app to an external entity, the store object, and we connected the App component to that state through a subscribe method. We also saw how to unsubscribe as well, and we implemented a simple notify all subscribers method that needed to be called on every store state change. Since we're managing the store externally, children component now

have the option to subscribe to that state directly instead of relying on props. We saw an example of that with a system timer that manages a timestamp value. We've placed that on the top-level state that is managed by the store object, and we first passed that value to a Timestamp component as a prop, and then saw how to use the storeProvider function to also make a component connected to the store's state. In the next module, we'll analyze the performance of this simple application and see what options we have to optimize it.

Performance Optimization

Understanding shouldComponentUpdate and componentWillUpdate

In this module, we're going to talk about performance optimization and what tools we can use to help us measure things to be optimized. The advice here is, if you want to optimized something, you should measure it first and measure it after the optimization, because otherwise you don't really know if your optimization actually made things better. In a React application, most of the optimization concerns happen during the update operation of a component. The component lifecycle methods help us identify any areas in the code that could use some optimization. So let's make sure we understand those first. There are 5 lifecycle methods that apply to an update process in a React component. componentWillReceiveProps, which has an argument of those nextProps that are received. Every time the parent component re-renders a child component, the componentWillReceiveProps is invoked with the new passed in props, even if those props are identical to the current props. So a call to componentWillReceiveProps does not really mean that the component is going to rerender. Also, if the component sets its state internally, this would generally not trigger the componentWillReceiveProps lifecycle method. So this method is actually not super useful for performance analysis. shouldComponentUpdate, has 2 arguments of the nextProps and the nextState for a current update operation. This lifecycle method is very important, because we can actually use it to cancel an update process. Returning false from this lifecycle method tells React to not re-render the component at all. This is very useful for pure components. If the component is being passed the exact props and state, and we know it's pure, which means its output will be exactly the same, we can use an if statement in a shouldComponentUpdate lifecycle method to compare current and next props and state and return false if they're the same. That's exactly what the React Pure Component class does. However in some cases, we might need different logic to short-circuit the React re-rendering

process based on other conditions. We'll see an example of that in this clip. The other lifecycle method that's helpful when analyzing why a component has re-rendered is componentWillUpdate, which is reported immediately before the rendering step. If we expect a component to not re-render, this lifecycle method should not be invoked. Also if this lifecycle method is being invoked while the rendered output is exactly the same, we basically have a wasteful render. Let's see an example of that. Let's work with the SearchBar component. The SearchBar component is a regular component, so I'm going to go ahead and define shouldComponentUpdate, and I'll just return true here, and I'll also define componentWillUpdate. And in here, let's just have a log statement updating SearchBar. Very simple. Let's test that. So every second we are updating SearchBar. However, we don't really need to update SearchBar at all. SearchBar doesn't need to be re-rendered every second. It's being re-rendered because the global State is changing and the App component is re-rendering, and that makes it re-render all the children component, including SearchBar. So in shouldComponentUpdate, if we just return false in here instead of true, and retest now. The SearchBar component is not being re-rendered at all. Right? Because we're short-circuiting the update process for that SearchBar component. However, that means we can't also type in the SearchBar component because it's not going to re-render at all. So to fix the problem without breaking the SearchBar component, we would compare nextProps with currentProps and nextState with currentState and return false, if they're exactly the same. That's actually what React pure component does. So we can just use pure component here. So let's test that. With a pure component, the SearchBar is not re-rendering, but we can type in it. And when we type it re-renders, right? Because we typed six letters, it re-rendered six times, but that's about it. It doesn't re-render every second. So here's my advice, always use pure component, unless you have a reason not to. Okay, so let me clear that up. And now let's target another component. Let's target the Timestamp component. Same thing, we will do a shouldComponentUpdate in here and just return true and the componentWillUpdate lifecycle method and console. log updating timestamp. Let's take a look at that. And the Timestamp component is getting updated every second, which is actually a valid case. Right? Because we are updating the timestamp every second. So in this particular case, React pure component doesn't add any value really, because we are updating the component every second. However, if this component is being updated for different reasons, other than the every second update, then a pure component would help. But we're going to need to dig deeper to know if the component is being updated for different reasons. However, let's assume that this Timestamp component is needed to render only the time part, not the date part, and actually not include the seconds, just hours and minutes. In that particular case, I don't need the Timestamp component to re-render every second, I just need it to render every minute. So let's test that. We can simply

