Building Applications with React and Flux

by Cory House

Start Course

Bookmark Add to Channel Download Course

Table of contents Description Transcript Exercise files Discussion Learnin

Course Overview

Course Overview

Hi everyone, my name is Cory house, and welcome to my course, Building Applications with React and Flux. I'm the principal consultant at reactjsconsulting.com. I've enjoyed working heavily with React since it was open source in 2013. React is one of the world's most popular technologies for building web apps today and for good reason. It's fast, it offers an elegant programming model, and it boasts a huge ecosystem of existing libraries and components. In this course, we're going to build a realistic data management app to explore the core features of React, Flux, and React Router. Some of the major topics that we'll cover include generating new React apps with create-react-app, handling state and props, performing client-side routing with React Router, creating reusable React components, and implementing one-way data flows with Flux. By the end of this course, you'll know how to build a realistic and scalable React app that supports creating, updating, and deleting data. Before beginning the course, you should be familiar with JavaScript, HTML, and some basic CSS. I'm assuming that you're new to React, and don't worry if you aren't up to speed on all the latest JavaScript features. I'll introduce modern JavaScript features along the way. I hope you'll join me on this journey to learn React, React Router, and Flux with the Building Applications with React and Flux course, at Pluralsight.

Intro

Intro

Hi, and welcome to Building Applications with React and Flux. I'm Cory House. In this course, we're going to use React, React Router, Flux, and a variety of related technologies to build a realistic course management application. You can find me on Twitter as @housecor or on my consulting site at reactjsconsulting.com. So who is this course for? Well, there are three core audiences for this course. If you're new to React and want to learn it by building a real app from the ground up, then this course is for you. I assume no previous knowledge of React. And if you're building a real app with React, you'll likely need to define routes so that users can navigate through your app, share deep links, and transition smoothly between different app states. React Router is a popular routing option that we'll use in this course to handle our routing. Finally, maybe you enjoy React, but you haven't yet tried out Flux. In this course, you'll learn how to implement unidirectional data flows using the Flux pattern, including actions, dispatchers, stores, and the various new jargon that comes along with using Flux. We'll explore the core concepts behind Flux and then use it to handle events and data flows in our app. Next, let's briefly look at the high-level course outline.

Course Outline

Here's the high-level course outline. In the first half of the course, we'll quickly set up our development environment. Then we'll explore React from scratch, including core concepts like JSX, components, lifecycle methods, Hooks, reusable components, component composition, and forms. We'll also implement rich client-side routing using React Router, the most popular routing solution for React. In the final two modules, we'll explore Flux in detail, including actions, dispatchers, and stores. We'll clean up our data handling by utilizing the Flux pattern for unidirectional data flows. Throughout the course, we'll build a course management application that supports creating, reading, updating, and deleting course data using these technologies. When we're done, we'll have a clean, scalable solution for rich, component-based web applications. And at the end of the course, I'll wrap up with a short challenge for you to put your new-found skills to work by enhancing this new application that we've created. In this module, I'll begin by clarifying the target audience for this course. Then I'll answer an important first question, why React? We'll explore a few innovative ideas that React and Flux help popularize. And I'll provide a high-level introduction to the technologies that we'll be using throughout the course.

Who Is This Course For?

You might wonder how this course differs from my React and Redux course. These two courses are designed to be complementary. This course is for beginners, and my React and Redux course is a bit more advanced. In this course, I introduce React, React Router, and Flux for people who are completely new to these technologies, and in my Redux course I assume that you already know React. Of course, we'll use Flux for data management in this course, and I use Redux in my more advanced React and Redux course. We'll use create-react-app to quickly get started in this course. In my more advanced React and Redux course I show how to build your own custom development environment from scratch using Babel, webpack, ESLint, Node, npm, and more. I use React Router in both these courses, but in this course I'm going to introduce React Router. In the Redux course, I assume you already know React Router. To keep things simple, in this course, we won't cover testing. I explore automated testing using Jest, Enzyme, and React Testing Library in the Redux course. Finally, I deliberately build a very similar app in both of these courses. I'm doing this deliberately so that you can easily compare the final result of these two technical approaches.

Why React?

Let's begin by answering the big question, why React? These days, there's a wide variety of ways to create rich web apps using components. Some of these are pure component libraries while others listed here are full-fledged application frameworks. But the point is each offers a unique way to handle web components. Of course, React is a popular option to consider. But what makes React special? Why should you choose it over all these other options? Let's consider the unique qualities that React has to offer. React is a project from Facebook that handles a very specific concern, creating components. Because of this focus, React is lightweight. This makes it easy to create complex UIs by composing many simple components together. Since React is so focused on this one goal, it can even be used along with other frameworks. So why should you consider React over all the other component options out there? Well, React is known for being extremely fast and responsive. React can scale to extremely large and complex UIs because it's very efficient about how and when it manipulates the DOM. React is designed to help eliminate layout thrash by using a virtual DOM behind the scenes. React offers a simple model for composing components. You can easily nest components within other components and pass data down to child components in a way that looks and operates very similar to HTML attributes. Unlike larger, full-featured frameworks like Angular, Vue, and Ember, React is easily pluggable into existing applications too. Since it's just a view layer, it can be easily integrated with other

technologies. So you can use React in certain places to try it out, or, of course, you can create an entire app using React, which is what we're going to do in this course. React components are isomorphic friendly. This is a fancy way of saying that they can be rendered on both the client and the server. React doesn't need the DOM in order to render. Isomorphic rendering can increase perceived loading performance, it avoids repeating code on the client and the server, and it offers a simple path to search engine optimization. Since React is a very focused library, it's simple to learn. The API is small, and there aren't many moving parts and concepts to master. Finally, React is battle-proven. It was created by Facebook and is used extensively on facebook.com, which is one of the most popular websites in the world. Today, facebook.com uses over 50, 000 React components. As you'll see in this course, React and Flux offer a number of interesting innovations, including JSX, which involves writing your markup in JavaScript; the virtual DOM, which enhances performance and efficiency by minimizing expensive updates to the DOM; isomorphic, also known as universal rendering, which allows you to render your React opponents on both the client and the server; and unidirectional data flows, which make your app easier to reason about by handling all your data flows in a predictable, explicit manner. Now, to clarify, I'm not claiming that React and Flux are the first libraries to utilize these patterns and technologies, but React and Flux were the first libraries to reach critical mass and help bring these approaches to the forefront. I like to joke that Facebook called it React for a reason. You see, when React was released, it created strong reactions. This tweet was in response to the initial announcement of React. Many people didn't take React seriously at first because it was so different from everything that came before. React and Flux deliberately ignored some commonly held best practices, so if you're new to React and Flux, I'm going to have to ask you to keep an open mind. Try to limit your initial urge to rage, quit, and run out the door. If you're willing to listen with an open mind today, I hope that I can sell you on rethinking the way that you're doing front-end development. Here's a few controversial ideas that we're going to cover in this course. First, React believes that HTML should be a projection of your application state rather than a source of truth. If you've ever worked in jQuery, then you probably understand what I mean here. jQuery caused us to make this mistake for years. We stored data in the DOM and data in JavaScript, so at any given point which was the system of record? Any time that the JS changed, all of our UI had to be redrawn. React's virtual DOM changes how you think about your app. Your app is a function of props and state. You don't store anything in the DOM. It's simply an implementation detail. Second, as you'll see, React takes the stance that JavaScript and HTML belong in the same file. For years, we've talked about separation of concerns as a common best practice, but is separation of concerns always a good thing? JavaScript and HTML are fundamentally intertwined, yet there's no strongly typed interface to help keep them in sync. So, as you'll see, there's some clear benefits to handling your

JavaScript and your markup in the same file. As we move on to Flux, you'll see that there are some key advantages to unidirectional data flows as opposed to the two-way binding that you might have experienced with other frameworks. Unidirectional data flows make our application easier to think about and will offer us a simple path toward updating our data stores when our app's data changes. Now this sounds crazy, but when using React, inline styles are actually useful again because they can actually increase code maintainability by avoiding the creation of bloated style sheets that are risky to change. React components allow you to utilize the power of JavaScript to write dynamic, deterministic, and component-specific styles that are potentially easier to maintain than traditional style sheets. And brace yourself because we've entered a new era. In React, all variables should be global. Global variables are awesome. Okay, I'm kidding. This was just a test to make sure you're paying attention. Hopefully you didn't just rage, quit, and turn off the course. In actuality, React with Flux does an excellent job of encapsulating and managing state, but the other previous points that I just mentioned are indeed legitimate arguments, and I'm going to make these throughout the course. And I want to clarify that while these ideas were initially controversial, many well respected companies have embraced React and are using it on large scale production systems today. Not just Facebook, but many other names you'd likely recognize like PayPal, Dropbox, Netflix, Amazon, and Microsoft. So React may be new to you, but it's actually a battle-proven library that's been used in production since 2013. Today, many of the world's most respected companies use React.

Tech Overview

One of the hardest parts of creating React app is making a decision on all the related libraries to build a full application. Remember, React is merely a library for building UI components, so it doesn't have strong opinions on how to handle routing, builds, deployments, and so on. Let's walk through the core technologies that we'll use to build our app. Of course, React will be our component library. We'll use React Router to handle client-side routing, and we'll use Facebook's Flux to handle our apps data flows. Let me briefly introduce React Router and Flux next.

React Router Overview

Since React is a small, focused view engine, it has no opinion on how to handle routing built in, and for small apps, you may have no need for a fully-featured router at all. But as your app grows, you'll likely find that you want to split your application into multiple, client-rendered pages with deep linking. And that's where a client-side router becomes handy. React Router handles routing

client side so that your app doesn't have to post back to the server. This means that your page transitions are typically instantaneous, which is quite nice. React Router offers a simple, declarative approach to routing. You configure your routes using React components that React Router provides, and you can nest your routes, which means that it can support complex UIs that are many layers deep.

Flux Overview

Flux is a unidirectional data flow library created by Facebook to complement React. This means the data updates flow in one direction unlike two-way binding libraries that automatically try to keep the UI and data in sync. This tends to make applications easier to debug and scale by avoiding complex interactions that can occur between multiple view and view models in traditional two-way binding approaches. Flux is more of a pattern than a library. It's really just a branding name for Facebook's implementation of a unidirectional data flow pattern. And there are many alternatives to Flux, the most popular being Redux.

Picking a Development Environment

When starting a new React project, you need a development environment. Today, there are literally hundreds of starter kits and boilerplates to choose from for React. The site javascriptstuff.com documents around 200 starter projects for React. Back in 2015, I created my own that I called react-slingshot; however, shortly after I created my boilerplate, Facebook published their own called create-react-app. Today, it's the most popular way to create a new React app. I highly recommend create-react-app, so it's what we're going to use within this course. However, you don't have to use an existing boilerplate. It's surprisingly easy to create your own development environment, and there are some advantages to doing so. First, it helps you understand how all the technologies that we're using with React fit together, and with this knowledge, you'll feel more comfortable customizing your dev environment of choice later. I show how to create a React development environment from scratch in my React and Redux course. But for this course, we're going to use an existing development environment. We'll use create-react-app because it's the most popular way to create a React app today. This is a powerful and reliable development environment that's supported by the Facebook team. Create-react-app requires no upfront setup, and it's easy to use, so it's very friendly for people who are new to React. We'll set this up in the next module. Create-react-app bundles many of the most popular development tools together into a luxurious, opinionated, and rapid feedback development environment.

Behind the scenes, it uses these technologies. Let me briefly introduce these technologies in the next few clips.

Node and npm Overview

Building an app with React and Flux doesn't require using Node, but Node certainly helps make our job easier. Node allows us to run JavaScript on the server using Google's fast V8 engine. Node now includes npm, which is Node's package manager. Npm includes hundreds of thousands of useful JavaScript libraries, so we'll use in npm to pull down the libraries that we need for our project. And if you haven't used Node, don't worry. We'll walk through the few things that you need to know along the way. Node comes with its own package manager called npm. Now, npm is how we'll install and manage the dependencies that we need to run our app. You're likely already familiar with the concept of a package manager if you code in a server-side language. .NET developers use NuGet, Ruby has RubyGems, Java has Maven, Python, pip, PHP, PEAR, and Node's package manager is npm. Npm provides a quick, easy way to install Node packages. And once you've installed Node, you simply type npm and the name of the package that you'd like to install.

