# Building Applications with React and Redux

by Cory House

Start Course

Bookmark Add to Channel Download Course



# Course Overview

## Course Overview

Hi, I'm Cory House. I really love React, and if you're watching this, I'm betting that you do too. Now I assume you already know the basics of React, yet React presents a simple problem, decision fatigue. You may understand the basics of React, but can you build something big, complicated, interactive, and testable with it? React is just a library, so chances are, you're going to need a variety of other tools, libraries, and patterns to build something real. Redux is one such library. Redux has eclipsed a long list of alternatives to become the de facto flux implementation. So we're going to explore Redux from the ground up, and you'll see what it's been so widely embraced. Yet even after you've chosen well with React and Redux, you still have two hard problems to solve. Problem 1 is what libraries should I compose with these two technologies, and problem two is once I've decided, how do I connect it all together in a way that makes sense and is maintainable and testable. This course answers both of these questions in a comprehensive manner. We'll build a real-world style React and Redux application from the ground up. With one command, we'll lint our code, run our tests, transpile ES6 to ES5, bundle, minify, start up a web server, open the application in the default browser, and hot reload our changes all along the way. Trust me, once you experience React development this way, you won't want to go back.

# Intro

## Course Outline

Hi there, I'm Cory House, and welcome to Building Applications with React and Redux. React is one of the world's most popular UI libraries, and Redux has become React's most popular state management library. Along with React and Redux, we're going to use a variety of other popular tools, including React Router, Babel, Webpack, Jest, and much more. In this course, we'll use some of the most popular and powerful tools that are available for building web apps today. Before we get started, I'd love feedback. I'm active on Twitter as housecor, and I offer React consulting and training services at reactjsconsulting.com. Here's the high-level course outline. Most modules are around 20 minutes. In this module, I'll quickly discuss the intended audience for this course so that you can decide if this is the right course for you. I'll also share why Redux has become so popular. Then, we'll set up an awesome development environment. But if you're in a hurry, you can skip this module and download the finished environment. We'll move on to explore the many approaches to creating React components. Then, we'll build our initial app structure and set up React Router so that we have a useful app to work with. After setting this foundation, we'll be ready to explore Redux in detail for the rest of the course. We'll build a course management app from scratch so that we can learn Redux by example. It will support creating, editing, and deleting course data. And it will make calls to a local mock API. We'll wrap up the course by exploring automated testing in both React and Redux, as well as creating a production build. Most of the first half of the course is conceptual, and most of the last half is coding. Alright, with the outline out of the way, the next question is who's this course for? Let's address that next.

## Who Is This Course For?

Now I don't want to waste your time, so let me clear this up right now. I assume you already know React. Now if you like learning by example, you can likely get a lot out of this course by just following along, but if you're totally new to React, then check out Building Applications with React and Flux. This is my introductory course on React, which also covers Flux and React Router in later modules. These four modules will give you an intro to React in about an hour. To clarify, you don't have to know Flux to learn Redux, but Redux certainly builds on Flux's unidirectional dataflow pattern. So if you already know Flux, you'll certainly find it easier to quickly pick up Redux. I may make some comparisons between Flux and Redux in this course, but I don't expect you to know Flux. That said, if you'd like to get up to speed quickly on Flux, check out the short,

20-minute module that introduces Flux in my Building Applications with React and Flux course. You also don't have to know modern JavaScript. This course is a great way to learn modern JavaScript by example. I'll provide short explanations of modern JavaScript features that I'm using throughout this course. We'll put a number of new features to use, including modules, let and const, enhanced object literals, default parameters, template strings, classes, arrow functions, promises, destructuring, and the spread operator. So this course is a great way to learn modern JavaScript by example.

## How Is This Different from the React and Flux Course?

You might wonder how this course differs from my React and Flux course. These two courses are designed to be complementary. In short, my previous React and Flux course is for beginners and introduces React and Flux for people who are completely new to React. In this course, I assume you're already familiar with React, so I'm using more advanced tools that are more popular in the real world. Obviously, instead of Flux, we'll be using Redux. And since this course isn't geared toward beginners, I chose to use more advanced tooling. So instead of ES5, we'll be using a variety of modern JavaScript features. We'll transpile our code using Babel. This will assure that our app will run in all browsers. I'm sticking with React Router for this course, so we'll use it to handle our routing needs. Instead of Browserify, I'm using Webpack since it's more powerful and also the most popular bundler in the React community. And since Webpack is so powerful, we don't need Gulp anymore. So instead, I'll use npm scripts to create our automated build. I'm going to stick with ESLint since it remains the most popular and powerful way to lint JavaScript today. In my previous course, I didn't cover testing, but this is an important topic. So I'll use Jest, Enzyme, and React Testing Library to write tests for both React and Redux. In the Flux course, I used sublime text. In this course, I'm going to use VS Code to show off the power of a full-blown JavaScript IDE. I think you'll like what you see. Finally, we're going to build the same app in this course that we built in my React and Flux course. I'm doing this deliberately so that you can easily compare the final result of these very different technical approaches. I think you'll find it insightful to open up this app next to the React and Flux app that I built in the previous course.

## Why Redux?

So why Redux? There are countless libraries in the React ecosystem, so why has Redux become so popular? Shopping for a Flux implementation to go with React used to feel like walking into an ice cream shop. There were so many options. They all taste great in their own way and solve

some basic problems. But they do so in their own unique ways. Vanilla Flux was the first Flux implementation that was provided by Facebook. However, Redux has easily become the most popular flavor of Flux. And here's some quick evidence. Redux has far more stars than Flux, in fact over twice as many. Over 40 thousand stars for an open source project is pretty impressive. In a short time, Redux has become a mature and popular alternative to Facebook's Flux. And in 2016, Redux's creator, Dan Abramov, joined the React team at Facebook. So why has Redux become so popular? Well I believe there are seven things that make it special. First, Redux centralizes all of your application's state in a single store. Unlike Flux alternatives, Redux enforces keeping all state in a single, centralized object graph. This makes the app easier to understand and avoids the complexity of handling interactions between multiple stores. It also helps avoid storing the same data more than once. Since there's only one store, you're also less likely to accidentally store the same data in multiple places or to request data that you already have available on the client. As you'll see, setting up Redux requires less boilerplate that Facebook's Flux as well. Your container components are subscribed to the Redux store automatically, so you don't have to wire up event emitters to subscribe to the dispatcher. As you'll see, there's no dispatcher at all. Redux's architecture is friendly to server rendering your React components, commonly referred to as isomorphic or universal JavaScript. And Redux uses an immutable store, which has a number of benefits, including performance. But it also enabled interesting features, like hot reloading, which means that you can instantly see your changes in your browser without losing client-side state and time-travel debugging, which allows you to step forward and backward through state changes in your code and even replay interactions. It's quite impression. Finally, Redux is quite small. It has a small API and only weighs about 2K. Now I just said a lot of nice things about Redux, but nothing is perfect. I love this tweet by Phil Webb. It's so true. If too much is going on behind the scenes, we complain that there's too much magic going on. And if it requires too much code to get things set up, then we complain about boilerplate. So there's an obvious tradeoff at play here. Some people complain that Flux required too much tedious and redundant boilerplate for building real apps. Thankfully, Redux requires less boilerplate than Flux, and it offers more power in the process. The great news is in my eyes, Redux avoid overabstracting and becoming magic. Of course, this means you still have to write a lot of what some might consider to be boilerplate with Redux. What's important is to understand that there's no right answer on this continuum, but I do believe that it helps to understand approximately where Redux sits on this line. There are portions of Redux that you might consider magic, but keep in mind that Redux is small enough that it's totally feasible to dip into the source code if you're curious.

## Summary

In this short introductory module, we discussed why Redux is worth learning. we saw that Redux has become React's most popular data management library, and we touched on a number of good reasons why. In the next module, we'll set up our dev environment. We'll use Webpack, Babel, Jest, ESLint, and more to build a powerful React development environment from the ground up. And if you happen to already know these technologies, I'll provide you a finished environment so that you can skip ahead.

# Environment Build

## Intro

In the last module, we discussed what makes Redux so compelling. But before we start writing code, let's set up a powerful development environment so that we can move quickly and get rapid feedback along the way. When starting a new React project, you need a development environment, and today there are literally hundreds of starter kits and boilerplates for React to choose from. The site javascriptstuff.com documents around 200 starter projects for React. Back in 2015, I created my own called react-slingshot. However, shortly after I created my boilerplate, Facebook published their own called create-react-app. It's easily the most popular way to create a React app today. I highly recommend create-react-app, and I use it some of my other React courses here, on Pluralsight. However for this course, I want to show you how to create your own React development environment. Now this is totally optional, but as you'll see, it doesn't take very long. We'll build something similar to create- react-app. So why bother? Well first, this will help you understand how create-react-app works. And second, with this knowledge, you'll feel more comfortable customizing your dev environment of choice later or potentially building your own dev environment from scratch. In this module, we'll build a React development environment. Now if you're eager to jump into building with React and Redux, or if you're already familiar with the technologies that we're going to use in this module, such as Webpack, Babel, ESLint, and npm scripts, then you can safely skip passed this module by downloading the finished environment. Click on Download exercise files under the Exercise files tab. So here's what we're building. Our development environment will compile JSX, transpile our JavaScript, lint our JavaScript to enforce coding standards and find potential bugs, generate our index.html file, reload any time that we hit Save, and do all of this with a single command. As you'll see, this provides a rapid

feedback development experience. In this module, we'll create a development environment from scratch using Node, Webpack, Babel, ESLint, and npm scripts. Alright, let's get started.

## Install Node

To get started on our environment setup, install Node. We'll use Node to run our development environment. Make sure you're running at least Node version 8. And if you already have Node installed or are concerned about upgrading and breaking existing projects, you have two options. You can install Node 8 or newer now to follow along and then reinstall your old version when you're done with the course. Or you can run multiple versions of Node using nvm on Mac or nvmwindows on Windows.

## Open Initial Exercise Files

To get started, click on Exercise files at the top of the course page on Pluralsight. Then, click on Download exercise files under the Exercise tab. You'll receive a ZIP file. Double-click on the file to unzip it. Inside the ZIP, you'll find a folder for each module. Open 03-environment-setup since that's the folder for this module. Inside each module folder, you'll find a before and after folder. To follow along with me in this module, use the content in the before folder. Or if you decide to skip past this module, you can use the content in the after folder. But be sure to read the README for instructions on how to set up your environment.

## Visual Studio Code Intro

In this course, I'm going to use Visual Studio Code as my editor. Now you don't need to use VS Code to follow along with me, but it's a free download if you want to try it, and I'll show you some handy ways to customize it for React development. I enjoy VS Code because it's fast, it has a huge extension ecosystem, and it has a built=in terminal, which is handy for running our React app via npm scripts.

## Prettier Intro

This step is optional, but I'm going to use Prettier in this course. Prettier is a code formatter. With Prettier, you don't have to mess with indentation or long lines. You don't need to think about tabs versus spaces or code formatting at all. Prettier formats the code for you.

## Configure Prettier

I've copied the files from the starter folder under 03_environment_setup from the exercise files, and I've opened the directory inside of VS Code. Again, see the earlier Create Project clip for instructions on how to download these files. Throughout the course, I'm going to use the Prettier extension. Prettier is a tool that autoformats code. So when I save code, it will reformat automatically. I'd suggest that you do the same so that your code formatting matches mine. So click on Extensions and search for prettier like I just did and click Install. I don't have an Install button since I've already installed it. After you've installed it, go to your Preferences and look up your settings. Within Settings, search for formatOnSave and make sure that that is enabled. This way every time you hit Save, Prettier will autoformat your code. Once you've enabled that setting, you can prove that it works by coming into package.json. And when you change the indentation and hit Save, you should see that it automatically properly reindents your code. Alright, let's begin creating our app.

## Review Initial Project Setup

To get started, let's review what's in our starter kit. If you look at the README, there's a markdown file that contains some information about how to get started and also some information on common things to try if you happen to get stuck. If you click up here on the magnifying glass that says Open Preview to the Side, then you can view the markdown file in Preview mode, which makes it easier to read. Package.json lists all of our dependencies. And the package.json that you see may be slightly different than what I'm showing here because I will update versions over time as newer versions come out. But you can see these in our production dependencies, and then down here is a long list of dev dependencies. I'll introduce these packages as we go through the course. But if you're curious, you can go over to the README file, and I've described what each of these packages are for. The package-lock file is generated when you run npm install. And I've included a gitignore that ignores some common files, including our Build folder where we will write our production build. The src directory is where we will place our source code. The only thing I have in here to start us off with is a favicon.

## Create Initial App Structure

Now I've already run npm install. But if you haven't, then you'll want to open up the terminal and run npm install. Nice thing about the built-in terminal in VS Code is it opens automatically to the project directory. To open the terminal, you can go up to View and then select Terminal, and you'll

also see a relevant shortcut for your operating system. (Working) After you run npm install, you should see a node_modules folder over here that contains all the node modules that we just installed. Let's begin building our app by adding some code to the src directory. This is where all our source code will sit. I'm going to add our first file, which will be index.html, and I'll add a little boilerplate into here. Nothing too interesting here, except we do have a div right here. This is where we will mount our React application. And again, as you see me paste in these files, feel free to pull these files from the after folder for this module. You don't have to type all this yourself. First, we need to create our entry point for our app. This will be index.js. So let's create another file in the root of src called index.js. Inside, we're going to import React and also import render from react-dom. I'm going to declare a simple function component called Hi just to test that everything's working so far, and it will return a paragraph that contains some text. Then down below, we will render Hi and point it to document.getElementById app because that was the DOM element that we created where we want to mount our React application. So now we have the necessary source files. In the next clip, let's use Webpack to run our application locally.

## Webpack: Intro

Webpack has become the most popular bundler in the React community for good reason. It's extremely powerful and extensible. We'll use Webpack to bundle or compile JavaScript into a single minified file that works in the browser. Webpack also includes a development web server, so we'll serve our app locally during development via Webpack. Webpack is also used by create-react-app behind the scenes.

## Webpack: Core Config Settings

Webpack is configured via a JavaScript object, and typically the config file is called webpack.config .js, and it's placed in the project root. However, it's common to have a separate config for development and production. So out here in the root, I'm going to create a file called webpack.config .dev .js. In the last module, we'll create a production.webpack .config. This is going to end up being about 50 lines of code. So again, if you don't want to type all of this, you can grab the finished config file from the after folder for this module. First, I'm going to import Webpack itself, and I'm going to use the common JS syntax here since we're working in Node. With all of our React code, we'll use ES modules. But at the time of recording, Node still lacks support for ES modules, so we'll use Node's CommonJS-style syntax here. I'm also going to import path, which comes with Node, this will let us work with paths, and one plugin that we're

going to configure, which is HtmlWebpackPlugin. First, let's declare the Node environment, so we will say process.env .NODE_ENV = development, and this is important for our Babel plugin so that it knows that we're running in development mode. To configure Webpack, we export a JavaScript object. To export objects in CommonJS, we say module.exports. And so here I will declare an object that configures Webpack. I'm going to begin by setting the mode to development so that Webpack knows to run in development mode. This sets the Node environment to development and disables some production- only features. Next, we choose our target, which will be the web. We could set this to node if we were using Webpack to build an app running in Node, and that would change the way that Webpack bundles our code so that Node could work with it instead of the browser. Next, we'll set the dev tool, and I'm going to set it to cheap-module-source-map. There are a number of dev tools to consider, but this one is generally recommended for development so that we get a source map for debugging. Remember, source maps let us see our original code in the browser. Because we're going to transpile our code with Babel, source maps will let us see the original code that we wrote when we view it in the browser. Next, we'll declare our apps entry point, which will be in the src directory, the index file that we just created. And we can omit the .js on the end. This is also the default for Webpack, so we could leave it out, but I wanted to put it in just for clarity. Next, we can declare where we want Webpack to output. Now this is a little bit strange because Webpack doesn't output code in development mode. It merely puts it in memory. However, we do have to declare these paths so that it knows where it's serving from memory. So I'm going to say path.resolve here and use the __dirname variable that gives us our current directory name and then say build right here. So although it's not actually going to write a file to build, in memory, it will be serving from this directory. And the public path we will set to a slash. This setting specifies the public URL of the output directory when it's referenced in the browser. We'll set the filename for our bundle to bundle.js. Again, a physical file won't be generated for development, but Webpack requires this value so that our HTML can reference the bundle that's being served from memory.

## Webpack: Dev Server

We're going to use Webpack to serve our app in development too. So we're going to configure the dev server right here. You could choose to serve your app using any Node-based web server, such as Express, too. And in fact, I show how to use Express to serve your app in my Building a JavaScript Development Environment course. But here, we'll just serve our app via Webpack since it's simple and also works great. I'm going to set stats to minimal. This reduces the information that it writes to the command line so that we don't get a lot of noise when it's

running. I'm going to set overlay to true, and this tells it to overlay any errors that occur in the browser. I'll set historyApiFallback to true, and this means that all requests will be sent to index.html. This way we can load deep links, and they'll all be handled by React Router. I'm going to add three more lines here. DisableHostCheck, I will set to true, and I'll explain these last three in a moment. We're going to set headers equal to Access-Control-Allow-Origin to a * and https to false. These last three lines are necessary due to an open issue in Webpack when using the latest version of Chrome. Once it's resolved, we should be able to remove these.

## Webpack: Plugins

Alright, we're almost done configuring Webpack. Next, let's declare a plugin. For plugins, you specify an array. To start, we're going to configure just one plugin, which is going to be the HTML Webpack plugin that I imported at the top of the file. It accepts an object for us to configure the plugin. We're going to tell the plugin where to find our HTML template, which is in the src directory and index.html and also tell it where to find our favicon, which is in that same directory.

## Webpack: Loaders

Alright, we're down to the last property that we need to declare, which is module. We tell Webpack what files we want it to handle, and we do that by declaring an array of rules. The first rule is going to be for our JavaScript. So we will tell it how to find our JavaScript files, and this involves a little regex fun. I'm going to tell it to look for either JavaScript files or JSX files. And I'm going to tell it to ignore node_modules since we don't need it to process any files in node_modules. Finally, we can use the use property to tell Webpack what to do with these JavaScript files. We want to run Babel on these files, which we haven't configured yet. We'll do that in an upcoming clip. But to run Babel on these files, we will call Babel-loader. So this little snippet of code will run Babel on all of our JavaScript, and Webpack will bundle that up for us. Webpack understands a lot more than JavaScript. We can also have it process our CSS. So let's add a a second test in here for CSS. Again, I'll write a regex. And this time, we will use two different loaders, the style-loader and the css-loader. When I hit Save, we can see the whole file reformat because Prettier is reformatting our code automatically. So on line 40, the combination of css- loader and style-loader will allow us to import CSS just like we do JavaScript, and Webpack will bundle all of our CSS into a single file. And that's our webpack.config, about 44 lines of code. Now if you've never seen Webpack before, this probably felt intimidating. I can completely understand that. But keep in mind that we just configured most of our development

environment in less than 50 lines of code. That's pretty impressive. Now a few lines up, we told Webpack to run Babel on our JavaScript files. So in the next clip, let's configure Babel.

## Babel Intro

Babel is a popular JavaScript compiler that lets us use next-generation JavaScript today. So why do we need Babel? Well, two reasons. First, Babel will transpile modern JavaScript features down so that they run in all browsers. Yes, modern JavaScript support is quite good in browsers today, but some holes remain depending on the browsers that you need to support and the modern JavaScript features that you choose to use. And remember, we can't send JSX to the browser. JSX is invalid JavaScript. So Babel will transpile our React JSX down to JavaScript. Let me briefly touch on JavaScript's history to help clarify why Babel remains so important. This table is from Wikipedia's ECMAScript page. ECMAScript is JavaScript's official name. JavaScript was officially released in June 97, and it moved through its first 5 versions over a 12-year period. ES5 was released way back in December 2009, and it's been supported in all browsers for many years. In June 2015, the TC39, which is the committee that decides what goes into JavaScript committed to yearly releases. So ever since June 2015, we've gotten a new JavaScript release every year. The ES6 release is also known as ES2015. And since the ES2015 release, JavaScript releases have been named by year, but you may still hear people say ES7, ES8, and so on. Today, most browsers have great support for the vast majority of modern JavaScript features. That's why you see all this green here on this graph. However, there are still some obvious holes, such as ES import support. And if you have to support older browsers, like Internet Explorer, it lacks support for most ES6 and newer features. So Babel transpiles modern JavaScript down to assure that it runs in all browsers that you need to support. You tell Babel what browsers you need to support, and it transpiles your code as needed. The second reason we need Babel is to compile our JSX. We can't send it down to the browser because JSX isn't valid JavaScript. So Babel will transpile our JSX down to JavaScript. If you click on Try it out on the Babel website, you can see how Babel transpiles JSX. Note that the browser ultimately just receives calls to React.createElement.

## Configure Babel

Let's configure Babel. Babel can be configured via a .babelrc file, but I prefer configuring it via package.json. Here I have package.json open. And if you look in package.json under devDependencies, you'll see a few Babel- related packages right here. So let's scroll down to the bottom, and we'll add a new section to configure Babel. Babel will look for a section in

package.json with the key of Babel, and we can set multiple presets for Babel. We're going to use just one preset, which is babel-preset-react-app. This single preset will tell Babel to transpile our JSX, and it will also allow us to safely use a long list of modern JavaScript features, including object spread, class properties, dynamic imports, and much more. Now this is the same Babel preset that's used by create-react-app. So you can dig into their preset if you're curious about all the details on how this configures Babel. In short, just understand that this one preset means that we can use modern JavaScript and a number of experimental features too, but our code will run in all recent browsers because Babel will convert it into a form that all recent browsers can understand. Next, let's set up our first npm script to start our app.

## Start Webpack via npm Script

To automate our build process, we're going to use npm scripts. Npm scripts are a simple automation tool that's built into npm. Let's scroll to the top, and here's the section called scripts where we can declare our new npm scripts. Let's declare an npm script that will start our application. So we will call it start. This key that we declare will determine how we call it on the command line. What we want to call is webpack-dev-server, and we're going to tell it where to find our configuration, which is the webpack.config .dev .js that we created in a previous clip. Finally, I'm going to configure it to run on port 3000. Alright, let's open our command line. And again, if you can't remember how to open it, go up to View and select Terminal, and you can see the keyboard shortcut for your operating system. Since we called it start, I can say npm start. Uh oh, it fails. Looks like I made a typo in my webpack.config on the devTool. Let's go up here. Ah, yes. DevTool should be all lowercased. Now thankfully Webpack gave me a helpful error message that it didn't understand that property of devTool. Let's try again. Hey that's a good sign. Project is running on localhost 3000. And oh my word, look at the beauty! It says Hi. We have our HelloWorld. So with 50 lines of Webpack config, a single line of npm script, and a reference to one Babel preset, we have our React app running successfully.

## Debugging and Sourcemaps

We can debug our code too. If we come back over here to index.js, I can type the debugger keyword right here in my component and then come back over to the browser. If I click on Inspect, open up the dev tools and then reload the page, we can see that we hit a debugger. I'll size this up a bit for you. And the great thing is Webpack is generating a source map for us. We can see that it says source mapped from bundle.js. So we are seeing our original code that we

wrote right here in the browser so that we can debug it. So if you haven't seen this trick, debugger is a handy way to set a breakpoint. Browsers will stop any time they see it as long as you have the dev tools open. This is nice because you don't have to go hunt the line down in the browser.

## Handling EADDRINUSE Error

I do want to show one other thing to keep in mind. If I open a second window here and type npm start again, you'll see that it will fail, and it will fail because we're already running our application. The error that you will receive Is EADDRESSINUSE. Or, in fact, it says EADDRINUSE. Just keep in mind, this is telling you that you already have something running on port 3000, and probably that would be another instance of our same application. So if you happen to see this as you go through the course, go find your other terminals and hit Ctrl+C to kill the process. Next up, let's configure ESLint so that we're quickly notified when we make mistakes.

## ESLint Intro

Linting in important so that you're quickly notified of potential issues in your code. ESLint has become an extremely popular way to lint JavaScript. ESLint will alert us when we make mistakes. We'll put it to use along with some additional React extensions that will help enforce best practices while working in React.

## Configure ESLint

When we hit Save, ESLint will run. Now we could configure ESLint via a separate file. But much like Babel, I prefer configuring ESLint in package.json. ESLint will look for an ESLint config key inside package.json. I'm going to paste in our eslintConfig, and then we can talk through it. It doesn't quite fit on the screen, but very close. Again, you don't need to type this. Just grab it from the after file for this module. We begin by telling ESLint to use the recommended settings as a starting point. This enables a number of recommended linting rules automatically. Then I am adding an ESLint plugin with React's recommended settings, as well as plugins that will validate our imports when we import files. So this will help alert us if we type the path to a file that doesn't exist. Since we're using Babel, I'm setting the parser to babel-eslint to ensure that ESLint can understand our code. Under parserOptions, I specify the version of ECMAScript that we're using, which will be ES2018. I specify that we'll be using ES modules and that we're going to be using JSX since we're writing React code. Under env, we specify the environments that we're going to

be working in. Each of these environments tells ESLint to expect certain global variables. The browser, node, ES6 JavaScript, and Jest all have unique global variables. We're going to use these tools in this course. Then under rules, I override a few of the recommended defaults. I'm going to use the debugger keyword inconsole.log frequently in this course. So this will avoid annoying ESLint warnings. Unlike errors, warnings just display a message instead of blocking progress via a build error. This setting is required by eslint-plugin-react. We're telling it to detect the version of React that we're using. Otherwise, it will throw a React Version Not Specified error. This final settings declares that this is the root ESLint config for this project. This way, if you're already using ESLint and you have an eslint.config in your user folder, it won't apply to this project. So this setting will avoid my linting rules conflicting with yours. Alright, now that we have ESLint configured, we could run it via the command line. But it's easier to just tell Webpack to run it for us since ESLint doesn't have FileWatch built in. So Webpack's built-in FileWatch will end up watching our files, recompiling our code, and linting our code each type that we hit Save. So let's configure Webpack to work with ESLint. This involves a very small change down here under JavaScript. We're already referencing Babel Loader. We'll reference a second loader, which is ESLint Loader. These rules are processed from the bottom up. So this says run ESLint first, then run Babel. Now let's see if ESLint works. Open up the command line and run npm start if you're not already running the app. Then, open up index.js, and I'm going to remove the debugger since we don't need it anymore. Then up here, I'm going to add a global variable and hit Save. The moment that I hit Save, you can see that ESLint is working. It says My global is not defined. So the No Undefined rule that is part of the recommended settings found our issue. You can also see that I get some red squigglies right here in VS Code. You may not see this. If you like the built-in feedback within VS Code, then you can come over here to Extensions, and the extension that I'm running is ESLint. So if you install this extension, then you will get ESLint feedback within the code itself. Otherwise, you'll need to watch the terminal to get feedback from ESLint. Now that linting is all set up, we'll know when we make many common mistakes in our code, and we can programmatically enforce our team's coding standards. The linting errors will display immediately in our console when we hit Save. Now that we've proven that ESLint works, if I delete my global variable and hit Save, it completes successfully. We can still see the old linting error in the terminal because it doesn't clear the console. So be sure to pay attention to the last line. If it says compiled successfully, then you know there are no linting issues. Our dev environment is ready to go. There's more that we could certainly add, and in later modules, we'll add a mock API and automated testing support. For a variety of other ideas, check out Building a JavaScript Development Environment here, on Pluralsight. For now, let's wrap up this module with a summary.

## Summary

We've come a long way in this module. We now have a powerful and rapid feedback development environment that we'll utilize throughout the course. We're transpiling via Babel, bundling via Webpack, linting with ESLint, serving the app via Webpack's built-in web server, generating script and CSS references in index.html via Webpack, and automatically loading changes when we hit Save. We're tying all this goodness together with npm scripts. Now we're primed and ready for building something real. But before we put all this to use, in the next module, let's discuss the various approaches to consider when creating React components today.

# React Component Approaches

## Intro

If you're watching this course, I assume that you're already generally familiar with React. But there are a few important React- specific topics that I want to discuss before we dive into Redux. In this module, I have two core goals. First, we'll discuss the many approaches that you can consider for creating React components today. You may be surprised how many ways you can create a React component these days. And we'll wrap up by contrasting container versus presentational components. You'll see that each has a unique purpose. Understanding the difference will help us write clean, reusable components and help us assure that our application is easier to maintain and scale. Let's get started by exploring the surprising number of ways that you can create a React component.

## Four Ways to Create React Components

This may surprise you, but there's currently at least four common ways to declare React components today. There's createClass, ES classes, functions, and arrow functions. Let's review each of these approaches, and along the way, I'll clarify the styles that we'll use in this course.

## createClass Component

React.createClass was the original way to create a React component when React was first launched. You can still use this style today, but most people prefer to use the more modern approaches that I'll show in a moment. So we won't use this style in the course.

## Class Component

When working in modern JavaScript, you no longer need the createClass function to create a class because JavaScript has classes built in. Here I'm using a JavaScript class. As you can see, this style uses the extends keyword to extend React.component.

## Function Component

A third option is to declare a function component. It has a simpler syntax. React assumes that the return statement is your render function. The only argument is the props passed in. We'll create many functional components in our app.

## Arrow Function Component

Finally, you can create functions via the arrow function syntax. This allows you to use the concise arrow syntax. With concise arrows, you can omit the return keyword if the code on the right-hand side of the arrow is a single expression. And if you have multiple lines of JSX, then you can wrap the JSX in parentheses, and that makes it a single expression. That said, you don't have to use an arrow function, so feel free to continue using the function keyword if you prefer. Also, in modern JavaScript, the var keyword should generally be avoided. Instead, we should use let or const. For arrow functions, use const to assure that the component isn't accidentally reassigned.