make the Timestamp component renders just the hours and minutes using this expression, which I totally got out of Stack Overflow. So this casting method takes a second option here, and we can specify what to display in there. So we have hour and minute only now. And if we re-render the Timestamp component now, we see just hours and minutes. But the Timestamp component is still ticking every second. So actually 59 of those update operations are going to be wasteful. We don't need the component to re-render every second, we just need it to re-render every minute. Here's the thing, even if I converted this component to a pure component, that's not going to help me. I'm gonna comment out shouldComponentUpdate and leave it just to pure component and see if that's going to help me. Will it? It will not! It's still ticking every second. Why? Because actually our nextProp as read from store is going to be different, we are receiving a different timestamp. However, we're only displaying just the hours and minutes of that timestamp. So we have two solutions here, either we'll make the container component pass just the hours and minutes and just display that, which is actually really simple to do. We just move this method into here. And let's test that. So now it's not re-rendering every second, because the prop itself is being passed differently and the pure component is doing its thing here and saying I don't need to re-render. But let's assume this wasn't an option. I need the timestamp value to be passed to the Timestamp component and I need the render method to cast that value into just the hour and minute string. What can we do to make the component not re-render every second? We can use a custom shouldComponentUpdate. We can do something like the currentTimeDisplay, which is what we're rendering. And the nextTimeDisplay is the exact same statement, but instead of this. props nextProps, just like that. And we can return currentTimeDisplay does not equal nextTimeDisplay. So if what I'm displaying next is different than what I'm displaying currently, go ahead and re-render; otherwise, don't re-render. So we can test that. And now my time component rendered the hours and minute, and it's not being re-rendered every second. And that happens thanks to the custom condition that we just added in shouldComponentUpdate. And when the minute ticks, we'll go ahead and actually update the timestamp, because the next timestamp display is different than the current timestamp display. So that's a valid case for some custom logic inside your shouldComponentUpdate, unless you don't want the actual timestamp value in the Timestamp component and you can just pass it what it needs to re-render. Of course in here, we are repeating this logic, so we should probably extract it into a function. So I'm going to go ahead and do that off-camera. So we now have a time display function that returns the string that we are interested in, and we are using this time display function in the render method. And in shouldComponentUpdate we are comparing the time display for the current timestamp with the time display for the next timestamp. And if they're the same, we don't need to update. So let's test one more time and make sure the Timestamp component is rendering and it gets

updated when it needs to, but not all the time. So we didn't use the nextState here, we can remove that. And also we can remove the will component update. So I'll leave this in the code as an example of an actual use case for shouldComponentUpdate with some custom logic. But again, this is probably not the best way to do that, this is just an example.

Profiling Components in Chrome Using the react_perf Flag

We can use the Chrome performance tab to record and analyze the performance of an application as it's running. This is very helpful when analyzing why React components render and re-render. The report here shows very useful metrics on which we can zoom and analyze, but it gets better. React actually support a very useful flag that will give us even more details related to the React application. All we have to do is add a react_perf query string value here and record the performance of any interaction we want with the application. So let's do that and see what we get. Reload the app with react_perf while the Chrome performance tab is active. And here's where you're going to get differently. You're going to get this user timing section, and in this user timing section, let's go ahead and zoom on that, you'll see the actual React components here, and this is a visualization of every component mount update, unmount operations. So here's the App component, which took 24. 8 milliseconds to render. And during that time we rendered the Timestamp, we rendered the SearchBar, and we rendered an ArticleList, and the ArticleList rendered many other components in here. So you can zoom in to see what is going on in here. You'll see how, for example, in the Timestamp component, this is the time that timestamp container took, and this is the constructor for the timestamp container, this the render for the timestamp container, which rendered the timestamp. So you'll see a lot of detail for every operation in the component lifecycle. So this is helpful to take a general look and see if something is weird. If one component is actually taking longer than it should, or if an unexpected operation is happening in the wrong time or place. The numbers that you see here are all relative, because this is running in development mode. So actually in production, things are going to run faster, but this is still a useful indicator of what's going on in development. You can use this visualization to see if an update is happening by mistake. And you see the update tree here, so how deep the tree is being updated and how often the updates are occurring are all things that you can quickly visualize by just looking at these performance metrics. So this should actually also work in IE Edge, I think, but I haven't really tested that. But eventually this should work in all browsers, because React is using the standard user timing API to integrate these metrics.

Perf Addons: Avoiding Wasteful Rerenders with PureComponent

If you're interested in more details about what is going on in the React application, React has a performance tools addon that is really helpful, react-addons-perf. So let's go ahead and bring this to our application, yarn add -- dev react-perf-addons. And we'll go ahead and import this performance addons in our code. I am going to import it in the App component in here. So I'm only adding the react addons perf as a development dependency, because we're just going to use it in development mode. And we can use this perf addons in many ways, but the simplest really would be to just place it on the window. So window. Perf = Perf and then interact with this perf tool within the chrome console. But I think that's actually going to break our server in here, because we are doing server render and window is not defined on server render. So actually if you want to use window in a server rendered component, you have to put it behind an if statement. And that if statement, basically is if typeof window does not equal undefined, then you can use this statement, otherwise you will do nothing. I think this will make the app happy. And now if we re-render the application, we would have the performance object on the window. So this is actually the full api of the performance object. We can do a start and a stop, and we can print DOM exclusive, inclusive, operations and wasted. So let's go ahead and test those. We do Perf. start, that's a function, this would start the measurements. And our application is ticking every second really, and its re-rendering the whole tree every second, so I know that we have some wasteful operation going on. So let's go ahead and stop that. Just do stop, and then take a look at the wasteful operation, with Perf. printWasted, just like that. And you would see how we have so many wasted operation in this application. So ideally this printWasted function should output nothing, because otherwise you have a lot of renders that are totally unnecessary. So we can solve this problem now that we can measure it. However, before we solve this problem, we should see a consistent measurement. We should have a way of always seeing consistent numbers here. So to do, that I am going to invoke Perf. start and Perf. stop in particular points in the application. So how about in componentDidMount we do a Perf start. Now sometimes when you do that directly you get this error that the mark did not exist, and this is a specific issue with the Perf component. There are some restrictions on using it, but one thing we can do is to put the perf start code in a set immediate call to be invoked in the next tick of the event loop. So set immediate function and then perf start inside that function and that would be okay. And I'll go ahead and set timeout. And do that after exactly five seconds and perf stop in here. So after five seconds, we'll stop the performance and we'll go ahead and Perf. printWasted. So hopefully doing this would give me consistent numbers of the wasted operations. Let's go ahead and test. So after 5 seconds I should see the wasted table, and I see 25 article list components were rendered wastefully, Article container as well. The App component itself was wastefully re-rendered five times and so is the article timestamp and search component. Okay so let's retest this and make