Babel Overview

In this course, we're going to use modern JavaScript features that may not be supported in older browsers, so Babel will transpile our code so that it runs in all browsers. With Babel, we can use all the latest JavaScript features and even some experimental features that aren't technically part of JavaScript yet. Our code will still run fine cross-browser. Babel is built in to create-react-app, so Babel will convert our code behind the scenes to ensure that it will magically work everywhere. Let me step back for a moment because to understand where JavaScript is and where it's headed, it helps to understand its history. Version 1 of JavaScript was born in 1997. It was created in a few short days by Brendan Eich. Who would have thought that it would have such longevity? A year later, ES2 was released, and a year later we saw ES3. So far, so good. But then they tried to tackle a much more aggressive release, and no one could come to an agreement, so ultimately ES4 never happened. JavaScript stagnated for a full decade before ES5 was finally released with a number of notable enhancements. Then the waiting game began again. It took another six years, but ES6, also known as ES2015, was worth the wait. We picked up tons of features, including arrow functions, const, let, modules, classes, destructuring, generators, and much more. The other huge piece of news was a commitment to doing annual releases starting in 2015. The

ES6 release was so huge that it apparently wore everyone out because ES2016 only had two features, the exponent operator and array.prototype .includes. So not much, but hey, this sure beats 10 years of stagnation. ES2017 added async/await for handling async code, and ES2018 added rest/spread and dynamic imports. Many people are already enjoying these features today by using transpilers like Babel that automatically add in support for older browsers. Today, most browsers have great support for the vast majority of modern JavaScript features; however, there are still some obvious holes, such as ES import support. And if you have to support Internet Explorer, it lacks support for most of the ES2015 and newer features that I just mentioned. So that's the beauty of Babel. We can use all these modern features, and it will work in all browsers.

Webpack Overview

Webpack is a popular and seriously powerful bundler for the web. Webpack can bundle npm packages so that they run in the browser, and you can even bundle styles, images, and more using webpack. You can import these sorts of assets just like you to JavaScript files, and webpack will bundle them for you. And the good news is you don't have to configure Webpack yourself because it's built in to create-react-app, but I wanted you to be aware that it's running behind the scenes when we start our app.

ESLint Overview

ESLint is a linter. It will check our code for common issues. ESLint finds common typos and syntax errors, and it can also enforce your coding standards. When we start create-react-app, ESLint will watch our files and report any issues that it finds on the command line. ESLint is highly configurable, and you can extend its behavior by adding plugins, but we'll use the default behavior that comes with create-react-app in this course. So that concludes an overview of the technology that we'll be using. Let's wrap up this module with a summary.

Summary

Let's wrap up this module by reviewing what we've covered so far. We began by considering what makes React special. React is fast, composable, simple, and battle-proven. It's used by some of the best-known companies in the world today. React includes many interesting features, including JSX and the virtual DOM. We reviewed the core technologies that we'll be using to create our example app: React for components, React Router for routing, Flux for data flows, Node and npm for packages and automation, Babel for transpiling both our modern JavaScript

and out React components, sebpack to bundle our application up for the browser, and ESLint to catch errors along the way. In the next module, we'll use the most popular option for creating React applications today, which is create-react-app. With a single command, we'll set up a development environment with nearly all of this built in.

Environment Setup

Intro

These days, there's a wonderful variety of free tools available to do rapid client-side development. In the previous module, we discussed a few of these tools. In this module, we'll set up our environment so that we receive rapid feedback as we build our React app. In this module, we'll install Node. We'll install VS Code as our editor, although you're free to use the editor that you like best. We'll configure Prettier to autoformat our code. Then we'll run create-react-app to generate our app. We'll run the app and review what's inside. We'll also add a few other packages that we're going to use throughout this course, including React Router for routing, Flux for state management, and Bootstrap for styling. We'll close out the module by configuring a mock API that will host our course and author data that we're going to use to build our course management application.

Install Node

To get started with our environment setup, install Node. We'll use Node to run our development environment. Make sure you're running at least Node 8. To see what version of Node you're currently running, you can open the command line and type node space -v. If you already have Node installed and you're 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 in nvm on Mac or nvm-windows on Windows.

Install VS Code

In this course, I'm going to use Visual Studio Code. VS Code has quickly become extremely popular for JavaScript developers. You don't need to use VS Code to follow along, but it's a free

download if you want to try it, and I'll show 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.

Configure VS Code Extensions: Prettier and ESLint

In this course, I'm going to use Prettier. Prettier is a code formatter. With Prettier, you don't have to mess with indentation or long lines. You don't have to think about tabs versus spaces. In fact, you don't have to think about formatting your code at all because Prettier formats your code for you. Throughout the course, I'm going to use the Prettier extension 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. Linting is important so that you're quickly notified of potential issues in your code. ESLint is an extremely popular way to lint JavaScript. ESLint is built into create-react-app, and it will alert us when we make mistakes, so we'll put it to use throughout the course. I'm running the ESLint extension in VS Code. This gives me feedback in the editor. When linting issues are found, the code will be underlined with a squiggly line, and the warning or error will be displayed in the editor when I hover over that line. I suggest installing this extension so that your editor works the same as mine. Without this extension, you'll need to watch the terminal to see the output from ESLint.

Run create-react-app

[Autogenerated] Let's get rolling in this clip. Let's use, create React up to create or react application. To begin open a command line in the directory where you'd like to start building your app. Then run this command to generate your app via create react out. So hit enter To execute this command the N p X command executes, create react app and automatically downloads create react at If you haven't run this before, this command tells, create react up to generate a new application called PS dash flux. You can give it a different name if you like. By changing that last parameter, the install process might take a couple minutes, but all speeded up for this video. If you see this output than the install is complete, great react up places our application and a folder that matches the app name that we chose. Let's open the APP folder N. V s code. I've opened the project's root directory in V s code. If you've opened the correct folder, you should

see a directory structure like this on the left. The S code has a built in terminal. Let's open the terminal so that we can start her application go up to view and then down to terminal. You can also see the keyboard shortcut listed here for your operating system down here. Type in P M. Start in the terminal and note that one nice thing about the V s code terminal is it opens to the directory for your project. If you run in P m. Start and see this screen than your applications running successfully. Let's review the project structure package dot Jason is the configuration file. It lists are projects, name and version, and it marks. Our project is private so that we don't accidentally publish this. 10 p.m. Our project depends on three packages. React, which is the react package. React Dom, which will allow us to render react to the browser and React scripts, which contains build scripts and configuration files. React Scripts contains configurations for the various tools that create React up includes, such as Babel. Web Pack, Yes, lint in jest. It's handy because with great react up, we don't have to configure any of this blow dependencies. His script. These air, called in P M scripts in P M scripts are useful for automating tasks when we type in p M start the START script will run, which runs our application locally for development. As we just saw, the build script generates a min ified production build. You can run this via in p m run build the test script runs automated tests, be a jest, and we can see this run by coming down to our terminal hitting control, see to stop our process and then saying in P M test notice that we can omit the run key word for this command. But in P m run test works as well, since we haven't changed any code justice smart enough not to rerun the tests. But if we hit the letter a, then it will rerun the tests and we can see the one built. An automated test passes. The final in P M script is eject. If you want to have complete control for configuring babble, Web pack and the other tools that create react up uses, then you can run the eject script. This command exposes the configuration files to you, but there's a downside. Once you do, you can't easily upgrade to future versions of create react up, so we're not going to eject in this course, and you typically don't need to eject since create react. APP offers a number of configuration options that don't require ejecting, including CSS modules, sass and excellent configuration changes. And if you're interested in learning how to create a react development environment from scratch, then you can see how to do that in the environment Setup module of building applications with reacting Redox After this course, moving down the file. The E s lent config references create react APS default. Excellent configuration. So this will give us some basic checks for code quality. And finally, the Browsers list section specifies what browsers were supporting. Create React up uses these settings to determine how to build the APP. The Source directory contains the application source code index dot Js is the applications entry point, and it references the APP component, which is the one react component that is included with this project. This is the react component that we're seeing rendered when we run in P M. Start in the browser. Some CSS is also included, as is a

service worker dot Js file, which may be useful. If you'd like to build a progressive Web app, we won't cover this aspect in the course. The public folder contains static assets like index dot html, a favor icon and a manifesto Jason File, which could be useful if you create a progressive Web app. Next up, let's install a few other dependencies.

Install Flux, React Router, and Bootstrap

Create-react-app includes a lot of useful tooling, but there's a few other dependencies that we'll need throughout the course, so let's install them now. I'm going to paste in the command. Be sure to use this exact install command to ensure that you can follow along with me. I'm specifying these exact versions, and I can then update the course as new versions come out over time. Flux is a state management library. We'll use React Router to handle client-side routing, and Bootstrap is a popular CSS framework. We'll use Bootstrap to style our app. To begin, let's remove some files that we don't need so that we can start from scratch. To do so, delete all the files in the src directory, except for index.js. Then open up index.js and delete everything inside. Now we have a clean starting point. Let's import Bootstrap so that our application is styled via Bootstrap. Now you might find it weird that I'm importing CSS just like JavaScript, but create-react-app supports this because behind the scenes it uses webpack to handle CSS. So create-react-app will inject the CSS into our application. Quite handy. So now Bootstrap styles will be applied as we add Bootstrap-related CSS classes to our app. To finish setting up our development environment, let's close out this module by setting up a mock API.

Why a Mock API?

Instead of hitting a real API, we'll host a mock API that simulates making async calls to a server. I'm using a mock API for convenience in this course so that we don't all end up hitting the same API and wiping out each other's data, but I actually recommend using a mock API any time that you're building a client-side app. Here's some reasons I'm a big fan of mock APIs. This pattern 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 the final API is returned, then you can create a mock API and begin development. A mock API helps me move independently when a separate team is handling the 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, then I get to decide when to do so. It's no longer a blocking issue for building the user interface. This 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 happens to be down or broken at any 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 rapid 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 potential performance problems. Of course, you'll want to 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 the responses. You can get a feel for how your app performs when the API calls are slow by using a set timeout in the mock API to delay the responses. A mock API gives me a handy tool for automated testing. Since the data is local, it's both fast and reliable, and you don't have to mock calls since your mock API is already a mock. And since the data is both local and deterministic, you can even write tests that utilize the data, and they won't be slow or flakey since these are local calls. Finally, you can easily point to the real API later using an environment variable. For all these reasons, I prefer to code against a mock API for all my projects. In this course, we're going to use JSON Server as our mock API.

Create Mock API

All right, let's get started and set up our mock API using JSON Server. Next, let's set up a local mock API. This will give us an endpoint for requesting and changing course and author data. To get started, we need to install a few more packages for the mock API. Since these packages are all development tools, I'm adding the -D flag so that they're listed under dev dependencies when they're written to package.json. Cross-env is a handy library for setting environment variables. Npm-run-all will allow us to run multiple scripts at the same time. And json-server will serve our mock data. And again, be sure to use these same versions. Next, we need to copy some files over from the course exercise files. To find the files that you need, look in the before folder for this module. I've already copied the files over, so let's look at what I placed. I have a tools directory that I've copied over and placed in the root of the project, so you should see it's at the same level as the src directory. Let's take a look at the files inside. ApiServer will serve the API that we use in this course. This file uses Express with JSON Server to host a mock API and simulated database using a JSON file. JSON Server provides a command line interface, but I'm using the more advanced module config to add custom server-side validation down here at the bottom. You don't need to understand this since it's unrelated to React and Flux, but if you're curious how this works, I've provided comments throughout, and you can also check out the JSON Server docs. If

you happen to know Express, then most of this code will look quite familiar to you, but you don't need to understand this since it has nothing to do with React itself. Second file is createMockDb. This script 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. Finally, mockData.js contains the mock data. This data will populate our mock database. This mock data contains an array of Pluralsight courses, and down at the bottom, also an array of authors, as well as the data structure for creating a new course. To put these scripts to use, let's open up package.json, and we'll create a new script called start:api. This script will use Node to call tools/ apiServer.js. We want to recreate the mock database each time that we start our API, so let's create another script called prestart:api, and this script will run node tools/ createMockDb.js. 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. So if you accidentally corrupt the data or get an odd data-related error at any point in the course, hit Ctrl+C to kill the process and restart the application. The mock database will be recreated. Now let's open the command line and run the start:api script to see if it works. Say npm run start:api. Great, it looks like it's running on port 3001, so let's check the browser. If you open up localhost 3001, then you should see this landing page from JSON Server. We can see that it created two endpoints for us, one for courses, which returns the course data, and another one for authors, which returns a list of authors. And here's what makes JSON Server so awesome. It automatically supports changing data too. So we can make POST, PUT, and DELETE calls to manipulate the data, and it will write to db.json behind the scenes. So this db.json file that was generated here simulates a database, and as we make calls to our mock API, you'll see this db.json file change over time. And also, to avoid us accidentally deleting all the data and having to start over, remember that each time that you start the application this db.json file will be regenerated. So this will protect us in case we write a bug and accidentally corrupt the data or delete it all. Each time that we start the app, the data structures and mock data will be copied over into db.json. And since I have Prettier installed, I can also hit Save, and then the db.json file will be nicely formatted so that I can read it. Through the rest of this course, we're going to work with this mock API, so let's go back to our package.json and set it up to start the mock API each time that we start the app. Let's rename the start script to start:dev, and then we can create a new start script right above it. In here, we're going to use the npm-run-all package, which provides a command called run-p. This will allow us to run both our app and our API at the same time. Run-p stands for run parallel. We provide this command with a list of scripts that we'd like to run at the same time. We want to run start:dev and start:api. Don't forget your comma. Now, if we jump down to our terminal, we should be able to hit Ctrl+C to stop JSON Server. And now if we run npm start, it should start up both the application and our mock API. Create-react-app clears the