## Functional Component Benefits

It's recommended to use functional components instead of class components when possible. So why prefer to use functional components, well there's a number of reasons. First, plain functions are generally preferable to classes since they're easier to understand, and they avoid class-related cruft, like extends and the constructor. Function components avoid the annoying and confusing quirks of JavaScript's this keyword. Doing so makes the component easier to understand. Avoiding classes also eliminates the need for binding. Given how confusing JavaScript's this keyword is to many developers, avoiding it is a big win. Functional components transpile smaller than class components. They produce less code when run through Babel. This leads to smaller

bundles and therefore a little bit better performance. Function components also have a higher signal-to-noise ratio. As I discuss in my Clean Code course, great code maximizes the signal-to-noise ratio. You can go a step further on simple components. With a single-line return statement, you can omit the return and parentheses. If you do this and also use destructuring on props, the result is nearly all signal. If you destructure your props, then all the data that you use is now specified as a simple function argument. This means that you get improved code completion support compared to class base components. And since functional components are just a function, assertions tend to be very straightforward. Given these props, I expect this component to return this markup. Function components offer improved performance as well since as of React 16, there's no instance created to wrap them. Finally, classes may be removed altogether in the future. With React Hooks, which I'll discuss more in a moment, function components can handle nearly all use cases. For all these reasons, prefer functional components.

## When to Use Class vs. Function Components

So you may be wondering when you need to use a class versus a function component. Well the answer depends on the version of React that you're using. With React versions lower than 16.8, function components lack some key features. Only class components support state, refs, and lifecycle methods, like componentWillMount, componentDidMount, and so on. But you could use function components everywhere else. However, as of React 16.8, function components are radically improved By adding a new feature called hooks. With React Hooks, you can use function components for almost everything. The useState hook handles state, useEffect handles side effects, so they effectively replace lifecycle methods. UseRef allows you to add a ref, and useMemo allows you to avoid needless rerenders for performance. Now that hooks are here, componentDidError and getSnapshotBeforeUpdate are the only two things that a function component can't do. So everywhere else, prefer function components. Regardless of your React versions, it's recommended to use function components when possible. And if you're using React 16.8 or newer, you can use function components for nearly all your components thanks to the power of hooks. That said, in this course, I'm going to use both class components and function components since many people still work in class components. We'll also use hooks later in the course.

## Container vs. Presentation Components

We just saw that there are multiple ways to create React components. There's another decision that you have to make. Are you going to create a container or a presentation component? This may be new jargon to you so let's discuss it. Container components are concerned with behavior and marshalling data and actions. So these components have little or no markup. You can think of container components as the backend for the frontend. Remember, components don't have to emit DOM. Container components are mainly concerned with passing data and actions down to child components. This means that they're typically stateful. When working in Redux, container components are typically created using Redux's Connect function at the bottom of the file. In contrast, presentation components are dumb. They're nearly all markup. Container components pass data and actions down to presentation components. Presentation components receive functions and data that they need from a container component. Container components know about Redux. They have Redux-specific code inside for dispatching actions to the store and connecting to the store via Redux's Connect function. Presentation components typically know nothing about Redux, and this is a good thing. It makes your presentation components more reusable and easier to understand. Presentation components just rely on props to display UI. They have no dependencies on the rest of the app, such as Redux actions or stores. Presentation components don't specify how the data is loaded or mutated. Finally, container components are often stateful. They contain state, but they pass down to presentation components; whereas presentation components are typically not stateful. They're just plain functions. And again, most your components in a Redux app should be presentation-style components. There's a lot of jargon around describing these two kinds of React components. We just talked about container versus presentational, but you'll also hear people say smart versus dumb, stateful versus stateless, controller view versus view. I consider all of these terms to be more or less synonymous. Now obviously you need at least one top- level container component for your application, but you might be wondering when to create other container components. Dan Abramov said when you notice that some components don't use props they receive, but merely forward them down, then it's a good time to introduce some container components. So when you find yourself having to update multiple layers of components just to pass data down to the child component that actually uses the data, that can be a good time to add some container components that connect to the Redux store.

## Summary

In this short module, we explored the various approaches for creating React components, including createClass, ES classes, functions, and arrow functions. In this course, we used both ES

class components and function components, and I'll show how to use hooks to eliminate the need for using class components in most cases. We wrapped up by discussing the uses of container and presentation components and alternative terms for this, such as smart and dumb. We saw that container components will typically connect to our Redux store and then pass that data down to our child components. In the next module, it's time to dive back into the editor and use the environment that we've already set up to start coding some React components in order to build out our application's initial structure.

# Initial App Structure

## Intro

Enough concepts. It's time to get into the code. In this module, we'll fire up the editor and create our first few React components that will form our app's foundation. We'll create our application's first pages, and we'll create a layout that's utilized on all these pages. We'll use React Router for configure our routing and set up navigation between our app's first few pages. Let's dive in.

## Create Home Page

In this course, I'm assuming you're generally familiar with React. So I'm going to paste in some of these initial React components, and then I'll explain how each works at a high level. It's a common convention to place your React components in a folder called components. So let's create a new folder under src called components. Our app is going to have multiple pages. I like to keep the components for each section of the site in a separate folder. So let's create two subfolders under components called home and about. Under home, let's create a new file, and we will call it HomePage.js, and note that I start it with a capital letter. Under about, let's also create AboutPage.js. First, let's go over to HomePage, and I'll create it. This is a functional component, and it is pulling in react-router-dom's Link component, which we will use to link over to the AboutPage that we're about to create. Notice that it's linking to about right here. You have a little bit of introductory text right here, and then I'm using a class that comes with Bootstrap called jumbotron. So this will give us a large greeting on our home page.

## Create About Page

Let's go add an AboutPage as well. The AboutPage just has a little message about the technologies that we're using. Again, this is declared as a functional component. I'm choosing to use an arrow function here. Note that since I'm using the concise arrow syntax, I don't need to use the return keyword. It's implied since there's a single expression on the right-hand side of this arrow. And one final note. Notice that I am exporting default at the bottom of both of these components. That's easy to forget. But if you forget to export, you won't be able to import it later. And I should clarify, I need to use the react-router-dom link here so that it doesn't post back to the server. We want to handle all of our routing in this application on the client. So we will use React Router to handle our routing. Of course, we haven't actually configured our routes yet, so let's configure our app's entry point to work with React Router next.

## Configure App Entry Point

Let's open up index.js so that we can begin configuring our application's entry point. First, let's import the browser router from React Router. And I will alias it as Router. Be sure to import it from react-router-dom, which is the version of React Router that works for the web. Now down here, we can wrap our application with the Router component. Pull this over into the middle. While we're in here, let's also import Bootstrap's CSS. So I will import bootstrap/dist/css/bootstrap.min .css. So this will import the minified version of Bootstrap. Remember, we configured Webpack to handle CSS as well. So Webpack will bundle this up for us and inject a reference to that bundled CSS into our index.html file. Also, instead of calling this simple Hi, let's take this out, and we will assume that we'll have an App component. So let's import App from. /component/App. We haven't create this yet, so we'll go do that in the next clip, but I'm going to reference it right down here. So you can see we've got some red squigglies here because ESLint import has noticed that this isn't a valid import. Before we create that App component, there's a few global styles that I'd like to add. Let's create index.css out here alongside index.js. I'm going to paste in a couple simple styles here that will set a max-width and also style our navigation that we're to create in an upcoming clip. Now let's reference that here in index.js. Now that we have our entry point configured, let's go create our App component.

## Create App Layout

The App component will always display because we called it here in our app's entry point, index.js. Let's go create our App component in the root of the components folder, App.js. And again, be sure to capitalize it. We'll begin by importing React. We'll also import the Route

component, which comes with react-router-dom. We use this to declare our routes. Then, let's import the components that we need, the HomePage component, as well as the AboutPage component. I'll declare our component as a functional component. Inside, we'll return a div, which will wrap our application. This div will have a class of container fluid. This is a class that comes with React Bootstrap. Inside, we'll use the Route component that we imported up above. We first need a route for our HomePage, so that will have an empty path. And for the component, we'll set that to the HomePage. Now note that I've provided the exact prop here on line 9. I provided that so that only the empty route matches. Otherwise, this route would also match any other route since / is in every route. For our second route, we'll copy the first route, remove the exact prop, set the path to about, then set the page to AboutPage. Finally, don't forget to export default at the bottom. We should now be able to load the app, although you may see an error like this about an unresolved path to module because I had it running while we were referencing that. So I'll hit Ctrl+C to kill the app and then npm start. So now the project is running again. And keep that in mind. If at any point you have an error that you believe you have resolved, such as an unresolved path, you might need to kill the app and restart it. And hey, look at that. We have a running application. And if I come over here and click on Learn more, I'm taken to the About page. Success. Of course, it would be handy if we had a header so that we could navigate between the About and the Home page. So let's create a header in the next clip.

## Create Header

Next, let's create a header component that will hold our navigation. I like to keep components that aren't tied to a specific page in a folder called common. So let's create a common folder under components. Then inside common, create a new file called Header.js. To begin, import React, and we'll also import the NavLink component from react-router-dom. Think of this component like an anchor that react-router-dom manages for us. So the NavLink component will make sure that react-router-dom handles all our links. I'm going to declare an activeStyle, which we will use to color any active links in our header. Now let's return a nav element. Inside the nav element, we'll declare a NavLink, and we'll set the path to /. So this will be a link to the HomePage. We'll set the activeStyle prop to the activeStyle we displayed up above and set the exact prop on this NavLink so that only the HomePage will match this particular NavLink. I'll set the text to Home. Now let's copy this NavLink. For the second link, I'll set the to property to about since this is for the About page. I'll remove the exact prop since it's not necessary here and set the text link to About. Then let's add a pipe character between our two NavLinks to visually separate them. Finally, don't forget to export default on the header. Now let's jump over to App.js

and put our header to use. We can import the header, and then we'll use it right above our Route component. By declaring it above our Route component, it will always render. So React Router will watch the URL and then render the relevant route based on the path properties that we've declared here on line 11 and 12. So if the URL is empty, our HomePagee will display. And if the URL contains /about, then our About page will display. Come over to the browser and great. We now have navigation, and we can see that the color of our links reflects whether it's active or not. So we can navigate between our two pages successfully. But what happens if I request a page that doesn't exist? Well, we don't see anything at all. Let's handle invalid routes next.

## Create 404 Page

If we try to load an unknown URL, we just see the header. So let's handle this better. Let's create a page that will display when an unknown route is requested. This is commonly called a 404 page. Let's call the file PageNotFound.js. I'm going to place it in the root under components. This component really couldn't get simpler. It's a one-liner. Again, I'm using the concise arrow syntax so I can omit the return keyword. And since this fits on one line, I don't even have to wrap it in parentheses. Now let's jump back into App.js and configure our 404 page. To handle 404s, I'm going to import a second component from react-router-dom, which is called switch. Switch allows us to declare that only one route should match. What we can do is wrap our route declarations in switch. And now as soon as one of these routes matches, it will stop looking for other matching routes. So switch operates a lot like a switch statement in JavaScript. Let's declare a new route, and we will set the component equal to PageNotFound. Now notice what just happened there. It added in an import for me. So VS Code is smart enough to figure that out. Notice that I don't have to declare a path here because what we're saying is if none of the routes above match, then this PageNotFound should load instead. Now when we come back over to the browser, we can see our 404 page renders. So we have about. But if I come up here and mess up the URL and hit Enter, we get our beautiful Page not found. Great. Now I'd say that we're ready to start building out our course management functionality.

## Create Course Page

Let's get busy creating functionality. The app that we're building is for administering course data on Pluralsight. So imagine that our target user is a Pluralsight employee that's managing course data. Let's first create the CoursesPage component. So we'll create a folder under components called courses. Inside courses, create a file called CoursesPage.js. I will import React. I'm going to

declare this component as a class since we're going to add some lifecycle methods and state to this component. Now that said, I could declare this as a function and use the useState and useEffect hooks, which were added in React 16.8. But here I'm going to use a class for people that are familiar with classes. And then a bit later in the course, I'm going to use React Hooks, like useState and useEffect, to create a different component, so that way you can see two different ways to handle stateful components and side effects. We need to extend React.Component. I'll declare my render function. And to start, I'm just going to put in an H2 tag. Of course, don't forget the export default. I meant to call this CoursesPage plural, so let me fix that typo. Now let's update App.js to reference our new page. Now let's add our import for CoursesPage, and I'll copy this line. We'll have a new path, but the path for this is going to be courses, and the page here is going to be Courses. So now we've declared a route for our CoursesPage. Next, let's update our header component to have a link to the CoursesPage. I'm going to copy the NavLink right above and the pipe as well. Change the path to courses, remove the exact prop, and set the text here to Courses. We should now be able to jump over to the browser and see a new Courses link and load our Courses page. And good. That works successfully. And with this structure in place, we're ready to dive into Redux. But first, let's close out this module with a summary.

### Summary

In this short module, we created the initial structure for our React and Redux application. We created our first few React components, and we set up React Router and a navigation header so that we can route between pages. Now that we have a solid app to build upon, we're ready to begin exploring Redux in the next few modules.

# Intro to Redux

### Intro

In the first few modules, we've created a solid foundation with a robust development environment and a solid application structure using React and React Router. In this module, we'll explore how Redux can manage our app data flows and what problems Redux solves. We'll begin by asking the most obvious question, do I need Redux? Like any tool, it's useful in certain contexts and not necessary for everyone. Then we'll consider Redux's three core principles. From there, I'll show how these three principles impact Redux's design. I'll contrast Flux with Redux in detail. This

section is useful for those who already know Flux. Then we'll wrap up by reviewing a simple example of the Redux flow. Alright, let's dive into Redux.

## Do I Need Redux?

I want to begin with the big question that you should ask yourself first. Do I need Redux at all? No tool is perfect for every job. I think it helps to think about complexity on a spectrum. On one end is the simplest tool that requires no setup. And on the other end of the spectrum is a complex, powerful, and scalable tool that requires significant setup. If you're building an ultra simple application, perhaps all you want to use is plain JavaScript. However, an app doesn't have to get very complex before you realize that adding plain JavaScript gets painful. As you start adding interactivity, forms, logic, and want to create reusable pieces, React becomes very worthwhile. Sure, it's more complex to learn than plain JavaScript, but that quickly pays off. The clear component model, the virtual DOM, synthetic events, and the ability to think about your app in terms of small, pure, reusable components are just a few of the great features that make React so popular and help us manage complexity in our apps. Now as data flows get more complex, you may find yourself displaying the same data in multiple places. You may have a large number of potential state changes that are hard to manage. You may find it helpful to handle state changes in a single spot for consistency, testability, and heck, your own sanity. At this point, there are two popular options to consider, React's built-in context or React with Redux. As you move to the right on this slide, capabilities increase, but so does complexity. There's no setup time required for you to start writing plain JavaScript. But to handle more complex apps, you have to accept that the initial setup takes some time. I want you to keep this in mind as we set up Redux for the first time. There's quite a few moving pieces to get right. But once we do, we'll have a very powerful and scalable foundation for handling seriously complex applications in a manner that's both testable and maintainable. It will provide us rapid feedback along the way too. I mentioned React's context in Redux can solve some similar problems. So let me step back and explain how each works and what problem they're trying to solve. Imagine this is your app's React component tree. Each circle represents a React component in your app. The top circle represents your top-level app component, which passes data down to these child components via props. What if these two components need to work with the same data. They're in very different parts of your app, so how can they work with the same data? For example, imagine these components in blue need user data. Each could make an API call to retrieve user data from a database, but that could be wasteful and redundant. And how would they communicate to assure that the user data that they're using stays in sync? There are three popular options to handle this. The first option is to

lift state to their common parent. In this case, it means moving the user data all the way up to the top-level component since that's their only common ancestor. This is annoying though because it means that you have to pass data down on props through all these other components. These components didn't need the user's data. You're just passing the data down on props to avoid storing the same data in two spots. We added the user prop to these six components merely to pass the data down to the two components that need it. This problem is commonly called prop drilling. That said, lifting state does work, and it's a good first step for small and mid-size applications. But on larger apps that need to display the same data in many spots, lifting state becomes tedious, and it leads to components with many props that exist merely to pass data down. The second way to solve this problem is to consider React's context. With React's context, you can expose global data and functions from a given React component. To access this global data, you import the context and consume it in your component. So it's not truly global data. You must explicitly import the context to consume it. For example, the top-level component could declare a UserContext.Provider. This provider would provide user data and relevant functions to any components that want to consume it. So the two components that need the user data can import that UserContext and thus access the user's information via UserContext.Consumer. Since context can also expose functions for global usage, the consuming component can also call the createUser function that's actually declared way up here in the top- level component if it's exposed via the UserContext.Provider. If you haven't tried context, it's certainly worth a look. It's build into React, it's quite elegant, and it's easy to learn. The third option to consider is Redux. With Redux, there's a centralized store. Think of the store like a local client-side database. This is a single spot where the app's global data is stored. Any component can connect to the Redux store. The Redux store can't be changed directly though. Instead, any components can dispatch an action, such as Create User. When the action is dispatched, the Redux store is updated to reflect the new data. And any connected components receive this new data from the Redux store and rerender. So does Redux sound useful to you? If so, let's move on and learn about the three principles at the foundation of Redux's design.

## When Is Redux Helpful?

The bottom line is you can build impressive applications with just React. But Redux can be useful for applications that have complex data flows. If you're writing an app that merely displays static data, then Redux isn't likely to be useful. If you need to handle interactions between two components that don't have a parent- child relationship, then Redux offers a clear and elegant solution. When you find two disparate components are manipulating the same data, Redux can

help. This scenario is often the case when your application has non-hierarchical data. Also, as your application offers an increasing number of actions, such as writes, reads, and deletes for complex data structures, Redux's structure and scalability becomes increasingly helpful. The most obvious sign that you'll want something like Redux is if you're utilizing the same data in many places. If your components need to utilize the same data, and they don't have a simple parent- child relationship, Redux can help. I like the way Pete Hunt boils all this down. You'll know when you need Redux. If you aren't sure if you need it, then you don't need it. When building a new app, don't automatically reach for Redux. Add Redux when it feels necessary. Here's my suggestion. Do these steps in order. Begin with state that's local to a single component. Just use plain React state. Then lift state to a common parent when more than one component needs the data. If you start feeling pain from lifting state, then try using context or Redux when lifting state isn't scaling well. Each item on this list is more complex, but also more scalable. So consider the tradeoffs.

## Three Core Redux Principles

Redux has three core principles. The first is that your application's state is placed in a single, immutable store. By immutable, I mean that state can't be changed directly. Having one immutable store aids debugging, supports server rendering, and it makes things like undo and redo easily possible. In Redux, the only way to change state is to emit an action, which describes a user's intent. For example, a user might click the Submit Contact Form button, and that would trigger a SUBMIT_CONTACT_FORM action. The final principle is that state changes are handled by pure functions. These functions are called reducers. That sounds complicated, but it's not. In Redux, a reducer is just a function that accepts the current state in an action, and it returns a new state. To help provide a kick start for those who are already familiar with Flux, in the next clip, I'll compare Flux with Redux.

## Flux vs. Redux

You don't have to know Flux to work with Redux. But if you happen to know Flux, that will certainly help you pick up Redux more quickly. If you're not familiar with Flux, that's just fine. Go ahead and skip this clip. Flux was the precursor to Redux. But today, Redux is much more popular than Flux. That said, if you're curious about Flux, I cover it in detail in Building Applications with React and Flux. Alright, let's begin our comparison of Flux and Redux by discussing what they have in common. Flux and Redux are two different ways that you can handle state and data flows in your React applications. Both Flux and Redux have the same unidirectional data flow

philosophy. Data flows down, and actions flow up. So the first similarity is that both Flux and Redux enforce unidirectional data flows. All data flows in one direction. They also both utilize a finite set of actions that define how state can be changed. You define action creators to generate those actions and use constants that are called action types in both as well. They both have the concept of a store that holds state, although Redux typically has a single store, while Flux typically has multiple. While these core concepts exist in both, as you're about to see, they differ in a variety of ways. Let's explore how they're different. Redux introduces a few new concepts. Reducers are functions that take the current state in an action and then return a new state. So reducers are pure functions. Containers are just React components, but their use is specific. Container components contain the necessary logic for marshalling data and actions, which they pass down to dumb child components via props. This clear separation helps keep most of your React components very simple, pure functions that receive data via props. This makes them easy to test and simple to reuse. The third new concept is immutability. The Redux store is immutable, so in an upcoming clip, we'll discuss approaches for working with immutable data in your reducers. Flux has three core concepts: actions, dispatchers, and stores. When actions are triggered, stores are notified by the dispatcher. So Flux uses a singleton dispatcher to connect actions to stores. Stores use eventEmitter to connect to the dispatcher. So in Flux, each store that wants to know about actions needs to explicitly connect itself to the dispatcher by using eventEmitter. In contrast, Redux doesn't have a dispatcher at all. Redux relies on pure functions called reducers, so it doesn't need a dispatcher. Pure functions are easy to compose, so no dispatcher is necessary. Each action is ultimately handled by one or more reducers, which update the single store. Since state is immutable in Redux, the reducer returns a new, updated copy of state, which updates the store. Let's contrast Flux and Redux further by exploring the specific ways that they differ. In Flux, stores do more than one thing. They don't just contain application state. They also contain the logic for changing state. Redux honors the single responsibility principle by separating the logic for handling state. Redux handles all state- changing logic with reducers. A reducer specifies how state should change for a given action. So a reducer is a function that accepts the current state in an action and returns an action. Flux supports having multiple stores. So in a Flux app, you may have a user store and a product store. You can have as many stores as you like. In Redux, you typically only have one store. This sounds constraining, but as you'll see, there are some significant advantages to the simplicity of having a single store. Having a single source of truth helps avoid storing the same data in multiple places and avoids the complexity of handling interactions between stores. One common struggle in Flux is how to deal with stores that interact with one another. In Flux, the stores are disconnected from each other, though Flux does at least provide a way for stores to interact in order via the waitFor

function. In contrast, Redux's single store model avoids the complexity of handling interactions between multiple stores. This conceptually simpler model also provides some unique advantages that we'll discuss further on in upcoming slides. In order to handle more complex stores with many potential actions, you can utilize multiple reducers, and you can even nest them. See in Flux, stores are flat. But in Redux, reducers can be nested via functional composition just like React components can be nested. So Redux gives you the same power of composition and nesting in your reducers as you have in your React component model. In Flux, a dispatcher sits at the center of your app. The dispatcher connects your actions to your stores. In Redux, there is no dispatcher because Redux's single store just passes actions down to the reducers that you define. It does so by calling a root reducer that you define. Reducers are pure functions. So in Redux, there's no need for Flux's eventEmitter pattern. See in Flux, you have to explicitly subscribe your React views to your stores using onChange handlers in eventEmitter. In Redux, this can be handled for you using React-Redux. React-Redux is a companion library that connects your React components to the Redux store. Finally, in Flux, you manipulate state directly. It's mutable. In Redux, state is immutable, so you need to return an updated copy of state rather than manipulating it directly. You'll see how to do this in an upcoming clip when we talk about reducers. So those are the major differences in a nutshell. Now let's explore Redux at a high level by reviewing a simple example of data flows in Redux.

## Redux Flow Overview

Let's explore Redux in more detail. Let's look at how actions, reducers, the store, and container components all interact to create unidirectional data flows in Redux. Let's consider an example to understand this flow. An action describes a user's intent. It's an object with a type property and some data. The data portion can be whatever shape you like. The only requirement is that an action has a type property. If you've worked with Flux, then this should look familiar. This concept doesn't change in Redux. Here's an action for rating a course. Imagine that you were rating my course on a scale of 1 to 5. Let's just say you rate it at a 5. Come on, be my friend. You know you want to. Okay anyway, if you did click to rate it as a 5-star course, then this is the action that would be produced. This data portion on the right can be whatever you like. You could pass multiple separate pieces of data here or just one or more separate objects. This action will ultimately be handled by a reducer. A reducer is a fancy name for a function that returns new state based on the action that you pass it. So as you can see, the reducer receives the current state in an action, and then it returns a new state. Reducers typically contain a switch statement that checks the type of action passed. This determines what new state should be returned. Once

this new state is returned from a reducer, the store is updated. React rerenders any components that are utilizing the data. Your React components are connected to the store using a Redux-related library called React-Redux. We'll write all this together in the next module.

### Summary

In this short module, we took a quick look at Redux at a high level. We discussed the various scenarios where Redux is useful, which basically summed up to if you need it, you'll know it. I covered the three core design principles of Redux, that state is immutable, actions trigger changes, and reducers return updated state. We saw how Flux and Redux are similar, but they differ in key ways, particularly around Redux's lack of a dispatcher, its immutable single store, and the store subscription approach. We wrapped up by reviewing a simple example of unidirectional flow. Now that you're generally familiar with actions, stores, and reducers, in the next module, we'll explore these four concepts in more detail.

## Actions, Stores, and Reducers

### Intro

We've discussed Redux's core concepts in the previous modules, but we don't know enough yet to start writing code. So now's the time to dive deeper into the core pieces of Redux. In this module, we'll look at actions, the Redux store. We'll discuss immutability in detail since you need to understand how to handle this concept in JavaScript before you write reducers. And we'll close out by discussing how Redux handles state updates with reducers. Alright, let's start by exploring actions further.

### Actions

In Redux, the events happening in the application are called actions. Actions are just plain objects containing a description of an event. So here's the plain object. This is the action. An action must have a type property. The rest of its shape is up to you. Here I'm passing some data under a property called rating. This could be a complex object, a simple number, a Boolean, or really any

value that's serializable. The only thing that you shouldn't try passing around in your actions are things that won't serialize to JSON like functions or promises. Actions are typically created by convenience functions that are called action creators. Here's an action creator called RATE_COURSE. Typically, the action creator has the same name as the action's type. Action creators are considered convenience functions because they're not required, but I recommend following this simple convention. By using these action creators to create your actions, the spot where you dispatch the action doesn't need to know about the action creator structure. The app that we're creating will work with course data, so it will have actions like loadCourse, createCourse, and deleteCourse. When actions are dispatched, it ultimately affects what data is in the store. So let's discuss the store next.

## Store

In Redux, you create a store by calling createStore in your app's entry point. You pass the createStore function to your reducer function. The Redux store honors the single responsibility principle because the store simply stores the data, while reducers, which we'll discuss in a moment, handle state changes. You might be concerned that there's only one store in Redux, but this is a key feature. Having a single source of truth makes the app easier to manage and understand. The Redux store API is quite simple. The store can dispatch an action, subscribe to a listener, return its current state, and replace a reducer. This last feature is useful to support hot reloading. The most interesting omission here is that there's no API for changing data in the store. Now that's a good thing. It means that the only way that you can change the store is by dispatching an action. That's why I'm showing the padlock icon on the store. You can't change it directly. The store doesn't actually handle the actions that you dispatch. As you'll see in a moment, actions are handled by reducers. To understand reducers, we need to discuss immutability first. So let's do that in the next few clips.

## What Is Immutability?

Immutability is a fundamental concept in Redux. So in case you're not familiar with working with immutable data, let's discuss what it is, why it's useful, and how to handle immutability in JavaScript. You might be wondering how you can build an application that doesn't mutate state. I mean if I can't mutate state, doesn't that mean that no data can change? Not at all. It just means that instead of changing your state object, you must return a new object that represents your application's new state. It's worth noting that some types in JavaScript are immutable already,

such as numbers, string, Boolean, undefined, and null. In other words, every time you change the value of one of these types, a new copy is created. Mutable JavaScript types are things like objects, arrays, and functions. To help understand immutability, let's consider an example. Imagine that our app state holds my name and my role. In a traditional app, if I wanted to change state, I'd assign a new value to the property that I want to change. So here I'm mutating state because I'm updating an existing object to have a new value for role. Let's contrast this approach with the immutable approach to updating state. Here you can see that I'm not mutating state. Instead, I'm returning an entirely new object. This is important because Redux depends on immutable state to improve performance. We'll get to why that's a big win in a moment. First, you might be thinking, yuck, do I have to build a new copy of state by hand every time I want to change it? Thankfully, no. That would be very impractical on an object with many properties. Let's look at some easy ways to create copies of objects in JavaScript. If you haven't dealt with immutability before, you might be wondering how to easily create a copy of an existing object. There are multiple ways to get this done using plain JavaScript, including Object.assign, the spread operator, and immutable-friendly array methods, like map, filter, and reduce. Let's briefly look at each. Object.assign creates a new object that allows us to specify existing objects as a template. The first parameter is the target, and then it accepts as many source objects as you want. Let's look at an example. Here I'm saying create a new empty object. The first parameter is the target, so we're just creating a new empty object. And then we're mixing that new object together with our existing state and also changing the role property to admin. So the result of this statement is effectively a clone of our existing state object, but with the role property changed to admin. Each new parameter that we declare on the right-hand side overrides anything on the left-hand side. And we can declare as many arguments as we want for Object.assign. One word of warning though. When using Object.assign, it's easy to forget the first parameter should be an empty object. If you leave it out, you'll end up mutating the state instead of creating a new object. So make sure you pass an empty object as the first parameter when you're trying to copy an object. Another option is the spread operator. The spread operator is three dots. Whatever you place on the right is shallow copied. Much like Object.assign, values that you specify on the right override the values on the left. So this creates a new object that is a copy of state, but with the role property set to admin. You can use spread to copy an array too. We'll use spread in our Redux reducers to update state by returning a copy of our current state with the desired changes included. And here's an important word of warning. Both Object.assign and Object.spread only create shallow copies. So if you have a nested object, you need to clone it too. Notice how the user object has a nested address object. This means that this line doesn't clone the nested address object. It remains referenced in the userCopy. To avoid this issue, you need to manually clone any nested

objects too. But to clarify, you only need to clone the nested object if you need to change a value in that nested object. In fact, cloning a nested object when you don't need to change it leads to unnecessary rerenders. Let me explain this further. You might be tempted to reach for deep merging tools like clone-deep or lodash.merge, but avoid blindly deep cloning. Here's why. First, deep cloning is expensive. It will needlessly slow your app down. Deep cloning is typically wasteful since you don't need to clone all nested objects. You only need to clone the objects that have changed. Deep cloning causes unnecessary renders since React thinks that everything has changed when, in fact, perhaps only a specific child object has changed. So be strategic. Only clone the subobjects that have changed. Object.spread and Object.assign look intimidating, but don't worry. You'll get used to them fast. That said, there are a variety of libraries out there for working with data in an immutable-friendly manner. One of the openstack popular options is immer. With immer, you can write mutative code, and immer will handle the change in an immutable manner behind the scenes. Note that it looks like I'm mutating the state property under address. But I'm changing a draft state here. Immer sees that I want to change a value in the nested address option. So it will clone the nested address object and return the existing user object with a new address object nested inside. So immer does the most efficient thing for me, and it allows me to write simple code that looks like I'm mutating data because I can mutate a draft, and it handles the necessary clones behind the scenes. So what about arrays? Well, JavaScript has a variety of array methods. But when working in Redux, there are some that you should avoid. Array methods like push and pop mutate the array, so they should be avoided. If you want to use this, you must clone the array first to avoid mutating the original array. Instead, prefer using immutable-friendly array methods like map, filter, reduce, and find. All of these methods return a new array, so they don't mutate the existing array. Remember, you can also use the spread operator to clone arrays as well. I'll show how to put some of these to use later in the course. In summary, there are many ways to handle immutability, but Object.assign, the spread operator, and immutable-friendly array functions like map, filter, and reduce are the most popular approaches. We'll use these approaches in the course. Another option to consider are libraries that make working with immutable data easier, such as immer, seamless-immutable, which creates immutable data structures in plain JavaScript, React's addons-update, which is another helper for working with immutable data, or Immutable.js. So great. We now have a clear way to easily make a new copy of an object that includes some updates. This will be useful when we need to update state in Redux in our reducers. But why are we bothering with immutability at all? Let's answer that next.