sure we're getting the exact same numbers after five seconds, and we are. So this is consistent measures. Now I know if I start doing some changes, if these numbers go down or go up, I know that my changes are causing that, and not the fact that we are measuring different durations or different cases in the App component. We're always getting 25 in article lists, 25 in article container and so on. So there are many reasons why a React component gets wastefully re-rendered, and we can avoid most of these reasons by using pure components. So let's take a look at these components. So the container components are all wastefully re-rendering, because we're basically force updating those every time the state of the store changes, which is every second. So that makes sense, I'll come back to those later. But let's see why this article list component is being re-rendered, that does not make any sense. Right? Let's take a look at article list, and the article list here is a function component. And here's the thing, the article list component does not really need to re-render at all, because it doesn't change. What changes is the global time, so the article list component and all the articles really don't need to re-render here. But the thing is, function components always re-render, because React doesn't really optimize function components. So if you're not using a higher-order component that optimizes the performance of a function component, then using a function component is actually a bad idea, because it will always re-render no matter what. Now we have two solutions here. One solution is to call the article list component directly, instead of calling it as a component, because it's a function. Right? So instead of doing that, I can call this function directly. I could do something like, ArticleList, call it with an object that pass in the articles, just like that. And that actually is going to work and make us avoid the unnecessary render in the article list component. So let me test that. You'll see the application is still rendering correctly, and the wasteful table doesn't have my article list container. So that's one way of voiding an unnecessary render in React application. However, by doing so we really lose some value of this component. If we take a look at the react dev tools at this point, we're not going to see the article list component, we're just going to see a div here. There's no article list component surrounding the article container. But it is better, in terms of performance. However, how can we keep the article list component as an actual component in our tree, but not wastefully re-rendering its content when it doesn't need to? The answer is, don't use a function component, use a pure component instead. So I'm going to take the exact same return and instead of using an article list function component, I'm going to use a React. PureComponent ArticleList, and return exactly what we had before. And instead of props. articles, it's now this. props. articles. So the exact same thing, but instead of a function component, we're now using a pure component. And here's the thing, that will get rid of the wasteful re-render in the article list component. Let's test. So the application is still re-rendering through a component, and I don't have the article list component reported as a wasteful render,

thanks to the pure component. Because if your props and state are exactly the same, we don't need to re-render. This is an optimization that we can do. So I've probably said this a few times already, but here is the advice, always use pure component, unless you have a reason not to. And the other advice here, avoid function components, unless you are using higher-order components that optimize the performance of function components. For example, if you're using react-redux, then react-redux is going to optimize the performance of any wrapped component. So using function components in that case is okay. But if you're not using react-redux, you're much better off using pure components than using function components. The React team is working on optimizing function components, but until they do, avoid them. So we still have five instances of wasteful renders here. Let's see if pure components is going to solve that. I'm going to go ahead and change the Article component into a pure component, instead of a function component and see if that's going to help me. And let's take a look. So I previously had a twenty five wasteful re-render in the Article component, and I think those are going to go away with just using pure components, and they did. Let's take a look at the other components. So we still have four instances of wasteful re-render, three of them are actually for the Container components, which are all provided through the storeProvider. And there's also the App component itself, which is being wastefully re-rendered five times. So sometimes using a pure component is not going to be enough. For example, if we changed the App component into a pure component, that's actually not going to help us. And I'll go ahead and test that. Refresh and you'll see that the App component is still being wastefully re-rendered five times. So using a pure component is recommended, but just know that sometimes it's not enough. You have to dig deeper to know why the component is being re-rendered.

Making Store-connected Components Subscribe to Partial State

In the previous clip, we changed the App component into a pure component, but that did not help us, because the App component is still wastefully re-rendering five times in our measurement period. So to optimize the performance of the App component, we need to carefully analyze what is going on. And that means looking at render method of the App component and what render method that is using. And by carefully looking at the render method of the App component, we know that it's using the articles object and also the search term from the state. But the App component initializes its state as the full state of the store. And then for every change in the store, it locally sets the complete state of the store as its own state. But here's the thing, the App component doesn't really need the full stored state. It only needs the articles and the searchTerm out of the state. So the fact that we have more properties on the