terminal, so we can only see the output from create-react-app. But if I come over here and refresh 3001, we can see that our API is still returning data. And if I come over here to 3000, we can see that our app is running as well. Although, remember we cleaned out index.js, so it's a completely empty white screen. While we're setting up our mock API, it's also useful to create a few JavaScript files that will make it easy to make calls to our mock API. I've already copied these files over under the src directory in a file called api. So look within the exercise files and the before folder for this module, and you should find the api folder which contains three files. CourseApi contains functions that will get courses, save courses, and delete courses. AuthorApi contains functions that will get authors, save authors, and delete authors. And apiUtils centralizes the handling of our API responses. Notice that I'm using fetch to make API calls. Fetch is built into modern browsers, so we can make API calls without installing an extra library. Fetch has a promise-based API. W00:21: 43.03 is called when the asynchronous call is complete. And if an error occurs, the catch function is called. Notice that I'm specifying different HTTP verbs for each call. Fetch defaults to passing GET, so I don't have to specify in 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, if I scroll down, you can see the call to delete a course. For that, 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, let's jump over into package.json. Inside, we can set the start:dev script to run cross-env. This is a package from npm that we installed that allows us to set environment variables in a cross-platform friendly way. The environment variable we want to set is REACT_APP_API_URL, and we'll set it to the URL for our mock API, which is localhost 3001. Create-react-app looks for any environment variables that start with REACT_APP and allows us to replace those in code. So our reference here to process.env .REACT_APP_API_URL will be replaced by create-react-app with the value of the environment variable that we just declared. Finally, remember that you can only run one instance of the app at a time; otherwise, you'll get this warning: EADDRINUSE. That's because the mock API runs on port 3001, so if you accidentally have two instances of the app open at the same time, you'll see this warning. The solution is to find the other instance, which is likely running in another VS Code window or another terminal if you're using a separate terminal. When you find that other instance that's running, hit Ctrl+C to kill it. If for some reason you can't find that other instance, you can always reboot to make sure that you clear the use of whatever's running on 3001. Our development environment is all set now. Let's summarize what we've covered so far next.

Summary

Our initial app structure is now complete. We installed a variety of technologies. We're using Node to run our development environment, I'm using VS Code as my editor, Prettier for code formatting, React Router for routing, Flux for state management, and Bootstrap for styling. We generated our app via create-react-app, and we configured a mock API that uses JSON Server behind the scenes. In the next module, let's explore some of React's core concepts.

React Core Concepts

Intro

Now that our development environment is set up, let's learn some key React features so that we can put it to use. In this module, we'll begin by comparing React's model to the MVC pattern. Then we'll explore one of the most unique aspects of React, which is JSX. We'll look at how React's innovative virtual DOM helps deliver exceptional performance and an excellent developer experience. We'll consider separation of concerns and discuss how React requires you to rethink how you separate your concerns. With these core concepts settled, we'll look at different ways that you can declare React components. And we'll close out this short module by creating our first React components. All right, let's begin.

React vs. MVC

Perhaps you're familiar with the traditional MVC model. MVC stands for Model-View-Controller. React doesn't dictate how you handle data flows, routing, or other concerns in your application. It focuses on being a simple, composable, component library. It's this simplicity that makes React such a powerful and flexible solution to component-based app development. So some people say that React is the V in MVC. In actuality, React can fulfill all three. You can build rich, complex apps with just React. React supports local component state, which is a lot like a model, and Flux provides an alternative place to store state outside of React. You can create components focused on logic, which end up operating a lot like controllers that you're used to in MVC. This pattern is called controller views. A controller view is basically a fancy term for React components that handle data concerns and compose other dumb components together. Some people call this separation smart versus dumb components, or container versus presentation components. We'll

check out an example of this in a future clip. But in short, controller views promote reuse and separation of concerns.

JSX Introduction

Let's discuss the unique way that React handles markup called JSX. React supports an optional XML-like syntax that's called JSX. You can use it to handle your markup. It looks almost identical to HTML, and that's why I put quotes around HTML. It's not actually HTML, but it's nearly identical in syntax. There are some minor differences, such as using className instead of class and htmlFor instead of for. Of course, a browser wouldn't know what to do with JSX, so JSX must be compiled down to plain JavaScript. Babel handles that for us. So really, JSX is an abstraction over plain old JavaScript calls, and it's completely optional. You can use JavaScript to declare your markup if you prefer, but the vast majority of people use JSX because it looks like the final HTML, which makes it easier to write, easier to read, and much more approachable for designers. Now before I show you JSX, I want to warn you it's a bit like the first time that you see an ugly baby. At first, you're going to have to hold back a cringe, but after you give it some time, I think you're likely going to fall in love. Trust me. And with that, let's take a look at JSX. Oh, and try to imagine an ugly baby here. This is my son; he's clearly adorable. This is a React component that displays the word About in a header. Now most of this is plain JavaScript, but what's this here in the middle? Well, that's JSX. As you can see, JSX looks like HTML, but it's sitting in the middle of your JavaScript. Now you're probably wondering how in the world this works. It's not valid JavaScript as is, and that's why JSX has to be compiled into JavaScript so that your browser can understand it. Compiling JSX to JavaScript is pretty simple. Babel will transpile our JSX down to a function call. React.createElement is a function that will generate the HTML that we specified up above. Notice how the code on the bottom is just plain JavaScript? The browser will understand this just fine. Here's a slightly more complex snippet of JSX. As you can see, the JSX looks like HTML, and as I mentioned, there are only a few minor differences. Now this JSX compiles down to plain old JavaScript. As you can see, it becomes a call to React.createElement. The function is passed the name of the tag that you created, an object that specifies the attributes that you'd like to set, and finally, the markup that should sit inside. This final parameter will contain calls to other elements if you have nested markup. Let's look at another example. Here I am on babeljs.io, and I clicked on Try it out so that I could view their REPL. If you come over here to the left, I suggest turning off these other presets that we don't need. The only one you need checked is react for this particular demo. What we're going to do is type out a React component so that we can watch our JSX transpile. Most people use Babel to transpile their JSX to JavaScript, so it's helpful to use the

Babel website to try it out. Then we can watch our transpile happen live on every keystroke. On the left, let's enter a simple React component. Inside, it will render a header. This call is converted into a call to React.createElement. The first parameter is the tag. The second parameter contains any attributes that I've specified on the tag, such as style, required, event handlers, classes, and so on. This is null since I didn't specify any attributes on my h1 tag. The final argument contains the child content, which in this case is the word About. To see how attributes are handled, let's add a className here called example. Note that I declare className instead of class. This is one of the few differences between HTML and JSX. Attributes are passed as the second argument to React.createElement. They're formatted as an object. Let's add some more markup so I can show nesting. I'm going to wrap this h1 in a div, and inside this div we'll also put in a paragraph that says, This is the about page, and close the paragraph tag. Now we can see the third argument to React.createElement contains a list of child elements. That's why you see the h1 and the paragraph tags are passed as the third argument to the first React.createElement call. I'll paste in a more complex example. This shows a table structure being compiled down into multiple nested calls to React.createElement. The XML-style syntax of JSX has the benefit of balanced opening and closing tags. This helps make trees easier to read than functions calls or object literals. So yes, you can avoid JSX if you want to do it the hard way and write the JavaScript yourself, but JSX is easier to read, easier to type, and it's much more friendly to designers, so I recommend using JSX. Create-react-app will automatically transpile our JSX for us via Babel so that we don't have to worry about it. React's JSX supports inline styles as well. Inline styles are declared by passing a JavaScript object to the style property. That's why you see two sets of curly braces here. The first set says, I'm about to write some JavaScript in JSX, and the inner set is for declaring the styling object. This style would set the header to white with a black background, and it would be 20 pixels high. Note that properties are CamelCased. That's why the background color is backgroundColor instead of background-color here. And note that I don't specify the height in pixels. It's inferred. If you want to use units other than pixels, then you can specify the value as a string with the desired unit. But remember, this styling approach is totally optional. You can use plain CSS with React. It's not recommended to use React's inline styles as your primary styling approach; however, the style attribute can be useful for occasional use when assigning dynamic styles. So let me summarize some key differences between HTML and JSX. cCass is className, for is htmlFor, and attributes are CamelCased. So, for instance, instead of being all lowercased, tabIndex is CamelCased. There are a few other minor differences outlined in the React docs, but the differences I've just shared are the most common items that you'll run across. And there are other benefits to JSX too. In JavaScript, we enjoy clear error messages that point us toward the line number where the error occurred. But think about HTML. When you make a

typo in HTML, things often just silently fail. Browsers were designed from the beginning to be very liberal in what they accept, but it's a lousy place to be writing any kind of logic because when you make a typo in HTML you typically have no idea what line failed. Contrast that experience with working in JSX. Since JSX is JavaScript, when you make a typo, the JSX often won't compile. It will throw an error and reference the exact line where the error exists. If you forget to close a tag, it will throw an error at that line. This is a huge win. JSX honors the philosophy of fail fast, fail loudly. This helps make debugging much easier by drawing attention to mistakes on specific lines as early as possible.

Virtual DOM

Next, let's discuss an important React feature called the virtual DOM. I think it helps to start with a story. Imagine that I do professional photography. I have a complicated stage setup like you see here. Now I have an assistant who sets up all these lights, and it's a lot of work for him to get all this in the right spot. I could just show him a picture like this so that he knows how I need the lights set up. But here's the thing. Since the lighting setup is generally the same day to day, the most efficient thing for him to do is to compare my desired setup to the current setup. I likely want nearly the exact same setup, but without this one light or with this one light added. Obviously, the most efficient thing for him to do is to compare the current setup to my desired setup. But what if he didn't try doing that comparison? What if instead he ended up tearing everything down and rebuilding it blindly based on my picture? That would be a lot of time-consuming manual setup, and that would be quite slow. In the same way, React's virtual DOM saves time. Just like the stagehand manually setting up these lights, updating the DOM is slow. So to minimize updates to the DOM, React's virtual DOM compares the current state of the DOM with the new desired state and then determines the most efficient way to update the DOM. See, updating the DOM is an expensive operation. And by expensive, I mean that it requires a lot of processing because redrawing a large section of the DOM is inefficient. React recognizes this problem, which is why they created an abstraction over the DOM called the virtual DOM. Traditional two-way binding approaches don't have a virtual DOM, so when state changes, they immediately update the DOM to reflect new state. In contrast, React monitors the values of each component's state, and when a component state changes, React compares the existing DOM state to what the new DOM should look like. It then determines the least expensive way to update the DOM. This sounds complicated, but it's all behind the scenes. This approach avoids layout thrashing, which is when a browser has to recalculate the position of everything when a DOM element changes. There's nothing extra that you have to do to enjoy this performance benefit.

When you update a component's state, it happens automatically. The comparison happens in memory, so it's very fast. For an example of how the virtual DOM helps, many traditional frameworks say, oh, it looks like this array has changed, so I'll just iterate over the array and redraw the entire table. That's a real problem if you have a thousand row table and you end up removing a row. Instead, React compares the old state to the new state to determine the cheapest way to make the change. So React simply removes the row that was just deleted from the DOM. This is much less expensive than redrawing the entire table. And it's hard to argue with the results. Khan Academy transitioned from Backbone to React. Notice how much smoother the React version renders. This graphic shows how React and Backbone behave when a computationally expensive operation is performed. Note that the Backbone version is choppy, whereas the React version is smooth. This is because React is only updating the DOM with the text that the user is typing, while Backbone is rendering everything from scratch on each change. This makes React much more efficient. The virtual DOM isn't merely about performance. React also offers synthetic events. Synthetic events abstract away browser-specific event quirks and allow React to optimize the performance of attaching event handlers for you behind the scenes. This means in React you can attach event handlers to each row in a table, and React will intelligently attach the event handler at the highest level for performance reasons. The virtual DOM also allows React to render on the server where no DOM exists at all. And because the DOM is abstracted away, React can even be used for implementing components in native applications using React Native. In summary, the virtual DOM is fundamental to React's performance and flexibility.