## Why Immutability?

You might be wondering why state is immutable in Redux. In Flux, you simply change state. But in Redux, it's a bit more complicated than that because state is immutable. Instead, each time you need to change your store's state, you must return an updated copy. So why make state immutable? There are three core benefits to having immutable state: clarity, performance, and what I like to call awesome sauce. Yeah, that's why I listed it last. But trust me, it's worth the wait. Let's consider each of these benefits in more detail. First, immutability means clarity. In many application architectures, I find myself wondering, hey, where did that state change? What line of code just changed that value from 0 to 1. Redux's centralized immutable store means that I no longer ask myself these silly and time-consuming questions. When state is updated, I know exactly where and how it happened. It just tell myself, it was in the reducer, stupid. Just chill out. Everything will be alright, Cory. The sky isn't falling. Sorry, I find excessive motivational talk helpful when I'm debugging. Anyway, when state changes in a Redux app, you know where it occurred. You know someone wrote some code in a reducer that returned a new copy of state. Trust me, this is luxurious because it means that you're clear about what file to open to actually see state changing. In traditional apps, many files could potentially be manipulating state. In Redux, you don't have to wonder where the state update occurred. As long as you're using Redux to handle your state, then you know that it occurred in your reducer. The second big benefit of immutability is performance. Let's consider an example to understand how immutability in Redux helps improve performance. To understand why immutable state is so useful for performance, imagine that we have a large state object with many properties. If state were immutable, Redux would have to do an expensive operation to determine if state is changed. It would have to check every single property on your state object to determine if any state had changed. But if state is immutable, suddenly this expensive operation of checking every single property is no longer necessary. Instead, Redux can simply do a reference comparison. If the old state isn't referencing the same object in memory, then we know that the state has changed. This is extremely efficient. And behind the scenes, React-Redux can use this simple reference comparison to determine when to notify React of state changes. It won't rerender the component if nothing is changed. So immutability doesn't just make your app more predictable and easier to think about. It also helps improve performance. React-Redux includes a variety of complex performance optimizations behind the scenes that rely on immutable state. So the great news is, you get these performance improvements for free. The third big benefit of immutability is what I like to call awesome sauce. Immutability helps support a truly amazing debugging experience that is unlike any other technology that I've worked in. This means that you can travel through time as you debug. You

can go back in history and see each specific state change that occurred. And as you go back in time, you can undo specific state changes to see how that changes the final state. You can even turn off individual actions that occurred so that you can see what the state would look like if a specific action in history had never happened. And finally, you can play all your interactions back with a click of a button and even select the speed at which it plays back. So now you're hopefully sold on the benefits of immutability. Let's discuss some approaches to make sure that we don't accidentally mutate state.

## Handling Immutability

Let's talk about some concrete ways to enforce immutability. JavaScript doesn't have immutable data structures built in. So you might be wondering, if stores are immutable, how do we make sure that that happens? Well there are three approaches to consider. First, the simplest way is to just educate your team and trust them. If you're on a small team, this might be sufficient. But you have to be very careful in code reviews and make sure that everybody remembers because if state is mutated in Redux, it will introduce a bug. If you want to put in a programmatic safety net, then you can install redux-immutable- state-invariant. This library displays an error when you try to mutate state anywhere in your app. We'll run this in our app so it will warn us if we accidentally mutate state. Be sure you only run this in development because it does a lot of object copying, and that can degrade performance in production. Finally, if you want to programmatically enforce immutability, you can consider some of the libraries I mentioned earlier, such as immer or Immutable.js. Immer freezes objects so that they cannot be mutated, and Immutable.js creates immutable JavaScript data structures. Seamless-immutable is another interesting option to consider. Again, I recommend Immer, but there are many more to consider listed on the Redux docs. Alright, now that we have a good understanding of immutability, we're ready to explore how state updates are handled in Redux. So next, let's talk about reducers.

## Reducers

Now that we have a good foundational understanding of immutability, we're ready to discuss how data changes are handled in Redux using reducers. To change the store, you dispatch an action that is ultimately handled by a reducer. A reducer is actually quite simple. It's a function that takes state and an action and returns new state. That's it. You can think of a reducer like a meat grinder. With a meat grinder, you put in some ingredients and turn the handle. The, uh, results, well, they come out the other side. In the same way, with reducers, you pass in some ingredients, in this

case the current state and an action, and it returns new state. So if you don't like the meat grinder metaphor, let's try one that's a little more cuddly. Reducers sound scary at first, but they're really like a fluffy bunny. They're so approachable, so simple, and so tasty. Wait, I don't eat rabbits, I swear. Never mind. Here's a reducer that's handling incrementing a counter. Reducer functions just look at the action passed and return a new copy of state. So, for example, if the action passed was INCREMENT_COUNTER, then it would increment the counter and return the new state. The reducer knew what state needed to be changed by looking at the action passed, and it updated the state accordingly. However, there's something wrong with my example. I'm mutating state right here. As we've discussed, in Redux, state is immutable. So in other words, it can't be changed. Let's update this example to return a new copy of state instead. This example does not mutate state. I'm using object.spread to create a new copy of state. Let's dissect this line. Here I'm saying create a new object by copying the existing state. I'm using the spread operator. On that new object, set the counter property to the current value plus 1. So this statement copies the existing state object, but the copy has the counter property incremented by 1. Remember that reducers must be pure functions. This means that they should produce no side effects. You know you have a pure function if calling it with the same set of arguments always returns the same value. Because reducers are supposed to be pure functions, there are three things that you should never do in a reducer. You should never mutate arguments. You should never perform side effects, like API calls or routing transitions. And you should never call non-pure functions. A reducer's return value should depend solely on the values of its parameters, and it should call no other non-pure functions, such as date.now or math.random. This way, the reducer stays pure. It simply takes the current state in an action and returns new state. So no mutations or side effects should occur. It's just a pure, predictable result. Remember, instead of mutating state, you return a copy of what was passed in, and Redux will use that to create the new store state. I mentioned earlier that you typically only have one store in Redux. So while you might think that having one store would be limiting and lead to huge monolithic stores that are hard to manage, in practice, it's not a problem. You could manage slices of your state changes through multiple reducers in Redux. When a store is created, Redux calls the reducers and uses the return values as initial state. But you might wonder, if we have multiple reducers, which one is called when an action is dispatched? Well, the answer is all of them. All reducers get called when an action is dispatched. The switch statement inside each reducer looks at the action type to determine if it has anything to do. This is why all reducers should return the untouched state as the default. This way if no case matches the action type passed, the existing state is returned. So for example, if I dispatch the DELETE_COURSE action and my app has three reducers, one for courses, one for authors, and one that handles loading status, all three of these reducers will be called. But only the

reducer that actually handles the DELETE_COURSE action type will do anything. The others will simply return the state that was passed to them. Each reducer only handles its slice of state. In fact, each reducer is only passed its slice of state, so it can only access the portion of state that it manages. So while there's only a single store for Redux, creating multiple reducers allows you to handle changes to different pieces of the store in isolation. This makes state changes easy to understand, and it avoids issues with side effects. Just remember, all the reducers together form the complete picture of what's in the store. One final note on reducers. You might wonder if there's always a one- to-one mapping between reducers and actions. Nope. The Redux docs recommend using reducer composition. This means a given action can be handled by more than one reducer. Bottom line, remember, each action can be handled by one or more reducers. And each reducer can handle one or more actions. Alright, we've discussed actions, the store, and reducers. Let's summarize what we covered in this module.

## Summary

Let's wrap up with a quick summary. In this module, we saw actions and action creators. Actions represent a user's intent, like createCourse or emptyCart. Each action must have a type property. Otherwise, its shape can be whatever you like. We saw that the Redux store has a simple API with only four functions: dispatch, subscribe, getState, and replaceReducer. We learned what immutability is and how to handle it by simply returning a new copy of state. We saw how reducers are pure functions, which means that they have no side effects. For a given input, they'll always return the same output. Most apps will have multiple reducers. This way each reducer can handle a separate slice of the store. Remember that reducers are like a meat grinder. They take state and an action, and they return an updated state. No wait, they're like a bunny. They're simple and approachable. Like the rest of Redux, they're just plain old JavaScript. Nothing fancy. It's about time that we put all this knowledge to use and start writing some code. But before we do, there's one important piece missing. How do we connect our React components to the Redux store? We'll see how to do that in the next short module.

# Connecting React to Redux

## Intro

We've explored nearly all the Redux pieces that we're going to use in this course. But the one obvious remaining question is, how do I connect my React components to Redux? The great news is Redux pulls this off in an elegant way. So in this module, we'll begin by quickly reviewing the difference between container and presentation components. Then we'll check out React-Redux, the Redux companion library that will connect our React components to the Redux store. You'll see how the Provider component wraps the application so that the Redux store is available. And you'll see how to use the connect function to connect React components to the store and specify what properties and actions you'd like to attach to your component. We'll wrap up our discussion with a summary style inspired by the great Kathy Sierra that I call a chat with Redux. This will help you understand Redux by thinking about each of the pieces as different people that are having a conversation. The rest of the course is almost exclusively writing code, so let's wrap up these final key concepts.

## Container vs. Presentational Components

In a previous module, we talked about container and presentational components, also known as smart and dumb components. To understand React-Redux, it's important to understand these two types of React components. Redux's documentation uses the terms container and presentational components, but I personally prefer to call these smart and dumb components since containers contain all the smarts that are necessary to support the dumb presentational components below. So let's contrast container and presentational components. Container components are focused on how things work. They handle data and state so that all the child components below can simply receive the data and actions they need via props. That's why they're called presentational components. They're focused on how things look. Container components are the only components in your system that are aware of Redux at all. This is a great thing because it means your child components are simply presentational components. They just receive data and actions via props and contain markup. Container components subscribe to Redux state, while presentational components read data from props. In a similar way, container components actually dispatch Redux actions, while presentational components fire off actions by invoking the callbacks that are passed down to them via props. So in this way, a presentational component isn't tied to a specific behavior. Its behavior is passed down from a container component via props. With this difference established, let's discuss how to connect our React components to Redux.

## React-Redux Introduction

So what actually connects your React components to Redux? Well, that's the React-Redux library is for. We looked at the flow of Redux on a previous slide. React-Redux handles this final piece. It connects your React container components to Redux. React-Redux is a companion library for Redux. It's a separate library because Redux isn't merely useful for React. Since Redux is simply a way to handle state, you can use Redux with other libraries. So you could use Redux with Angular, View, Ember, JQuery, or even Vanilla JavaScript. That said, Redux was initially created as an alternative to Facebook's Flux, and most people today are using Redux with React. So in this course, I'm going to assume that you're using Redux with React. So we will use the React-Redux library to connect our React components to the Redux store. React-Redux ties your React components together to Redux, and it consists of two core items, the Provider component and the connect function. The Provider component is utilized at your app's route. It wraps your entire application. This is how the Provider component attaches your app to the Redux store. Connect is a function provided by React- Redux that, not surprisingly, connects your React components to the Redux store. Let's take a look at each of these in more detail. The Provider component attaches your app to the Redux store. You declare the provider once in your app's entry point. So you use the Provider component to wrap your app's top-level component. If you're curious how this works, the Provider component uses React's context to pull this off. So the Provider component makes the store available to all child components without you having to pass the store down explicitly. You only need to use this once in your root component. Now let's discuss the second piece of React-Redux, which is the connect function. The second important piece of React-Redux is the connect function. This function wraps a component so it's connected to the Redux store. With this function, we can declare what parts of the store we'd like attached to our component as props. We declare what actions we want to expose on props as well. Let's contrast this approach with Flux. In Flux, you wire your components to the store in componentWillMount, and you have to do this manually. When your components are removed from the page, you need to remove the change listener that you set up in componentWillUnmount. Finally, you need to wire up a change handler that you referenced in componentWillMount and componentWillUnmount. Here, the change handler is getting all authors from the store any time the author store changes. This boilerplate code is required when you're working with plain Flux. But when you use Redux, this is all handled in a more terse and elegant way. Let's check it out. When you use Redux with React, you use a function called connect. This function connects your React component to the store, so the function is certainly named well. You pass connect two arguments, mapStateToProps and mapDispatchToProps. he first argument specifies what state you want to pass to your component on props, and the second argument specifies what actions you want to pass to your component on props. I want to point out some benefits to Redux's

approach over plain Flux. First, you don't need to write boilerplate code to subscribe and unsubscribe from your store. The connect function that comes with React- Redux does that for you. With Redux's style, you can clearly declare the subset of state that you want to expose to your container component. In traditional Flux, when you wire up a change handler to a store, the entire store's data is exposed. And finally the previous point is important because by declaring carefully what specific data you need, Redux can give you performance improvements behind the scenes. It makes sure that your component only renders when the specific data that you've connected changes. This helps improve performance by avoiding unnecessary rerenders. Now let's explore the two arguments that we passed to connect, mapStateToProps and mapDispatchToProps.

## mapStateToProps

The connect function accepts two parameters. Both are typically functions, and both of these parameters are optional. The first parameter is mapStateToProps. This function is useful for defining what part of the Redux store you want to expose as props on your component. When you define this function, the component will subscribe to the Redux store updates. And any time it updates, mapStateToProps will be called. This function returns an object. Each property on the object that you define will become a property on your container component. So in summary, the mapStateToProps argument determines what state is available on your container component. This is a logical place to filter or otherwise transform your state so that it's most conveniently shaped and sorted for your component's use. Let's consider a simple example. If you're building a simple app, you may have only one reducer and one container component. In that case, you'd just want to pass down all of your state. But as your app grows, you'll likely want to create multiple components to manage different pages or sections of your app, and you'll likely want to create different reducers to handle different slices of your store. This is an example of a simple mapStateToProps function that makes all of your state accessible to the component via props. With this setup, I could say this.props .appState within my component to access any state that's handled by my appState reducer. Now what if I only want to expose part of more store state in the component? Then I can specify the specific pieces of state that I want to expose via props here. Each object key will become a prop on my component. Being more specific is a win too because it helps with performance. Remember, React components rerender any time that props change. So you should only pass in props that your component needs. This way each connected component only rerenders when the props that it needs have changed. I'll show more examples as we jump back into the code in the next module. One important thing to note is every time the

component is updated, the mapStateToProps function is called. So if you're doing something expensive in there, you'll want to use a library like Reselect for memoizing. Memoization is like caching for function calls. Each time a function is called, Reselect checks whether it was just called with those same parameters. And if it was, it doesn't call the function. Instead, it just returns the memoized value. This is useful for increasing performance by avoiding unnecessary operations. So if you're doing expensive operations in your mapping, for instance filtering, a list, or making expensive calculations, then memoization can help assure that these expensive operations occur less often. With Reselect, the selectors that you write in mapStateToProps are efficient because they're only computed when one of the arguments change. So if you're doing expensive work in mapStateToProps, consider using the Reselect library. Alright, so we've discussed the first parameter we passed to connect. Now let's turn our attention to the second parameter, mapDispatchToProps.

## mapDispatchToProps

The second argument that we pass to connect is mapDispatchToProps. This argument lets us specify what actions we want to expose as props. So this is conceptually very similar to mapStateToProps. The difference is this argument determines what actions we want to expose instead of what state. MapDispatchToProps receives dispatch as its loan parameter. It returns the callback props that you want to pass down. The bindActionCreators function you see here is also part of Redux. To clarify its use, let's step back and consider the different ways of passing actions to components using Redux. Redux is lightweight and rather unopinionated. So there are many ways to handle mapping your actions to props. Remember, mapDispatchToProps is how you expose your actions to your components. There are four options. You can ignore it, wrap manually, use bindActionCreators, or return an object. Let's look at these four in more detail. Option 1 is to ignore the mapDispatchToProps function altogether. It's an optional argument when you call connect. When you omit it, then the dispatch function will be attached to your container component. This means you can call dispatch manually and pass it an action creator. However, there are a couple of downsides to this approach. First, it requires more boilerplate each time you want to fire off an action because you have to explicitly call dispatch and pass it the action that you'd like to fire. Second, this means that your child components need to reference Redux-specific concepts, like the dispatch function, as well as your action creators. If you want to keep your child components as simple as possible and avoid tying them to Redux, then this approach isn't ideal. Option 2 is to manually wrap your action creators in dispatch calls. Here I'm specifying the actions that I want to expose to my component explicitly. One by one, I wrap each action

Cceator in a dispatch call. When you're getting started, this is a nice option because manually wrapping action creators makes it clear what you're doing. But as you can see, it's rather redundant. Each of the actions that I've specified above in mapStateToProps could be accessed under this.props in the component. As a third option, you may prefer to use the bindActionCreators function that ships with Redux. With this approach, you call bindActionCreators, and it will wrap the actions passed to it in a dispatch call for you. Now the props created by these two examples will be slightly different. Notice that the prop that will be exposed to component will be called actions in this case. So accessing the bound actions with this approach would require saying this.props .actions .actionName. In summary, bindActionCreators wraps your action creators in a dispatch call for you to save you a little bit of typing. A fourth option is to declare mapDispatchToProps as an object instead of as a function. In this example, assume that loadCourses is an action creator. When you declare mapDispatchToProps as an object, Redux's connect will automatically wrap each action creator and dispatch for you. I like this approach because it's quite concise. Here are the four patterns side by side. I tend to prefer option 4 since it's the most concise, but you will likely find it useful to mix and match these different approaches. And we'll use a number of these approaches in upcoming clips.

## A Chat with Redux

Well great news. We've went over all the core players in a Redux app: actions, reducers, the store, and React container views. That's a lot of concept, so it's easy to get confused at first. I found it helpful to think about these players as people that have different roles who interact with each other. So here's an example conversation that I play through my head. React says Hey CourseAction, someone just clicked this Save Course button. The action says Thanks React! I will dispatch an action so that reducers that care can update state. The reducer says Ah, thanks Action. I see that you passed me the current state and the action to perform. I'll make a new copy of the state and return it. The Redux store say Ah, thank you for updating the state reducer. I'll make sure that all connected components are aware. React-Redux says whoa, hey, thanks for that new data Mr. Store. I'll now intelligently determine if I should tell React about this change so that it only has to bother with updating the UI when it's necessary. Finally React goes ooo, shiny new data that's been passed down via props from the store. I'll update the UI to reflect this. And that's how data flows through Redux in a unidirectional manner. We've covered a lot of ground, so let's close this module with a summary.

## Summary

In this module, we reviewed the important differences between container and presentation components. We'll only connect container-style components to Redux. Our presentation component will know nothing about Redux. They'll just receive what they need via props. We're going to use the React-Redux library to connect our components to Redux. We'll wrap our app in the Provider component and connect our container components to the Redux store using connect. We saw that mapStateToProps lets us declare what state we want to expose on our container components via props and that mapDispatchToProps lets us declare what actions we want to expose via props. We saw that there are many different ways to handle mappingDispatchToProps. And that's it. Set off the fireworks because your head is now brimming with useful knowledge. And no more slides for a while because it's time to start coding like the geeky ninja JavaScript rock star that you are. And in case you're wondering, I have no idea what this picture is about either. And no, this isn't me, I swear. I mean seriously, who gets up in the morning and says, you know, I think I'll go code in a rainstorm while dressed like a banana.

# Redux Flow

## Intro

We've spent the last few modules building a strong foundation. Now we have all the pieces in place to finally dive into the code for the rest of the course. There are very few slides from here on out. We have the knowledge that we need, so let's put it to use and start coding. Before we start setting up Redux, I want to warn you. Redux will likely feel intimidating at first. There are a lot of moving pieces to set up, which is going to make it look like adding a feature is a lot of work. There are eight steps to initially set up Redux. But after this initial setup, things will get much easier. Adding additional features requires much less work. So you'll notice that things get easier as we progress through the course. So please be patient. After we set up the first feature, you'll see that adding more features takes much less time and also has a clear predictable flow. If you're confused after we add this first feature, don't worry. I was too. We'll add a few more features, and I think you'll find that the repetition will help the flow sink in. In this module, we'll build our first feature using Redux. We'll create a simple form for adding courses. We'll define actions and action creators. We'll set up our Redux store. We'll handle state changes via a reducer and ultimately complete the loop by wiring our first container component, which will connect to our

Redux store using Redux Connect. In this module, we'll experience the full flow of working with Redux by building a single feature from the ground up.

## Create Simple Add Course Form

For our first use of Redux, let's create a new course. And to do that, we'll build our first container component using the CoursesPage component as our starting point. We're going to need a form to input a course, so let's declare some state. I'll declare a constructor, which accepts props. Within the constructor, the first thing you need to be sure to do is call super props, and then we can declare our state. Here we're going to have a course object, and that course object will just have one property, which will be title for now just to keep things simple. And I'll show a simpler way to declare state in an upcoming clip. And, of course, you could also choose to use hooks here as well, and we'll also put hooks to use a little later in the course. So lots of ways to get things done in React these days. Now let's go down to render and add a course form. So I'm going to delete the existing H2. We'll begin by declaring a form. Inside that form, we'll put our H2 with a header for courses and an H3 underneath that that says Add Course since we're adding a course. We'll declare an input for the type of text and an onChange handler that calls handleChange. We'll declare that in a moment. For the value, we'll set it to this.state .course .title. Then we'll declare an input with a type of submit and set the value to Save. This will be our Save button. Hit Save and Prettier reformats. We need to declare this change handler, so let's come up here and make that happen. Say handleChange will accept an event. First, let's create a copy of our existing course object, which we'll set to a const. Again, in modern JavaScript, we should prefer using const or let and avoid using var. What we want to do is use the spread operator, which is three dots, to copy the current course from state. And then we want to set the title to a new value. That new value is getting passed in on event.target .value. Now that we've created a copy of our course object that reflects the new title, we can call setState and pass it that new course. So let's talk through what we've done here. Up here on 15, we used object spread to copy the state, and then we overwrote the title by setting it to the target.value passed in on the event. Any values that we specify on the right- hand side override the left-hand side. And we could specify as many arguments as we like here within this object literal. Now down here where we're setting course, I can omit the right-hand side here since it matches the left-hand side. This is called the object shorthand syntax. And remember, we copied React state up here because we should treat React state as immutable. So we cloned our existing state, made a change to it, and then called setState with that new object. Technically, I should use the functional form of setState here since I'm setting state based on previous state, but I'll show how to do that in an upcoming clip.

## Binding in Classes

Let's load this in the browser and see if it works. Great, we can see our course form. But when I try to type in it, nothing happens. This tells me there's an error. If we click on Inspect, we can see Cannot read property state of undefined. The issue is that the this context is not what we expect right here. Our function is inheriting the this context of the caller, which, in this case, is the change handler. To fix this, I need to bind the this context to our instance. Now one method for binding to consider is to go down in the render and call .bind to this on line 26. So this will bind our handleChange function to our instance. Now I can come over here and type in the field as expected. This works because now the this context is getting bound to our class instance. But this approach isn't ideal because using bind within render causes a new function to be unnecessarily created on each render. So let's undo this and look at another approach. We could go up into the constructor, and we could call this.handleChange = this.handleChange .bind to this. So this is called binding in the constructor, and this approach is preferable to the first one because now the function is bound once, so it won't be reallocated on every render. If I come over here, we can see the form still works as expected. However, there is an even simpler approach. Let's remove what we have here in the constructor, and let's change our handleChange function here to be what looks like an arrow instead. This is called a class field, and at the time of this recording, this is a stage 3 feature. So this will likely be officially part of JavaScript somewhere around summer 2020. But we can safely use class fields today because Babel will transpile this so that it works cross-browser. And this works because arrow functions inherit the binding context of their enclosing scope. Basically, arrow functions don't have a this binding, so the this keyword inside references our class instance. In summary, I suggest using this approach to avoid binding issues in event handlers on class components. So we're going to use arrow functions on class fields throughout the course. We can use a similar approach up here to simplify our constructor. We can get rid of the constructor declaration and the this. on the front of state and our extra curly brace. Now we have a simpler way of declaring state. This still initializes our state when the component is instantiated, but I prefer0 this approach because it requires less code, and we also don't have to remember to call super. And as a side note, we're going to look at React Hooks a little bit later in the course. One of the reasons React Hooks are so attractive is we can write function components for everything. We no longer have to use classes. And one benefit of functions is we don't have to deal with the confusion caused by the this keyword and the binding that we're seeing here. So we'll use classes for now, but you'll see how hooks simplify a little bit later in the course.

## Handle Submit

Now that our change handler is working, let's add a submit handler to our form. I'll say onSubmit= this.handleSubmit. Now you might be tempted to attach an onClick handler down here to the Save button instead, but that's not recommended for accessibility and usability reasons. Why? Well, because users should be able to submit a form by hitting the Enter key. With this onSubmit handler up here, the Enter key will also submit our form. Let's come up here and declare our handleSubmit. And for now, let's just alert this.state .course .title. We'll jump over here to the browser, type in a course, and hit Save, and we don't get an alert. We can see that the entire page posts back, and that's why we're not seeing the alert. What we need to do is prevent the form from causing the page to reload. So our handleSubmit function will receive an event, and we can call event.preventDefault right here to keep it from posting back. So this tells the browser to prevent its default behavior. If I try to submit it though, it will fail because we'll have the same binding problem that we did before. Let me show what I mean. Come over here and hit Save. Again, it fails. And if we come down here in Inspect, we see that same error as before. So let's convert this over to an arrow as well. Now when we jump back in the browser, there we go. Great. Now the form alerts the title that we've entered. Now we're ready to start wiring up Redux.

## Create Course Action

We have a form that's set up and ready to send data. So it's time to wire up Redux. To begin setting up our Redux flow, let's create a few folders where we'll keep our Redux-related files. First, I'm going to place Redux-related files under a folder called Redux. So let's create a new folder under src called Redux. This is handy since we tend to work with multiple Redux-related files at the same time. But totally optional. Redux doesn't have any opinions about how you structure your folders. Inside this Redux folder, let's create a new folder called actions. This is where we'll store all our actions, such as saving a course, deleting a course, or loading courses. Inside this folder, let's create a new file. And we'll call it courseActions.js. This file will hold our course-related action creators. And since we're creating a course, let's call our first action creator createCourse. We will export a function, and that function will be called createCourse, and it will accept a course as an argument. Our action creator returns a plain object, and it must have a type property. We're going to set the type to CREATE_COURSE. And since this is a constant, I'm going to put it in all caps. Again, that's not required. It's just a common convention. Then for our payload, we're going to pass along the course that is passed in as an argument. Remember, this function is called an action creator because that's just what it does. It creates an action. The type property specifies the action's type. And for now, I'm just hard-coding a string here for

CREATE_COURSE. And remember, the only requirement for an action is that it has a type property. The rest of its shape can be whatever works best for you. For this action, we're going to pass our course data. Now again, I can shorten this call a bit by removing the right-hand side here because the left and right-hand side match, and that is called the object shorthand syntax. We've created our first action and action creator. So next, let's create our first reducer.