state that changes, makes the App component re-render wastefully. Now there are a few solutions to this problem. The simplest solution is to customize the shouldComponentUpdate operation of the App component. And basically, we can have a condition here, to return true, only if the nextState. articles does not equal the current state. articles or when the nextState. searchTerm does not equal the current state. searchTerm. And that actually should solve our problem. Let's test. Re-render, and the App is working fine. The search is working and we don't have the App component as a wasteful render. The timestamp actually just ticked, so everything is working. But we don't have the App component as a wasteful render. And all of what we needed to do is to customize the shouldComponentUpdate and only actually re-render if we have a reason to. However, while this solution works, it's not really the best solution, because this means that there is a dependency between the render component and the shouldComponentUpdate. So every time we want to use a new property on the state inside the render function, that means we need to change the shouldComponentUpdate logic to include that thing, so we can actually do better than that. There's another solution to this problem. Instead of subscribing to the whole state, we can just subscribe to a portion of the state. So we can, for example, have a function in here. Let's name it appState, and this function reads the store state. So instead of reading the whole state in here, we can just read the articles and the searchTerm out of the store state. So destructure articles and searchTerm out of the store state and then return those, articles and searchTerm, just like that, because this is the exact state that we want out of the store. And now, instead of reading the state of the store directly, we can always read this. appState in here and also in here as well. And if we do that, the state of the App component only changes when either the articles or the searchTerm changes. So let's go ahead and test that. The app is rendering. I still search, things are working. We'll verify that the time is ticking and I do not have the App component as a wasteful render. So the application is working exactly the same, but now the App component only subscribes to a partial state of the store. And also the timestamp is still ticking. Now you might ask, how is it that the timestamp is still ticking when the App component is not really aware of the timestamp state? And the reason is, the timestamp component reads the timestamp state out of the store directly, it doesn't need the App component to read that. So we made the App component independent of that timestamp change, and now App component only re-renders when either the articles object or the searchTerm changes. And that's a much better way. In fact, we should always do that, we should always make a connected component subscribe to the part of the state that it's interested in, not the whole state. Because subscribing to the whole state will cause us to do wasteful re-renders. This also applies to the remaining cases of the wasteful re-renders, which are all the container component. Because the container component right now just force update themselves when the

state of the store changes, but really they don't need to. They only need to force update, if the state of the component that they contained is changing. And we can actually do this optimization really easily. So let's take a look at the contained components. The Article component is contained. It depends on the author, so it doesn't really depend on the state at all. The searchBar component doesn't depend on any extra states, as well. The timestamp component does depend on an extra state. So the timestamp component needs to force update when the timestamp property on the state changes, but nothing else, only when the timestamp property of that state changes. So the extra props function here is key. If we're returning from extra props, a property that depends on the store states, then we force update the component if that property changes on the state. So we can generically do that in the storeProvider function. Let's try to do that. Instead of force update, let's think about doing a setState call, but only for the part of the state that is actually relevant. So let's call this part, this. usedState. And I'll make that into a function. So this usedState function would be only the properties from the state that are being used by the wrapped component. And we can compute these properties from the extra props function. So we have two cases here. Sometime we return an extra prop that uses the state. And sometime we return an extra prop that doesn't really use the state. To simplify the logic of what we need to do next, let's just assume that everything in the extra props is a usedSate. So the usedState is actually the extra props call itself. So I'm going to create a usedState function and just return the extra props function, invoked with the context store and this. props. And I'll go ahead and use it here as well, so this. usedState. And now the container component only sets the state that's being used by the wrapped component. And I think that is going to get rid of some of these wasteful re-renders. Let's test. So the app is rendered correctly, but I still the article container, the searchBar container and the timestamp container. So let's go ahead and dig deeper. I'm going to kill this force update. And to figure out why these container component are still re-rendering, we need to do an inspection inside the componentWillUpdate. And in here, we can just simply console log this. state and the nextState, just to see what's going on in here. Let's go ahead and test that. So we have a few cases here. Let's first focus on the searchBar container, which is being wastefully re-rendered once and here is the call here. So it started with a null state and the nextState had an empty object. So we can easily fix that. So this is actually happening because I don't initialize the state. So let's go ahead and initialize the state, which is exactly the same, state = this. usedState. That actually should fix the searchBar component, because now it will always be an object. So let's test that. However, I'm getting this. usedState is not a function, and that's actually because we need to put this. usedState before this operation. Let's make sure things are working now. And they are, and now let's take a look at the wasteful renders. The wasteful renders is now only the timestamp component. So by making the higher order wrapped

component subscribes to only the part of the state that is being used, we fixed the wasteful renders of both the searchBar component and the articles component. And now the only wasteful renders is happening in the timestamp component. So why is the timestamp component being wastefully re-rendered every second? Well, because we have the logic of time display in here, inside the timestamp component itself. So the timestamp container is still ticking every second and it's still changing the timestamp value. And the display portion, the timestamp component, is only re-rendering every minute. So we can fix this problem by moving this logic to the timestamp container component. So instead of using a shouldComponentUpdate here, and instead of rendering this. timeDisplay timestamp in here, we can do something like this. props. timestampDisplay and call the timeDisplay function in here, so this. timeDisplay is what we're interested in. However, I don't have access to the component instance inside the extra props. But here's the thing, I don't need access to the component instance, because this timeDisplay function doesn't really depend on the component instance, so I can actually make it into a static function. And instead of calling it with this. timeDisplay, I call it with Timestamp, the component itself,. timeDisplay. So let's test that. Now the application is re-rendering, but I don't see the timestamp display at all. So let's fix that. So the property here is timestampDisplay. All right, let's test. I see the timestamp and it's not being wastefully re-rendered. In fact, the wasted table operation is empty. You see that? You don't have any more wasteful operations. And the timestamp component is still working; every minute the time is going to change and the timestampDisplay is going to change and the timestamp container is going to re-render correctly. When it does, it's not really being wastefully re-rendered, because it's re-rendering different content to the DOM. So now we can remove this componentWillUpdate lifecycle method, because we just used it to figure out why the component is being re-rendered. So let's do one final test. Make sure the search is working, and it is. Make sure the timestamp component is re-rendering, and it is. So at this point if you don't want to do any more performance optimization analysis, we can practically remove the Perf code. So both the setImmediate and the setTimeout calls can go away in here. So this analysis of why the component updated, is very important. In fact, there's a package that you can actually use to see why components are updating unnecessarily. And it gives you exactly the values before and after and it tells you if an unnecessary re-render is avoidable. So if you don't want to do what we did manually, you can also use that package to give you some clues of why some component are re-rendering unnecessarily.