Rethinking Separation of Concerns

Now, the idea of putting HTML in your JavaScript may seem wrong to you at first. Doesn't this totally ignore separation of concerns? Well, let's explore that argument for a moment. If you've been doing front-end development for long, you may have worked with Angular, Ember, or Vue. It likely never occurred to you, but these frameworks effectively put JavaScript in HTML. Angular uses ng-repeat to add looping logic to HTML. Ember uses handlebar-style syntax to effectively add JavaScript in HTML, and Vue does something similar, using v-for="user in users". Each of these syntaxes are functionally equivalent even though they look very different. But the bottom line is each of these are effectively putting JavaScript in your HTML. Though unfortunately, notice that you have to learn a proprietary domain-specific language to work with any of these. So you have to ask, instead of effectively putting JavaScript in your HTML, why not put HTML in your JavaScript instead? That's what React does with JSX. So JSX isn't really that wild an idea. It's

simply the polar opposite of the mindset that's been so popular over the last few years. Instead of enhancing HTML to support simple logic, data binding, and looping semantics, with React, you use JavaScript to define your markup. And this has some clear advantages since JavaScript is a far more powerful technology than HTML. This means that you can enjoy all the power of JavaScript when composing your JSX. Here's an example of me using JavaScript's map function to map over an array of users. Notice that I didn't have to learn any new syntax. This is just plain JavaScript. Regardless of technology that you use, you don't really have separation of concerns. HTML and JavaScript haven't really ever been separate anyway. Sure, one is a markup language, and the other is a programming language, but to create any significant app, these two technologies must be carefully kept in sync. Think about this. In server-side languages such as Java or C#, we have the luxury of strongly typed interfaces. These interfaces allow us to separate concerns, but enforce a common interface that must be implemented. However, keeping JavaScript and HTML in sync is tricky, and it's a source of many bugs. It's tricky because there's no explicit interface between HTML and JavaScript. You have to manually keep these two technologies in sync, or your application will crash. And often, when these two get out of sync, the application fails in unpredictable and hard-to-debug ways. So instead, with React, we need to change the way that we think about separation of concerns. Many of us have been taught to think about JavaScript, CSS, and HTML as separate concerns, but they're not. They're fundamentally intertwined. Putting them in separate files merely separates technologies, but not their concerns. Because the real concern is a single feature in our user interface, such as a button, a date picker, or a modal. These components may each use HTML, JavaScript, and CSS to work, but those three technologies must be carefully composed for our actual concern, in this case the button or the date picker, to actually work as expected. So React acknowledges that our actual concern is the component, and thus, it places JavaScript, HTML, and, optionally, CSS all in the same file. So in React, you think about your component as the concern that's worth separating, and the nice thing is you can open a single file and trust that changes that you make there are specific to that one file. So, in summary, HTML and JavaScript are totally intertwined concerns. Putting them in separate files fails to separate concerns. It's merely a separation of technologies. JSX acknowledges the tight coupling of HTML and JavaScript, and thus allows you to compose your HTML in a JavaScript file using JSX. As you'll see, this integration of intertwined concerns actually makes debugging easier.

Four Ways to Declare Components

Okay, this sounds crazy, but there's currently at least four common ways to declare React components today. There's createClass, ES class, function components, and arrow functions. Let's review each of these approaches, and along the way I'll clarify the styles that we're going to use in this course. React.createClass was the original way when React was first launched, and it works great in ES5, which is the version of JavaScript that's been around since 2009. Or course, you can still use this style today, but most people prefer to use more modern approaches, so we're not going to use this style in the course. Today, JavaScript has classes built in. Here, I'm declaring a React component using a JavaScript class. Notice that I use the extends keyword to extend React.Component. You can optionally initialize state in the constructor. And you declare what JSX it should return via the render method. We'll use this style for some of our components in the course. A third option is to declare a function component. This has a simpler syntax. React assumes that the return statement is your render function. The only argument is the props that are passed in. We'll create many function components in our app. You can also create functions via the arrow function syntax. This allows you to use the concise arrow syntax. With the concise arrow syntax, you can omit the return keyword if the code on the right-hand side of the arrow is a single expression. 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 just use the function keyword that I showed on the previous slide if you prefer. Also, in modern JavaScript, the var keyword should be avoided. Instead, we should use let or const. For React components that use arrow functions, use const to ensure that the component isn't accidentally reassigned.

Class vs. Function Components

It's recommended to use functional components instead of class components when possible, so let's discuss why we should prefer function components. Plain functions are generally preferable to classes since they're easier to understand. They eliminate class-related cruff like the extends keyword and the constructor. Function components avoid the annoying and confusing quirks with JavaScript's this keyword, so function components are easier to understand. Dumping classes also eliminates the need for binding. Given how confusing JavaScript's this keyword behavior is to many developers, avoiding it is a big win. Functional components transpile smaller than class components, and they produce less code when run through Babel. Functional components require less code. This translates to less noise. 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 the parentheses. If you

destructure your props in a functional component, 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-based components. We all know that a function that takes a lot of parameters is a code smell, and when you use destructuring in your function components, the argument list clearly conveys your component's dependencies. Thus, it's easy to spot components that need attention. And since functional components are pure functions, your assertions are very straightforward. Given these values for props, I expect them to return this markup. Function components offer improved performance as well. As of React 16, there's no instance created to wrap them. Finally, classes may be removed in the future from React's core. With React Hooks, function components can handle virtually all use cases. So for all these reasons, prefer writing functional components over class components. So let me summarize when you should use class components versus functional components. The answer depends on which version of React you're using. With React versions lower than 16.8, function components lack some key features. Only class components can support state, refs, and lifecycle methods like componentDidMount. So, before 16.8, you'll need to use class components if you need state, refs, or lifecycle methods. React 16.8 added a feature called Hooks. With React Hooks, you can use function components for almost everything. At the time of this recording, there's only two things that function components can't do, which is componentDidError and getSnapshotBeforeUpdate. Both of these are rarely used, so otherwise, you can use function components everywhere else. In summary, regardless of your React version, it's recommended to use function components when possible. And if you're using React 16.8 or newer, which we are in this course, then you can use function components for nearly everything, 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 with class components, and you'll find plenty of examples of class components in older applications. We'll also use function components with Hooks a little later in the course. Now let's wrap up with a summary.

Summary

In this module, we discussed two foundational concepts in React. JSX looks like HTML in your JavaScript, and it compiles down to JavaScript. This may seem like a bad idea, and it honestly did to me at first, but in the next module, we'll put JSX to work, and I suspect you'll grow to really appreciate the compile-time error checking and colocation with related variables for rapid app development. And the virtual DOM provides exceptional performance by carefully diffing current state with future state and updating the DOM strategically in the most efficient way. As you'll see, it also enables a simple mental model. In React, there's a clear separation between data and the

DOM. The DOM is merely a representation of current state. This makes apps easier to understand and test. And the synthetic event system gives us performance for free by doing the right things behind the scenes and abstracting away browser quirks. Finally, the virtual DOM enables other innovations like simple server-side rendering and support for building mobile apps via tools like React Native. Finally, you can declare React components using either a class or a function. We'll use both styles in this course, but it's recommended to prefer function components since functions are generally easier to create and maintain. So that's React in a nutshell. Now it's time to dive in and start writing some React components. In the next module, we'll create our first components and see how easy it is to compose components together to create complex and easily maintainable web apps.

Creating React Components

Intro

Now that we've set the foundation by covering a few core concepts in React, we're ready to start building components. In this module, we'll begin building our example app, which is a web-based tool for managing Pluralsight courses. So let's begin by diving right into the code. In this first coding module, we'll create our initial app structure including routing and a centralized layout for our application. Alright, open up your editor of choice. Let's get started.

Demo: Create Function Component

To create our first React component, let's create a folder where we're going to keep our components. Since we're keeping all our source code under the src folder, I'm going to create a new folder under the src folder called components. Inside the components folder, let's create a new file, and we'll call it HomePage.js. This component will display our app's home page. First, we're going to import React from 'react'. We're going to use the ES import syntax throughout the course. Native JavaScript imports were added to the language in 2015, so this line says import an npm package called React, and set it to a variable called react. We can declare a component using either a class or a function. For our first component, let's use a function. React component names should begin with a capital letter, so let's call the function HomePage. Beginning with a capital letter is important for two reasons. First, React components are instantiated, and JavaScript classes and functions that create an instance traditionally start with a capital letter.

Second, React assumes that any elements in JSX that start with an uppercase character are React components, and any elements in JSX that start with a lowercase character are assumed to be native HTML elements. Function components render whatever JSX we return. For this function, let's return a div with a header and a paragraph inside. So we'll declare the div, and I'm going to say className here instead of class. This is one of the rare differences between JSX and HTML. I'll set the className to jumbotron, which is a class that comes with Bootstrap. I'll set an h1, set it to Pluralsight Administration, and then let's put a paragraph below that that says React, Flux, and React Router for ultra-responsive web apps. And I'll hit Save. The important piece that I'm missing here is a return statement. Easy to forget. I'm going to wrap our JSX in parentheses, and that happens automatically when I hit Save. Prettier is smart enough to add those parentheses around our multi-line JSX so that it's a single expression. By default, everything in each file is private. That's because create-react-app is configured to use ES modules, and everything in an ES module is private by default, so we need to export our component so that it can be used by other files. We'll do that down here at the bottom, and we will export this default on line 12. We could export just the home page without the default keyword, but traditionally, if only a single item is being exported from a file, you will use a default export. The benefit of a default export is it requires a little less code to import, and the file doing the import can decide what to name that import. That's it. Our first React component is complete. In the next clip, let's run it.

Demo: Create App Entry Point

To render our app, we're going to use ReactDOM. React supports rendering to other targets like React Native for mobile apps, but since we're building a web app, we'll using ReactDOM. Our app's entry point is index.js, so let's open that up. Create-react-app is configured to look at this file first and then look at the imports within this file to figure out what other files make up our application. That's why I say it's the entry point for our app. First, let's import React, and since we're going to use ReactDOM, we can import it as well. I'm going to use a named import for ReactDOM. And what we will import over here is the render function. Named imports are a handy way to get a reference to a function inside the element that we're importing. So this creates a const called render that references ReactDOM's render function. The render function is what will render our app. For now, we want to render our home page, so let's import our HomePage as well. Now we're ready to render our application. Render accepts two arguments. The first argument is the component that we want to render. In this case, it's our HomePage component. JSX supports self-closing tags much like HTML. For our second argument, we need to specify the DOM element where we want to place our application. To determine what DOM element we want

to target, we can go over into the public folder and look at index.html. Right here, there's a div already provided that we can use as a target, and that div has an ID of root, so this is where we will mount our application here within the body tag. To reference it, we'll say document.getElementById, root. We should be ready to run the app. Again, if you don't see the terminal, you can go up to View and down to Terminal, type npm start, and hit Enter. The app should open in your default browser. If it doesn't, open the URL that's listed in the terminal. I'm running Chrome. Great! We can see our home page render successfully. Right-click and select Inspect. This opens up the DevTools in Chrome. Then click on Sources. Under Sources, select static/js, then click on main.chunk .js. Now we can see our application's compiled source code. This code is being served from memory via webpack. Now, this code is a little tricky to read because it's different than our source code. Babel transpiled it, and webpack has transformed it to run in the browser. But notice that no JSX is being sent to the browser. Instead, there are calls to the createElement function in here. Remember, each call to JSX is converted into a call to createElement, so plain JavaScript is being sent to the browser. Babel also allows us to use modern JavaScript today even though some features that we're using may not be supported by all browsers yet. Great! We've created our first React component. In the next clip, let's create another component and learn how we can navigate between pages.

Demo: Create Class Component

You can create React components using a function or a class. We used a function for our first component. Let's create our about page using a class so that you can see the difference. So let's create a new file. We'll call it AboutPage.js. First, we'll import React. Then we can declare a JavaScript class using the class keyword. We'll call our component AboutPage. To make the class a React component, we need to extend a base class called React.Component. Class components have just one required method, which is render, so let's declare that. In render, we declare our JSX. For now, let's just put a header here, so we will return an h2 that says About. Finally, don't forget we need an export default AboutPage at the bottom. Let's try rendering this class component. Open up index.js, and we can import it right here below the HomePage. And down here, instead of calling the HomePage, I'll call the AboutPage. I'm already running my application. You can see that I do have an ESLint warning that my import on line 4 isn't necessary, but I'll go ahead and leave it for now. If we jump over to the browser, we can see that the About page is rendering successfully. So now you know two different ways to render React components. I recommend using function components since functions are typically easier to work with than classes, but you'll find a lot of class components in older React code bases since earlier versions

of React only supported class components. Let's go back to our AboutPage and make a few tweaks. First, let's add a paragraph under the heading right here. I'll say, This app uses React, and hit Save. But notice that we get a helpful error message down here in the terminal. It says, Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment? Remember, our JSX is compiled down to nested function calls. There can only be one top-level function, so this means we can only have one top-level element in JSX. A simple solution to this problem is to wrap our adjacent JSX elements in a div. So we could come up here and add a div like this. I can move this div to the end. If I hit Save, Prettier reformats. And this works fine, but it means that we're rendering a div that we don't actually need. Instead, we can use a React fragment, which we can call like this, React.Fragment. React fragments allows us to wrap adjacent elements, but React won't actually render anything to the screen. Even better, we can declare a fragment using an empty tag, which is a shorthand way to specify a fragment. So we can take out the explicit call to React.Fragment, and it's implied. If we run this in the Babel REPL, we can see that it outputs the same JavaScript. I recommend using this syntax for fragments. This is useful anytime you have adjacent JSX elements. So far, we've created two pages, a home page and an about page. In the next clip, let's configure some simple routing so that we can navigate between the two pages.