## Create Course Reducer and Root Reducer

We've created our action. Now we need a function that will handle the action, and that's where reducers come in. In Redux, you handle actions within reducers. A reducer is a function that accepts state in an action and returns a new state. So let's create our first reducer. Let's create a folder. We'll call this folder reducers. In this folder, we'll create our first reducer. We'll call it courseReducer.js. Inside, we're going to export a function, set it as the default export, and we'll call this function courseReducer. Remember, all reducers accept state and an action as their arguments. Now for this state argument, we need to decide how we want to initialize state. We'll use the default argument syntax to specify that state should be initialized to an empty array because this will end up storing an array of courses. Inside our reducer, we'll create a switch statement that looks at the action type that gets passed in. The action type that we just created was CREATE_COURSE, so let's look for that. If the action type is CREATE_COURSE, for now I'm going to do this, action.course. Now don't do this. This is the thing that might feel natural to you at first to push a new course into that array of courses. But we can't do this because this mutates state. State is immutable. Instead what we need to do is return an updated copy of state. So we can copy the existing array, which is held in state and then add the course that was passed in on action.course. Remember that whatever is returned from our reducer becomes the new state for that particular reducer. So this will update our Redux store by adding the new action passed in on action.course to our Redux store. It's also important to always declare a default on a Redux reducer. And for our default here, as usual, we will return state. This way if some other action is dispatched that this particular reducer doesn't care about, for instance maybe it's an action that's related to author data or products, some other data that's not related to this reducer, then it should just return the state untouched. And again, up here I'm using the spread operator to clone state and then also clone the course passed in. So this ends up creating a new array that contains all of the existing courses and that one additional course passed in on our action. Also for this course, we're going to keep it simple and store the courses as a simple array of objects. But for large datasets, you can consider storing by ID instead. To arrange by ID, you can declare an object instead of an array. Each key is the item's ID. This can provide faster lookups on large datasets

since you can access an element directly by ID. For example, in our current data structure, we would have to use a tool like find to find the course with an ID of 2. But with the data structure below, we could reference course ID directly. For more information on this pattern, check out Normalizing State Shape in the Redux docs. Now you might look at this switch statement and say yuck. A common concern about Redux is the existence of these switch statements in reducers. Now this doesn't bother me. But if you don't like the switch statement, keep in mind that there are alternative approaches to consider. The switch statement is the most common approach, but it's fine to use if statements, a lookup table of function, or even create a function that abstracts this away. But keep in mind that each reducer handles a specific slice of state. So even though in Redux you have one store, your reducers let you slice up the management of your store's state changes into a number of separate functions. And these functions are called reducers. We're going to have multiple reducers in our application, so now's a good time to go create our root reducer that will compose our different reducers together. I'm going to place this in the reducers folder and call it index.js. Inside, I'm going to reference combineReducers from the Redux package. So I'm using a named import to get that particular function from the Redux package. Then I'm going to import the reducer that we just created, which was called courses and is in the same directory. Now let's create a const called rootReducer, and we will call combineReducers. This will combine all of our reducers together. For now, we only have one reducer. So this isn't actually necessary, but we're setting this up now to prepare ourselves for the future. Finally, so that we can use this reducer, I'm going to export default rootReducer. So notice that we pass combineReducers an object, and each property is set to a corresponding reducer. Now since we only have one reducer, we admittedly didn't need to set this up. But we'll add a second reducer in an upcoming clip, so this will be necessary very soon. Again, since this is an object, I can omit the right-hand side since it matches the left-hand side. Also up here, since I'm exporting default for my reducer, I can name this import whatever I like. So if I go over into courseReducer, we can see that the function is called courseReducer. But with a default export, you can decide what to name it when you import it. So I am naming it courses here instead because this will impact how we reference that particular data in our Redux store. I'll show you what this naming impacts in a moment. Now that we've created our first reducer and our rootReducer, let's shift our focus to creating our Redux store.

## Create Store

Next up, let's create a Redux store. In Redux, there's a single store. So in the Redux folder, let's create a new file and call it configureStore.js. There's no need to put this in a folder since we'll

only have one of this type of file. When creating a store, it's useful to define a function that configures the store because we'll call this function at our application's entry point. This way the store will get configured when our application starts up. First, let's import createStore from Redux. This is the function we call to create a Redux store. Let's also import our rootReducer from our reducers folder. Note, since I called index.js, I don't have to say /index here. That is implied. Now let's declare our configureStore function. We will export default a function and call it configureStore. This function will take one argument, which is initialState. So we could initialize our store with some data when we configure it. Inside, there's just one line. We're going to call createStore and pass it two arguments, our rootReducer and the initialState that was passed into this function. And that's it. This is all it takes to configure our store. However, while we're in here, we can add a few optional pieces of middleware to enhance our store. Remember, Redux middleware is a way to enhance Redux with extra functionality. To work with middleware, let's add another import up here called applyMiddleware. The third parameter for createStore accepts the applyMiddleware function. And the middleware that we would like to apply is called reduxImmutableStateInvariant. So let's go import that mouthful. Thank goodness for autocomplete. Now we can pass it to the applyMiddleware function right here. When I hit Save, Prettier moves that down to a separate line for readability. I also need to add parentheses on the end of my call to reduxImmutableStateInvariant because it is a function that we need to execute. And this is very important. Do not forget these parentheses because without them, you'll get a very strange and unhelpful error. So this crazy-named middleware is helpful because it will warn us if we accidentally mutate any state in the Redux store. So you can think of this as a safety net. Let's make one final tweak. Let's configure the Redux DevTools. They're handy for interacting with our Redux store. To do so, let's go back up here and update the import for Redux to have one more reference to compose. Then, let's come down here and configureStore and declare a new variable called composeEnhancers. This is going to look pretty crazy, but we're going to call window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE. Oop, you don't want that S or it won't work. Details. And then put two underscores on the end, two pipes, and the word compose. So this will add support for Redux DevTools. I'm going to hit Save here. And yes I admit to you, that's crazy-looking. But the Redux DevTools end up declaring this very unique global variable to determine whether the dev tools are available at all. So they chose this name to avoid name collisions, and that's why it has these underscores on the front and the back as well. And with this, we can now come down to our call to apply middleware and prefix it with a composeEnhancers call because this composeEnhancers variable that we declared on line 6 gives us a function that we can call. So composeEnhancers calls apply Middleware, and reduxImmutableStateInvariant is a piece of middleware that we're using. And now we'll be able to interact with our Redux store

using the Redux DevTools in the browser. Now I know this looks intimidating, but the good news is you only have to do this once. Just set it and forget it. So we've just added reduxImmutableStateInvariant as middleware for our Redux store, and we've configured the Redux DevTools, which we'll use in an upcoming clip. Now that we've configured our store, we need to put it to use at our application's entry point. So let's take care of that in the next clip.

## Instantiate Store and Provider

We're nearly done configuring Redux. We've created our first action. We created our first reducer. We've set up our store configuration. And now we need to update our app's entry point to instantiate our store. So let's open index.js out here in the root of our src directory. First, let's add an import for configureStore. And under the import, we can instantiate the store. I'm not going to pass in initiateState to our store. But if you were creating a server-rendered app, you might choose to do so here. Now let me explain. You might be confused about the difference between passing initialState here and setting the initialState within our reducer. Now currently, our reducer sets the initial state using a default parameter in the reducer. So passing initialState here is merely for overwriting the default parameters that we specify in our reducers. So when would you end up passing initialState to the configureStore call? Well, if you're wanting to rehydrate your store using some separate state that's passed down from the server or stored in local storage, then this would be a good place to do so. We now have a configured instance of our store that's set to a constant called store. But what do we do with it? That's where our companion library comes into play. If you're using Redux with React, then you'll want to use the React- Redux library. So let's go back up and import that. I'm going to import Provider, and I like to alias it as ReduxProvider just for clarity, but that's totally optional. We'll import it from React-Redux. Provider is a higher-order component that provides your Redux store data to child components. Now let's come down here in render and call ReduxProvider, and we will pass it our store on props. (Working) And I need to get my closing tag right. Now our app will be able to access our Redux store because our entire app is being wrapped in the Provider component. So good news. This wraps up the boilerplate of our Redux configuration. Now we can end up putting our store to use by connecting our first container component to work with Redux.

## Connect Container Component

Okay, I know that was a lot of plumbing and a lot to take in. But the good news is, most of the work that we just did only has to happen once. As you'll see, now that we have this infrastructure

in place, adding additional behaviors won't take long. Now that we have the Redux infrastructure set up, the only remaining piece is to update our CoursesPage component to connect to Redux. And at the top, let's import connect from React-Redux. Then jump down to the very bottom of the component. And instead of exporting our plain CoursesPage component, we're going to decorate our component using connect. Remember, the connect function connects our components to Redux. And I typically call these components container components. The connect function takes two parameters. The first is mapStateToProps, and the second is mapDispatchToProps. Then we take the results of this and call CoursesPage. Now you're likely squinting at the oddity of having two parentheses side by side here. This is just two function calls. The connect function returns a function, and that function immediately calls our component, which is the CoursesPage. To help you understand what this is actually doing, we could create a const right here and call it connectedStateAndProps and set it to a call to connect and pass it mapStateToProps and mapDispatchToProps. And then down here on this line, we could call connectedStateAndProps and pass it the CoursesPage. But typically, people just do this on one line. So I'll undo what I just did. Now we have red squigglies under our arguments because we haven't declared those yet. Let's first declare mapStateToProps. This function determines what part of the state we expose to our component. And it receives two arguments, state and ownProps. Our store is simple right now, so it only has an array of courses. We'll just return that array of courses out of state. So we'll say state.courses. And when I hit Save, Prettier does reformat our code. When declaring mapStateToProps, be as specific as possible about the data that you expose to the component. For example, if you expose the entire Redux store, then the component will rerender when any data changes in the Redux store, and that's not good. So instead, be specific. Request exactly the data that your component needs and no more. As you can see, we're not using the second parameter, ownProps. This parameter lets us access props that are being attached to this component. That's why it's called ownProps because it's a reference to the component's own props. Since we don't need ownProps, I'm going to remove that argument for now. Alright, we've taken care of mapStateToProps. Now let's talk about mapDispatchToProps. This argument lets us decide what actions we want to expose on our component. This is an optional parameter, and so for the moment, I'm going to go ahead and leave it out. When we omit this parameter, our component gets a dispatch property injected automatically. So we can use it to dispatch our actions. Now to be able to dispatch our first action, we need to be able to fire off that CreateCourse action that we created earlier. So let's go to the top and import our actions. I will import * as courseActions from our Redux folder. Now we have access to the CreateCourse action in our component. So down here on 19 where we were alerting, instead, let's call the createCourse action in our saveCourse function. So we'll say

this.props .dispatch and then call courseActions.createCourse. And we will pass it the course from our state. We don't need our alert anymore. Now line 19 looks pretty verbose, and don't worry. I'm showing you to the ugly way first. I'll show you how to simplify this in a moment. Since we're not declaring the mapDispatchToProps function, the connect function that we're calling below automatically adds dispatch as a prop on our component. Dispatch allows us to dispatch our actions. So here, we're dispatching the createCourse action and passing it to the course. Remember, you have to dispatch an action. If we merely called courseActions.createCourse without wrapping it in dispatch, then it won't do anything. It would just be a function that would return an object. You can see here, we're getting an ESLint warning because dispatch isn't declared as a prop type for our component. Prop types help us specify the props that our component accepts, and this helps us catch errors when we're not getting the data that we expect, and it also documents each component's interface. So let's add a PropType declaration. I'll go back up here to the top and import PropTypes. Then let's go down under CoursesPage, and we can say CoursesPage.propTypes = dispatch PropTypes.func .isRequired. So now we've clarified that we expect dispatch to be passed in to the CoursesPage component, and it will be passed in because connect automatically passes dispatch in if we omit that second argument, which was mapDispatchToProps. If we scroll back up, we can see that now our linting warning has gone away. And great news. This wraps up our Redux flow. So in the next clip, let's update our CoursePage component's render function to display our data, and then we can step through the full Redux flow to see how this works.

## Step through Redux Flow and Try Redux DevTools

Our CoursesPage is now connected to the Redux store, and the list of courses is available on this.props .courses because we set that up down here in mapStateToProps. So one final detail before we jump over to the browser and try this. Let's update the render function so that it displays the list of courses from the Redux store. I'll display the list of courses right here below our form. So we'll say this.props .courses .map, and we'll map over that list of courses. Remember, map is a function built into JavaScript that returns a new array. For each of our courses, we'll put in a div, and I will set the key to the course.title and display course.title. Again, I'm using the concise arrow syntax here. I can omit the return keyword because I've wrapped our whole expression in parentheses. Again, we're getting a warning over here because now we're expecting courses on props, but we didn't declare that down on our PropTypes. So let's go down here and do that. And I should mention up here on line 36, any time we're iterating over an array, React expects us to provide a key. Keys help React keep track of each element in an array, and they're

useful for performance reasons. A key should be unique to the array. So ideally, we'd set the key to an ID. We'll do that later. But for now, the course title will do. Let's open up the terminal and see if it's cranky in any way. It looks like we're good because we have resolved the ESLint warnings that occurred earlier. Again, notice that you can see warnings in my terminal, but those are old warnings. Pay attention to the last line that says Compiled successfully. That's the latest information. So assuming this all just worked, let's jump over to the browser and give it a shot. Cross your fingers. I'm going to enter a course and hit Save. Hey, look at that. Enter a 2, enter a 3, enter a 4. Redux is working. We just implemented the entire Redux flow. Let's jump back over to our code though and set some breakpoints so that we can see the entire flow in action. First, let's go up here where we dispatch our action. Set a debugger here. Then let's go over into our Redux folder into courseActions. Set a debugger here. This would be step 2. Step 3, we'd expect the reducer to handle this action. So let's set a debugger here. And then finally, once that new course is added to our Redux store, we would expect that our mapStateToProps function will get called again because there will be new data available. So let's set a debugger right here. Now that we've set these debuggers, our browser will automatically stop on each one of these debugger statements. I'm going to hit F8 just to let it load that first time. Well in fact, I'm going to reload this, and we can see that initially we have an empty array of courses getting passed in. So we can see our Redux store. Hit F8. Now let's enter our first course, hit Save, and what we expect is we've landed first in our submit handler. So here we are just about to dispatch our action, and we can see that our course is there in state. Although I will warn you, notice I can't inspect this.state .course. This is an annoying quirk right now with the source maps getting generated. They're not 100% reliable when you use these arrow functions. So that is a downside to using these class properties is sometimes your debugger will lie to you here. But trust me, this.state .course is valid. One workaround you can do here is console.log these values out, and you will see that they are set. So let's hit F8 and see where we go next. Now we're over in the action creator as expected. So we're about to return this action. And we know that this action will get handled by our reducer. So let's hit F8 again, and there we go. We landed over in our reducer. And now we can see the action.course is set as we expected to that new course, which has the title that I just entered. Our existing state is an empty array. I can hit F8 again, and now we land at the final piece, which is mapStateToProps. We can see now that our Redux store contains a single course. So now we will return this new data, and this will cause our render function to be called again. So our course renders right down below. We can also watch the Redux flow using the Redux DevTools. I am running the Redux DevTools extension, and I am using Chrome. So go out to Chrome's extensions and search, and you can find the Redux DevTools extension. If I click on Redux, this will open our dev tools. We'll size this back just a little so we can see what's going on.

But now, if I put in course 2 and I hit Enter, it's going to hit our breakpoints again. If I come back over to Redux, we can see that we have called CREATE_COURSE twice. We can also see that our state right here has two courses inside of it. We can see this laid out as a tree or as a chart or as just raw JSON. So multiple ways to see our Redux store. Ssince our breakpoints get in the way of me toying with the Redux DevTools, let me go back here and undo my breakpoints. I think I got them all. I'm just going to do a search for debuggers. Oop, there's one more. There we go. So now let's pull up those Redux DevTools, and watch me say test, test2, test23. Now we can see these different courses, and we can even see, if I can go back through here, and I can see that at this point, at init, courses were empty. And then after calling CREATE_COURSE, I had one and called CREATE_COURSE again, and we had two. Called CREATE_COURSE again, and we had three. So we can time-travel through the history of our Redux store. We can also drag this way to go to different points and then play back through. It's a very impressive debugging experience. We now have a nearly complete view of Redux, so congratulations. But I mentioned that there are cleaner ways to handle mapDispatchToProps. So let's check that out in the next clip.

## mapDispatchToProps: Manual Mapping

When we wired up the call to dispatch the CREATE_COURSE action earlier, I mentioned that there was a cleaner way to get this done. How? Well, that's accomplished with the second function that we passed to connect, which is mapDispatchToProps. This function determines what actions are available in this component. So let's declare it. It receives dispatch. That's its sole argument. And the actions that we choose to return here will be available within our component on props. So let's return the createCourse action. Now there are multiple ways to implement mapDispatchToProps, but for now, I'm going to do the mapping manually so that you can see what I'm doing. We're going to wrap our action in a call to dispatch. And then inside here, we will call courseActions.createCourse and pass it the course that was received. Again, if I didn't wrap this in dispatch, then nothing would happen. And that's an easy mistake to make. Many people try to call action creators directly, but that just returns an object because it's not wired to anything. You have to pass your action creators to dispatch to actually fire off the action because dispatch is the function that notifies Redux about an action. Now that we have this mapping set up, we've declared up above that our component will receive createCourse as a prop. So it will be wrapped in a call to dispatch for us. So let's update that now. So now we will say this.props .createCourse. That's much cleaner. We no longer have to wrap it in a call to dispatch because it was handled on mapDispatchToProps. However, we do have a green squiggly here because we need to jump down and update our PropTypes. We're no longer passing in dispatch. Now we are passing in

createCourse. Remember, when we pass mapDispatchToProps to connect, the dispatch function is no longer injected into our container components. are available. Let's jump over to the browser and test it. Great. It still works. In the next clip, let's look at some other ways to handle mapDispatchToProps.

## mapDispatchToProps: bindActionCreators

Using mapDispatchToProps helps simplify dispatching our action within our component. But our call down here in mapDispatchToProps is still quite verbose. Now Redux comes with a helper function to save us from having manually wrap our action creators in a dispatch call, and this function is called bindActionCreators. So let's come up here to the top and import bindActionCreators. Then we can put this to use down in our mapDispatchToProps function. (Working) That's a little simpler. Now note that bindActionCreators will accept a function or an object, so you can pass it all of your actions like I'm doing right here. And when you do that, it will return them all wrapped. Or you could just pass it one action in order to wrap it. BindActionCreators returns an object mimicking the original object, but with each function wrapped in a call to dispatch. So this one line ends up wrapping all of my course actions in a call to dispatch. Of course, we only have one course action. But as we add more, I wouldn't have to change this line. But now what we're passing in is all of the actions in the courseActions file. There's only one now, but there'll be more in the future. Now because of that, we should really call this actions instead because we're going to have a property for each one of the actions found in courseActions. And that means that up here, we're no longer passing createCourse in. We're now passing actions in, and actions is an object. So createCourse would just be one of the properties on that object. This means that up here, we would say this.props .actions .createCourse. Let's test again. Very good. It still works. We've looked at three ways to handle mapDispatchToProps. In the next clip, let's look at the fourth option, which is my favorite approach.

## mapDispatchToProps: Object Form

Let's look at the fourth option, which is to declare mapDispatchToProps as an object instead of as a function. Instead of declaring this as a function, we can declare mapDispatchToProps as an object. Inside this object, we can declare properties for each action that we want to expose. We can take all this out and say createCourse is equal to courseActions.createCourse. So this becomes quite concise. When we declare mapDispatchToProps as an object, then each property

is expected to be an actionCreator function, and that's exactly what courseActions.createCourse is. What happens is connect will automatically go through and bind each of these functions in a call to dispatch for you. Now I'm going to undo what I did here and stick with the bindActionCreators approach since I like how we won't have to change our mapDispatchToProps later as we add more actions. I also like how all our Redux actions are under a single actionsProp. We only have a single action now, but we'll soon add more. So bindActionCreators will save us a little bit of maintenance. Finally, I want to clarify that there's nothing magical about these functions or their names. These are merely arguments that you pass to connect. Now I prefer to define these functions separately for consistency and readability, but even the Redux example apps often declare these inline when they're simple. And you could also even consider using destructuring here in mapStateToProps to shorten your declarations further. Now I could omit the right-hand side. But to avoid more confusion, I'm going to undo that. We've now seen four different ways to dispatch actions in our container components. We're almost done with this module, but we still have one tweak to make. Instead of hard-coding strings in our actionTypes, it's a good idea to use constants to help us avoid typos. So in the next clip, let's make that tweak.

## Action Type Constants

Redux errors on the side of being unopinionated and explicit. So as you saw in this module, that means there's some boilerplate we have to write to get started. There are many options for reducing the boilerplate if you're interested. Check out the Redux docs after you get comfortable. There's a section on reducing boilerplate that discusses alternative approaches. Once you're comfortable with the patterns in this course, consider looking into these resources. Let's go over into courseActions.js. Now you might've cringed earlier when I used a hard-coded string for actionTypes. And hey if you didn't, maybe you should have. Magic strings are just typos waiting to happen. So to avoid typos, it's recommended to use constants instead. For this course, I'm going to create a separate constants file, and people often place action type constants in a separate folder called constants. But I find it preferable to place action type constants in the actions folder. So that's what we'll do in here. Let's create a new file and call it actionTypes.js. This file will hold the actionTypes used in our system. And since we're creating a course, the first actionType is going to be export const CREATE_COURSE = a string with the same value. And now that we have a constant, let's update our courseActions to use the constant. I will import it at the top. And then instead of calling this string, we can say types.createCourse. Now notice that I get autocompletion support right here. So now I can't make a typo. We can update our reducer as well. So let's open up our courseReducer. I'm going to copy this import. Put it in our reducer.

Update our path here. And right down here, instead of referencing the string, we'll now say types.CREATE_COURSE. Although I'm not getting autocomplete support because I didn't quite get that path right. There we go. Now it should help me out, and there it is. Let's make sure we didn't break anything. Come back over, hit test. Great, it still works.

### Summary

Way to go Programming Adventurer. If we were pair programming in person right now, I'd give you a high five because you're over the biggest hurdle. We just created a complete Redux flow using actions with actionType constants, a Redux store. We created our first reducer and a container components that's connected to our Redux store. There's a lot more details to explore in the next modules, but you've now seen the fundamentals and implemented it yourself. Bravo. However, there's a significant common use case that we've ignored so far. How do we handle making asynchronous requests, like API calls to a server? In the next module, we'll explore that by loading up existing course data via an API call on page load.

# Async in Redux

### Intro

So far, we've only created synchronous actions. What happens when we want to do async activities, like make API calls to the server? Here's the plan for this module. We'll begin by discussing the merits of creating and using a mock API throughout development. And we'll quickly set up a mock API by pulling in some files from the course exercises. Before we start calling web APIs with Redux, it's useful to understand Redux middleware. So I'll introduce the concept of Redux middleware. We won't need to write our own middleware though because there's a variety of mature Redux middleware libraries that are available for handling async flows in Redux. And once this foundation is set, we'll be back in the code for nearly the rest of the course as we implement asynchronous flows using Redux Thunk. Alright, let's get started by discussing the merits of mock APIs.

### Why a Mock API?

Instead of hitting a real API, we'll host a local mock API that simulates making asynchronous calls to a server. I did this for convenience here so that we don't all end up hitting the same API and wiping out each other's data. But I actually utilize and recommend this pattern most of the time when I'm building any client-side app, and here's why. This patterns allows you to start development immediately, even if the APIs that you need to consume haven't been created yet. As long as you can agree with the API team on the shape of the data that your final APIs will return, then you can create a mock API and begin your development immediately. A mock API helps me move independently when a separate team is handling web APIs. We don't have to move at the same pace. I'm not directly reliant on other developers delivering code in order to build the UI. If I'm also building the APIs, I get to decide when to do so. It's no longer a blocking issue for building the UI. That makes life much easier for both teams. It's effectively the rule of coding to an interface rather than an implementation. A mock API gives me an easy backup plan if the API is down or broken at any given time. I don't have to stop development. I can just point to the mock API and keep working. Hitting mock data is the fastest way to handle development because you can count on all responses being instantaneous. This means you're not hampered by slow or unreliable API calls in the early stages of development. Now you might be thinking yeah, but that could also mask performance concerns. Of course, you'll test against the actual API before deploying. But you don't have to wait until the real APIs are complete before testing how the app feels with slow APIs because unlike a real API call, with a mock API, you can control the speed of responses. You can simulate slow API call responses using tools like setTimeOut in your mock API. A mock API also gives me a handy tool for automated testing. Since the data is local, it's both fast and reliable. You don't have to mock calls since your mock API is already a mock. And since the data is deterministic, you can even write tests that utilize the data, and they won't be slow tests since they're local. Finally, you can easily point to the real API later by changing the import at the top of your thunks or sagas. Or you could even check a centralized config or environment variable that allows you to toggle between mock and real APIs via a single setting. For all these reasons, I typically create a mock API for my projects. We're going to use json-server as our mock API.

## Mock API Setup

Let's jump back in the code. It's time to set up our mock API. Before we start handling asynchronous calls with Redux, let's set up a local web API. To get started, we need to copy some files from the course exercise files for this module. Look in the before folder for this module. Inside the before folder, you should find a folder called tools. I've copied the three files in that

folder over into my application. Let's peek at what's inside. ApiServer will serve an API that we use in this course. This file uses Express with json-server to host a mock API and simulate a database using a JSON file. Json-server provides a command line interface, but I'm using the more advanced module config to add a custom server-side validation. If you're curious about this works, I provided comments throughout. Also, check out the json-server docs. MockData.js is a file that contains mock data. This data will populate our mock database. We're going to use this mock data for automated tests too. The mock data contains an array of Pluralsight courses, an array of authors, and a data structure for a new course. Finally, createMockDb will read our mock data and write it to a separate file called db.json. The db.json file will be read and manipulated by our API server. To start the mock API, let's add a script to package.json. Right above start:api, we'll call it prestart:api. Because we have prefixed it with the word pre, it will run before the start:api script. What we want to do is call createMockDb.js. So by convention, this script will run before the start:api script because it has the same name, but it's prefixed with the word pre. Let's now run the start:api script and see if it works. Now in this case, you need to put the run keyword in because to run most commands with npm, you need to provide the run keyword after npm. Npm start is one of those rare shortcuts where we can omit the run keyword. So we'll say npm run start:api. We can see that the mock DB was generated. We can see this new db.json file right here. And it contains our JSON. If I hit Save in here, Prettier will reformat it for me, which will make it easier to see. So this is basically a copy of that mockData.js. And you might wonder why we're making a copy. The important thing is having mock data static and consistent is going to be useful for us because we'll reference mockData.js within our tests, and also we may at some point make a mistake and corrupt this file. But since it's regenerated each time we start, we don't have to worry about this affecting our development. So json-server is now running on port 3001. Let's jump over to the browser and try it out. If I load localhost 3001, I can see both courses and authors are resources here. So if I go to /courses, here's our course data getting served up. And if I go to /authors, here's the author data getting served up. And the great thing about json-server is we can make POST, PUT, and DELETE requests to our API, and it will simulate create, updates, and deletes by updating that db.json file. So we'll be able to watch this file. And as we manipulate data, we'll see this file updated so that it can simulate rights to a database. Through the rest of the course, our app is going to call this mock API. So we need to start this mock API every time that we start our app. To do that, let's rename the current start script to start:dev. And above it, we'll create a new script called start. In here, I'm going to call run-p. This is a command that comes with npm run all, which is already installed since it's one of the packages listed in package.json. The run-p command allows us to provide a list of npm scripts that we want to run in parallel. That's what the p stands for. So we want to run the start:dev script, as well as the

start:api script. So this will run both of those scripts at the same time and send their output to the command line. Let's come down here, hit Ctrl+C, and say npm start to see if this works. We can now see that our application starts as it did before. But we can also see that json-server is running on port 3001, and our mock DB was created. So if we go back to the browser, we should still be able to load up data on our API on 3001. And if we go to 3000, we should be able to load our application. Excellent. Now our mock API is all set up. To call the API, it's helpful to have some JavaScript functions that will call the API for us. Open up the course exercise files for this module and copy the API folder into your solution under the src directory. I've already copied this folder into my project. That's why you see these three new files here. In JavaScript apps, I like to centralize all the API calls that my application makes, and that's what these files are for. CourseApi contains functions that will call getCourses, saveCourses, and deleteCourses. AuthorApi contains a function for calling getAuthors, and apiUtilis centralizes the handling of our API responses. Note that I'm using fetch to make API calls. Fetch is built in to modern browsers, so we can make API calls without installing an extra library. Fetch is a promise-based API. With promises, the then function is called when the asynchronous call is complete. And if an error occurs, the catch function is called. In courseApi, we can see that I'm specifying different HTTP verbs for each call. Fetch defaults to GET, so I don't have to specify an HTTP verb here. If we're saving changes to an existing course, that's a PUT. And if we're adding a new course, that's a POST. Finally, to delete a course, we pass the DELETE verb. These are common standards for RESTful APIs, and json-server honors these conventions. Finally, notice how I'm referencing an environment variable for the base URL. To configure this environment variable, let's use Webpack. Open up the webpack.config. First, we need to import Webpack here at the top because we're going to call a function that is available within the Webpack package. Now let's scroll down to line 25 where we begin declaring our plugins. And I'm going to call that new plugin, which will be webpack.DefinePlugin. And we pass the DefinePlugin a setting for our environment variable. We're going to call process.env .API_URL and set that to the results of JSON.stringify and put our base URL for our API in here. Be sure to put a comma after that declaration. With this setting, now anywhere that Webpack sees process.env .API_URL in our code, it will replace it with our API_URL. And note that we need to call JSON.stringify on our result since Webpack requires this format for values. Now that we have our API set up, let's discuss how to handle async calls in Redux using middleware.