Immutable Data Structures

Sometime when you use a pure component optimization, you'll run into unexpected problems. And this happens particularly when your state has mutable structures, like objects and arrays, and if you're actually mutating these structures. So let me demonstrate this problem with an example. Let's go to the state api class, where we manage the store. And we are managing the list of articles here as an object, which is a mutable data structure. So I'm going to do an example of mutating this structure. So let's assume that after one second, we received another article from the api. So to simulate that after one second, so I'll do a setTimeout operation that gets invoked after one second. And in here, we're going to have a fake article, and we'll append this fake article to the articles object that we have on the state. So I'll copy this fake article from the rawData, so we'll copy rawData. articles first, and we'll just give it a fake id. So now I have my data articles as an object, and if we want to add this fake article, one option would be to just do this. data. articles for this fakeArticle id = the fakeArticle. And I also need to notify the subscribers, right? So this. notifySubscribers call in here. And what we're doing here is we're basically mutating the state. And if we're mutating the state, the pure component optimization that we did in the App component is actually going to give us a problem, because as a pure component sees it, the articles object reference did not really change. So if we test what we just did, it's not going to work. I will not see this new article, again, because we are using a pure component. In fact, if we remove the pure component optimization from both the App component and the article list component, then what we just did should work. Let's test. So render the first list, and after one second the fake article should show up on the list. So the fact that we mutated our state is now preventing us from using pure component, which is bad. So I'm going to put back the pure components and go back to the same problem. Now we are changing the state of this component, but we are not re-rendering, because we are using pure component optimization. So how can we do the exact same operation without mutating the state? We can use a copy operation. So something like this. data equal, start by copying the current data, then change the articles object. And start by copying the current articles object from the data, and then add the new article. So in here, that would be fakeArticle. id is the fakeArticle. So what we're doing here is practically equivalent to the commented out line, but without mutating the state. We are copying the current state and then modifying a copy of the current state, which means for the pure component, the previous object and the new object are going to be different, and things should work, even though we have pure component. Let's test. Re-render, we get the list of articles first and after one second, a new article shows up. And again, we're still using pure components here and here. However, now we are not mutating our data structures. So here's the advice, always use immutable data structures, never mutate your data structures directly, because that's going to have performance implications on your application, eventually. However, when you start copying

object to avoid mutability, things might start to be harder than they should. Right here we needed a few lines of code to replace a single line of code. And we have to do these manual copy operations, which actually cost something, in terms of the resources available in the browser. So although we removed some complexities by not mutating the data directly, we added new complexities of needing to copy data around and use memory and cpu to do so. So the best of both worlds is to use some kind of immutability helper. Now you have options here. And the first option is the immutability helper package, which is a very simple helper that allows us to update an object without mutating it. So it actually returns a new copy of the object, and it uses syntax similar to MongoDB here as dollar sign set, dollar sign push. So this is actually the minimum you should use in a React application, if you're managing your own state and you want to utilize the pure component optimization. If you want a more advanced option, you can use the immutable js library, which has a lot more features. Like for example, you can work with lists and maps. And not only you can update these data structures in an immutable way, but you can also do operations like merge and deep merge and deep update. So it's a very rich library to work with immutable data structures. And the cool thing about it, is that it actually uses shared data to do these immutable operations in a very optimal way without wasting resources unnecessarily. But it does add complexities to the project. However, I think the complexities that it removes are much bigger than the complexities that it adds. So I've actually started using immutable js in all of my React projects, because I think the values it adds is simply great.

Summary

We talked about performance optimization in this module. We first talked about the update lifecycle methods that happen right before a component gets updated, and saw how two of these are very helpful for performance analysis and improvement. shouldComponentUpdate can be used to cancel an update process, and componentWillUpdate can be used to verify that an update is happening when it shouldn't. We talked about how React's PureComponent class optimizes the render process for pure components, but we also saw examples of how we can also use shouldComponentUpdate with custom logic. We looked at using Chrome's performance tab to visualize what's going on with the application and saw how React has a special react_perf flag that provides a user timing section in the recorded performance which visualizes the mount/unmount/update operations for every component in the app. React has another very handy tool to look at the performance of components, especially when they get updated, and that is the Perf addon. This tool allows us to start and stop measuring certain things about a React application, and then it will report those things out nicely. One helpful thing that it

measures is the wasted renders, which is when a component is being re-rendered, but its output remained exactly the same. Seeing those measurements allow you to know where in the application there is room for a good refactoring that would avoid any unnecessary re-rendering. We also learn a trick of how to subscribe to only the part of the state that is relevant to a component, because otherwise you'll have changes that drive unnecessary re-renders. This is only valid if you have an external state, but the same idea applies to props as well. A component should only receives props that are significant to its outputted render. Finally, we saw how simply mutating the data prevent us from using some optimization, which is bad. Using immutable data structure allows for much easier optimizations in React and it's a very good strategy to adopt globally. In the next and final module, we'll go over some best practices for deploying React applications to production. This also falls into the performance optimization category, but it's not an optional one. Running a React application in production without optimizing its build is a bad idea, and I'll tell you why in this next module.