Demo: Simple Routing

To navigate between the pages, we could import React Router. We'll use React Router later in the course. But for now, let's just do a simple setup that actually posts back to the server between pages. Over in the components folder, let's create another new file, and we'll call it App.js. The App component will have a simple job. It will decide which page to render. The App component will look at the path name and the URL to determine which component it should render. First, let's import React. Then, import the two pages that we've created. Let's declare the App as a function component, so I'll say function App. Inside, let's read the URL to determine which page to render, so I'll declare a constant called route, and I will look at window.location .pathname. This isn't specific to React because window.location is built in to all browsers. Now what we want to say is if the route is equal to /about, we want to show the about page, so we can return the AboutPage, like this; otherwise, we can return the HomePage. Finally let's come down here and export the App as default. Next, let's go over to the HomePage, and we can add a link to the AboutPage down here below the paragraph. I'll use an anchor, set the href to /about. Next, let's update index.js to call our new App component. I'm going to change this import to import App instead of HomePage, and we'll take the AboutPage import out because now we're going to call

the App component from our application's entry point. So now what we'd expect to happen is React is going to mount our App component and then look at the URL to determine which component should display. So if we come over to the browser, we can see that our home page displays with this new About link. If I click the About link, it loads the About page. And if we hit reload on the About page, it displays as expected. So this simple setup works; however, the downside is it's posting back to the server. It would be nice if it simply showed a different component when I clicked the link. In an upcoming clip, we'll implement React Router so that our app no longer posts back to the server between pages. But for now, this will be enough to get us started, and it helps you understand the core concepts. Now that we've created a couple of React components, let's create a centralized header that allows us to navigate our application. Under components, let's create a new folder called common. This is where we'll put any common components that are used throughout the application. React has no opinions about folder structure, so I'm just sharing my personal preferences here. In here, let's create a new file, and we'll call it Header.js. Inside, we'll import React, and we'll declare Header as a function component. Inside, let's use the nav tag to wrap our navigation, and we'll put in an anchor for our Home page, then a pipe character to separate our links, and a second anchor to the About page. We'll hit Save, and don't forget your export default down here at the bottom. I'm using a nav element instead of a div since it's more descriptive, and I'm using plain anchors for navigating between Home and About. Now that we've created our centralized header, we just need to reference it in App.js. Right now, App.js only displays the proper page for each URL. Let's enhance it to display the header as well. First, we need to import the Header. And we need to do a little refactoring now. Notice that right now we return either the HomePage or the AboutPage, but we want to always display the header. So first, let's extract our logic for finding the right page to a separate function. In JavaScript, we can nest a function inside a function, so we can declare that function right here within the App component. Let's call that function getPage. And we can move this curly brace down so that all this logic is inside the getPage function. Now that we've declared the function, we can call it right here below. So this should operate the same as before. But we want to display the header above each page, so we can put the Header tag above this call to getPage. Let's go ahead and wrap this in a div, and we'll use another class that comes with Bootstrap called container-fluid. And I'll move this closing div down below and hit Save. Now, notice that right here we want to call some JavaScript, but it's not being syntax highlighted. Now that it's sitting within JSX, for this to be read as JavaScript, we need to put it within curly braces. And we can remove the return statement, as well as the semicolon. Instead, we need the return statement up here to return our div that contains both the header and the page. Let's jump over

to the browser and see if this works. Great! We can now navigate between Home and About using our navigation. Our app is looking great so far. Let's wrap up this module with a summary.

Summary

We're making progress. In this module, we created our first React components. We set up an entry point using ReactDOM, which renders our application, and we created both function and class components. We also set up simple routing between our pages. We'll replace this setup with React Router a little bit later. Next up, let's explore more key concepts, including props, state, lifecycle, and keys. These skills will let us pass data to components, declare local state, run code at specific times, and more.

Props, State, Lifecycle Methods, and Keys

Intro

In the previous module, we created our first React components. We utilized custom routing and a centralized header to support client-side navigation. Now it's time to discuss the tools that we need to create more dynamic components. First, in this module, we'll discuss props, which allow you to pass data to your components, and state, which allows you to declare local data in your component that can change over time. Then we'll discuss React's multiple lifecycle methods for class-based components. Lifecycle methods allow us to initialize class components and attach behaviors at specific points in time. We'll handle dynamic child components by assigning keys to elements. And don't worry, we'll do plenty of coding as well. Along the way, we'll enhance some components to put these features to work.

Props and State

Data for a React component is held in two places, props and state. Let's discuss props first. Props is short for properties. Props allow you to pass data down to child components. Props look like HTML attributes. Here's an example component called Avatar. This component assumes that a username prop is passed in, so I say props.username to reference that prop. If this were a class

component, then I would say this.props .username. And here's an example of passing a prop to the Avatar component. So, note that props look a lot like HTML attributes. I could also pass a variable in by wrapping the variable in curly braces. And here's an important point: props are immutable. Since they're passed down from the parent, they're effectively owned by the parent, so you can't change props. If you want to change data that's passed in on props, you typically call a function passed in from the parent. So, what if you want to hold data that changes over time? Well, that's what state is for. Let's discuss that next. The second place that a React component holds data is state. Unlike props, state is mutable, so state is used full for holding data that your component needs to change over time. A common example is values for form fields. However, you don't set state directly. Instead, you call setState to change state within class components, or you can use the useState or useReducer Hooks in function components. We'll discuss Hooks a little later in the module. But as an example, to access a username stored in state, I'd say state.username in a function component or this.state .username in a class component. Here's an example of handling state in a class component. If I want to hold a value in state in a class component, I can initialize the state in the constructor. Note that I need to call super(props) first in the constructor. In the onChange function, I'm calling setState to set the value and state to the value that was just typed in the input declared down below. Alternatively, in a class component, I can also declare state using a class field. I recommend this approach since it avoids having to declare a constructor. As you can see, this requires less typing. We'll implement both props and state in upcoming clips.

Lifecycle Methods

Each React class component has a lifecycle, so there's a list of functions that allow you to run code at various parts of the component's lifecycle. Again, note that these special methods only exist on class components. You can do similar things on functional components using a different feature called Hooks that we'll discuss later. Let's quickly explore when and where each of these lifecycle methods are called and consider why each is useful. The constructor is called before the component is mounted, so this function is a great spot to initialize state and bind event methods. However, as I showed in the previous clip, you don't have to use the constructor to initialize state, and, in fact, I prefer to avoid it since it's a little less typing. Render runs any time that state or props change. Render typically returns your JSX, but you can also return arrays, strings, numbers, Booleans, or null. So, in this function, you declare what your component outputs. And note that this method is only required on class components. With function components, whatever you return is assumed to be what your component should render. Note that your render function

should be pure, so it shouldn't modify state or directly interact with the browser. ComponentDidMount runs immediately after the component is mounted. By the time this function is called, the component's DOM exists. So, this is a handy spot to access the DOM, set up subscriptions, integrate with third-party frameworks, set timers, and make HTTP requests. You can also safely call setState in this lifecycle method. ComponentDidUpdate is invoked immediately after the component's updates are flushed to the DOM, so this isn't called on initial render. This function allows you to operate on the DOM immediately after the component has been updated and re-rendered in the DOM. ComponentWillUnmount runs just before a component is unmounted from the DOM, so this is a good place to clean up by destroying any related resources, subscriptions, or DOM elements that were created when the component was mounted. This isn't a complete list, but these are the lifecycle methods that you're most likely to use.

Demo: Lifecycle Methods and State

Alright, enough concepts. Let's get back to coding and try out state and lifecycle methods. To create our first state and lifecycle method, let's create a page to display course data from our mock API. Under components, let's create a new file and call it CoursesPage.js. Let's create this component as a class. So, we will import React from react and then say class CoursesPage extends React.Component. Remember, with class components, we need to declare the render method, and this will declare what our component renders. To start, let's return a header that says courses and add our export default at the bottom. Be sure to hit Save. And now, let's jump over to Header. Inside of Header, we can add another link. Put the pipe in, the anchor, the href. This will go to courses, and the text will say Courses. And I hit Save. Prettier puts it on a separate line for me. Also notice that Prettier puts in a space within curly braces. This is a way to put in a non-breaking space within JSX. Next, we need to update our App component so that it will handle the routing for this new page that we just set up. We need to input the CoursesPage and then add another route check down here in the getPage function. I'm going to copy the AboutPage and change the path, as well as what it returns. Let's check out the browser. If you're not running the app, be sure to open the command line and run npm start. And now we can see our new link displays as expected, so our Courses page is displaying, and we can navigate between. Although I did want to put courses in the middle, so let's go back to Header and make courses the second link instead of the last link. Now I can take off this final pipe. Hit Save. There we go. I like that order better. Now that the Courses page is displaying, let's display a list of courses in a table. To do that, let's go over to CoursesPage, and we're going to import a function from the course API

file. And I'm going to show you a trick in VS Code to handle the import. I'm going to say import statement, and when I hit Enter, notice that it puts me over here on the right-hand side first. Here I can put in the path to our API file, which is courseAPI. And now, watch. When I hit Tab, it takes me over to that left-hand side so that I can declare my named import, and what it gives me is a short list of the exported functions from that file. So I get autocomplete support because I entered the right-hand side first. Nice little touch. So this function will allow us to call our mock API and will return a list of courses. Let's store the list of courses in state. In class components, we can initialize state in the constructor. We haven't declared a constructor yet, so let's do that. The constructor accepts props as an argument, and the first argument that we have to specify within the constructor is super(props). This ensures that the base class's constructor runs first. After that's happened, then we're ready to declare our state, which we can do by saying this.state. We can declare our state as an object. For our state, we want to have an array of courses. I'll hit Save so Prettier cleans that up. We can see that our linter is warning that we've imported getCourses, but we haven't put it to use yet. We'll do that in a moment. Now, this constructor approach works just fine, but you don't have to declare a constructor in a class component to declare state. We can actually omit the constructor and use a class field syntax instead, which looks like this. Quite a bit less code, and it's less error prone as well since we don't have to remember to call super(props) in the constructor. For the next step, what we'd like to do is request a list of courses when this particular component loads. To do that, we need to declare our first lifecycle method. Since we want to set some state when the page loads, let's use componentDidMount. When you start to type the word component, you can see other lifecycle methods that might be of interest to you in other scenarios, but we want componentDidMount, which is called immediately after the component is mounted, as we can see in the autocompletion support. This is the proper lifecycle method for making API calls. The component must be mounted before we call setState. Since the component is mounted by the time this is run, we know that we can set state in here. We're going to call getCourses to get that list of courses. And remember, if we go over to courseApi, we're using fetch to make our API calls within this file. Fetch has a promise-based API, and with a promise-based API, you declare a .then function to handle the response. Promises are a way to handle asynchronous calls, and they're built into JavaScript. Promises represent a future value. To handle or resolve promise, you call .then, so that's why I'm calling dot then right here. The function that I declare here will be called when the API call is completed. The getCourses API call returns an array of courses, so this anonymous function is going to receive an array of courses. So here on line 11, our API call is completed, and we have a reference to the array of courses that was returned from the API. This means that we're ready to set state. So you might think that we could set state by doing this, by saying this.state .courses are equal to courses. But don't do this. And in

fact, if I try to do this, I get a helpful error message. Do not mutate state directly. Use setState instead. The built-in ESLint configuration in create-react-app is here to help us out. Instead, what we should do is call setState, so we will call this.setState. SetState accepts an object that describes the new properties that we'd like to set in state. What we want to do is set courses to the courses that we just received, so we specify an object inside the setState call and set the courses property to the courses that we just received. And note that setState calls only update the properties that we declare inside the call to setState. So, this call only sets course data. If there were other data stored in state on this component, it would be unaffected by the code on line 11. Now, there are a couple of tweaks that we can make to this code. First, we could use an arrow function instead of this anonymous function, so we could declare our function using an arrow like this. And when I hit Save, Prettier will remove the parentheses around courses since there's a single argument to the arrow. Parentheses can be omitted on arrows if there's a single argument. Arrow functions offer a more concise syntax. With an arrow function, we can omit the function keyword, as well as the parentheses around the argument when there's a single argument. We can also omit the curly braces using what's called the concise arrow syntax. So this code now becomes a one liner. This line says get courses from the API, and when the API call completes, store the array of courses in state. We've configured the state and lifecycle methods that we need to display course data, so in the next clip, let's render the list of courses in a table.