## Middleware and Async Library Options

One of Redux's most powerful features is the ability to configure middleware that runs on every request. There's a variety of existing middlewares to choose from, or you can create your own. Redux middleware runs in between dispatching an action and the moment that it reaches the reducer. If you've ever used popular libraries like Express, then you're already familiar with the idea of middleware. Middleware is a handy way to enhance Redux's behavior. Redux middleware can handle many cross- cutting concerns, including handling asynchronous API calls, logging, reporting crashes, and routing. Here's an example of custom Redux middleware that logs all actions and states after they're dispatched. The signature for Redux middleware might look odd to you because it chains multiple function calls together using currying. But don't worry. You don't need to understand this to work with Redux. I just wanted you to be aware that you can write your own middleware to enhance Redux's behavior. The good news is the Redux community has already written a lot of useful middleware, so chances are you don't need to write your own. Instead of creating our own middleware, we're going to use some existing popular middleware to handle async API calls. In Redux, actions are synchronous and must return an object. So the question is, how do we handle async calls? Well, there are multiple libraries for handling async and Redux. Let's review the major players. Four popular libraries for handling async are redux-thunk, redux-promise, redux-observable, and redux-saga. Yes, some weird names. Redux-thunk is quite popular and was written by Dan Abramov who's also the creator of Redux. It allows you to return functions from your action creators instead of objects. Redux-promise allows you to use promises for async, and it uses Flux standard actions to bring some clear conventions to async calls. Redux-observable allows you to dispatch observables. So if you're already familiar with RxJS, then you may be interested in this approach. Redux-saga takes very different approach. It uses ES6 generators and offers an impressive amount of power with what's basically a rich domain-specific language for dealing with asynchrony. Redux-saga is impressive and certainly worth looking into. But it covers so much ground, that it really warrants its own course. Let's contrast the two most popular options, thunks and sagas. With redux-thunk, you actions can return functions instead of objects. A thunk wraps an asynchronous operation in a function. With sagas, you handle async operations via generators instead. Generators are functions that can be paused and resumed later. A generator can contain multiple yield statements. At each yield, the generator will pause. They're a powerful tool. As you'll see in the next module, thunks are a bit clunky to test because you have to mock API calls, and you have no easy hooks for observing and testing individual steps in the async flow. Sagas are easier to test because you can assert on effects, which simply return data. You don't have to mock anything, and your tests generally read more clearly. The benefit of thunks is they're conceptually simple. And much like Redux, the API surface area is very small. This makes learning thunks quite easy.

Sagas, on the other hand, are hard to learn because you have to understand generators and a large API. Once you do, there are many ways to introduce subtle bugs in your code if you don't fully understand the implications and the interactions of the effects that you're composing. That said, once you know sagas and see the elegance that generators afford, you may prefer them over thunks. The decision between these two libraries isn't easy, but I suggest starting initially with thunks since they cover most use cases well and are definitely easier to learn. We're going to use redux-thunk for our app, so let's take a look at an example of a thunk.

## Thunk Introduction

Normally, we can only return actions from our action creators. With redux-thunk though, we can return a function instead. Here's an example of a thunk for deleting an author. A thunk is a function that returns a function. Thunk is a computer science term. A thunk is a function that wraps an expression in order to delay its evaluation. So in this case, the deleteAuthor function is wrapping our dispatch function so that dispatch can run later. Depending on your programming background, returning functions from functions likely feels a little bit weird. But it's a common and powerful technique in functional programming. On the third line, I'm calling a regular action creator called deleteAuthor. But note that you don't have to call a separate action creator function. You could inline the action within the thunk if you prefer. Thunks enable us to avoid directly causing side effects in our actions, action creators, or components. Instead, anything impure is wrapped in a thunk. Later, that thunk will be invoked by middleware to actually cause the effect. By transferring our side effects to running at a single point in the Redux loop, in this case at the middleware level, the rest of our app stays relatively pure. Redux-thunk has access to the store, so it can pass in the store's dispatch and get state when invoking the thunk. The middleware itself is responsible for injecting those dependencies into the thunk. Thunks are passed dispatch automatically. They also receive getState as a second argument. The getState argument on thunks is useful for checking cache data before you make a request or for checking whether you're authenticated, in other words doing a conditional dispatch. Now redux-thunk is only 14 lines of code. On line 2, you can see the common Redux middleware signature that's required for all Redux middleware. If redux-thunk middleware is enabled, any time you attempt to dispatch a function instead of an action object, the middleware will call this function with dispatch itself as the first argument and getState as the second argument. You can optionally configure redux-thunk to inject a third argument of your choice as well. A common example would be injecting an HTTP library, like Axios, for making API calls. I want to clarify that middleware isn't actually required to support asynchrony in Redux. It just makes handling async

more elegant because you don't have to pass dispatch to each call. Without middleware, your components have to know which action creators are async since they must pass dispatch down to the action creator. So middleware isn't required to support async in Redux, but you probably want it. For example, without redux-thunk, you could handle an async action like this. Note that I have to pass dispatch in as an argument. With thunks, we don't have to pass dispatch or state as arguments. They're injected for us by the thunk middleware. This may sound minor, but it's important because it means that our components can call async actions the same way that they call synchronous actions. So the benefit of using middleware, like redux-thunk, is that components aren't aware of how action creators are implemented and whether they care about Redux state or whether their synchronous or asynchronous. Thunks keep our app code simple and consistent. In summary, here's the three reasons to use async middleware. It means that all your action calls are consistent. Without middleware, the signature of our dispatch calls would differ depending on whether they were synchronous or asynchronous. Async middleware like redux-thunk also helps keep our code pure. It avoids binding our code to side effects, and this makes testing our React components easier as well. So you don't have to use async middleware, but the vast majority of people do. We're going to use redux-thunk in this course.

## Remove Inline Manage Course Form

In the previous module, we set up our first data flow via Redux. We created a simple course that consisted of nothing, but a title. In this module, we'll begin by restructuring our work to better separate concerns. There are some problems with our current design. First, CoursesPage should be a container component. It's connected to Redux and should ideally not have much JSX inside. It's also both currently displaying courses and adding courses. For clarity, let's handle editing and adding courses on a separate page. So we can create a separate page called ManageCourses to handle the management of courses. To begin, let's remove the state, and we'll remove the change handler and the handleSubmit function. Then down here within render, we no longer need the form tag or the header for Add Course or these inputs for submit or text. So we'll remove this closing form tag as well. Now we do need to add in a fragment to wrap our render since otherwise we would have two top-level elements. We're going to solve these problems in a more elegant way on the ManageCourse page, which we'll create a little bit later. By the end of this module, we'll have an app that allows us to view, create, and update courses, all using Redux. For now though, we should still be able to run the app. If we come over here to courses, we just see an empty header. So let's turn our attention first to displaying a list of courses on initial load. To

do this, we'll call our new API to fetch a list of existing courses when the page loads, and we'll load the courses by dispatching an action with Redux. Let's do that next.

## Add First Thunk

Alright, enough talk. Let's get thunky. Sorry, I couldn't resist. The first step to using thunks is to enhance our store configuration. So open configureStore.js, which is in the stores folder. First, let's import thunk from redux-thunk. To add our thunk middleware, we'll come down here to line 13 where we call applyMiddleware, and we will add thunk as an argument to the applyMiddleware function call. Here we could add as many pieces of middleware as we like. Now that we've updated the store to utilize thunk middleware, we're all set to create our first thunk. We just added redux-thunk to our store configuration, so it's now part of our Redux middleware. So we're ready to create our first thunk to handle making an asynchronous call. For our first thunk, let's load courses when the app initially loads. So open up courseActions.js. To get started, let's add a reference to the courseApi since we're going to call it in our thunk. (Working) I like to put my thunks at the bottom of the actions file. So let's create a new function called loadCourses, and be sure to export that function so that we can call it. So this will be our first thunk. Every thunk returns a function that accepts dispatch as an argument. And yes I know it likely feels weird seeing a function that's returning a function. It's important that the function you return has this exact signature though because the function that we declare here is utilized by the redux-thunk middleware. And remember, thunk middleware passes dispatch as an argument to our thunk. That's how this inner function is going to get dispatch as an argument. This is the core benefit of redux-thunk. We don't have to pass dispatch in ourselves. So this way, our calling code looks the same for both synchronous and asynchronous calls. Now let's add code inside our thunk to load courses from the API. We're going to return courseApi.getCourses. And since it returns a promise, we will use .then to handle the response. The response is going to return a list of courses. When we receive that list of courses, what we'd like to do is dispatch an action. The action that we're going to dispatch is going to be loadCoursesSuccess, which we need to declare up above. We'll do that next. But we will pass it the courses that we receive. Since we're making a call to a promise, it's a good idea to declare a catch as well so that we could catch any errors that occur. For now, I'm just going to throw the error, but I'm putting the catch here to remind you that it's a good idea to declare it. I could choose to handle the error here by dispatching another action that lets the application know that the request failed. But for now, we'll just throw the error. And as you can see, I'm calling the API functions that we set up earlier rather than making an API call directly here. Again, I prefer to keep API calls separate in order to keep my thunks simple and to

centralize the way that I handle my API calls. Let's finish our work on this thunk. I haven't created the loadCoursesSuccess action yet, so let's do that next. First, let's open up actionTypes.js and add a new constant, which we will call LOAD_COURSES_SUCCESS. (Working) Then, let's declare our action creator, and we will say export function loadCoursesSuccess, which will accept courses. Inside it will return an action. That action will have a type equal to types.LOAD_COURSES_SUCCESS, and the payload for it will be courses. I'll hit Save, and Prettier reformats for me. Note that I'm omitting the right-hand side here. I could say courses equals courses, but that's un necessary because I can use the object shorthand syntax. To clarify though, you could declare this action inline if you prefer. In other words, I could take this action and put it right down here, dispatch like this. If you prefer that approach, that's totally fine. I tend to prefer creating the separate function just for clarity. If it's only used once, I can completely understand the logic of putting it inline here though. So it's really just a stylistic choice. This is a good point to discuss action naming conventions. Here I'm using the suffix success for two reasons. First, we already have a function called loadCourses, which is our thunk. Second, this action doesn't fire until authors have been successfully returned by our API call. So the suffix helps clarify that our async request was successful. Third, people often create a corresponding failure action type called LOAD_COURSES_FAILURE or LOAD_COURSES_ERROR. To help save us time and typing, I'm not going to create corresponding error actions for each thunk. But you might want to do so in a real app when you need to treat the failures of different async calls uniquely. Instead, I'm going to show you an approach for handling async calls in a centralized way that I find quite elegant. We'll do that in a moment.

## Handle Loading Courses in Reducer

Now we need to handle the loadCoursesSuccess action that we just created. We'll handle that in our courseReducer. So open up courseReducer, and we'll add another case statement. This time we'll be looking for LOAD_COURSES_SUCCESS, and we will return action.courses. Since whatever is returned from our API will simply replace what was in our state, all we have to do is return the courses here. Simple as it gets. Remember, whatever you return from your reducer becomes the new state. Now we have our actions set up and a corresponding reducer that handles the loadCoursesSuccess action. But the question is, where and how do we fire this off on load? Let's set that up next.

## Dispatch Actions on Load

To fetch course data when our app loads, we have a couple of options. First, we could load course data the moment that our app starts up, but that could be wasteful if users never end up navigating to the CoursesPage. Or instead, we could load course data when the course page mounts. So let's go with option 2 since that's more efficient. Down here in mapDispatchToProps, we're already injecting any actions from courseActions into our component under a prop that's called actions. So let's call loadCourses when our component mounts. To do so, declare a componentDidMount function inside our component. We will call this.props .actions .loadCourses. Remember that this returns a promise, so we will call .catch to catch any errors that might occur. If an error does occur, then we will just alert it. We'll say loading courses failed and then append the error to the end of the string. Again, remember with promises, you declare .catch in order to catch errors. Great. And with that final configuration, we're all set to try loading our app to see if the initial list of courses displays. Since we changed the Webpack configuration in a previous clip, if you already have your app running, be sure to hit Ctrl+C to kill the app and then rerun it via npm start so that our new webpack config is applied. Now if we go over to the browser, there we go. We can see our list of courses is loading great. If I open up the dev tools and go to the Network tab and reload, we can see our call to localhost 3001/courses, and we can see that response, which we are parsing. We're currently only displaying the course titles. So in the next clip, let's create a CourseList component that will display all the properties in this dataset.

## Create CourseList

We're currently only displaying the course title, but there's a lot of other data that's being sent down from our API that we could also display. We're also currently handling the presentation within our course page container, and that would ideally be handled by a separate component that just has presentation concerns. So let's go over to our courses folder and create a new file, and we'll call this file CourseList.js. I'm going to paste this component in. Again, you can get this file from the before file for this module. This is a functional component. I'm also using react-router-dom because we have a link in here that will link to a particular course. Notice that it looks for the course.slug, which is one other piece of data that's in our dataset. This is what we want to show in the URL. Our table header displays Title, Author, and Category headers. Then we map over the list of courses. And for each one of the courses, we output a table row. I'm using the course.id as the key and providing a button to be able to click to watch the course by clicking over and going to the Pluralsight URL. Down here, we show the course's authorId and category as well. Finally, down at the bottom, we have the propTypes declared. In this case, the only prop this component accepts is an array of courses. Although I just realized this should be propTypes.array.

Back up at the top, I should also clarify that I am destructuring props here in the function signature. If this looks odd to you, recognize that every functional component receives props as an argument, so I could choose to destructure props on a separate line within this component. This is equivalent to if I were in here and I hit Return, I could right here say const courses = props. So effectively, I'm taking this declaration and putting it inline right up here. And again, notice I'm able to use the concise arrow syntax so that I can omit the return keyword. I wrap all my JSX in a parenthesis so that it's a single expression and therefore is automatically returned. Now that we have this new component ready to display some course data, let's put it to use by updating CoursesPage.js. At the top, let's import CourseList. Then down in render, we can put it to use. We'll say CourseList, and we will pass it courses on props. This means we don't need these lines down here anymore. Let's load up the browser and see our results. Ah, this looks much better. And if you hover over the Watch buttons or the links to the courses themselves, we can see that the slug that's part of the course dataset is being injected into the URL. However, notice that the Author column lists the author's ID. In the next clip, let's display the author's name instead.

## Practice Redux Flow - Load and Display Author Data

The course data that we're receiving is normalized. It contains the author's ID, but not their name. So to display the author's name for each course, we're going to need to retrieve the list of authors. This is a good chance for us to practice the entire Redux flow. The process for loading authors will be the same as for loading courses. So I'm over here in actionTypes.js. Let's add a new actionType constant. Next, let's create a new file under actions called authorActions.js. This file will be extremely similar to courseActions.js, so why don't we come over here and copy it. And come over to authorActions, and it is going to be loading from the authorApi. We don't need the create story here, but we do need loadAuthorSuccess. It's going to accept a list of authors. The type here is going to be LOAD_AUTHORS_SUCCESS. The payload will be authors instead of courses. Down here, our thunk will be called loadAuthors. And instead of calling courseApi, we will call authorApi. And instead of calling getCourses, we will call getAuthors. That will return a list of authors, and we will call loadAuthorsSuccess. I just realized this should be plural, loadAuthors, and it will be passed authors. Next, let's go create an authorReducer. So under reducers, we'll create a new file. We'll call it authorReducer.js. And again, this is going to be very similar to our courseReducer, so let's just copy it, paste it over here, and change a few things. We're going to call it authorReducer. It will initially have an empty array for state. We don't need the create, but we do need a loadSuccess. So we'll have LOAD_AUTHORS_SUCCESS. We'll return the action, which will be that array of authors. Finally, and this is very easy to forget, but we need to go

update our rootReducer to reference this new reducer that we've created. So click over here on index, and let's import authors from. /authorReducer and then add it down here to the list of reducers that we're combining. Now that we've wired up the Redux flow, let's jump over to our CoursesPage. In mapDispatchToProps, we're currently only passing course actions in on props. We need to pass in the loadAuthor action too so that this component can request author data. First at the top, let's import authorActions. Then, let's jump down to mapDispatchToProps. Remember, this is where we specify the actions that we'd like to pass in on props. Right now, we're passing in all course actions under a prop called actions. Instead, we can pass the specific actions that we need in. For simplicity, I'm going to keep the actions under the actions prop. But then in here, we're going to change. Since we're only using loadCourses, we'll say that loadCourses is equal to bindActionCreators courseActions.loadCourses since that is the only one that we need. I'll copy this line and change this to say loadAuthors and then add our extra curly brace in. Oh, I also need to change this to authorActions. As I've shown here, you can pass a single function to bindActionCreators as well. Now in componentDidMount up top, we can make a call to load our author data. Again, it's going to look almost identical to our call to loadCourses. So I will paste this in and then change it to say loadAuthors. And if we have an error, then we will say Loading authors failed. Now we're loading author data when the courses page mounts. We'd like our course's array that's getting passed down to our course list to include the relevant author's name. To do this, we can enhance mapStateToProps. Let's go down here. What we'd like to do is weave in each author's name with each record. So first, instead of just returning plain courses, we can map over that list of courses. And for each course, we can return the existing course, but also add an authorName property to our object. To get that authorName, what we will do is look in our Redux store for that list of authors and find the author that has the relevant ID. And we'll return the name property. I do have an extra comma up here, so we'll fix that. We'll also return the list of authors, which we will set to state.authors. I'll explain why we need to pass authors in as well in a moment. First, there's a safety check that I should place above our call to map. Remember, course and author data are each requested separately and asynchronously, and we need to assure that they're both available before I can weave the author's name in right here. So let's do a check for whether author data is available. We'll say state.authors .length equal to 0. And if it's equal to 0, then we will return an empty array. Otherwise, we'll do what we were doing before. Hit Save, and Prettier reformats this. So I'm using a ternary operator. And let's talk through what this is doing. I want to pass in courses on props. If we don't have any author data yet, we're going to return an empty array. If we do have author data, then we're going to map over that array of courses, and we will enhance that existing array. Each element in that array will have an extra property called authorName, which we are setting by looking through the list of

authors to find the corresponding author that has the corresponding authorId. We set that record to that author's name. Now that we're including the author's name in our array of courses, we can jump over to our course list component and consume it. Let's go down here to where we were displaying authorId, and instead we should be able to say authorName. Let's check the browser and see if it worked. Aha, looking good. We now have the author name displayed instead. There's a final tweak that we can make. Right now, each time that we navigate to the Courses page, it's going to request courses from the API again. And we can see that if we go to the Network tab, clear this out. I'll go to Home, and I'll go to Courses, and there it requested both courses and authors. And each time I come to this page, it requests courses and authors again. That's rather wasteful. We really only need to request courses and authors once. So over in componentDidMount, we can fix this issue. Let's add an if statement right here. And we'll say if this.props .courses .length is equal to 0, that's the only time that we want to load our list of courses. I'm going to copy this line because we need to do the same thing down here for our authors, except we need to say if this.props .authors .length is equal to 0, then request the authors. Notice also that ESLint is pointing out that I need to update my PropTypes. So let's come down here and declare authors on PropTypes. What we have here works, but let's go back up to componentDidMount because I want to show how destructuring can make this cleaner. I can say const course authors actions are equal to this.props. Now that I've done that, I can remove the redundant this.props from our calls inside of here. (Working) Ah, that's better. Let's go check browser and make sure we haven't broken anything. Now when I reload the Courses page, it does request courses and authors. But when I switch between the pages, we're no longer making additional network requests. Much more efficient. Now we are dispatching actions on load, and we're weaving data form multiple APIs together in our mapStateToProps function. Again, this was a lot of moving parts because we just stood up a whole new slice of state for handling authors. This required us to create a new action file, reducer file, update our rootReducer, and so on. But as you're going to see, adding more features will now be relatively easy since our structure is in place. Remember, Redux is most useful for larger, complex apps. On trivial apps like this, the boilerplate feels tedious. But the larger the app, the more that this architecture will pay off in terms of scalability, maintainability, and clarity through consistency. Before we wrap up this module, let's make one more tweak. Let's centralize the declaration of our Redux store's initial state.

## Centralize Initial Redux State

We currently have two reducers, authorReducer and courseReducer. And now that we have multiple reducers, it's a little harder to picture our store's complete shape. So I like to declare initial state in a single spot. This way we clearly document the Redux store's shape. In the reducers folder, let's create a new file. And we can call it initialState.js. Inside, we're going to declare a simple object structure that says that courses is an empty array and authors is an empty array. Now we can open each reducer and reference this initial state. I'm going to import initialState from. /initialState. Then instead of hard-coding this array, I will say initialState.authors. I'm going to copy this line and then open my courseReducer and do the same thing. I'll paste in my import and then call initialState.courses. Now if developers want to understand the Redux store's shape, they can simply view initialState.js. This completes our initial coverage of async in Redux using thunks. Let's wrap us this module with a summary.

## Summary

In this module, we began by considering the merits of a mock API. I shared a variety of benefits to mock APIs, and I high recommend trying out the setup that we created for your own apps even if they don't use Redux. Remember, the mock API that we set up has no tie to Redux. It can be useful on any frontend app. Then, we configured our mock API using Express and json-server. I introduced the concept of Redux middleware and showed how Redux's middleware allows you to enhance Redux's behavior by running your own code on every call. We discussed approaches for handling asynchrony in Redux. We saw that there are multiple ways to get it done, including thunks, Redux promise, and sagas. We're using thunks because they're easy to learn and use. Once you get comfortable with thunks, consider looking into alternative libraries. We discussed approaches for handling async actions, such as having a suffix of success and error so that you can handle success and failure from your API calls by convention. And most importantly, we created our first thunks, which we're using to load course data. Now that we're comfortable with handling async calls for loading, it's time to add support for creating and updating courses to our app. So that's coming in the next module.

# Async Writes in Redux

## Intro

Now that we're comfortable using thunks to load course data asynchronously, let's put these new skills to use for creating and updating courses. This module is all code. We're going to create a full course management page that allows us to asynchronously create and update course data. We'll compose the form by creating a few reusable form inputs, and we'll initially populate the form using our Redux store's data via a combination of mapStateToProps and the useEffect hook. Alright, back to the editor. Let's keep coding.

## Create Manage Course Page

In the last module, we saw how to load data into the Redux store. In this module, you'll see how to add an editData in the Redux store. We're going to enhance the app to support adding and editing courses. We're going to handle adding and editing courses on a dedicated page. So let's create a new component in the courses folder called ManageCoursePage. To avoid creating components from scratch, I like to generate boilerplate code via my editor. For example, I enjoy using the React Code Snippets extension in VS Code. With that extension installed, I can type rcc to generate a React class component. When I hit Enter, it creates the boilerplate for me. So I recommend checking out the React Code Snippets extension. But in this case, let's copy the CoursesPage as a starting point because the ManageCoursePage is going to be similar in some ways. First, let's use the find and replace to replace the references to CoursesPage with ManageCoursePage. And I'll click this button over here to replace all instances. Let's remove the import for the CourseList since we won't be displaying a CourseList on this page. If someone loads our ManageCoursePage directly, we're still going to want to load our course and author data. So we'll go ahead and leave componentDidMount as is. Let's change the header down here to say Manage Course. Then jump down to mapStateToProps. We can greatly simplify this because we simply need to return state.courses and state.authors. And back up in render, I should've also removed the reference to CourseList. This gives us a good boilerplate for the ManageCoursePage. Let's jump to the top, and let's review the major sections. At the top, we import the core libraries that we'll need in any container component, React, PropTypes, and the connect function from React-Redux. Then we have the class component declaration, including its render function. And down below, we have our PropTypes declaration, the Redux mapping functions that determine what state and actions we'd like to access in our component, and finally the call to connect, which connects our component to Redux. What we have here works, but in the next clip, I want to show an alternative way to handle mapDispatchToProps.

## Implement Object Form of mapDispatchToProps

Earlier in the course, I showed four different ways to handle mapDispatchToProps. Let's use the object form of mapDispatchToProps now to simplify our code. If we declare mapDispatchToProps as an object instead of a function, each property will automatically be wrapped in a call to dispatch. This can end up simplifying our declaration. Let's show how this looks. So instead of declaring a function, we will declare an object. And for our object, we want two actions, loadCourses and loadAuthors. I'm going to remove the actions wrapper here. The wrapper isn't necessary. We'll simplify this down as best we can. So now we're no longer nesting these actions under a single object. This means that we need to tweak the calls above in componentDidMount. (Working) So here, we need to destructure loadAuthors and loadCourses. But this does mean we no longer say actions. here anymore, so these calls get a little bit shorter. And it also means that we need to go down to our PropTypes since we're no longer sending in actions. We are sending loadCourses, which is a func. I'll copy this. And we're also sending loadAuthors. I do need to put my comma up here. This means that we can go to the top of the file and remove the bindActionCreators import because we're no longer using it. We've now made mapDispatchToProps cleaner by using the object form. So now our mapping logic is more concise. Okay, a quick warning before I move on. I'm about to show a little tweak that makes mapDispatchToProps even shorter. But it is admittedly a bit confusing. You can decide if the benefit is worth the potential for confusion. We can use named imports up here on our actions. So I can call loadCourses since that's the only action that we're calling out of here, and I can use a named import here for loadAuthors. Now that I'm doing this, I can jump down into our mapDispatchToProps function, and I no longer need to say courseActions.. Now you notice, there's one more tweak that I can make. Because of JavaScript's object shorthand syntax, I can omit the right-hand side since it matches the left-hand side. And finally, we end up with a very concise mapDispatchToProps. This I like. Clean and simple. However, here's the confusing thing about this final tweak. We now have two different variables on the page that are assigned to the same name. I'm importing loadCourses and loadAuthor action creators at the top. Then I'm using mapDispatchToProps to pass them in on props under the same name. Since we're destructuring props at the top of this component, the action passed in on props is the one that's in scope. The function scope takes precedence over the module scope. So our calls to loadCourses and loadAuthor action creators are properly bound. I admit this is potentially confusing, so feel free to use the previous wildcard import style to avoid this naming conflict if you prefer. So I've shown you many different ways to handle mapDispatchToProps, but I think this one is my favorite. I like declaring it as an object instead of a function since it's easy to implement and typically requires less code. Just keep in mind that there are many ways to handle mapDispatchToProps, and you're

free to mix and match the different ways that I've shown you. In the next clip, let's set up routing for our new page.

## Configure Routing for ManageCoursePage

Since we're just created a new page, we need to update routing to support it. So let's open app.js and add the route. Import ManageCoursePage from courses/ManageCoursePage. Then down here, we need to add two routes. So I'm going to copy the existing route. And for our first route, we want /course/:slug, and we want that to load our ManageCoursePage. Our second route is almost the same, but it won't contain the slug. As you can see, we expect the second segment of the URL to provide the course slug. You could think of the slug like the course's ID. Like an ID, the slug is unique. But unlike an ID, the slug is friendlier to read in the URL than a number. Remember, our routes are wrapped in switch, so only one route inside switch can match. The moment the React Router finds a matching route, it stops searching. So if it finds a slug in the URL, it will stop here. This is why we need to make sure that the slug route is declared first. Otherwise, the shorter course URL would match, so our slug route would never load. Now that we've created the routes, let's see if we can load the page. I can click on Courses. And if I click on a course, I can now see the Manage Course header successfully. The Manage Course page is now rendering, but it doesn't need to be a class component. In the next clip, let's convert ManageCoursePage to use hooks instead.

## Convert Class Component to Function Component with Hooks

Let's convert this class into a functional component and use React's useEffect hook to replace componentDidMount. If you haven't tried hooks yet, they were added to React in early 2019 in React 16.8. Hooks allow us to handle state and side effects within function components. So we typically don't need to declare class components anymore. Switching this to a function component using hooks requires a few tweaks. First, switch the class to a function. Hooks only work with function components. I could just move this line up here, but I like to destructure props within the function signature because it's a little more concise. So let's cut this and paste it in right here. Now I can remove that declaration because in our function component, we'll be able to access all of these variables that we've destructured. Now let's go up to the top, and on our import statement for React, we're going to pull in the useEffect hook. We're going to use this hook to replace our usage of componentDidMount because useEffect lets us handle side effects. UseEffect accepts a function that it will call. In this function, we'd like the same logic to run. So let

me come down here and close our useEffect call. We no longer need to declare render in a function component because it is implied. So I'll take these extra curly braces out. Now when I hit Save, Prettier cleans up my formatting. So we have our return statement for our functional component, and then nested inside this function component is our useEffect hook. Right now, the way this is coded, this effect will rerun every time that the component renders. We want this to only run once when the component first mounts. So to do that, we can declare a second argument to useEffect. We declare that argument as an array of items for it to watch. And if anything in that array changes, it will rerun this effect. Since we only want this to run once, we can declare an empty array here, and this is effectively the same as componentDidMount. This will only run once when the component mounts. Let's hit Save and then jump over to the browser. Our hook-based component is rendering, so we're on the right track. I'm a big fan of hooks, and I recommend declaring function components over class components because they tend to be easier to work with and understand. We've now implemented our first functional component that uses React Hooks. In the next clip, let's create the form for managing courses.

## Create Course Form

To save some typing, grab three simple React components from the before folder for this module, TextInput and SelectInput, which are in the common folder, and CourseForm, which is in the courses folder. I'll paste these in one at a time and discuss each. Earlier in the course, we created a simple inline form that allowed us to create a course with a title. Now let's create a more realistic form that utilizes child components to help enforce enhanced styling and functionality. In the courses folder, create a new file and call it CourseForm.js. I'll just paste this in, and we can talk through it. I'm importing React and PropTypes. I'm importing a TextInput and a SelectInput, which we will create in a moment. This is a functional component, and I'm destructuring props in the argument. The form itself has a form tag with an onSubmit that will call the onSave passed in on props. I have a header that changes based on whether the course ID is passed in. If there's no course ID, then I label it addCourse instead of editCourse. Have a spot to display errors, which we'll implement. And then down below, nothing too surprising. I have inputs for each of the different pieces of data that we're going to allow someone to input, the title, the authorId, and the category. We're passing onChange handlers to these, as well as any errors that might occur during validation. Down at the bottom, we have our Submit button. And while it is saving, it will change the label from Saving to Save. And finally at the bottom, we have our PropTypes declarations. In the next clip, let's create the TextInput and SelectInput components that we're referencing here.