Production Deployment Best Practices

React's Production Build

The one optimization in your project that you should definitely do is to use minified production builds for all dependencies and for your own application as well. Right now, we have a single bundle. js file that is about 3 MB, which is a big number. Even half a megabyte would be a big number when working with just text files. But it's not really just about the size and its related network cost, it's also about the performance of code after it's been downloaded, and that is especially true for React. In fact, this is so important, the React team made so many signals to make sure that you don't run a development build in production. React will give out warnings about that, and recently the dev tools introduced an icon with colors about that. This red background here means that you are using a development build for React. And if you go to a site that uses React in production, that icon should turn blue on black. So why is this so important for React? Well, because React has many helper tools in development that we can do without in production, including things like prop-types validations and the useful detailed warnings. These tools that are designed for development make the development build larger and slower. Building React for production basically strips these development tools out to make the build smaller and

faster. Of course, developing a React application using a production build would be a bad idea as well, because, for example, instead of getting a helpful warning when something is used in wrong way, you would just get a number or something that's not helpful at all. Unfortunately, the process of deploying things to production in general is not an easy one. There are many configurations needed for various reasons, and it's important that we understand these configurations and their reasons. If you want to sync the code as of now, it's tagged in the repo as module-7-start.

Continuous Integration and Test Coverage

The very first thing that should be validated before initiating a deploying process is the green build. You have to make sure all tests are passing and your test coverage is decent. I've been keeping an eye on the couple of tests that we wrote for this project manually, but relying on a manual check like that in a project is not a good idea. You are going to forget. You should automate the tests check. There are two things you can do for that: first, if the whole test suite runs fast, you should run all the tests before each and every git push you do. To automate that, you can use git hooks, which allow you to write custom code to be invoked before and after certain git operations, like commit or push. But practically, the test suite will start taking longer and longer, and running it per git push is going to start frustrating you. This is when you should go async. Use a continuous integration service and get alerted when the build breaks. An excellent testing integration that plays well with GitHub is Travis. It automates the process for you on every push. Failing tests should be treated as a top priority in any project. Ok, tests are passing and life is good, but how much of this project is actually being tested? This is a very hard to answer question, but there's something that we get out of the box with Jest that can help us make an estimate. It's the coverage report. Now we only did a few example tests for this project, so I bet our test coverage is not good at all. Let's take a look at that. All we need to do is to add a dash dash coverage arguments to the jest command. I'm actually going to introduce a new script, let's call it verify tests, and I'm going to call this with jest, I don't need the --watch for this one, but we'll do --coverage. Okay, so let's run this one, yarn verify tests, and it's running just with --coverage, and this is the report. The only component that was tested is article list, so we have good coverage there. We have a little bit of coverage in the other files, but it's not enough. For example, only 27% of the storeProvider function is tested, and jest actually gives you the lines that are not tested, so this is a good indicator of where you should be writing more tests. So ideally this report should be all green, and your coverage is 100%. But even if your coverage is actually a 100%, it doesn't really mean that everything is tested properly. It just means that all the lines and functions were executed during the testing run, but sometimes you have to execute functions and

lines in many ways and with different inputs to test all the cases. But this coverage report is your first step. Get this degree and maintain 100% coverage all the time, and then start thinking about the more cases that you need to test certain things. There are also other options where you can actually fail the test if the coverage is not good. If, for example, the coverage fails below a certain point, you want to just give a signal to everybody that the test is failing, because the test coverage dropped. Because, again, unless you automate these things, you're going to forget to check things like coverage. Running this coverage command is going to introduce a new coverage directory under this project, so this is something that we should probably git ignore. So in here I'm going to ignore coverage, make sure it doesn't show up in the Git Status, and now we can commit.

Upgrading Dependencies

The following step is optional, but right before you put the application in front of people to test it, it might be a good idea to see if there are any non-major upgrades that you should try to perform first. Non-major upgrades should not break things, theoretically, but might start giving you some deprecation warnings, and now is a good time to decide on those if they start to happen. This step should only be carried out in a green state, and preferably, if you have a human QA process, you do this upgrade right before you hand the project to the QA team. Let's take a look at the upgrades available for us now. We do yarn upgrade-interactive. And it looks like we have a few. Most of the NPM packages follow semantic versioning, with three numbers: Major, minor, and patch. For example, the React update here is just a patch, so it should be safe to do. However, the Webpack upgrade is a major one, so that might not be backward compatible. So if we choose to upgrade Webpack here, we want to make sure that things did not break. Webpack is probably okay to upgrade here, because it's in our dev flow, so verifying its task is easy. I'm going to go ahead and upgrade everything. So the first thing to check after the upgrade is the tests. So we'll run yarn jest, and our impressive test suite is still green. That's good. Let's restart things. So first, let me reload the pm2 process. Take a look at the logs, and pm2 seems to be okay here. Let's restart Webpack. And we're now running on Webpack 3, and it's not giving out any errors. So let me verify the application itself, make sure nothing broke. And things seem to be working, so I'd say this seems to be a safe upgrade to do. So ship it.