Demo: Display Array in a Table via Map

Now we have the data that we need, so let's render the list of courses in a table. To do that, let's come down here below our h2 and declare a table tag, and we'll give that table a className equal to table so that Bootstrap styles our table nicely. We'll put in a thead to wrap our table's header, and then for the header row, we'll put in three headers. We'll display the title, the author's ID, and the category. Now that we've declared the thead, let's declare the tbody. Inside here we have an interesting question. Now we need to iterate over that list of courses and display a row for each course. To do this, we can use a built-in JavaScript function called map. And since we are within JSX, we need to begin by putting in a curly brace to say I want to write some JavaScript right here. So how do we iterate over an array? In React, people typically use JavaScript's map function to iterate over arrays. Map is on JavaScript's Array.prototype, so that means that you can call map on any JavaScript array. There's a long list of JavaScript array methods documented on MDN. We're going to iterate over this.state .courses and call map on that array of courses. Map expects us to pass it a callback function, which we'll declare using an arrow function. Inside here, we can return a table row for each one of the courses. For the first table data, we'll be outputting

the course's title, so we'll say course.title. And I'm going to hit copy to copy two more here. Here we'll output the course's author ID, and finally, the course's category. Notice again that we have a linting error right here because we have adjacent JSX elements. So, to resolve this issue, we can put in a fragment syntax around all of our JSX. And now, when we hit Save, Prettier reformats, and we have a valid HTML table ready to render. Map returns a new array. We're mapping over the array of courses, and for each course returning a table row. I'm using an arrow function here. And there are two benefits to using an arrow function here instead of an anonymous function, which would look like this. First, the arrow function requires a little less code, but more importantly, if you reference the this keyword inside an arrow function, it will automatically inherit the enclosing scope. We're not currently calling any instance methods inside our map call yet, but if we were, the arrow function means that the this keyword will reference the component's instance as expected. Why? Well, because arrow functions don't have their own this context, so they always inherit the this context of the enclosing scope. So I'm going to undo that change, and we'll keep the arrow function here. And in React, people typically declare the body of the map right here in line, but if you wanted, you could extract the body of this function out to a separate function called render course row if you prefer. That would look like this. Notice that the renderRow function is automatically passed the course. We don't have to explicitly pass it because JavaScript passes the course to renderRow automatically for us. You could even go further and declare a separate row component that renders a row if you like. Let's check the browser and see if this works. Hey, look at that. We've got our table of courses. I'm going to undo the separate function since I recommend declaring inline. It's a more common approach and a little bit less code, but again, you can do whichever you prefer. Trust me, you'll quickly get used to reading inline map functions like this. Let's check the browser, and it's still working as expected. However, if we open the DevTools, we can see that there's a warning down here. In the next clip, let's resolve this.

Keys

When you're creating multiple child components dynamically, it's important to provide a key for each child component. As children are added and removed, React uses the key to ensure that child components are properly reordered or destroyed. So in this example, I'm using JavaScript's map function to iterate over an array of courses. I'm specifying the course's ID field as the key. The key is often the primary key for the corresponding database record, but it doesn't have to be. You just need to declare a unique ID for each specific record. And the ID doesn't need to be globally unique. It only needs to be unique to the array that you're iterating over. The important

thing is the ID shouldn't change over time because React will use it to track each record in the array. A common mistake is to just use a counter or the index from map to assign the key. That's not ideal since the index will change when a row is removed or reordered. Think of the key like an ID in a database. It shouldn't change once the page is loaded. If you open up the DevTools, you can see that a warning is being thrown that each child in a list should have a unique key prop. That's because we need to add a key to each table row. Each table row is going to need a key because when we create multiple instances of a given element using React components, React needs a key so that it can keep track of these different elements. This way, as we remove and add elements, React can maintain proper state to make sure that the ordering of these elements doesn't get destroyed as we re-render the application. So we need to declare a key on the top-level element that's being rendered within our map call, so we'll add a key prop and set it to course.id. Typically, you'd use a primary key from the database as a key, and we're using the ID for the course since this is a unique key that we can rely upon that won't change over time. And remember, the key only needs to be unique for the array, not globally. Hit Save, and we should be set now. If we come back over to the browser, the warning is gone. Our app is looking great. Time for a summary.

Summary

Let's wrap up this module by reviewing some of the new concepts that we put to use. Props give us an elegant way to pass data down to child components. Props look a lot like HTML attributes. State is where we store mutable data in our components. We'll use React state more in upcoming clips. The various lifecycle methods give us many options for bootstrapping our components. They allow us to run our own code at different points in a class component's lifecycle. And when you're iterating over an array, you need to specify a unique key for each record. This way, React can track each unique record's position in the array. This is a performance optimization since it helps React update the DOM in the most efficient manner. Next up, it's time to discuss other important React concepts and features including Hooks, component composition, and PropTypes.

Hooks, Component Composition, and PropTypes

Intro

In the previous module, we covered many critical React features such as props and state. In this module, we'll explore a few more key features that you're likely to use on every app. We'll begin by exploring Hooks. Hooks perform a similar role as lifecycle methods, but Hooks only work with function components. We'll discuss component composition and separate our logic from our markup using the controller view pattern. Then we'll move on to PropTypes, which will help us clearly convey the data that we expect to receive in our child components.

Hooks

We just reviewed a variety of lifecycle methods that are supported in class components, but you don't typically need to use class components anymore. As of React 16.8, you can use functional components for almost everything via a feature called Hooks. So why are there two different ways of doing things? Well, Hooks are the most modern way to build React apps today. Hooks allow you to use function components for nearly everything, and there are some significant wins to using function components with Hooks instead of classes. Since no class is involved, function components avoid the common confusion around JavaScript's this keyword. Hooks use the same React concepts that we've already learned, such as state and props, so they're easy to pick up. Their design better models the way that React works, so this helps foster a mental model that tends to make React easier to understand. This tends to make functions with Hooks easier to work with. You're less likely to make mistakes when using Hooks, and you can share your logic between components by creating your own custom Hooks. React includes around 10 different Hooks, but there's really 3 that you'll most commonly use. UseState is for handling state, useEffect is for handling side effects, and useContext for consuming data via React's context. Let's briefly look at an example of each. Here's an example of the useState Hook. Note that this is a functional component. Again, Hooks only work with function components. This line declares a piece of state called email and a corresponding setter function called setEmail. The call to useState accepts a default value. I'm initializing the email state to an empty string. Now I know that this line looks odd to you. What I'm using is array destructuring. UseState returns an array, and the first element in that array is the piece of state, the second element in the array is the setter, so you get to choose the names for those two items. We're choosing to set the name of our state to email and the setter function's name to setEmail. Down below I use the state to set the value on the input, and I use the setter that we declared called setEmail to set the email to the new value the user enters in the text field. Note that you don't call setState. There's a dedicated setter for each piece of state that you declare with useState. You can also declare multiple useState calls, and you can put whatever data you like in state as well. You can store an object via

useState too. Doing so can be useful when the values are related. Another Hook you're likely to use often is useEffect. By default, useEffect runs immediately after each render. That's why it's called useEffect. It allows you to handle side effects that need to occur each time React renders. Here's an example of useEffect in action. You pass a function to useEffect. In this case, the document title is updated each time the button is clicked, so this example functions like a combination of componentDidMount and componentDidUpdate. However, it's important to specify a second parameter to useEffect here. The second parameter is called the dependency array. We need to specify the list of dependencies for our effect. This is where you list the pieces of state and props that are being referenced within your effect. This way the effect will only run when the count changes. So you can think about the dependency array as a list of reasons that the effect should rerun later. So it's important to remember the dependency array argument because if you forget it, it can lead to performance issues. For example, if you make a web API call in useEffect and you forget to specify a dependency array, then the API call will be run every time your component renders. That's not good. Thankfully, there's an ESLint rule built into create-reactapp that will warn us if we forget to properly populate the dependency array. Finally, you can also run code when your component is unmounted by returning a function from useEffect. The function you return will be called when your component is unmounted. You can think of the useEffect Hook as a combination of componentDidMount, componentDidUpdate, and componentWillUnmount. UseEffect lets you specify the code that runs when your component mounts, anytime a re-render occurs, and upon unmount. Another big benefit of Hooks is that you can share your logic between components easily. This sounds complex, but Hooks are just JavaScript functions, so you can basically cut some code out of an existing React component and paste it into a new file to create a Hook. It's surprisingly straightforward, but if you do so, you need to follow the rules of Hooks. Let's review a few of the rules that you need to honor. First and most obviously, you can only call Hooks from function components. You can't call Hooks from classes. And you can't call Hooks from plain JavaScript functions. They must be called from inside a React function component or from your own custom Hook. Second, Hooks must be declared at the top level. This means that you can't declare Hooks conditionally such as using an if statement, loops, or nested functions. For example, you can't wrap a Hook in an if statement so that it only runs sometimes. If you want your Hook to run conditionally, put the condition inside the Hook instead. This rule is important because of the way that React Hooks work behind the scenes. React stores each Hook call in an array to track the order that the Hooks are called. This is how React is able to maintain state over time. Each time your component renders, React notes the order that the Hooks are being called and retrieves the appropriate value from state behind the scenes.

Demo: Hooks

Enough talk. Let's try out Hooks. To try out Hooks, let's refactor our CoursesPage. We can convert this class component into a function component that uses Hooks. First, let's declare CoursesPage as a function, and we can remove these calls to extends. To declare state, we can use the useState Hook, so let's add an import up here for useState. Then, down here, instead of declaring state this way, we can use the useState Hook. The useState Hook returns an array with two values. The first value is the name of our state, which we're going to say is courses, and the second value will be the name of our setter method, which we will specify as setCourses. So we'll set this to the result of calling useState. Inside the call to useState we can declare the default value, so we will begin with an empty array of courses. This means we can remove our state declaration right here. Now we need to replace the componentDidMount lifecycle method. In function components we can do similar things as lifecycle methods using the useEffect Hook. So let's come back up top, and we'll also import useEffect. We'll call useEffect here within our function component. UseEffect accepts a function that it will call, so we'll declare an arrow function inside this useEffect Hook. Inside, we want to do something similar to what we did here with componentDidMount. Let's go ahead and move the componentDidMount body up into useEffect. We still want to call getCourses. And then instead of calling this.setState, we want to call the setCourses function that we declared on line 5. And instead of passing it an object, we pass it exactly what we'd like to set, which in this case is the courses that we received back. However, notice that we have a problem here. I'm declaring that the value I get back from calling getCourses is courses, but this name conflicts with the courses variable that we've declared to hold state up on line 5. One easy way to avoid this that's pretty common is to put an underscore on the front of this variable. So what we want to do is call setCourses and pass it the courses that we got back from our API call. We can now delete the componentDidMount function, and we can also delete render because it's implied in a function component. Whatever we return is what is rendered. So this means we should also jump down to the bottom and remove the extra curly brace. Hit Save. Prettier reformats, which tells me that we now have valid syntax. However, we do need to update render because state is no longer under this.state .courses. We can reference courses directly because that's what we called it when we called useState up at the top of the function. All right, let's save our changes and try this out. If you're not running the app, now's the time to open the terminal and run npm start. Great! Our form still renders as expected, and if I open up the DevTools, see no errors displaying, so we're in good shape. We've now converted our class component over to a function component that's using the useState Hook and the useEffect Hook. Now you might think that everything's okay, but there's a problem going on. If we scroll back over here and you look, notice that we're continually

making calls to our mock API at /courses. This is a very easy mistake to make when you declare useEffect. If we scroll back up here, what we need to do is declare a second argument on useEffect. I'm going to come down here and hit Ctrl+C to stop our application, and right here on useEffect we need to declare a second argument, which is called the dependency array. This is where we tell useEffect when it should rerun. Without a dependency array, useEffect will rerun every time that our function re-renders. So without this dependency array, what was happening was we were calling getCourses, which ended up calling setCourses on line 8, and then useEffect ran in an infinite loop. Each time getCourses was called and setCourses was called the function rendered again, which caused useEffect to make another API call. So now that we have this empty array, we're telling React that it should only run this effect one time. Now, if we jump back down and start the application, we should find that it still renders just like before, but we're no longer making repetitive calls to the courses API. It happened just once when the CoursesPage loaded. Now that we've converted this class component to a function component, let's just look at the dif side by side to see the difference. We say function instead of class, we don't need the extends keyword, and we declare the state using the useState Hook instead of a class field as we did on the left. Instead of a the componentDidMount lifecycle method, we used the useEffect Hook, and we declared an empty dependency array to say that we only wanted this to run once. The only other change we had to make was to remove render since that's implied in a function component, and down here our call to state got a little bit shorter because instead of saying this.state, we just say courses since that was the name that we selected up here when calling useState. Next up, let's talk about controller views.