## Create Reusable TextInput and SelectInput Components

Now we need to create the reusable components for handling textInputs and SelectInputs on forms. Since they're common components that are likely to be used in a number of spots, let's place them over here in the common folder. I'll say New File and put in TextInput.js and then paste in the implementation. Another functional component. I'm adding some classes that come from Bootstrap, wrapping our input in a div, putting the label in, and then also wrapping our input itself in a div with the className of field. So this will look very nice, and our consumers won't have to do much work to interact with this. These are basically the same components that I created from scratch in my React and Flux course. So check that course if you want to see these created by hand in a little more detail. What's handy about these components is they encapsulate the necessary Bootstrap markup and styling, which will make them look great and consistent. Let's go create our SelectInput as well. There's the SelectInput component. Nothing too different from the TextInput, except, of course, it accepts an array of options, which we do map over and set the key and the value and display the text passed in. Then down at the bottom, again, specify the PropTypes. These sorts of base-level components help greatly reduce complexity, as well as enforce consistency throughout the rest of our app. In the next clip, I think we're ready to put our course form to use.

## Call CourseForm on ManageCoursePage

Now that we've created the necessary components, let's reference ManageCourseForm here in ManageCoursePage. We need to pass course data to the course form. So we need to update mapStateToProps to pass an empty course that will get us started. I declared the empty course structure in mockData.js, so let's import that as well. Then, jump down to the bottom to mapStateToProps. We'll pass in that new course that we imported up above on props. And up here on PropTypes, let's also declare that empty course with a PropTypes.object .isRequired. Our form will need state to hold the form field values before they're saved, so let's jump up to the top and set some state on this component. Our form needs state to hold the form field values before they're saved. So let's set up some local state on this component. Since we're using React Hooks, let's go to the top and import the useState hook. UseState is a hook that lets us add React state to function components. We can declare our state down here inside our function component. I'm going to declare course as our state and the setter as setCourse. We will call useState. And for the default value, we'd like to set it to the course that's getting passed in on props. Let's talk through this. UseState returns a pair of values. We use array destructuring syntax to assign each value a name. The first value is the state variable, and the second value is the setter function for

that variable. UseState accepts a default argument. We're specifying that it should initialize our core state variable to a copy of the course passed in on props. Now we can see some red squigglies here, and that's because props isn't defined because we're not destructuring it up here in the signature. And this produces an interesting conflict because I can't destructure it right here or it will cause ambiguity between the course passed in on props and our course object declared on line 10. So instead what I'm going to do is use the rest operator to assign any properties that we haven't destructured to an object called props. Now we can see that this code no longer has an error. The rest operator uses the same three-dot syntax as the spread operator. So here the rest operator stores any properties that we haven't destructured on the left here in a variable called props. This allows us to reference props.course below and avoids us having two different variables called course. One other option you could consider is to alias course as initialCourse within the destructuring statement. Two different ways to resolve this conflict. But I'm going to go with the original approach. And when I hit Save, Prettier reformats. Before we move on, note that I'm using React state. Now you might be wondering why I'm not using Redux to hold this form state, and the reason is that's typically unnecessary, and it leads to extra complexity. Avoid using Redux for all your state. Plain React state remains useful for local state. Use Redux for more global values. So if you're wondering when to use local form state versus Redux, ask yourself, who cares about this data? If only one component or a small, related group of components cares about the data, then just use plain React state. Keep it local. Typically, with a form, only a single component cares about the unsaved data. So local React state makes sense for most forms, as well as for any state that a single component or a small subnet of components needs. Now we're ready to call the course form component. This means that we can jump back down to the bottom of our render function. We can eliminate the header since that is part of our course form component. We don't need the fragments anymore either. Instead, we will call CourseForm, and we will pass it the course and the errors, as well as the authors. Notice that I'm referencing errors, but we haven't declared that yet in state. Let's jump back up here quickly and do that. I'm going to copy this line just to make it easier. We're going to initialize errors to an empty object. We'll name the setter setErrors, and we'll name the state variable errors. So this state will hold any errors that occur when we run validation later in the course. Alright, this should get us rolling. Let's jump back to the browser and see how it looks so far. We need to load /course manually because we haven't yet created a way to navigate here. But great! We now have the form rendering, and it looks very nice in a vanilla, lame, oh man, I've seen this a thousand times before, Bootstrapy kind of way. Hey, I'm not a designer, but this looks fine to me. Now if you open the console, you can see that we're currently getting a few warnings because we're not passing down all the required props. We'll add those in a moment. So what's next? Well, if I try typing in any of

these form fields, not much happens. Looks like we need some change handlers. So let's fix that next.

## Implement Centralized Change Handler

We have our form displaying. So the next obvious issue is we can't type in the form fields. Why? Well, because they're managed components, and we haven't defined a change handler. So let's add a change handler. I'm going to declare a single change handler for all of our form fields. Now if you look at CourseForm, you'll see that each one of the inputs has a name, which corresponds to the property that it displays. For instance, the category input has a name called category, and it's value is set to course.category. This allows me to update the corresponding value in state with a single function. So let's declare that single change handler. We'll call the function handle change, and it will accept an event. First, I'm going to do destructuring of that event at the top, and we'll see why this is useful here in a moment. Then we'll call setCourse, and we will use the functional form of setState. Remember, you can pass either an object or a function to setState. I'm using the functional form of setState here so that I can safely reference the previous state as I set newState. Here I'm using JavaScript's computed property syntax so that I can reference a property using a variable. So for example, if the input that just changed was the title field, this code is equivalent to saying course.title. Quite handy. And this approach is also outlined in the React docs. Then we need to handle the value differently depending on whether we're working with the author ID or any other value. Since we need to handle the author ID as a number, I call parseInt to convert that value to a number. Otherwise, I use the value that was passed in on the event directly. Even though we're storing the ID as an int, it will be returned as a string in the event. So I'm using parseInt to convert the ID back to an int. Note that I'm destructuring here at the top. This shortens the calls below, which is nice, but it's also necessary in this case so that we can access the event inside the setState function. Without this destructuring statement, we get the error that this synthetic event is reused for performance reasons. Why? Well, that's because the synthetic event is no longer defined within the async function. Destructuring on the first line avoids that error because it allows us to retain a local reference to the event. Now that we've declared our change handler, we can come down here and pass it in on props. We'll say onChange=handleChange. Hit Save. Prettier reformats that. Again, this is a nice thing about functional components and React Hooks. Notice how we don't have the this keyword littered all over the place. We have nice, short calls to local variables that are scoped to our function. And with our change handler declared, we should be able to jump back over to the browser. And now,

we can type within the fields. That's a good thing. But we are still getting one PropType validation error because we haven't declared the onSave yet. So next, let's get that working.

## Add Save Course Thunk and Action Creators

Our form needs a save function, so we're going to need to create a new action. So open CourseActions.js, and we'll create a new thunk called saveCourse. And this is extremely similar to our LoadCourses thunk, so I'll just paste this in and point out the minor differences. We'll put it down here at the bottom. So this thunk looks quite similar to our LoadCourses thunk up above, except we are calling saveCourse instead of getCourses. And another difference is that we want to dispatch different actions depending on whether we are creating a new course or updating an existing course. We determine that by looking at weather a courseID exists. If there's an ID, then we know we want to update an existing course. Otherwise, we know we want to create a course instead. One other thing to note is although I'm going to pass the course in from the form, you may choose to access the store state within your thunk without having to pass the data all into your thunk. The second parameter that you see right here is optional. And if I had set a breakpoint and watched this parameter, I'd see that getState has all of our Redux store's state inside. Now in this case, I don't need this feature, but I wanted you to be aware of it since this can be handy. So our next step is to resolve these red lines down here because we haven't declared either of these action creator functions yet. So let's jump up top since I like to keep all the action creator functions here at the top, and we'll create these two functions. And we can see that we also need to go create the corresponding constant. So let's open actionTypes.js. I'll add these two new actionTypes in for CREATE_COURSE_SUCCESS and UPDATE_COURSE_SUCCESS. While we're in here, we can also jump up here and remove this first action creator because we're not using it anymore. Our CREATE_COURSE_SUCCESS and UPDATE_COURSE_SUCCESS action creators have replaced this old one. Now that our new thunk and action creators are set up, let's shift our focus to the reducer in the next clip.

## Handle Creates and Updates in Reducer

Now it's time to update the reducer. We're going to handle both the creation and the update separately since they each have their own action type. For handling create, we just need to add the suffix SUCCESS here on the end since that's our new action name signifying that the save succeeded. Our update course reducer will look slightly different. Again, since state is immutable, we can't simply change the appropriate index in the array. Instead, I'm mapping over the courses,

which returns a new array. When I find the course that has the ID that was just changed, I replace it in the array. This is handy since it returns a new array, and it allows me to replace a value in the array without changing the array's order. I'm also using the concise arrow syntax here. The return is implied because it's a single expression without curly braces after the arrow. This likely feels odd and confusing to you at first. And trust me, after you work with immutable data for a bit, you'll get very comfortable with these patterns and tools. You'll find that you often use the spread operator, map, filter, and reduce to get things done. Once you understand these built-in JavaScript features, you can accomplish powerful tasks without mutating state. We've now wired up our actions and updated our reducer, so we just need to update ManageCoursePage to put this to use. One thing I want to point out before we move on. Did you notice how adding support for create and update was quite quick now that our Redux infrastructure is all set up? See, things are getting easier.

## Dispatch Create and Update

Okay, we're back in the ManageCoursePage component. Now that we have the actions that we need, we can dispatch the saveCourse action when the user saves a course. So let's import the saveCourse action and then add it down here to mapDispatchToProps. And notice how easy it is to add with the object style of mapDispatchToProps. We should also jump up here and declare it on our PropTypes. Now that we're passing the bound saveCourse function into the component, let's destructure saveCourse up here at the top. This way calling saveCourse in our component will reference the saveCourse function that we just configured in mapDispatchToProps. Instead of the plain saveCourse function that's being imported up above. Now we can move down and implement the handleSave function. HandleSave will accept an event, and inside, we want to call event.preventDefault to keep the page from posting back. Then we can call saveCourse and pass it the course that's available in state. Remember, saveCourse is getting passed in on props, and it's being bound to dispatch via our mapDispatchToProps declaration below. Again, this can be confusing because it's also imported by the same name at the top of this file. But inside this function, our locally scoped bound variable that's being passed in on props takes precedents. If you find this confusing, rewatch the third clip in this module. In that clip, we were using a wildcard import for actions at the beginning. The wildcard import avoids the duplicate naming you see here in exchange for a little extra code in mapDispatchToProps. Now we can pass onSave to our course form. Hit Save and bam. That should do the trick. And if it doesn't, I'm going to throw my mouse across the room. I mean, come on. We just wrote a lot of code together. I can't afford public failure at this point. Let's jump over to the browser and see if I can keep my job. So let's

load up the Course page, and we'll have to change the URL manually because we don't have a way to navigate here directly yet. I'll put in a test title, select and author, and put in a test category. When I hit Save, nothing happens, and that's because we haven't set up any kind of confirmation or redirect yet. But if I click over here on Courses and scroll down, aha. Success. Our new course has been added successfully. Awesome. I get to stay being an author for now. Of course, there's two obvious problems. There's no way to navigate directly to our Add Course page, and when I hit Save, I get no feedback. So let's add a button for navigating to the Add Course page first.

## Redirect via React Router's Redirect Component

Let's put an Add Course button on the Courses page. In the next two clips, I'm going to show two different ways to handle redirects with React Router. In this clip, we'll use React Router's Redirect component. So first, we're going to import Redirect from react- router-dom, and now let's come down into the render function where we can add our button. I'm going to place the button right here below the header. I've specified some Bootstrap styles so it looks nice and also an inline style because, well, I'm a total rebel. OnClick I'm setting redirectToAddCoursePage to true. So let's go up to the top and add this value to state. Again, instead of a constructor, I'm using a class field to save some typing. Yes, I'm that lazy. Now we can use the state value in render to enable the redirect when redirectToAddCoursePage is true. So let's handle this right up above the header, although the exact placement doesn't matter much. Yes, this feels weird at first. Render has a side effect here. If the value in state is true, it will redirect to the Add Course URL. We're using JavaScript's logical and operator to pull this off, so the right- hand side will only evaluate if the left- hand side is true. If this line really makes you angry, don't worry. I'll show you a different way to handle this in the next clip. Let's hit Save though and see if this works. CoursesPage now has an Add Course button. And when I click it, I'm successfully redirected. Amazing. It worked on the first try. I shouldn't act so surprised. Next, let's set up a redirect to the course list when a course is saved.

## Redirect via React Router's History

Our save works great, but we should redirect the user back to the list of courses when the save is complete. In this case, I'm going to show a different way to handle redirects via React Router. Notice that I'm back in ManageCoursePage. This time, let's use React Router's history to handle a redirect. Our call to the saveCourse thunk returns a promise. So we can chain a .then onto this

call. Inside the then, we can call history.push and redirect to /courses. So this says after the save is done, use React-Router's history to change the URL to the Course List page. For this to work, we need to destructure history at the top because it's being passed in on props. We also need to declare this down here on PropTypes. Now you might be wondering, where's this history object even coming from? Well, any component that's loaded via a React Router route gets the history object passed in automatically. Isn't that nice of React Router? So kind. Okay, let's try it out. I'm over on the Add Course page. Let's enter some random text and a random category and hit Save. Aha, look at that. We got redirected successfully. However, there's still some room for improvement. We don't get any notification when the save succeeds. We'll fix that issue soon. But first, we have a bigger problem. If you click on any existing course, it doesn't populate the form. Let's fix that next using mapStateToProps.

## Populate Form via mapStateToProps

As we just saw, if someone clicks an existing course, the form isn't populated. So let's write some code to make that work. In Redux, that means we have some work to do in mapStateToProps. Right now, we're always passing an empty course to our container component. Notice that I'm referencing newCourse right here. What we need to do is look at the URL and determine if the user is trying to add a new course or edit an existing course. To get parameters from the URL, there's a handy second parameter on mapStateToProps. It's called ownProps. This is automatically populated by Redux. OwnProps lets us access our component's props. So in this case, it means that we can access some routing-related props that are populated by React Router based on the route defined for this component. So to read the course slug from the URL, we can add a single line of code. I'll say const slug = ownProps.match .params .slug. Now let me clarify why this line works. If we look over in App.js, we can see that the route for this component says that the second segment of the URL has a placeholder that represents the slug. This means that it will be available on ownProps.match .params .slug. So now we want to set the course to the requested course if there's a slug in the URL or to an empty course otherwise. We can do that like this. Declare a constant for the course. And we'll say if there's a slug, we want to get a CourseBySlug. This function doesn't exist yet, so we'll need to create it. But we'll pass this function state.courses and the slug. Otherwise, we'll set it to a new course. So if the user is requesting an existing course, we need to get the course that they requested. How do we do that? Well, it should be available in state. So we just need to pluck the right course out of state on this line. Let's put this logic in a separate function for readability. I'll place the function right here above mapStateToProps. There we go. This function accepts the list of courses and the course

slug that we're looking for. I use JavaScript's built-in find method to get the course requested. And if it happens to not be found, then I just return null. Functions like this are commonly called selectors. Why? Because it selects data from the Redux store. You could also place this function in the course reducer so that it could be called by other components. For performance, you could even memoize the response with a library like reselect. For more information on this pattern, check out selectors in Redux. Now let's jump back down into mapStateToProps, and we can update this line and remove the right-hand side altogether because now we are setting course to course up here. Again, I could say course right here, but it would just be redundant. So I'll use the shorthand syntax. Strategic variable naming for the win. Let's go ahead and run this and see if it works. Let's come over to the list of courses, and then I'll click on a course. As we can see, it doesn't work. Let's open up the terminal and see if it gives us any hint. We don't see any errors down here in the browser. We do have a warning about setErrors, but you can ignore that because we will resolve setErrors in the next module. If we set a breakpoint over here in the code, we can see what the problem is. I'll hit Save. When we are calling getCourseBySlug, the courses array is empty. And, of course, it's empty because our API call hasn't completed yet. Remember, it's an asynchronous call so it may take a moment for it to succeed. So our Redux store starts out with an empty array of courses. Therefore, we're not going to find any courses yet. So to fix this, we need to come over here and AD a check. If we have a slug and state.courses .length is greater than 0, then we want to get the CourseBySlug. Otherwise, we will return the new course. So let's remove our debugger. This should fix our problem. Remember, mapStateToProps will run each time the Redux store's state changes. So when there are courses finally available, then we will get the CourseBySlug. If we go to the Courses page, we should be able to click on any one of the courses and, aha, success. It reads the slug from the URL and populates our form accordingly. We should also be able to edit data on our course and then hit Save and hey look at that. It's reflected. So it looks like we're in pretty good shape. However, there is a bug lurking here. What if I try to load a specific course page directly? Let's click on one of the courses. And now if I load this page directly, I'll just come up here to the URL and hit Enter, notice that it's not populating the form. Now this can seem confusing at first, but it's an easy mistake to make. Remember over here in ManageCoursePage, this course that we're passing down to CourseForm is declared in state up here at the top. So it is a copy of props.course. And we needed to make that copy so that we could hold data in state as we edited the course. So hopefully a light bulb just came on. See state is being initialized right here in this useState call, and this happens on initial load before the list of courses is available. So this logic on line 18 is performed once based on the course data that's available when the component is mounted. The problem is when the component is mounted, no course data is available quite yet. It's an asynchronous call. There are multiple ways

to solve this issue. Let's explore the options next. We have the CourseForm populating in some cases, but it still doesn't populate when we load the page directly. Why? Well because when the props change, we need to update our component state. This can be easily solved by tweaking our useEffect hook. Right now, the useEffect hook only runs once when the component mounts. Instead, we want it to run any time that a new course is passed in on props. This way up here, we can say that if we do have courses available, we would like to set our course in state to the course passed in on props. So this will copy the course passed in on props over into state any time that a new course is passed in on props. Another solution to consider would be to declare a key on the route for this component up in App.js so that the component would remount when the key changes. But that's more work and likely more confusing, so I'm going to stick with this approach. Let's save our change and see if it works. Aha! With this change, we can now load the form directly, and it populates. Nice. However, if you watch closely, you can see the form quickly flicker from Add to Edit mode. So we still have some work to do in the next module. In the final clip for this module, let's wrap up by reviewing where we are so far and chart out where we're headed.

### Summary

In this module, we enhanced our application by creating a dedicated page for adding and editing courses. We saw how to create reusable form components that centralize and abstract away the complexity of dealing with Bootstrap's expected markup structure in CSS classes. And we populated the form on page load using a combination of mapStateToProps to initially populate the form when mounted and the useEffect hook, which updates our local copy of state when props change. In the next module, let's polish the application's user experience. We'll create new dedicated actions and reducers so we can properly manage and display the status of asynchronous calls. We'll also add some elegant error handling while we're at it.

# Async Status and Error Handling

### Intro

Our application is coming together nicely, but there's still room for improvement. We're completely ignoring async status and error handling, so in this module we'll use React, Redux,

promises, and some other tricks to polish our application's user experience. Here's some specific issues that we'll resolve in this module. We'll add a loading indicator, display feedback upon clicking Save, handle API failures, display server-side validation, and add client-side validation, as well as optimistic deletes to help mask a slow API. And we'll close out by trying async/await instead of promises. This short module is all code, so we're going to jump right back to the editor and continue coding. Let's first review a few key usability issues in our app. Our app still has problems, but some of its issues are hiding because so far our mock API is so fast that we don't notice. So, let's change the mock API so that it takes 2 seconds to response to each request. I'm in apiServer.js, and here on line 30, we can specify a delay for each one of our requests. I'm going to set it to 2000. This will delay all requests by 2 seconds. After hitting Save, be sure that you hit Ctrl+C to kill your app if you're running it, then hit npm start to rerun the app. Now when we load the app and click on courses, you can see that we wait 2 seconds before the course data displays. And we have no indicator telling us that things are loading, so this is pretty clunky. If I come in here and click on a course and then hit Save, our form experience is bad too because we sit there for 2 seconds without any indication that the application is responding. We can't tell that there's a save in progress. If we go to the manage course form and hit refresh, then we have a different problem. We can see that the app takes around 2 seconds to load before it switches over to the edit mode, so we see an empty form for a couple seconds. Ouch! All of these issues tie back to the same problem. We're not properly handling async status. We need to display a spinner so that the user knows that async calls are in progress. So in the next clip, let's create a component that we can display when async calls are in progress.

## Create Spinner Component

Our app doesn't currently notify the user when an API call is in progress, so let's solve that. Let's create a new component in the common folder called Spinner.js. This component is quite simple. It's a div with a class assigned to it and some text inside. But notice I am referencing a CSS file, so let's create a corresponding CSS file as well, Spinner.css. This is a pure CSS spinner. Don't waste your time typing this CSS out. Just grab the file from the module's exercise files. Look in the after folder. To try this out, let's add this to the CoursesPage. I'm going to display it down here on line 36. And let's go to the top and import the Spinner. Now let's jump over to the browser and see how it looks. Great! It works! Of course, the obvious next step is to only show this when there's an API call in progress. So next, let's set up a pattern for keeping track of when API calls are in progress.

## Create API Status Actions

Now that we have a spinner to display when API calls are in progress, we need to keep track of when they begin and end. To make that happen, let's create a new action constant called BEGIN_API_CALL over here in actionTypes. Then, under actions, let's create a new file. We'll call this apiStatusActions because these will be actions that relate to the current status of our APIs. The most obvious action will be BEGIN_API_CALL, so let's create that. First, import * as types from '. /actionTypes. ' Then, we will declare BEGIN_API_CALL. So this is our action creator for BEGIN_API_CALL. Next, let's open up our initial state file and add some more data to that. We will store the apiCallsInProgress. And this will be a number that we initialize to 0 because we'll be tracking the number of API calls that are currently in progress. Remember, theoretically we could have multiple API calls in progress at the same time. So each time one completes, we'll decrement this number, and each time we start a new one, we'll increment this number. Now let's create our API status reducer. So, come over to the reducers folder, say New File, apiStatusReducer.js. This will follow the same basic structure as our other reducers. Notice that I'm calling the initialState that we've centralized, and it accepts an action like all other reducers. Then, if the action type is BEGIN_API_CALL, then we will increment the number of API calls in progress by 1 by returning state + 1. Notice that I'm using an if statement instead of a switch. For simple reducers like this, I find an if statement sufficient. This will also help me pull off a little trick in a moment. Remember, Redux doesn't force you to use a switch statement. Anytime an API call begins, I'll increment state by 1. State, in this case, is the API calls in progress counter that we just added to initialState. Remember, each of our reducers is working with a specific slice of state. In other words, you could think of the Redux store being split up and composed of a bunch of separate reducers. So this reducer will keep track of when API calls begin, but how do we know when they end? For that I'm going to use a trick that Dan Abramov, the creator of Redux, suggested to me when reviewing this demo. I'm going to use a convention. Recall how our thunks ultimately dispatch an action that ends in the word _SUCCESS when the complete. So, I can use that _SUCCESS suffix as a signal that the API is completed. This helps me avoid manually dispatching a separate end API call action every time that an AJAX call is completed. To do this, let's add an else statement that checks to see if the action type ends in _SUCCESS. First, above this reducer, I'll add a helper function. So this function will determine whether the action type ends in _SUCCESS by using substring. We can put this to use down here in our else statement. So if the action type ends in success, we will decrement the number of API calls in progress. Before we move on, I want to note something else important. We're now handling the same action type in multiple reducers. For instance, loadCourses' success will be handled by our course reducer, as well as our API

status reducer. So remember, each reducer is simply a slice of state, so a given action may impact multiple reducers. And a given reducer will typically work with multiple actions. So there's a many-to-many relationship here. And this is why it's useful to keep your reducers and actions each in separate folders. Some people try to group actions and reducers together in feature folders, but I don't recommend it. Why? Because each action may be handled by multiple reducers. Actions and reducers shouldn't typically be tied to a single feature or portion of the app, so avoid placing them in feature folders. Doing so is a sign that you don't need to put that state in Redux at all. We have one final step to make since we've created a new reducer, and this one's always easy to forget. We need to add the reference to our root reducer. So, open index.js in your reducers folder, import the apiCallsInProgress from apiStatusReducer, and then reference it down here within combineReducers. Again, it's easy to forget this when creating a new reducer. This is the most common mistake that I make in Redux. But if you ever set a breakpoint in a reducer and you see that it's not getting called, it's likely because you forgot to add it to the rootReducer.

## Call Begin API in Thunks

Now that we've created our new action and reducer, let's update our existing thunks to call it. Open the courseActions and authorActions files and let's update them to play along. We're going to import the beginApiCall, then down in loadCourses, before we call the API, we need to dispatch beginApiCall. And be sure to include the parentheses so that the action creator is invoked. I'm going to copy this line and jump down to the savedCourses, where I'll do the same thing, right above the return for calling courseApi. Then, let's jump over to authorActions and also add it here. So, I will call dispatch right above the call to authorApi, and again over here to authorActions. Now you might be thinking, could we handle this in the author and course fetch calls instead over in the API folder? Then we wouldn't have to remember to add this dispatch call to every thunk. That's true, but there's some benefits to our current approach. We don't have to pass dispatch in our API calls. Now, I could also choose to make the actual fetch calls within my thunks instead of over in the API folder, but I prefer to keep my fetch calls in a separate file so that I can deal with that concern in isolation. Another advantage to my current approach is I can decide to not show a preloader for some thunks. As you'll see, this is useful when I want to do an optimistic create, update, or delete. In other words, sometimes I want to immediately update the user interface when someone clicks a button without them having to wait for the response to return. This technique is called optimistic updates, and it makes UIs feel extremely responsive because they always respond instantly, no matter the speed of the API call. I'll show this a bit

later. All right, we've wired this all up, but if you check the browser, you'll see that that preloader is still displaying the whole time. That thing just never gets tired. It's still displaying because we haven't added any code that hides it based on status. So let's do that final piece next.

## Add Spinner to Course Pages

Now we have all the infrastructure in place for properly tracking API calls status in our Redux store. Let's put this new information to use. I'm in CoursesPage.js down here in the mapStateToProps function. What we want to do is return a Boolean that tracks whether an API call is in progress. So, we need to pass another prop to our component. Instead of passing the number of calls in progress as a prop, let's create a Boolean called loading, and we'll set it to state.apiCallsInProgress are greater than 0. Because we know that the app is loading if at least one API call is in progress. Since we've added another prop, let's come up here and add it to PropTypes as well. Now we can move on up a bit and put this to use and render. Instead of always displaying the spinner, we want to show the spinner when this.props .loading is true. Otherwise, we'd like to do these other things below. So I'm using a ternary operator to selectively hide the rest of our page. Now notice this does create a problem because both course list and button are peers, so we need to add a parent wrapper element. Again, I'm going to use the fragment syntax here. And I also need to close my curly brace up from line 37. Notice how VS code shows our matching braces. When I hit Save, this will reformat and become a little easier to read. So now we're saying if loading is true, render the spinner; otherwise, render our button and our course list. And remember, I needed to add the fragment wrapper here because jsx requires a single top-level element for each expression. Instead of fragment, I could wrap this in a div, but fragments are preferable when we don't need the div because it avoids creating a needless wrapper element in the DOM. And with this complete, we should be able to jump over to the browser and see this in action. Let's go to the courses page and hit refresh. There's our loader. Hey! And there's our data! Our API is a little slow, but I think I know the root cause. So we've solved one of our loading problems. But this issue still remains. If I come over here and hit Refresh, we do see an empty form for 2 seconds. Let's solve this problem with a slightly different approach next. Now that we know when an API calls is in progress, we can put this information to use to improve the user experience in other places. However, you don't have to use this information everywhere. You may find being more specific is useful, so that you can decide when to show the spinner. I'm now in the ManageCoursePage. And as we just saw, if we load this page directly, it looks broken. It shows the add form at first until the course and author data is available. So we need to hide the form until we have this data. We could use the same pattern as the

previous clip. However, if we do, then the form will disappear and display a spinner the moment that the user hits Save. Instead, I'd like to continue showing the form and disable the Save button while the save is in progress. I think this feels more natural. To make this happen, first, let's import the spinner at the top. Then, let's jump down to the bottom of our render function, and above our CourseForm, let's add some code right here. If authors.length === 0, or courses.length == 0, then we know we don't want to show the form yet because loading needs to complete first. So, let's call the spinner in that case. Otherwise, we'll go ahead and show the CourseForm. That's simple enough. Let's try it out. I'm back over on the Edit Course page. If I hit Refresh, I see the spinner, and there we go, there's our form. Now the form is hidden until the necessary data is available. However, we still have another problem. When I hit Save, I don't get any feedback that an API call is in progress. It would be nice to disable the Save button when we hit Click, so that we have some visual feedback that the Save is in progress. Let's fix that next.

## Add Form Submission Loading Indicator

Right now when you click Save on a course, it provides no feedback at all. It would be nice to at least acknowledge the click and disable the button with a friendly message so that you know that the save is in progress. This is easy to pull off with a little local state. So, let's do that. Before I do, I want to clarify that one of the common questions in Redux is when to use local state. In this case, local state is useful because this is fleeting data that the rest of the application won't care about. We could run this interaction through the Redux flow, but in my opinion, it would just be unnecessary overhead since so few components care about this data. First, let's add some additional state via the useState hook. We'll store this in a Boolean called saving, and call the setter setSaving. We'll initialize this state to false. Now let's scroll down to the handleSave function, and when the save starts, we need to setSaving to true. We don't need to set it back to false again after the API call completes, since we're going to redirect to another page. Now we can pass this as an additional property on the course form. Now the data that we need is getting passed down to the CourseForm, so let's open it up. When we originally created this, I already had it wired up to accept a Boolean for saving, which determines whether the Submit button is disabled and what the label says. So, we don't need to make any changes here. We're actually all set. So, let's check this out in the browser. We'll click on a course and hit Save. The input disables and shows a different message, and then redirects. So now we have a nice feedback experience, so we know that that call is in progress. Our form is starting to feel polished, but it would be nice if we notified the user when the action is complete. So, next let's add a toast. We need a friendly way to tell the user when actions are successfully completed. For that, let's use an open source

React component called ReactToastify. This is already installed since it was listed in our package.json in the beginning of the course. To configure ReactToastify for use throughout the site, let's import the toast container and the CSS that comes along with it. Now we can place the toast container anywhere within render. I'll set it down here below the switch. To configure ReactToastify, we can add a few props. I'm going to set autoClose to 3000, which means that it will automatically hide the toast after 3 seconds, and I'm going to configure it to hide the progress bar because it comes with a built-in progress bar, but we don't really need that. And that's all it takes to globally configure ReactToastify. Now that we've configured it, let's jump over to ManageCoursePage.js, and we'll put it to use. At the top, let's import toast from react-toastify, so this is the method that we will call to display the toast. Then, we want to display the toast after the save is completed. So, before we call history.push, we will say toast.success, and then we can decide what message to display. We'll say Course saved. That should do the trick. Let's try it out. Click on a course, hit Save, there we go. Toast displays. Slick! It animates in and then it hides itself automatically after 3 seconds, just like we configured in app.js. Our app is looking quite polished now, but there's a significant concern left. What about error handling and validation? Let's address those next.