Separating Vendor Files

The bundle file for this application currently includes all the front-end dependencies like React, axios, and the lodash functions we used, and we need to deploy that to production. After that, we're going to make changes to our application and deploy a new version of this bundle. js file. However, if you think about it, when we deploy version 2, we're likely going to change just our application, not the vendor dependencies like React. When we deploy a new release of this bundle. js file, we would be invalidating the whole cached file in all browsers that used the application. This is wasting some good cache. We do not need to invalidate the whole thing, we only need to invalidate the part of bundle. js that we actually changed. This is why it's currently a good idea to split this bundle. js file into two files, one that represents our own application, and another to represent all the vendor files that would not be changed as frequently as our application. Webpack can help us with that. We need to do two configuration changes to make that happen. First, instead of this array in the entry file, we can do an object here. This object will have two properties; one property for the vendor files and another property for our own app. So we can take what we had before here and make that our own app, and then introduce another array for all the vendor files. Actually, the Babel polyfill is also a vendor file, so I'm going to remove it from here, keep just the DOM renderer in our app, and put the Babel polyfill file in here. And also we need to add react react-dom, we also used prop-types and axios, and we also used a few lodash functions. So we have lodash. debounce and we also have lodash. pickby. All of these packages are vendor files that would not be changing as frequently as our own application. We also need to include this commons chunk plugin for Webpack to do the actual separation. This receives a number of options. I'm going to copy an example here, we're doing this explicit vendor syntax. So we did the vendor, and now we need to include this CommonsChunkPlugin. This is something that we need to include as a plugin, so we do plugins is an array of plugins and we include the CommonsChunkPlugin. And we need to require Webpack, since we haven't done that before. Now before we test, since now we have two entry points, the output should be two files not just one bundle. js file. With Webpack we can use this syntax, we put name in square brackets like that, and that means for every entry point generate an output file that matches the entry point property in here. So we'll have vendor. js and we'll have App. js. So I think we can go ahead and test. Just restart Webpack. And now you should see two bundled files, App. js, which is our own application, and vendor. js, which is everything else. Now we also need to change the views ejs file to include both of them. So we need vendor. js and we also need App. js. This way when it's time to release a new version of our application, we only change App. js. Vendor. js can stay cached in all the browsers that previously used this application. However, when it's time to invalidate the cached version of App. js in all the browsers, we need to do something different here; otherwise, the browsers might not actually fetch the new release. So it's a good idea to have

something like a version here, so version 1. And also this is something that you want to do for vendor. js as well. So when we release version 2 of our application, we would change that into two. And when we upgrade the dependencies of our application and release a new vendor. js file, we would change that version separately. This versioning is also something that you should automate as well. And to do that, one option would be to generate this index. ejs file every time we release a new App. js file and make that part of the production build for this application. Now that we have two bundled files, we need to ignore these separately. So, public vendor. js and app. js, because both of these are generated content that should not be in source control.

Minifying Bundled Files

The next task is to minify these bundled files. Right now we have these sizes and we can use Webpack to minify them. So let me create a new script for that. I'm going to call it build-webpack. And basically we need to do webpack -p. So let me test that real quick. Yarn build-webpack. And you'll notice how the sizes for the files that we bundled went down significantly. The vendor. js file went from about 3 MB into just 260 KB, so that's one step that you should definitely do in production. Let's test our application now, and notice how the React dev tools icon is now blue on black, that means we're using a production build of React. And if you take a look at the network now, we are loading vendor. js and App. js with their minified versions here. So this makes the resources needed in production ready. However, we still need to do one more thing. We still need to build Node itself, because in development, we used babel-node and using babel-node in production is probably a bad idea.

Using Different Babel Configurations

We need to run Node. js in production without using Babel. Since we are using some JS features that are not yet supported natively by Node, we need to compile our Node project before running it in production. This is simple. We'll introduce a new script for that. Let's call it build-node. Since all of the files for this project are under the lib directory, we can use the Babel cli command to compile the whole lib directory into, say for example, a build directory. The command is babel lib -d build, so read all the files under the lib directory and compile everything into a new directory called build. We also need to use the --copy-files option. This is to copy any non-JavaScript files as well. However, building for Node is actually different than building for browsers. The latest Node. js supports most of what we used in this project while user browsers are likely to be still lagging behind, in terms of supporting recent JS features. For example, we used the async/await

feature in this project and that is supported natively in Node. We don't need Babel to do anything here when we compile for Node, but we do need Babel to compile that into something else for browsers. This means we need to introduce different Babel configurations for each target. We'll use the Babel configuration inside package. json for node itself, and then we'll introduce different Babel configuration for Webpack. To compile for node, instead of using the default configuration for the env preset, we can configure that. So basically we do an array here, and we'll configure env with a different object, and inside that object we specify the targets property, and that targets property is also an object for each target, and the only target that we care about here is node, and we specify a value of current in here, which means when you compile for node use the current running node. So for example, if the current running node supports things like async/await, babel would not compile that into something else. Also instead of compiling with stage 2 presets, we can specify the list of plugins that we actually used and we used 2 of them. We used class properties, and the plug-in name for that is transform-class-properties. We also used the object spread operator, and the plug-in name for that is transform object rest spread. So Babel allows you to be this specific about the plugins that you're actually using and what you need to compile. So let's go ahead and test this new script. We do yarn build-node, and you'll notice how we now have a build directory that was compiled out of the lib directory. It has the exact same structure, but inside the build directory, all the features that are not yet supported by node, like for example, import, will be converted into something that node understands. But features, like for example, async/await are not compiled, because those are supported by node natively. So the build directory is something that we should ignore as well, because it's generated. If we now run Webpack to compile the React application for browsers, it's going to use these exact same settings. But I don't want to do that. I don't want to compile for the node target. I want to compile for the browser's target. And if I don't specify a configuration object for the env presets, it will assume that we want to compile all the features into something that the browsers understand. So to override the Babel settings for just Webpack, we can change the configuration that we do for the rules in here. When we tell Webpack to use the babel loader on all the js files, instead of just specifying the babel loader here, we can specify an object. And in that object we do the loader is babel loader. However, we can pass a set of options. And in here we can override the presets to be used by Webpack when compiling JavaScript. So in here we can go back to just using React env without any configuration and stage 2, as well. So let's go ahead and test. We do yarn build-webpack. So now we're compiling babel with two configurations, one configuration for Webpack, which is what we're going to take to the browsers, and another configuration for just node.