Composing Components

React makes it trivial to compose components together. It's fundamental to the way that React scales. The simplicity of composing components is one of React's greatest features. In this example, we have two simple components. ProfilePic holds an image with a path that gets populated based on the username provided on props, and ProfileLink contains a link to the user's profile based on the username passed in on props. The Avatar components uses these two components. You reference the child components within the parent component. Nice and simple. Also, there's an important concept of ownership at play here. The Avatar component owns the ProfilePic and ProfileLink components. By ownership, I mean the Avatar component is setting the props for these components. Since a component can't mutate its own props, they remain set to whatever value the owner sets them to. This leads us to the concept of controller views. So what's a controller view? Well, a controller view is a React component that has child components. Why

am I using the term controller view here instead of just calling it a React component? Well, controller views are a name for a top-level React component. A controller view controls the data flows for its child components. It does this by setting props on its child components. And as you'll see in the Flux module, controller views interact with Flux stores. As I mentioned earlier in the course, there are other related terms that you might also hear such as smart versus dumb components or container versus presentation components. The idea here is simple. Creating separate components for logic and markup can make your components easier to maintain and reuse.

Demo: Controller Views

Let's check out a demo of creating controller views by refactoring our CoursesPage component. To demonstrate separating logic and presentation, let's extract the course list JSX from the CoursesPage component. This will keep our CoursesPage component focused on logic and also give us a chance to try out passing props to a child component. Under components, let's create another new file, which we'll call CourseList.js. First, import React, and then let's declare a function called CourseList. It will receive props as an argument. Although I haven't been typing the word props on our other function components, all function components automatically receive props as their sole argument, and the parent component that calls it can decide what arguments to pass in on props. At the bottom, we need to export default CourseList. Now let's go get the markup from CoursesPage. So we can cut all of this. But before we paste it, let's put in our return statement and a parenthesis, and then we can paste it within that return statement. We're going to receive that array of courses from the parent component, so let's change this call to props.courses .map because we're expecting courses to be passed in on props. So now it's quite easy to understand this component. It needs to be sent an array of courses on props, and it outputs a table. Let's jump over to the CoursesPage and call this component. We'll jump to the top and import it and then reference it right down here. So now we're calling the component; however, we need to pass data down to our new component. And we can do that using props. Props look a lot like HTML attributes, but remember we're expecting to get an array of data called courses, so let's pass the array of courses down on a prop called courses. Since our prop is passing down JavaScript, we put it within curly braces. Again, any time you want to write some JavaScript in JSX, you wrap it in curly braces. This is your way of saying, hey JSX, I'd like to write some JavaScript in here. However, if you wanted to pass a string value down, you could do so by wrapping it in quotes just like an HTML attribute. For example, if I wanted to specify some custom header for this component, I could pass down a header prop and pass that as a string, like this.

But we don't need the header, so I'll take that out. Again, props look a lot like HTML attributes, but when you pass values down to React components, they're called props. Let's hit Save, and the page should still render the same, so let's confirm. Great! Page is still rendering just fine, but now we've separated our logic from our markup. Note that the CoursesPage is now quite simple. It focuses solely on state concerns. It makes an API call, populates state, and passes that state down to the CourseList. You can think of this as separation between smart components and dumb components. Courses page is the smart component. It's smart because it goes out, gets data, sets it all up, and then passes it down to the dumb component below, which is our CourseList component. The CourseList does nothing but define some markup, and it receives an array of courses via props. This makes it easy to understand and test in isolation. I enjoy separating components like this, but to clarify, it's totally optional. If you prefer larger components that handle state, logic, and markup, that's fine too. But now that we're passing props down to a child component, it's time to discuss prop types next.

PropTypes and Default Props

When you create components, they're going to accept props. Props allow the parent component to pass data and functions down to child components. So, how do you document the props that your component accepts? Well, that's what PropTypes do. PropTypes allow you to document the data and functions that you expect to be passed to your component via props. PropTypes are a separate package from React, but they're documented on the React site and supported by the React team. You can specify that certain properties are required and also specify the data types for each prop, such as a number, a string, or a Boolean. PropTypes are declared as an object, traditionally below your component. With PropTypes, you specify a validation function for each property. When validation fails, a warning is logged in the console, so you're notified at runtime. This is an example of a PropType declaration. This CoursePage component is requiring an author object, requiring a function that should be called when the user clicks Save, requiring a function that should be called to validate input, declaring any errors passed into our form that should be of type object, and requiring a function that determines if any errors have occurred. You can add validation to require arrays, Booleans, functions, numbers, objects, and strings. But note, for performance, PropTypes are only run in development, so think of this more as a way to document your expectation and catch issues during development than as a way to enforce these rules in production. This is also a good time for me to mention development versus production mode. Again, PropType validation only runs in development mode. The production version of React omits PropTypes for performance. So, be sure that you deploy the appropriate version to each

environment, and remember that prop validation won't run in production. Since we're using create-react-app in this course, if you run the production build, it will automatically use the production version of React. For optional props, you may find it useful to declare a default. Default props can be declared below the component much like you declare PropTypes. You declare default props by adding a defaultProps property to your component. So this declaration sets errors to an empty object if the errors prop isn't passed to the CoursePage component. Let's check out a demo of declaring expected props via PropTypes. PropTypes allow us to specify the data that our components accept on props. Let's add PropTypes to our CourseList component. First, import the PropTypes package, so import PropTypes from prop-types. Then jump to the bottom of the file, just above the export, and we will say CourseList.propTypes. Now notice that I'm calling a lowercase p here because this is a property that I'm declaring on the CourseList function. And yes, this likely looks weird to you. We're declaring a property on a function. And yeah, in JavaScript you can add properties to a function because everything is an object. It's crazy, huh? Anyway, by convention, if React sees a PropTypes property on your component, then it will validate the props that are passed in. We want to declare that courses are an array, so we can say PropTypes.array .isRequired. So we're declaring that this component expects to be passed an array of courses, and that if it isn't, then a runtime error should be logged to the console in the browser. Now this check will only happen during development. For performance reasons, the production version of React doesn't check PropTypes. Let's see if this works. Jump over to the CoursesPage, and let's stop passing down the courses prop. I'll just put a 2 on the end here so that we're no longer passing down a prop called courses. If we jump over to the application, we can see that it fails. The app fails because our CourseList component is trying to iterate over an array that doesn't exist. But if you open up the DevTools, you can see that a helpful error message is displayed. This message was written because of our PropTypes declaration, and it says that the prop courses is marked as required in CourseList, but its value is undefined. We could go back to our CourseList component and declare a default prop. So we could say CourseList.defaultProps and say that courses defaults to an empty array. Now, if the component doesn't receive a CourseList prop, then it will default courses to an empty array. So if we jump over to the browser, we can see that we no longer get an error. But for now, let's undo our change to defaultProps here. We'll use defaultProps a little bit later in the course. Let's go back to CoursesPage, and we'll put our courses prop back as it was. So the app should be rendering just fine again. We can also be more explicit in our PropType declaration to declare the shape of the object that this array should hold. So we can say arrayOf, and inside of here we can call PropTypes.shape, which allows us to specify the properties that are expected on each object in the array. So we expect there to be an id, and that ID should have a type of number, and it's

required. We expect a title, and it should have a PropType of string and is required. We expect an authorId, which should be a number that is required. And finally, category, which should be a PropTypes.string and also required. Hit Save, and that reformats a bit. Now I'm being more specific. Each object in the array must have these properties, otherwise we'll be notified in the console. This is also helpful because it documents the object's shape, and if you have multiple components that use the same objects, then you can centralize this declaration by placing it in a separate file and importing it. You can see how to centralize PropTypes in the more advanced Redux course. And note that you can omit the isRequired on the end if the property is optional. And if we check the browser, looks like all is well. No issues. Let's close this module with a summary.

Summary

In this module, we continued to explore how to compose React components. We looked at Hooks, which allow us to manage state and side effects in function components. With Hooks, you can manage state using the useState or useReducer Hooks. We didn't look at useReducer, but it's a more advanced alternative to useState. You can run code at different times using the useEffect Hook, and you can share logic between components using Hooks too. Hooks are JavaScript functions with a few extra rules. And note that there are other handy Hooks built into React that we didn't cover, but they're used less often. We saw that controller views are smart components which pass data and functions down to their children via props. We refactored the monolithic CoursePage component so that it passes its data down to a child component via props. So CoursePage is now a proper controller view. Alternative terms are container versus presentation or smart versus dumb. Finally, we used PropTypes to document and validate props during development. You can specify data types and whether the props are optional or required, but remember, these are only run in development mode. Next up, let's add client-side routing to our app using our React router.

React Router

Intro

In the previous module, we saw how React components give us the power to create rich web UIs using small, composable components, but now we need a new to route between the different

parts of our application. To get that done, we need to select a router, so in this module, we're going to implement client-side routing using React Router. We'll declare routes, handle URL parameters, link between pages, perform redirects, implement a 404 page to display when a page isn't found, and we'll prompt the user before performing a page transition.

Overview: Key Components

React is a small, focused, component library, so it has no opinion on how to handle routing. For small apps, you may have no need for a fully-featured router, but as your app grows, you'll likely want to split your app into multiple pages with deep linking. And that's when a routing library like React Router comes in handy. React Router is an open-source project, so it's not officially affiliated with React, but it's the most popular routing library for React apps today. With React Router, you declare your routes via components, and you specify the component that should load for a given URL. I think you'll find React Router's declarative approach easy to understand. Let's review the three key components that come with React Router. First, you wrap your app with a router component. There's a variety of router components depending on the type of app that you're building and the way that you want to structure your URLs. But for most web apps, you'll want to use the BrowserRouter. The BrowserRouter uses the HTML5 history API so that you have clean URLs. There will be no hashes in the URL. This is the recommended approach for modern web apps. We're going to declare our routes using the Route component. This component accepts props so that you can declare the component to load for each URL. We'll also use the Link component to create anchors. These anchors will be managed by React Router so that they won't actually post back to the browser. JavaScript will capture clicks on these links and load the requested route on the client. No postback required. This will give us instantaneous navigation between pages, which is quite nice. There are a few other components that we're going to use, but these three are the core API. Nice and simple. React Router comes bundled with multiple routers. The HashRouter places hashes in the URL, the BrowserRouter uses HTML5's history API to provide clean URLs, and React Router also includes a MemoryRouter. This router keeps the history of your URL in memory. It doesn't read or write to the address bar. So this router can be useful for automated testing in non-browser environments like React Native. We're going to use the BrowserRouter since we're building web apps for modern browsers. If you need to support old browsers, then you might prefer to use the HashRouter instead.

Demo: Initial Configuration

Let's use React Router to add routing to our app. To begin configuring React Router, let's import it in our app's entry point, which is index.js. We need to choose which router to import. We're going to use the BrowserRouter since it's recommended for modern web apps. So let's import it. We will import. I'm going to say import statement so that we can put the right-hand side in first. React-router-dom is what we want to import. And then I'll hit Tab, and it'll move me over here to the left-hand side, and I get autocomplete support now. So I'll say BrowserRouter. I'm going to alias the BrowserRouter though as Router. So this will allow me to say Router down below even through what I've imported is called BrowserRouter. This isn't required; it just makes my calls down below a little bit easier to read. Now let's wrap our App component in Router. Move this after the App component. By wrapping our top-level component in the Router component, we can declare routes and all of our app's components now. In the next clip, let's declare a few routes.