## Handle Server-side Validation and API Errors

Our form is looking good, but what happens when the API throws an error? Let's find out. Click on Add Course, and then click on Save on an empty course. Notice that we receive an error back from the API. We get a 400 bad request back that the title is required. The form is broken now, though, because our Submit remains disabled, and the user can't see the error unless they happen to be a developer who likes checking the console. Yuck! So let's open apiServer.js and see how this works. If I scroll down here, our API has server-side validation that requires some basic information. It calls this validateCourse function. If we go to the bottom of the file, we can see that to validate a course, it checks to make sure that title, authorId, and category are populated. If we jump over to apiUtils.js, we can see that the error handler is already parsing error responses from the API. The handleError function then takes over, which logs the error in the console and then throws the error so that functions higher up the stack can handle it as desired. In a real app, you'd likely want to log the error to some error logging service as well. To handle this response, we need to catch the error in ManageCoursePage. We're calling saveCourse, which returns a promise, but notice we don't have any error handling here because we haven't added a catch yet. So let's add a .catch. The catch will receive an error as an argument, and the first thing that we should do is set saving to false because we're going to stay on this page if an error occurs, so we

want to make sure that we re-enable the Save button so somebody can try again. Then, we can call setErrors, which was declared up above in our use state call, and we can set the onSave error to error.message. I'll hit Save, and it reformats. Now notice that I'm setting an onSave property on our errors object here on line 58, and I'm setting it to the error message. To understand why I'm setting onSave, it's helpful to go look at the course form. At the top of our course form, we have a section that looks for errors.onSave, and then displays those errors within a nicely styled div. And again, we didn't have to write this code, we just pulled this out of the course exercise files to save some time. But thankfully this is already wired up to handle displaying our errors. So, we don't need to change any other code in order to display our validation message now. Let's go to the browser and try it out. Let's hit Save on this empty form and see what happens. We wait our 2 seconds, and great, there we go. Our server-side validation message was parsed properly, handled within our catch statement, and then displayed up here at the top of our form. So we now have server-side validation from an API displaying error messages. This is much better, but there is a subtle error that's lurking behind the scenes. After we get this validation error, if I come over here and click on courses, uh-oh, there's our spinner, and it never goes away. Why? Well, because Redux thinks that there's still an API call in progress. We can see this if I click over on the Redux dev tools, and I click on the State tab here, we can see that there's still one API call in progress. The real problem is when an API call fails, our Redux store isn't notified that one less API call is in progress. So we need to enhance our Redux configuration so that it decrements the number of API calls in progress if an API call fails. Let's do that next. If we open courseActions and look at the saveCourse thunk, we can see that we're currently simply throwing the error. Redux isn't being notified that the API call is completed, since we only do so for the success case. Remember, any action type that ends in success decrements the count of API calls in progress. So, we need to also decrement the count of API calls in progress when the API call fails. That's why the spinner is continuing to display. The fix for this is simple. We need to dispatch an action when an API call errors. So let's open up actionTypes.js and add another actionType for API_CALL_ERROR. Then, let's add a corresponding action creator in apiStatusActions.js. Simple enough. Now we can jump over to the apiStatusReducer. Let's update this to handle the API call error the same way that it handles a success. So right here we'll say if the action.type is equal to types.API_CALL_ERROR, or the actionTypeEndsInSuccess, decrement the state by 1. So now both errored API calls and successful API calls will decrement the number of API calls in progress. Now that we've updated our actions and reducers to handle API call errors, let's update our saveCourse thunk to dispatch this action when an error occurs. So, open courseActions, and then at the top we need to import apiCallError from our apiStatusActions, and then let's jump back down. Within our catch, what we need to do is dispatch apiCallError and pass it the error that we

received. This should do the trick. Now if an error occurs, our API status will still get updated to reflect the completed call. Let's give this a shot. I'll click on Add Course. Come over, hit Save. We saw our server-side validation, and if I click over here on state, the number of AJAX calls in progress is now 0, just like it should be. So if I click over here on Courses, I no longer see that spinner there either. Now that we found a pattern that works, let's jump back in the code and enhance our other thunks as well. I'm going to copy this line and then go up here to our catch for our other thunk and do the same dispatch here. Then let's jump over to authorActions.js and make the same change. Come to the top, import apiCallError, and then down here on our catch dispatch apiCallError and pass the error. Now our app properly handles apiCallErrors, but it would be nice to avoid getting such errors at all, so in the next clip, let's add client-side validation.

## Implement Client-side Validation

We're not doing any client-side validation yet, so if you click on addCourse and hit Save, you have to wait for the server to respond to know whether you have any validation errors. Let's add client-side validation so that users get validation feedback immediately instead of having to wait for the server to respond. Open up manageCourse page and let's add a new function above handleSave called formIsValid. This is a pattern that I enjoy for handling errors. First, I'm destructuring here at the top, just to shorten my calls below. I declare an empty errors object, and if I find an error, then I set a property on the errors object that corresponds to the field's property in state. I call setErrors and pass it the errors object. Remember, errors is already initialized as an empty object since we declared it in a previous module. Finally, I return a Boolean to determine if the form is valid. The form is valid if the errors object has no properties. I use Object.keys to do this. It returns an array of an object's properties. So if there are no properties on the object, then I know that no errors were found. Now that we've declared our validation function, let's call it down here in handleSave. We'll call it at the top, and we'll say if the form is not valid, then there's nothing more to do here. We will return. And the errors will display because up on line 56, we called setErrors to update state. So if the form isn't valid, we can just return early. If we scroll down, we can see that we're already passing an errors object over to the course form. And if we open the course form, you can see that each of the inputs accept an error prop, and note that the error's property that it's looking for corresponds with the name of the input. So errors.title is for the input of title. So, we're setting the relevant error for each input. Let's try this in the browser. Now when I hit save, there we go. Instant feedback. We didn't have to wait for our slow API. And if I fill out the form, we should still be able to submit it as expected. Great! It worked, and there's our new course. To close out this module on asynchroning and errors, I want to show you a handy way to make your

app feel faster. We're going to use a pattern called optimistic updates in order to handle deleting our courses.

## Optimistic Deletes

There's one obvious feature missing from our application. We're loading, saving, and updating courses, but what about deletes? Let's add support for deletes, but handle deletes optimistically. Let me explain. Currently when the user saves or updates an existing course, they have to wait for the API call to complete before they're redirected back to the course list page where they can continue working. This assures that we don't show them the updated record until we know that the save completed successfully. However, we can handle deletes a bit differently. Instead of waiting for the API call to complete, we can do an optimistic delete. The word optimism is all about assuming the best. And that's just what we're going to do. Instead of waiting for the API call to complete, let's remove the course from the UI immediately. This is a popular technique for making user interfaces feel extremely responsive. To set this up, let's begin by creating a new actionType constant. I'm going to add it down here with a nice long comment that explains what's going on. We're going to call this DELETE_COURSE_OPTIMISTIC, since we're being optimistic here. We shouldn't call it success because we're going to be optimistic and update the store immediately without waiting for the delete API call to complete. This name choice is also important because remember, in our apiCallStatus reducer, it's currently looking for any action types that end in _SUCCESS and reducing the number of API calls in progress. If we had kept the old name, then our API status would be incorrect because we'd remove one call from the count, even though we never added a call to the count when the AJAX call was initiated. So I'm including this comment above this action type to help clarify my intent. Now let's open up the courseActions file and create a new thunk for handling deletes. I'll add it down here at the bottom. Note how this differs from our other thunks. I'm immediately dispatching the deleteCourseOptimistic action so that our UI is immediately updated to reflect this deletion. Also, note that I'm not dispatching beginApiCall, or handling any API call errors. Why? Well, since we're going to immediately respond to the user's click without waiting for the API call to complete, there's no need to display the preloader. Next, let's create the action creator at the top for deleteCourseOptimistic. Now that we have our action creator, there's one final piece. We need to update courseReducer. Down here, let's enhance courseReducer to handle this new action type. We'll have a case of types.DELETE_COURSE_OPTIMISTIC. For this, what we want to do is return the array of courses, but omit the one course that was just deleted. To do that, let's use JavaScript's filter function, which accepts a predicate. A predicate is a function that returns true

or false. So what we will do is compare to course.id, and we want to make sure that the course.id is not the action.course .id. At the beginning of this line, I did forget to put in my return statement. I'll hit Save. So this will return an array of all the courses, but with one course omitted, the course that has the course ID that we're trying to delete. JavaScript's built-in filter function is handy because it returns a new array. So we don't have to worry about mutating state here. We're returning a new array with one course omitted from the list. With the Redux portion now set up, let's jump over to CoursesPage.js. It makes sense to have the Delete button on our list of courses. So first, let's go down to mapDispatchToProps. We're going to need a new action, and that action's going to be deleteCourse. Then let's go up into our component and implement a handleDeleteCourse method up above render. We want to output a success message to say that the course was deleted. And notice that I'm outputting this success message before I even fire off the action to delete the course because we aren't going to wait for that action to succeed. We're going to immediately tell the user that the course has been deleted. And this action will fire immediately as well. It won't wait for that API call to complete. We do need to jump to the top of the file and import toast from react-toastify. And down here within render, let's go down to CourseList, and we need to pass this new prop over to CourseList. It will now receive onDeleteClick as a new prop. And we'll pass it this.handleDeleteCourse. Finally, we need to update the CourseList component to handle a delete course link. So, open up CourseList. At the bottom, let's add that new propType. Then, let's jump to the top of the file and destructure this new prop. There's onDeleteClick. Let's add an extra column where we will place our button. Don't need a header for this one. Finally, let's scroll down and implement our button. So we'll do that as the final column right here. I'll just paste it in. So we have a button with some bootstrap classes on it to make it look nice. I have the onClick handler that will pass in the course, and then a label of Delete. We should be all set. Let's hit Save and see if this works in the browser. Our Delete button is rendering. When I click it, hey, look at that! It immediately deletes the course. Or at least it immediately removes it from the list, and we see our toasts pop in right away, even though behind the scenes, these network calls are still in progress. If I come in here, go to network, I can click Delete a couple times, and I can see still pending, still pending. It takes it a little bit of time to actually handle that delete, but we get instant responsiveness up in the UI. So this can be a very handy pattern if you have slow APIs. Now you might be thinking, isn't this dangerous? Aren't we lying to the user? And what happens if the API call fails? Let's finish this module by exploring how to handle errors on optimistic calls like this. Optimistic calls are a trade-off. It makes the app feel responsive, but if the API call ultimately fails, then we need to tell the user, which can admittedly lead to some confusion. If this call to deleteCourse fails, we're not currently handling it. So, let's add a catch to handle errors. We'll call toast.error this time instead, so this will style our toast in

red, and say that the Delete failed. Then, we will tack the error message onto the end. The toast also accepts a second argument where we can pass options, and I will set autoClose equal to false. Now if the API call fails, the user will be notified with a toast that remains on the screen until they close it. Let's deliberately break the delete call so that we can confirm that this works. Open up courseApi.js. And down here on 21, let's change this from courseId to some invalid number. I'll just put 1000 in here. So now we'll be trying to delete a record that doesn't exist. Let's try it out. I'll click on Delete. It says course deleted, but after 2 seconds when the API call actually fails, we get the Delete failed toast. So, yes, it's a bit unfortunate that this happens a moment after the success message. But this is the risk that we take to make the app feel more responsive by using an optimistic delete strategy. And before we finish up, be sure to open up the courseApi file and undo our breaking change in the deleteCourse function. To wrap up this module, let's explore an alternative way to handle asynchronous calls in JavaScript called Async/Await.

## Async/Await

Throughout the course, we've used promises to handle async calls. But we can also use async/await. Async/await is an alternative approach to handling async calls in JavaScript. It uses promises behind the scenes, so code that uses Async/Await can easily interact with other code that's returning promises. Here's an example function. With Async/Await, you must first declare the function as asynchronous via the async keyword. Then, when you make an asynchronous call inside, you can use the await keyword to wait for the response. The function will pause execution and continue when the async call completes. You can have multiple awaits in a single function. You can store the result of the async request in a variable and then return it. Since Async/Await uses promises behind the scenes, this line returns a promise. And unlike promises, you can use traditional try catch for error handling. To try out Async/Await, let's convert one of our existing functions that currently uses plain promises. Let me quickly show you how you can refactor the handleDeleteCourse method to use Async/Await instead. Add the async keyword. Then, with Async/Await, we handle errors differently. We use a try catch style. So, we will try this call, and I will await this promise. Since we're using a try catch style, we declare the catch using a traditional try catch instead of an arrow function. So with Async/Await, you decorate the function with the async keyword. This is a benefit of Async/Await, because async functions are clearly marked. You use the await keyword to decorate any async calls. The function will pause on each await keyword and continue when the async call is completed. This often leads to code that's easier to read. Just keep in mind that this is still using promises behind the scenes. You can think of

Async/Await as syntactic sugar over promises. And that's it. What a module! Let's wrap this one up with a summary.

### Summary

All right, our app is in great shape. We fixed our app issues. We added a loading indicator. We're displaying feedback upon clicking save via a toast. We're handling API failures. We added both server-side validation and client-side validation. And we implemented optimistic deletes to help make our app feel responsive, even if the API is slow. And we wrapped up by trying out Async/Await. Now there are two large topics that I've saved for the end; testing and production builds. So in the next two modules, let's explore how to create automated tests for both React and Redux.

# Testing React

### Intro

Testing the front-end can be tricky. With React, life has gotten significantly simpler. React is built on the idea that components produce HTML based on props and state. This simple design makes testing React components quite straightforward compared to many alternative UI designs. Here's the plan. If you're new to automated testing, just deciding what to use can be a major hurdle, so we'll begin by reviewing the long list of options to consider in this space, including testing frameworks and helper libraries. Once we've clarified our testing stack, we'll spend the rest of our time back in the editor writing tests. We'll use Jest as our testing framework, and we'll test the same React component using two popular options, Enzyme and React Testing Library. This way you can see the benefits of each approach. As you'll see, we can simulate a render without opening a real browser, pass properties to our component, simulate interactions, and assert on the results of these interactions. Let's begin by discussing testing frameworks and libraries.

### Testing Frameworks and Libraries

Our first decision is what testing framework to use. There's a wide variety of JavaScript testing frameworks available. Jest as from Facebook. Jest is easy to set up and has quickly become the most popular testing framework for React. Jest also comes bundled with Create React App.

Mocha is also popular because it's highly configurable and has a large ecosystem of support. Jasmine is similar to Mocha and Jest, but Mocha tends to be more configurable, so some people prefer Mocha over Jasmine. Tape is arguably the leanest and simplest library of the bunch. Its simplicity and minimal configuration are its key strengths. Tape also avoids the global variable configuration that exists in Jest, Mocha, and Jasmine. Finally, AVA is another interesting option to consider. We're going to use Jest because it's popular, easy to configure, fast, and offers an excellent command-line interface. Jest also has some unique features like Snapshot testing that allow us to easily generate and store serialized snapshots to help protect us from regressions. Helper libraries make testing React and Redux easier. Let's discus the three most popular options, which are React Test Utils, Enzyme, and React Testing Library. React Test Utils is a library that's specifically for testing React components. It's built and supported by Facebook. This library is powerful, but unfortunately it has a rather verbose API. React Test Utils offers two ways to render your React components. The shallowRender lets us render just the component that we're testing without rendering any of its children. That's why it's called shallowRender. It renders only one level deep. Shallow rendering is useful to constrain yourself to testing a component as a unit and ensure that your tests aren't indirectly asserting on behavior of child components. When testing with shallowRender, no DOM is required. ShallowRender returns an object that mirrors what we expect to see in a real DOM. But sometimes you want to simulate interactions with the DOM, such as clicks and change events. That's when shallowRender is no longer sufficient. Instead, you can use renderIntoDocument. It actually renders the component into the DOM. So this function requires a DOM to be present. Although this doesn't mean that you have to fire up an actual browser to use it. Libraries like JSDOM_)__ offer a simulated DOM that runs a node. That allows you to interact with the DOM as though you're in a real browser. Finally, by rendering into the document, you can simulate multiple interactions. However, I mentioned that React Test Utils has a rather clunky API. Let's just look at the functions for querying and interacting with the DOM. FindRenderDOMComponentWithTag is for finding a specific DOM element. ScryRenderedDOMComponents finds components by tag name. Yes, pretty weird name. And once you have a reference to an element, you can do things like simulate interactions. You could simulate clicks, keypresses, et cetera. Of course, there's much more to React Test Utils, and you can check out the docs on that, but due to the verbose and low-level API, many people prefer to use other popular alternatives. The ReactTestUtils documentation recommends my two preferred alternatives, Enzyme or React Testing Library. Let's contrast the API of Enzyme versus React Test Utils. With Enzyme, you just call find, and you pass it a css selector to be able to get what you want. Now to clarify, Enzyme is using React Test Utils behind the scenes, but it creates a much friendlier API on top of it. Enzyme's find function is very versatile because it accepts CSS

selectors, so if you already know how to write CSS, or if you've worked jQuery in the past, since jQuery also accepts CSS selectors, it's quite easy to learn Enzyme. And, of course, Enzyme's API offers much more than just find. Keep in mind that Enzyme is an abstraction. Just like a theater production behind the scenes, there's a lot going on. Ultimately, Enzyme is calling React Test Utils behind the scenes, and it uses JSDOM to create an in-memory DOM to simulate the browser. It also uses a library called Cheerio that provides those CSS selectors that I just mentioned. With Enzyme, these three libraries come together to create a powerful and concise testing API and environment without having to configure them to work together, and without having to open an actual browser to run our tests. React Testing Library is an alternative to Enzyme. It offers a much smaller API and helps encourage writing tests that resemble the way that your software is used. This tends to lead to tests that are less brittle and helps encourage writing accessible applications. Like Enzyme, we can simulate interactions with our React components without opening a browser. Both Enzyme and react-testing-library are great options for testing React components, so we'll test the same component using both of these technologies so that you can decide which library you prefer.

## Configure Jest

To configure Jest, let's start in package.json. First, let's add a script that will run our tests via Jest. Well, that couldn't get simpler, huh. Hit Save, then let's create our first test to prove that this setup works. I'll create it in the root of the src directory and call it index.test .js. We declare our tests with the it function. The first parameter is a description of our test. We're just going to say it should pass for now. And the second parameter is a function that contains the body of our test. I'm going to expect true to equal true. So I'm pretty confident this should pass. Let's open up the terminal and say npm t to run our tests. You could also say npm test or npm run test. Jest automatically finds any files that end in .spec .js or test.js. And great, we can see that our one test is now passing. And if I jump back in the code, set this to false, and hit Save, I can rerun our tests, and I should see it fail. And it does. So I'll undo my change here, keep our passing test. We don't want to have to rerun our tests manually. It'd be nice if they ran every time that we hit Save. So let's change our test script to enable watch mode for Jest. This tells Jest to watch our files and rerun our tests when we hit save. Now when I say npm t, it will continue watching our tests. We're also presented with a set of commands for running only a subset of our full test suite. Let's make one other tweak. To configure Jest, we can add a section to package.json. This tells Jest to ignore imports for various file types like images, videos, sounds, and styles. So although WebPack can handle these, this config tells Jest to ignore these imports. And don't waste your time trying to

type this; just pull this in from the course exercise files for this module. We also need to create fileMock.js and styleMock.js files that I'm referencing here. These files tell Jest to ignore any imports for these file types, so that you can write tests on files that import styles, images, and so on. First, let's create styleMock. There's not much here. So this tells Jest that if it is sees us importing CSS, it should ignore it. The other file we need to create is fileMock.js. And this tells Jest to stub our any file imports that might exist within our systems under test. All right, Jest is now configured and we've written our first test. Next, let's write a snapshot test using Jest.

## Test React with Jest Snapshot Tests

Jest offers a rather unique feature called snapshot testing. Snapshots store a record of a component's output. So, snapshots can be useful for documenting expected output and regression testing protection. Under the courses folder, let's create a new file. We'll call it CourseForm.Snapshots .test .js. To clarify, though, you don't have to have snapshots in the file name. I'm just putting it there to help label the different styles of tests that we're writing in this module. First, we need to import four items. We need to import React, since we're going to run a React component. Our system under test will be the COurseForm. We're going to use react-test-renderer to render our component, and I'm pulling in some mock course and author data for use in our test. This is why I like to centralize our declaration of mock data. It's useful for populating our mock API and our tests, so we don't have to repeat creating these data structures for our testing. We could simply create a snapshot test for a component to assure it renders, but honestly, I don't find that very useful. Instead, I like to create snapshots tests with helpful names to describe my intent. For our first snapshot tests, let's assure that the label on the Save button within our form is properly set when we set the saving prop to true. So, I will label this test as sets submit button label to Saving when saving is true. Remember, this button's label should change when we set the saving prop to true, so that the user knows that a save is in progress. To render our component, we're going to call renderer up above. Renderer returns a tree, which is an object that represents the output of our React component. We call the create method and pass it the component that we'd like to render, which is our CourseForm. Our CourseForm takes a number of different props, so I'll paste these in and close my tag. I'm using the mock data that we imported up above to get the first course out of the array of mock data, and passing in the array of authors also from mock data. Then I'm using jest.fn for the onSave and onChange props. Jest.fn creates an empty function, so that we don't have to declare our own for the test. You can also assert that this mock function has been called. Finally, I'm setting the saving prop. This is the whole point of our test. And remember, with Boolean props, the existence of the prop infers truth, so I don't

have to explicitly type = true here. Now we're ready to write our assertion. We will expect the tree to match snapshot. Jest has assertions built in. Assertions are how you declare expected behavior in your tests. With Jest, you call expect and chain it together with a variety of other methods that are listed here in their docs. I'm going to open up the terminal, and when I hit Save, we can see that Jest finds our new test and it writes a new snapshot to the filesystem. Snapshots are written in the same directory as the test under a folder called Snapshots with underscores on the front and the back, just to help assure that it doesn't conflict with other folder names. Let's take a look at the Snapshot file. Inside the Snapshot file, you can see the output of our component has been stored. This is the entire output from rendering the component with the props that we selected. And down here at the bottom, we can see the label for the Submit button is being set to saving as expected. Now let's write another Snapshot test that tests the output when we set saving to false. To do that, let's copy our existing test, paste it in, set saving to false, and change the name, sets submit button to Save when saving is false. So this test that it sets the submit button label to save when saving is false. Let's hit save again. When I open up the terminal, we can see that another snapshot test has been written. So let's go look at our Snapshot file. It wrote the second snapshot here at the top. And if I scroll down, we can see that our new snapshot has the label that we expected. Let me show you another little trick. If we jump back over to our test, I can hover my mouse over toMatchSnapshot and I can see the contents of that snapshot. To get this behavior, I'm running a handy VS code extension called snapshot-tools. It shows my snapshots in line. This is quite handy, especially if you have many snapshots because this is showing only this particular snapshot. Snapshot files can get rather long, so this avoids having to dig through the file. I can scroll through the full output right here. And I'm only seeing this particular snapshot. To assure that our snapshot tests work, let's open up CourseForm and change it. Let's jump down to where I set Saving, and let's say that I changed the label right here. When I hit Save, let's open up the tests, and now we can see that one of my snapshots have failed. If I scroll up a bit, we can see that it expected it to have saving..., but instead it received just the word saving. Now if this were an intentional change, I can hit the letter U. If I scroll down, we can see the output tells me this as well. So, if I hit the letter U, it will update my snapshot file. Notice how it now says that the snapshot has been updated. And if I come back over into the snapshot file, I should be able to see that reflected down here at the bottom. There it is. Notice how I no longer have the three dots. So the value of snapshot tests is they point out anytime that your React components rendering changes. So this can protect you from accidental changes. Let's go revert my changes, hit Save, and again, the snapshot test says, oh, we're not getting what we expected, because now we updated the Snapshot, it was expecting it to be without the dots. So I'm going to hit the letter U again, and it will update our snapshot test back to that original setting. And keep in mind, with

snapshots it's important to name the test well, so that when it fails, other developers are clear whether the snapshot test should be updated. All right, we've written a couple of snapshot tests. In the next clip, we're going to test the CourseForm component using two other popular testing approaches, Enzyme and react-testing-library.

## Test React with Enyzme

In this clip, let's configure Enzyme. To use Enzyme with React, we need to configure the Enzyme adapter. So in the Tools directory, let's create a new file called testSetup.js. Configuring Enzyme requires pulling in an adapter for the particular version of React that we're using. This configures Enzyme to work with React 16. As newer versions of React are released, you'll use the corresponding adapter for that version. We also need to tell Jest to call this adapter, so open up package.json and we'll add one more section here in the Jest config. Jest will run any items that we declare under the setupFiles array. I'm calling the test setupFile that we just configure for Enzyme. And with those quick changes, we're ready to write our first test in Enzyme next. First, let's add a test for the CourseForm component. So, let's create a new file under courses called CourseForm.Enzyme .test .js. Again, Enzyme doesn't need to be in the filename; Jest will find any files that end in .test .js. Let's begin by importing what we need. We need React, since we're going to be running React. The CourseForm will be our system under test, and we're going to use the shallow function from Enzyme. There are two ways to render a React component for testing with Enzyme, shallow and mount. Shallow renders a single component in isolation. With shallow rendering, so actual DOM is created. This makes shallow rendering a little faster than mount, and allows you to test a React component all by itself. Mount renders a component along with its children. We're going to use shallow first. First, I want to show you a handy pattern. I like to place a factory function at the top of my tests in order to call my React components with some default values. So let's paste that in first. Now this pattern isn't specific to Enzyme; in fact, we'll use it for react-testing-library as well. But the benefit of this pattern is it keeps each one of our tests below nice and simple because it avoids having to repeat all this boilerplate code for each test. In this factory function, I declare defaultProps as an object and I accept an object that contains arguments to override the defaults. I use the spread operator to blend the two together, then I render the component using Enzyme's shallow function. I use the spread operator to assign all the props to the component. So with this little helper function, our tests will be quite simple. First, let's add a test that confirms our CourseForm component renders a form and a header. To begin, let's call the render CourseForm, which will return a wrapper for our component. We don't need to pass any arguments to the function, since we're going to accept the default props that were

declared up above. And now we're ready to write our assertions. First, we expect wrapper.find form .length toBe 1. So we are expecting to find one form. I'm going to copy this line and make a little change here for our second line. We're expecting h2 .text, and instead of calling linked on this, we will expect the text to equal Add Course. Enzyme's find function allows us to find the form tag in the component. The find function accepts a CSS selector as an argument, so if you know CSS, then you know how to write selectors in Enzyme. For example, to find an element by ID, I'd append my selector with a pound sign. For example, pound sign, ID I'm targeting. To find an element by class, I'd append my selector with a dot; for example, .class I'm targeting. To find an element by tag name, you don't need a prefix character. In this test, I'm looking for the form tag, so I'm passing form to find. And we expect there to be only one form tag. Let me hit Save and open up the terminal so we can see the results. Uh-oh, looks like my test failed. Ah, I'm glad I saw this. So, I forgot to hit Ctrl+C to stop my previous test process. We have reconfigured Jest, so we need to hit Ctrl+C to stop the tests, and then hit npm t again to restart them. There we go. And our new test passes. When writing these tests, if you have any issues along the way, you can also use console.log. So right here, I could say console.log wrapper.debug, and this will output debugging output to our terminal. Now I can scroll through and see the markup that we're working with. This is what was returned from our render CourseForm, which is calling shallow on our component. Wrapper.debug can be really useful when you're wondering why your selectors aren't matching. Sometimes it's because it's not rendering what you expected. I'm going to comment this out. Let's add a second test down here below, and I want to show you another way that we could be a little more targeted in testing our Save button label's behavior. Here we're also testing that it labels save buttons as Save when not saving. So this is accomplishing the same goal as our snapshot test, but doing so in a more focused way. Let's make our test is passing. And good, looks like it is. Now let's write the same test, but with saving set to true. So the only difference between this test and the one above is that I'm passing a parameter to render CourseForm. This will end up setting the saving prop to true, so now we expect the button text to be saving … instead. Let's see if this passes. Good. Again, all our tests are passing. So we've seen a few examples of using Enzyme shallow. Let's try out mount. With Enzyme shallow rendering, no actual DOM is created. Only the component that you're testing is created in memory. So no child components are available. With mount, child components and the actual DOM are rendered. To see the difference, let's add a test for the header component. So I'll create a new file over here in the common folder. I'll call it Header.test .js. Let's begin with the necessary imports. We need React. Header is our system under test. I'm pulling in both mount and shallow because I'm going to show the difference between these two approaches in this file. And I'm importing MemoryRouter because the header component expects to be run as a child of react-router and

thus received react-router's props. This final import isn't necessary for shallow rendering, since shallow doesn't actually render the component. First, let's put in our shallow render. So this tests that three NavLinks are in our Header component. We render our Header component, and then we find the NavLink components that sit inside. So, notice that with shallow render, I can search for the React component tag. And then assert that there are three links inside. Let's contrast this test with a test that uses mount. Note that with shallow rendering, I search for the name of the React component, which is NavLink. And since no actual DOM is rendered, that makes sense. I'm basically searching on the raw JSX. But when I use Enzyme's mount, a full DOM is created in memory using JS DOM behind the scenes. So my NavLink is ultimately rendered into an anchor tag. That's why I'm searching for an anchor tag here on line 19. Also, with mount, I have to pull in React Router's memory router because the Header component expects to be run as a child of React Router and receive React Router's props. This isn't necessary with shallow, since shallow doesn't actually render the component. In summary, shallow rendering is fast and lightweight and it allows you to test a component in isolation. But mount creates a more realistic test by actually rendering the component and its children. So use mount if you want to test the final DOM, use refs, or test interactions with child components. Now that we've written these tests, if we open up the terminal, we can see that all of our tests are passing. Enzyme is very popular, but next I want to show an alternative to Enzyme that I also enjoy called React Testing Library.