Running Node.js in Production

One final optimization that you should do in production is to run multiple node instances per processor core. When we we just run a node command, it creates a process that uses a single CPU core. If you're running the command on a machine that has multiple CPU cores, you should start a different node process for each core. On this machine, I have four logical CPU cores, which means I need to run four different node processes. Node has a built-in module to help with that. The Cluster module. I cover details about using the Cluster module in the Pluralsight course about Advanced Node. js. The PM2 tool that we've using in this course wraps the Node. js cluster module and offers a much simpler syntax to take advantage of Node's multi core load balancing capabilities. All we need to do is use the -i argument with the number of instances we need to start. And we can use max to use all available CPU cores. We can also use the PM to reload command to do a zero downtime reload of all the running instances. So let's create a new script to run node in production. I'll copy the dev script in here, and let's call this script start-prod, and it's the exact same command, except I don't need to watch and I don't need the interpreter to be babel node, because we're running natively through node. We're also going to run this with the build directory. So instead of lib here, we're going to go with build. So consider the NODE_PATH to be build and then start build/server. js. And go ahead and use -i max to use all the CPU cores on this machine. Since we're using PM2 for production and development, we could also name the process. So let's do - - name and I'm going to name this process as appProd. And let's name the dev process as appDev. So now if I check the logs of all the running processes, I would know which process is actually running. Another thing that we should do for the node process is to run it with NODE_ENV=production, so if any script is using this NODE_ENV special environment variable, it will be set to production in that case. So we can test now. We can take a look at all the running PM2 processes with yarn PM2 status, and I see the server which was our development process before we named it, so we can delete all the processes now. So now we have a clean PM2 status. We can run our start prod script, so yarn start-prod. And you'll notice how PM2 started 4 processes for the appProd project. This is because on this machine I have four different CPU cores. And PM2 is also going to load balance between these 4 processes automatically. I don't even have to configure any ports or anything. It will do that internally using node. js cluster module. So we can test the application now and make sure everything is working, as expected. This application is now ready for production. We can push all the changes to a server, run yarn to install dependencies, and then build for webpack, build for node, and then yarn start -prod to start the server with production settings for PM2.

Summary and Course Wrap Up

In this last module, we talked about the process of building an application for production. We first talked about tests and how we should automate the run. We also saw the excellent coverage report that comes out of the box with Jest. We looked at how to split vendor files that do not change frequently into their own bundled file. We used Webpack to minify bundled files, and then we talked about Node. js in production, how to use Babel differently for Node, and the value that PM2 brings to the table. I like to be in control, configuration-wise, when working with a React project. I'm not a fan of starter templates or tools that isolate you from the configurations. Those are good for beginners and for prototyping something quick. However, even when you do your own configurations, duplicating that per project gets old fast. This is why I am experimenting with this new tool which I called reactful. I've put most of the configurations that we went over in this course in this tool. The tool has two main purposes. First, create a new full-stack React project that is server-side ready with full customizable configurations. So what you get there is ejected by default, and second, and more important really, it gives you command to assist you with the React project workflow. Now this is still a work in progress, but the tool already supports things like creating different types of React components and even working with Redux. Give it a try and let me know what you think! If you want to test something quick about React, I wrote a web playground for React, and this one does not need any configurations. It'll always have the latest React and ReactDOM, included among a few other things like axios and Immutable. js. It's also a simple JavaScript REPL. Check it out at jscomplete. com/repl. If you're planning to use React with REST APIs, you should definitely take a look at GraphQL, because I think it's a much better solution to working with Data. One other course that I recommend after this one is the Advanced Node. js at Pluralsight. You've seen how much of Node. js we had to deal with in this course. A solid understanding of the core Node. js runtime itself will make your full-stack JavaScript experience much better. Thanks for taking this course. I hope you've enjoyed it and it was a good educational experience for you. I hope you now you feel a lot more comfortable about React. js and full-stack JavaScript in general. Don't forget that you can always ask me questions about any of my Pluralsight courses using the jscomplete slack channel. Come say hi if you don't have any questions, or if you have any kind of feedback about this course, good or bad. Come chat with me and tell me what could I have done to make this course better for you, and what should my next course be about. I actively blog on Medium about React. js, Node. js, and many other topics. Keep an eye on my feed for more practical advice and tips about React. js. Thanks again, and I'll see you in the next course.

Course author

Samer Buna

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

Course info

Level	Advanced
Rating	★★★★½ (216)
My rating	★★★★★
Duration	3h 54m
Released	21 Jul 2017

Share course