## Test React with React Testing Library

React Testing Library is a compelling alternative to Enzyme. It's unique because it encourages you to write tests based on what your user sees. This tends to lead to tests that are less brittle than Enzyme's tests and it also helps encourage writing accessible React components. Earlier we wrote some tests for our course form using Enzyme. Now let's try writing the same tests using react-testing-library for comparison. Over here in courses, create a new file, name it CourseForm.ReactTestingLibrary .test .js. Let's begin with some imports. We'll need React. Then, react-testing-library gives us a render function, which we will use to render the component, as well as a cleanup function, which we need to run after each one of our tests, so we'll set that up in a moment. CourseForm is our system under test. First, let's wire up our cleanup function to run after each one of our tests. To do that, we will call afterEach and pass it cleanup. You can also centralize this configuration, but we'll just keep it simple here. Next, let's use the same pattern that we used in our Enzyme tests earlier. We'll add in a factory that will create our component, and this will help keep our tests below nice and simple. The only difference here is that we are calling render, which comes with react-testing-library instead of shallow, which we were calling

over in the Enzyme version. First, let's test that it should render the Add Course header. So we call our helper that will render the course form, and the render function up on line 18 returns an object with a number of different methods inside. So, we are using destructuring here on line 22 to get our hands on the getByText function. The react-testing-library docs provide a number of different methods for querying our React component's DOM. But in this case, we're looking for particular text. So this will search through our React component's output and look for the text Add Course. The getByText function has an assertion built in, so if it doesn't find the text that we pass, then it will fail. Next, let's write our test around the label for the Save button. Remember how we wrote those earlier in Enzyme. First, let's test that it should say Save when it's not saving. Again, I use getByText to find that label. Notice that I don't have to look for a particular element; I just tell it look anywhere in the output for this particular text. So this is a little less precise than what I was doing in Enzyme, but it also is a little less work. If Save weren't sufficiently unique here, then I could look for a particular ID or DOM element on the page as well. Finally, let's add the opposite test that checks to make sure that it says Saving... when it is saving. Notice again that I set Saving to true when I'm calling renderCourseForm here so that we can override the default setting up above. And again, I use getByText to make this assertion. If I open up the terminal, great, we can see that all of our tests are passing. Now unlike Enzyme, React Testing Library doesn't distinguish between shallow and mount. Components are always mounted. This is because React Testing Library has the philosophy that you should focus on what the end user sees. So, the component that you test and its children are rendered We just used getByText in these tests, but the render function returns a variety of queries for finding elements, such as getByTest ID, getByLabelText, getByAltText, or getByTitle. But notice that these queries focus on elements that the user can see. This helps encourage writing tests that are less brittle because it makes you less likely to test implementation details. I also like that unlike Enzyme, I don't have to explicitly call expect to make an assertion. By making my query, the assertion is automatic. One other feature that I appreciate on React Testing Library over Enzyme is the way the debugging is handled. I can destructure right here to get a reference to debug, which is also returned from renderCourseForm, and then I can call that function. When I do, if I open up the terminal, I get the output right here, but unlike Enzyme, the output is nicely color-coded, so this makes it easy to see what we're getting back from our call to render. Also, you can optionally pass in an element that you want to debug, but if you don't, then it outputs the results of callingRender. Of course, we've only scratched the surface on these two libraries, but I generally prefer React Testing Library since the tests tend to require less code and I like how it encourages focusing on what the user sees. We've seen a couple ways to test React components, and in the next module, we're going to shift our focus on how to test Redux. But first let's wrap up this module with a summary.

## Summary

In this module, we reviewed the long list of testing frameworks to choose from, but ultimately we used Jest. For helper libraries, we wrote the same test using both Enzyme and React Testing Library. We saw the many potential places that you can run React tests, including n browser__, via a headless browser, or in-memory. We ran in-memory tests using JS DOM's virtual DOM, and then ran them via Node. In this module, we wrote the simplest tests. We tested React presentation components. Now that we've seen how to test simple React components, in the next module, we'll wrap up our testing coverage by going deeper. We'll explore how to write unit and integration tests for Redux.

# Testing Redux

## Intro

We just saw how React's design lends itself very nicely to testing. And as you're about to see, Redux was designed with testing in mind from the beginning as well. In this module, we'll focus on how to test all the pieces of Redux. We'll write tests for React components that are connected to Redux, and all the discrete pieces of Redux, including action creators, thunks, reducers, and the Redux store, which is really just a final integration test. And we'll continue to use Jest to write our tests. Let's begin by writing a test for a connected component.

## Overview: Testing Connected React Components

To continue exploring testing, let's look at a more challenging problem. How do we test connected components? When testing React components, there's two core concerns to consider. First, we can test the component's markup. Given a certain set of props, we should expect to get certain output. For container components, the service area should be minimal here since we'll save our exploration of testing markup for the presentation components. Remember, markup belongs in presentation components, so ideally the only JSX in a container component is a reference to a child component. Second, we can test the behavior. Given a click, scroll, drag, change, what happens? Do we get the expected behavior? We've already seen some examples of testing the markup, so for testing connected components, let's focus on testing behavior. Our container components should have very little markup anyway, since that should be handled by our presentation components as much as possible in order to separate concerns. The tricky thing

about testing container components is they're all wrapped in a call to connect. And the connect function assumes that our app is ultimately wrapped in a provider component, so our container components don't export the component that we wrote. Instead, they export the component wrapped in a call to connect. There are two ways that we can handle this. First, we can wrap the container component with React Redux's provider component within our test. So with this approach, you actually reference the store and pass it to the provider and compose your component under the test. The advantage of this approach is you can actually create a custom store for the test. So this approach is useful if you want to test the Redux-related portions of your component. If you're merely interested in testing the component's rendering and local state-related behaviors, you can simply add a named export for the unconnected plain version of your component. As you'll see, if you forget to do one of these two things when trying to test a container component, you will at least get a handy error message that guides you in this direction.

## Testing Connected React Components

All right, let's test a connected component. Testing a connected React component can seem a little tricky at first. To see how to test a connected React component, let's create a new file here to test ManageCoursePage.js. Let's begin by pulling in the necessary imports. We need React. We're going to use mount from Enzyme, although I could also use React Testing Library here. We're going to use a variety of mock data from our mock data file, and ManageCoursePage is our system under test. First, let's declare a factory to render our component. This is the same pattern that we used in the previous module. Although I am showing another way to handle components that rely upon React router, here I'm just stubbing out history by putting in an empty object because this component will still render fine with that declared as an empty object. But I could, of course, pull in memory router as I showed in the previous module. Note that I'm calling mount, so this will render our component and its children in-memory. Also, note that I have to pass in all the props that the component requires, including those that were getting injected by Redux. We're going to run this component without Redux, so it's our responsibility to pass all the necessary props in. This is why I like to declare a centralized factory function. It helps keep the rest of our tests simple. First, let's write a test to assure that our form displays a validation error if someone tries to save the course without a title. Before I talk through the details of this test, note that I'm using mount rather than shallow up above. Remember, shallow is called shallow for a reason. It only renders one layer deep. We need to test this component's interactions with its child component; more specifically, the title input that sits in the CourseForm component. So, we need

to use mount so that a full DOM is created in-memory. Enzyme uses a library called JS DOM behind the scenes to create this in-memory DOM. So, let's talk through what this test does. I render the component. Then, I find the form and I simulate a form submit using Enzyme's simulate function. Simulate allows you to simulate user interactions like clicks and hovers. So, at this point, I've submitted an empty form, so an error should be displaying. Thus, I look for the error, which should have a class of alert. Since there may be multiple errors, I just look for the first one. Finally, I expect that the text of the error should be Title is required. So, hit Save and let's see if our test passes. Uh-oh, we get an error. If I scroll up, it's a pretty scary looking error. If I finally get to the top, it says could not find store in the context of Connect. Either wrap the root component in a provider or pass a custom React context provider to Provider and the corresponding React context consumer to Connect in connect options. Ooh, that's a mouthful! Now, thankfully that is quite helpful, though, so we have two options. We can wrap our component in Provider, or we can export the raw component. Here's what wrapping the Provider might look like. Of course, I would need to import the Provider component and also import a store and instantiate it to pass into the provider. Instead of doing this, though, I'd prefer to option two, which is to update our component so that it exports the raw, unconnected version. This will allow us to test our component directly without the complexity of setting up Redux's provider and store. So, let's open up ManageCoursePage to make this happen. So that we can test our component directly, we just need to add an export right here. So now this file exports two different things. It exports an unconnected component, and then down at the bottom of the file, it exports the connected version as the default export. Now that we're exporting our unconnected component, we can come over into our test and put it to use. Let's go up to the top of our test file and instead of importing the default export, we're going to import the named export, which is ManageCoursePage. So now line 4 is importing the plain component without it being wrapped in connect. The great thing about this approach is it doesn't break any existing import statements. Our default export will still work just fine. So if somebody imports the default export, that will continue to work. But adding these curly braces around it is a named import, which means I am no longer referencing the default export; I'm referencing the plain ManageCoursePage component. Now if we open up the terminal, we can see that our test is passing. However, there is a minor tweak to make, though. If I stop our tests and start the app, you can see that we now get a linting warning because ESLint is concerned that we're accidentally importing the wrong thing. So to solve this issue, we can open up App.js and we can see the ESLint warning right here on line 8, and I will disable it by adding this comment on the side. This allows me to disable a single line. Now when I hit Save, notice how down on the bottom of the screen we can see Compiled successfully. So, I've resolved that ESLint issue, and we can also see that the squiggly

went away. ESLint is just making sure that we deliberately want to import the default import from this file, since we are giving it the same name as the named import. It's a nice little safety check. So we've seen how to test a connected component; next, let's look at how to test mapStateToProps.

## Testing Action Creators

Action creators are so simple to test that unit testing them is a breeze. So, let's jump back in the editor and see how. Action creators are conceptually simple. They return an object. So unit testing an action creator is simple as well. We need to assert that it returns the expected object. Over here in our actions folder, let's create a new file called courseActions.test .js because we're going to write a test for one of our courseAction creators. Before we do this, I want to be honest. I find unit testing something this simple rather silly because we basically end up repeating ourselves verbatim in our test. I find the integration style store test that we'll write at the end of this module more useful, but let's go ahead and write this and you can decide whether it's worthwhile. To begin, let's pull in our imports. CourseActions is our system under test. We'll also need the action types from our actionTypes file, and we'll use the array of courses from our mockData. Notice that I'm using a describe block here to help organize our tests. I haven't shown this before, but it's optional if you find it useful. And if you run your tests in verbose mode, this extra describe block will provide some descriptive nesting in the output. We'll name this test should create a CREATE_COURSE_SUCCESS action. I'm going to use the arrange act assert pattern in this test, so first we will arrange our test. I'm going to test a variable called course to the first course in our mockData. Then, I will declare our expected action. This action will have a type of CREATE_COURSE_SUCCESS and the payload will contain the course. Now our arrange is complete, so we move onto the act. In here we will call courseActions.createCourseSuccess and pass it the course. Finally, we can close out our test by making our assertion. We expect the action to equal the expectedAction. So this test confirms that when I call the createCourse action creator with this specific course, I get an object back with the expected shape. And if you open the terminal and run the test, you should find that this passes. In the next clip, let's test a thunk.

## Testing Thunks

Thunks handle asynchrony. They often dispatch multiple actions and often interact with APIs. This makes them a little trickier to test. To test a thunk, it's going to require some mocking. We need to mock two things; our Redux store and any API calls that we make. We'll mock the store using

redux-mock-store and we'll mock our API calls using fetch-mock. Okay, back to the editor. Let's write our first thunk test. We just tested a simple synchronous action creator, but testing a thunk is a little more involved. We need to add three more imports up here at the top. We need to import redux-thunk. We'll use fetchMock to mock our fetch calls, and redux-mock-store to create a mock Redux store. Let's add our code to test an async action up here above our existing test. I'm going to begin with some configuration. ConfigureMockStore expects us to pass in an array of middleware. The only middleware that we need for our test is redux-thunk. So with these two lines, I've configured the redux-mock-store that we will use within our thunk. Next, let's use a describe block to group this test together. In this case, I'm going to show how we can nest describe blocks. So I'm going to place tests for Async Actions within this describe block, and every test that we put inside of here, this particular afterEach will run, reusing fetchMock to mock the fetch call that happens in our thunks. To keep our tests atomic, it's important to run fetchMock.restore after each test. This initializes fetchMock for each test. We're going to add our tests inside this describe block so that the afterEach will run for each of the thunk tests that we might add into here. Now let's add the body of our test. We're going to assert that our loadCourses thunk should create BEGIN_API_CALL and LOAD_COURSES_SUCCESS when we are loading courses. First, we configure fetchMock. We tell it to capture any fetch calls and return a body that contains an array of courses with the header set to application/json. So this mimics the response that our API would return, but avoids making an actual API call. So this will run instantly and reliably because it won't make a real API call. Then, I declare the actions that I expect to be fired from the thunk. I create a mock-redux-store by calling mockStore, and I initialize the store to contain an empty array of courses. Then, I dispatch the loadCourses action, which returns a promise. I call getActions on the mockStore, which returns a list of actions that have occurred. I assert that the list of actions matches the expectedActions we declared above. Yes, it takes quite a bit of code to test a thunk, but thankfully if you have multiple thunks, you can copy and paste this pattern and quickly tweak it for other thunks. Next, let's do something easier. Let's write a test for a Redux reducer.

## Testing Reducers

Oh, now here's one of my favorite things to test, reducers. Since reducers are pure functions, they naturally lend themselves to unit testing. In fact, the creator of Redux, Dan Abramov himself, said that "people who never wrote unit tests for front-end apps started writing them because it is just so easy to test reducers. " You don't need to mock any dependencies or simulate API calls because any new data that comes into your reducer comes in as an action. So, you can just call

the reducer with a state and an action and assert that its output matches what you expect. So testing reducers is as simple as it gets. Given this input, I expect this output. Since reducers have no side effects, they're easy to understand and test. Now let's jump into the code and create our first reducer test. Let's create some tests for our course reducer. To begin, over here in the reducers folder, create a new file called courseReducer.test .js. First, let's import the reducer that we need to test and the courseActions. Our courseReducer handles creating and updating courses. So, let's first create a test for how it handles the CREATE_COURSE_SUCCESS action. We're going to test that it should add a course when it's passed the CREATE_COURSE_SUCCESS action type. First, I arrange my test. Note that I'm not going to create a full course object here, since we don't need all the properties for this test. We just need an array and a couple of items to test this behavior. This is a principle I like to follow to keep tests readable. I try to leave out any information that isn't relevant to the test. In this case, the long list of properties on the object aren't relevant for assuring that our reducer handles a create properly. So I declare some initial data with these two courses, and then we declare a new course and the createCourseSuccess action. Now that we've arranged the necessary pieces, we can call the reducer and pass it the initial state and action that we declared. The reducer returns the new state. Since we just added a course, we expect there to be three courses. And the new course we added with the title of C should be the third one. We can do something similar to test the update. So let's add another test. Notice here that I need an ID property on our test course as well, since our reducer will use that information to filter out the existing course. I call the updateCourseSuccess action instead, and then within the act section, I get a reference to the updatedCourse and the untouchedCourse using array.find. I assert that the updatedCourse title has the value that we expect, which we configured up in the arrange section. And that the untouchedCourse still has its original title A, which we can configured up in initial state. Finally, I assert that the new state has a length of 3. This helps assure that we didn't accidentally add a new course to the array. Since we started out with 3 and we did an update, we would expect the new state's link to remain 3. Again, if you prefer to declare the full course object, that's just fine. You could also use our mockData if you prefer. As you can see, testing reducers is quite straightforward and one of the big benefits of using Redux. Let's wrap up our Redux test next by testing the store.

## Testing the Store

Let's wrap up this module by writing an automated integration test for the Redux store. When testing the store, we're really writing an integration test rather than a unit test because our goal is to assure that our actions, the store, and our reducers, are all interacting together as we

expected. So we're going to write some tests for the interaction of these three pieces. This is the last test that we're going to write, but frankly, I find this style gives me the most bang for the buck because it tests a lot of surface area with a small amount of code without being brittle in the process. All right, let's write an automated integration test for our Redux store. To test our store, we'll create an integration test rather than a unit test. We're going to test the interactions between an action creator, our store, and our reducer. To begin, over in the root of the Redux folder, create a new file called store.test .js. Going to begin by pulling in the necessary imports. We're going to use createStore, which comes with Redux. We'll pull in our rootReducer, our declaration of initialState, and our courseActions. We're going to test that our store should handle creating courses. Again, I'm following the arrange, act, assert pattern. In arrange, we create the store and pass it the rootReducer and initialState. We also declare a course object. Again, I'm omitting the other properties on a course to keep this test clean. Then, under act, we get the createCourseSuccess action and we dispatch it. Finally, under assert, we call getState to get the course from the store. And we expect it to match the course that we declared under arrange. We could even create an array of actions above and then assert the final result was what we expected. So, I could dispatch two createCourseSuccess actions and an updateCourseSuccess action, and then assert that the final store has two courses with the expected values. But I'll leave that as an exercise for you. And again, let's check the terminal and make sure it passes. All green. That's it! We've now seen how to test all the major pieces of React and Redux. Now it's time for a short summary.

## Summary

In this module, we wrote tests for each of our application's major file types. We tested our connected React components, and we tested a number of different portions of Redux, including action creators, thunks, reducers, and our Redux store. We're almost done! In the final capstone module, let's set up a production build process that prepares our app for actual use in the real world, and I'll close out the course with a list of challenges so that you can put these skills to use to enhance our demo app.

# Production Builds

## Intro

This may be the final module, but it's seriously important. Today's front-end development stack requires a modern and powerful production build to prepare the app for deployment. If you've opened up the Network tab in the browser, you might have noticed that the dev build for our application is seriously huge. That's obviously not going to work for production, so let's solve this. In this final module, you'll see how to make a real production build happen. To close out the course, let's set up our production build process.

## Production Build Plan Overview

Our development process doesn't generate any actual files. Everything is being served by Web Pack in memory. It just reads the files in the source directory and serves the processed files from memory. Now, obviously for production we need to write real physical files to the filesystem, so that a web server can serve them up. I prefer to follow the popular convention of having a source and a build folder where sources are source code and build is the production build. So we'll write our final production build to the build folder. Our goal for production is to bundle our entire application into three files, a minified and bundled JavaScript file, a minified and bundled .css file, and an index.html that references both of these files.

## Set up Production Redux Store

In this final module, we're going to prepare our application for production by creating an automated build that does the following. Runs our tests and lints our code, bundles and minifies our JavaScript and our CSS, generates source maps for both, so that we can debug any production issues, excludes dev-specific concerns, such as dev-only Redux store configurations, it should build React in production mode so that dev-specific features like property-type validation are eliminated for optimal performance, and the build will generate a bundle report so that we can see our app size and clearly see what packages are in our app's build. Finally, it will run our production build locally using a local webserver. This sounds like a lot of work, but you'll be surprised how little code we write. Let's make it happen. To begin preparing our app for production, let's configure our Redux store for production. Our current store contains code that we don't want to run in production, such as reduxImmutableStateInvariant, and the reduxDevTools configuration. So let's create a separate Redux store configuration for development and production. To begin, let's rename our existing configureStore to configureStore.dev .js. So this will remain our development configuration. Now let's copy the contents of this file and create a new file in the same folder called configureStore.prod .js. And

we'll paste in the development configuration as our starting point. Our production configuration will not need Redux dev tools, so we'll take that out. This means we also no longer have the composeEnhancers function, so we can remove that. This means we can also jump to the top and remove the compose import, as well as the import for reduxImmutableStateInvariant. And now the only middleware that we want to apply will be Redux thunk. So our production store configuration is much simpler. Now we just need to add a little code to call the proper store configuration based on our environment. So, let's create another file in this same directory called configureStore.js. For this file, I'm going to use CommonJS so that we can dynamically import during build time. We're going to look at the Node environment to determine whether to load the production configuration or the dev configuration. This pattern assures that our dev-related store configuration code isn't included in our production bundle. Now you might be wondering where the Node environment is configured. We'll take care of that in the next clip as we configure Web Pack for production.

## Set up Webpack

We already have a development webpack.config, and our production webpack.config will be quite similar, but it will differ in some key ways. Let's begin by copying our existing webpack.config, and then pasting it out here at the root. But I will rename it to webpack.config .prod .js. To begin, we need to import two more items at the top. We're going to use the MiniCssExtractPlugin, which will minify our CSS and extract it to a separate file. I'm also going to add in webpackBundleAnalyzer, which will create a handy report of what's in our bundle. Down here on line 7, we want to set the Node environment to production. Remember the code that we wrote in the previous clip is looking for this setting. On line 10, let's set the mode to production as well. This configures web pack with some useful defaults for production builds. For production, we'll use the recommended production source map, which is source-map. This is a little slower to create than the source map setting that we were using for development, but it's higher quality, so for production builds, it's worth the wait. Our entry point will remain the same, and our output settings will also remain the same, but unlike the dev config, web pack will actually write physical files to the build directory since we're in production mode. We don't need this dev server section for production, so we can remove all of that. Now let's tweak our plugin configuration. Remember, plugins enhance web pack's power. Under the define plugin, I'm going to add one more setting. The define plugin lets us define variables that are then made available to the libraries that web pack is building. React looks for this to determine if it should be built in production mode. Production mode omits development-specific features like property types for

performance and to help deliver a smaller bundle size. There's a few other plugins that I'd like to configure for our production build. We'll add them up here at the top. First, let's add webpackBundleAnalyzer. This is already imported up above. With this configuration, Webpack will automatically display a report of what's in our bundle when the build is completed. For production, we want to minify our CSS and extract it to a separate file. So let's add the MiniCssExtractPlugin. Webpack will pick the name for us and add a hash to it. This way the filename will only change when our CSS changes. So this supports setting far expires headers on your web server so your users only have to reload this file when it changes. The last plugin to configure is HtmlWebPackPlugin, which we've already been using in development. This plugin performs a number of functions. It generates our index.html, and it adds a reference to our JavaScript bundle and CSS bundle into the HTML for us. And this is handy, since the JS and CSS filenames will change over time, since they contain hashes so that we can cache them for a long time on our web server. For production, let's add a few more settings. With these extra settings, it also minifies our HTML, removes comments, and much more. These are all little performance enhancements to keep our HTML file as small as possible. For our final change, let's scroll down to our CSS config. I'm going to replace what we have here in our use statement with this. It's admittedly clunky to type, so I'm going to paste it in and then talk through. MiniCssExtractPlugin will extract our CSS to a separate file using the css-loader that you see here. And it will generate a sourceMap for debugging purposes. Down here, we're also using postcss, which can perform a variety of processing on CSS. We're using the cssnano postcss plugin to minify our CSS. Remember that loaders run from the bottom up. So the postcss-loader will run, and then the css-loader will take over, generate our sourceMap, and extract to a separate file. And that's it. Now that we have this config ready, of course we need to call it. Sounds like a job for an npm script. Let's do that next.

## Set up npm Scripts

To handle our production build, we need a few more npm scripts. First, let's add a script called build. This script will run Webpack and pass the config of webpack.config .prod .js. Add my comma on the previous line. This script will write to the build folder, so before we run the production build, it's a good idea to delete the previous build folder. So, let's add a script for that. We'll call it clean:build. This script we'll call the rimraf package, which is a cross-platform-friendly way to delete files, and we'll delete the build folder, and then we'll recreate the build folder using the mkdir command, which runs on all operating systems. It's also a good idea to run our tests before the production build, but our current test script runs our tests in watch mode. We only

need our test to run one time before the build to assure that they passed, so let's create a dedicated script that runs the test just once. We'll call it test:ci. Ci stands for continuous integration because this would also likely be a script you'd run on a continuous integration server. This calls jest without watch mode enabled, since we only want to run the test one time when we build. We'll use this to assure that we can't deploy an app with broken tests. Now let's create a script called prebuild. I'm going to place it up above the build script, although it's not required to sit here. I just like to put prescripts before the related script. This script will use run-p to run multiple npm scripts in parallel. Run-p is a command that comes with the npm run all package. We want to run the clean:build script, as well as the test.ci script before we run our build. So this will run both of these scripts at the same time, and if our tests fail, then the build will fail. We're almost done. It would also be handy to be able to run our production build locally to assure that the build runs properly. So, let's add a script that will serve our production build. We'll call it serve:build, and this will use a lightweight web server called http-server to serve what we write to the build folder. For convenience, we can run this server after our build is complete. We can do that by creating a script called postbuild. Like our prebuild script, this will run after the build script by convention because it has the same name, but with post on the front. We'll use run-p again to run a few scripts at the same time. We want to start our API and also serve our build. Of course, for a real production app, you'd want to hit a different API. Now that we have our script set up, let's give it a shot in the next clip.

## Run Production Build and Review Bundle Content

Okay, hit Save on your package.json and cross your fingers because it's time to see if this thing bursts into flames. Say npm run build. We can see all our tests run, and they all pass successfully. Then Webpack begins running our build. Running Webpack could take a moment, because remember it's minifying our code, which is a slow process that we only need to do for production. When our build is done, a tab opens automatically that contains our report from webpack-bundle-analyzer. Bundle-analyzer shows what's in our bundle. We can see that we're running the production version of React. The size of these boxes represents the size of the file. And for the big moment of truth, take your mouse and hover over bundle.js. Now we can see that our production build is only 64 KB. That's a huge improvement! Now that we're properly bundling and minifying our code, our app is only 63K g-zipped. That's about the size of this cat JPEG! Crazy! This lightweight result is the clear benefit of choosing focused libraries like React, React Router, and Redux. Nice work! And since each box represents its relative size in the bundle, we can see that React is about half of our bundle. We can also see that Redux is quite a small

dependency, only 2.2 K g-zipped. In fact, it's smaller than the react-toastify library that we're using to show toast, which is 5.6 K g-zipped. Over here on the right is the source code that we've written, which totals up to about 12 K. And as a side note, you could further improve the bundle size by code splitting your app. Check out React.lazy for how to do that.

## Run Production App Locally

If I open a second tab, I should be able to see our production build on 127.0 .0 .1 port 8080. Notice that this URL is output by http-server when it starts up our web server. We can see our app. We can click around. And it loads data successfully. Looks like we have a successful production build. Oh, and if you deploy this to production, be sure to configure your web server to direct all requests to index.html. This way client-side routing will take over and load the proper page. Otherwise, your route will fail to load when someone tries to load a deep link directly like /courses because your web server will look for a folder called courses that doesn't exist. The exact steps required will vary by web server, but on express, you can configure a single route that directs all requests to index.html. We're in good shape! I'd like to wrap up this course with a few challenges for you.

## Final Challenges

I believe that until you've really tried this on your own, you don't fully know it. So, I'd like to wrap up this course with a few final challenges for you. If you're looking for some ideas to try on your own, here are some suggestions. Add support for administering authors as well, and here's a hint; be sure to add some logic that makes sure you can't delete an author who has a course. Add filters for the course list at the top of the course list table. Hide the course list table when there are no courses. Message to the user if they try to leave the managed course form when they have unsaved changes. Enhance the client and server-side validation on the manage course form to be more strict about the data that you can enter. This is a surprisingly fun one; the challenge is showing a 404 on the ManageCourse page when an invalid course slug is provided in the URL. Here's a hint; add some logic to mapState to props. Show the number of courses in the header. This is a great example of how Redux's single store model pays off. You'll see that adding this is really trivial, and there's no worry of it getting out of sync. Add pagination to the course table in order to support large datasets. Sort the course table alphabetically by title by default, so that the last record updated or created isn't always placed at the bottom. Or to go a step beyond, add drop-downs above the table that allow you to sort by different columns. As a hint, mapState to

props is a good way to get this done. And the final challenge is try to keep the old course data so that users can view history and click undo to revert their changes, even after hitting Save. This should help you put your newfound skills to use. Happy coding! As we saw, a production build process requires some extra work, but it pays off huge for the end user. Our dev environment is around 1.8 MB, but it's only 63k g-zipped for production. That's a huge time and bandwidth savings, and it all happens via a single repeatable command. All right, you've made it! We've reached the end of our exploration of React and Redux. I'd love to hear your feedback in the course discussion. You can also reach me on Twitter at @housecor, or at reactjsconsulting.com. Congratulations! And seriously, thanks so much for watching!

## Course author

Cory House

Cory is the principal consultant at reactjsconsulting.com, where he has helped dozens of companies transition to React. Cory has trained over 10,000 software developers at events and businesses...

## Course info

| | |
|---|---|
| Level | Intermediate |
| Rating | ★★★★★ (1487) |
| My rating | ★★★★★ |
| Duration | 6h 39m |
| Updated | 12 Mar 2019 |

## Share course