Demo: Declaring Routes

Now let's use React Router's Route component to declare our routes. Open up App.js. Currently, we're doing our routing by watching the URL. Now that's going to be React Router's job. But to make that happen, first, let's delete the getPage function since we'll no longer need it. We still want the header to always render, but we want the proper page to render right here below the header. So first, let's import another component from React Router called Route. And be sure that you import from react-router-dom. Then we can use the Route component down here to declare our application's routes. First, we want a route for our Home page, and we want to load the HomePage component when the path is empty. So the Route component takes two props, the path, which declares the URL that it will look for, and the component it will load when the path matches. So this route will load the Home page when the URL is at the root. And I'm going to copy this line, paste it twice, save a little bit of typing. So for our second one, we want to call the CoursesPage, which will have a path of courses. And for our last page, we will have a path of about, and that will call the AboutPage. This should do the trick. Let's check the browser. Ah, looks like I imported the Route component incorrectly. I made a typo up on line 6. This is not the default export. I need a named import for Route. Now let's see if the browser's happier. Well, this is interesting. Our Home page is displaying along with the Courses page. When I come over to the About page, the Home page renders along with the About page. So it looks like our Home page is rendering on all of our routes. So why is that happening? Well, that's because our Home page route matches all three of the routes. Note that the path that we declared for the Home page just contains a slash. That's the path for the Home page, but the problem is the other two paths down

here have a slash as well. React Router will allow multiple routes to match, and since all three of the routes have a slash, all three of them match the path that we specified for HomePage. And that's why the Home page is displaying on all three of the pages. The fix for this is simple. We can add an exact prop to the HomePage route. This tells React Router that the route should only match if the path exactly matches. Now if we check the browser, things look great. Home only renders on the Home page. However, as we navigate around, notice that it's still completely reloading the app as we go from page to page. With React Router, we can avoid page reloads by navigating completely on the client. Let's explore that next.

Links and NavLinks

Let's talk about navigation between pages. React Router offers an abstraction over anchors called Link. The Link component creates anchors for you and allows you to specify the path that you'd like to link to, including any parameters. Let's consider an example. Imagine that we want to create a link to a user with an ID of 1. We'd likely want a URL that looks something like this. To support this URL, I define a route that has a path with placeholder for the user ID. The :userId on the end is the placeholder. To create links that point to this route, we can write some JSX and use the Link component, which is provided by React Router. We set the to prop on the path where we'd like to navigate. Note here how the user ID and the route and the link correlate. In the path prop for the route, I declared that the second segment should contain the user ID. Ultimately, when the JSX is compiled, this anchor tag will be generated, but any clicks on this link will be captured by React Router so that the routing will happen client side. So think of Link as a handy abstraction for creating anchors for navigating throughout your app when working with React Router. For navigation links, you may prefer to use the NavLink component instead. This component works the same as the Link component, but it accepts an extra prop called activeClassName. React Router will apply the class name that you specify when the route matches. This way, you can style active links in your navigation bar. So in this example, the active class will be applied to this component when the URL is /users.

Demo: Links

For React Router to handle our links, we need to use the Link component. This way, when we click on links, React Router will capture the click and avoid a postback to the server. Instead, it will route us to the proper page on the client based on the matching route. To set this up, let's open up HomePage.js. At the top, we can import the Link component from react-router-dom. Now we

need to change the anchor tag down here to instead call Link. Instead of href, Link accepts a to prop. And we don't need the leading slash anymore. We can put a closing Link right here. I'm also going to add a class name in here just to show how we can use a little bit of Bootstrap to make this look nicer. I'll give this a class of btn and a class of btn-primary. Now if we check the browser, we can see this nicely styled link that looks like a button. And when I navigate, notice that I don't post back to the server anyone. This all happened on the client, so it's instantaneous. There's no flicker anymore. Much better. Now what's happening is React Router is handling the routing, so React is simply showing a different component as we click around. That's why it's instantly responsive. In the next clip, let's update our navigation links in the header to use a similar component called NavLink.

Demo: NavLinks

We just used the Link component, but React Router also offers a NavLink component. It works like the Link component, but includes one extra trick. It will set a CSS class when the route matches the component's path. So this is useful when you want to style the link for the current route. For example, if we're on the Home page, it would be nice if the Home link was in a different color. This would clarify which page we're on. So let's use the NavLink to make it happen. To do that, open up Header.js, and let's import the NavLink from react-router-dom. I'm going to use Find and Replace to save a little bit of typing here. So I hit Command+F to get this dialog. What we want to replace is the opening anchor with NavLink. And then we want to click this button over here to replace all instances. Then we want to replace any closing anchors with a closing NavLink. Finally, we want to replace href with to. Now all our links use NavLink instead. However, the reason we're using NavLink instead of plain Link is because NavLink accepts an extra prop called activeStyle. We can specify a style to apply when the NavLink's route matches the URL. As I mentioned early in the course when I introduced JSX, in React, we can optionally declare styles using objects. The rules are straightforward. CSS keywords are CamelCased instead of being separated by dashes. So let's create a style called activeStyle. We can declare it up here above the return statement. We'll specify that the color for activeStyle should be orange. Now we can pass this to each NavLink via the activeStyle prop. I'm going to copy this and then paste it on the other two NavLinks. And I can also simplify this JSX a bit. We can have the pipe character within curly braces with spaces on each side instead of having two separate declarations. That's a little bit cleaner. So now this activeStyle will apply when our NavLink's path matches the URL. Let's jump over here, and hmm, it's sort of working, but looks like we've got this similar problem where Home is always matching again. We solve this problem the same way that we solved our route

declarations earlier. We need to add an exact prop to the first NavLink for Home. And now that active style will only apply if the URL is the root. So now our activeStyles are applying as expected, and it's no longer posting back to the server. React Router is doing its magic. So what if someone requests a URL that doesn't exist? Let's discuss that next.

Switch and 404s

So what if the route that we declared isn't found? Well, we can declare a 404 page to display when no other routes match. Notice the route I've declared for the 404 page doesn't specify a path. A route with no path will match all routes. But wait. If it matches all routes, you might wonder how do we avoid the 404 page always displaying? To handle this, notice that I'm using the Switch component. It works a lot like JavaScript's Switch component. Only one route listed between the Switch component can match. So when a matching route is found, React Router will stop looking. This way, by placing the 404 route last in the list, it will only match if none of the other routes above have matched. Since I have the routes nested inside a Switch component, if the first two routes don't match, then the bottom route will end up matching and display the PageNotFound component.

Demo: 404s

Let's add a 404 page to our app using the Switch and Route components. Let's create a page to display when a matching route isn't found. Create a new file in the components folder and call it NotFoundPage.js. Inside, I'm just going to paste in a simple function component. There's nothing new here, so I'll save a little bit of typing. And if you want to do the same, you can copy this out of the course exercise files in the after folder for this module. To put this to use, let's jump over to App.js. First, let's import the component. And since this component should display when none of the other routes match, we don't need to specify a path. We specify merely the component. However, if we jump over and check the browser right now, if you scroll to the bottom, you can see that the 404 page is always displaying. Even when I navigate around to other pages, that 404 page shows up. That's because it matches all routes the way I declared it right here. What we need to do is declare that it should only display if one of the routes above doesn't match, so we can use the Switch component to declare that only one route in the list should match. Let's come up here and import the Switch component. Now we can wrap our route declarations within Switch. I'll move this one down to the bottom. Hit save. The Switch component works a lot like a Switch statement in JavaScript. Only one of these routes will match. Once React Router finds a

matching route, it will stop checking the routes below. So order matters here. That's why it's important that I have this NotFoundPage down here as the last route in the list. So let's jump over to the browser and see if it's happy now. Much better. Things look good. But if I put in a bogus route and request it, our 404 page displays successfully. Next up, let's talk about redirects.

Redirects

So what if you want to programmatically change the URL? For that, you can use the Redirect component. For example, this will redirect our application to /users. Of course, if you just put this in render, the component will redirect the moment that it mounts. What you likely want to do is redirect after a certain activity occurs. So for that, we can use some state. Here I'm using state to hold a Boolean that determines whether I should redirect to the user's page. When it's true, I render the Redirect component. Yes, I admit it feels weird having a React component that redirects to a different page. And if you don't like this approach, you can directly interact with React Router's history object if you prefer. I'll show how to use React Router's history object for redirects a little bit later. You may also want to change URLs over time. In those cases, it's friendly to redirect users that request the old URL. From is the old path, and to is the new path. So from is the path that you want to redirect from, and to is the path that you want to redirect to. You can also specify params and query string parameters if you like, but that's typically not necessary since they'll pass right through to the new route automatically. We just saw how to handle redirects using the Redirect component, but you can also handle redirects in a more traditional approach. Any routes loaded with React Router receive a history object on props, so you can call history.push to programmatically redirect to a new page as well. We're going to use this approach a little later in the course.

Demo: Redirects

Now let's add a redirect to the app using the Redirect component. What if we need to change a URL at some point? It would be friendly if we redirected users to the new page. So to do that, let's import another component from React Router called Redirect. And down here we can specify that we should redirect from our old URL, which we'll pretend is about-page, and redirect the user to our current URL, which is about. Now, if I jump over to the browser and I request about-page, it redirects to About. So this technique is useful when you change URLs over time. We'll also use the Redirect component to redirect conditionally later in the course. Next, let's talk about URL parameters and query strings.

URL Parameters and Querystrings

At some point, you may want to pass parameters in the URL. React Router automatically adds this data to props under params and location. So, for example, here's a route with a placeholder. The course slug is declared as a placeholder. Placeholder's are prefixed with a colon, so we'll be able to reference that portion of the URL by name via React Router. The name here can be whatever you like. Since the named the placeholder slug, I can reference it by that same name. Imagine that we load this URL. Here, the slug is clean-code, and the query string parameter has a value of 3. In this case, the component's props will be populated like this. Props.match .params .slug will be set to clean-code. Again, I say .slug here because that's what I called the placeholder in the route up above. For example, if I rename slug up here to id, then I'd say .id down below. The query contains an object with keys and values that correspond to the key-value pairs in the query string. That's why it's set to an object with a key of module and a value of 3. And the path name contains the entire path.

Demo: URL Parameters

Let's put URL parameters to use to pull the course slug from the URL. So far, our app only lists the courses in a table. Let's create a dedicated Manage Course page where we can add new courses and edit existing courses. This will give us a chance to try reading URL parameters using React Router. First, let's create a new component called ManageCoursePage. In the next module, we'll build a form for editing an existing course. For now, let's just display the course slug from the URL. Any route loaded by React Router has a variety of props injected automatically. As usual, we need to import React from react. And I'm going to declare a functional component using an arrow function instead here just to show another way to declare functional components. It will accept props. And then inside of here I will return the results. I'm going to use the fragment syntax since we're going to have adjacent elements inside. We're going to have an h2 that says Manage Course. And now, below this h2, the question is, what do we want to display? Well, in the next module, we're going to build a form for editing and adding courses, but for now, let's keep it simple and just display the course slug from the URL. You can think of the course slug a lot like the course's I'd, but the slug is a URL-friendly version of the course title. Now, any route loaded by React Router has a variety of props injected automatically. If I set a debugger in the component, then we'll be able to see the values getting passed into this component. But for now, just trust me. I'm going to call props.match .params .slug. This will read the slug placeholder from the URL. Finally, at the bottom, be sure to export default ManageCoursePage. Now that we've created this new page, we need to create a route for the page, so let's open up App.js. We can import it. And

then down here we'll add a route with a path of course, and the component that we want to load here is ManageCoursePage. So if the URL is /course, we expect to load this new ManageCoursePage. Finally, let's go enhance the CourseList component to display each course title as a Link. So let's jump up here and import Link from react-router-DOM, and then down here, instead of merely outputting the title, we can use the Link component. And we want to Link to the course page, and then after that, we want to add on the course's slug to that URL. We want the body of the Link to display the course's title, and then we need to close the Link right here. So now, instead of outputting just the course's title, we're going to have a Link to /course/ whatever the slug is for that particular course. So let's see if it works. If we come over here, go to Courses, now we have links for each of the courses. And if we hover over them, we can see the URLs getting generated. So if I click on one, I'm taken over to the Manage Course page. However, the slug isn't being displayed. That's because if we want to read the slug from the URL, we need to declare the page's router accordingly. So over here in App.js, we need to add a placeholder for the second segment of this URL. To declare a placeholder in React Router routes, you add a colon and then the name of the placeholder. So what we're expecting to receive in this second part of the URL is the slug. Now that I have called this slug over within ManageCoursePage, this call will work because slug will be passed in as one of the params from React Router. Since this component is being loaded by React Router under routes, props.match .params will be populated by React Router. And now if we come back over, there we go. Now the slug is displaying as expected. And if we navigate around to different components, we can see that slug change. And in fact, if I come over here and set a debugger and open the DevTools, now if I refresh, the browser will stop on this line, and we can inspect what's getting passed in on props. We can see that these props, history, location, and match, are all props that are getting passed in automatically by React Router because this route was loaded by React Router. If we look under match, this is information about the matching URL, and we can see that under params there's the slug. So this is why I said props.match .params .slug to get this data from the URL. We can see the path that matched as well. And we can also see other metadata, information about the current location and its path, as well as the history and various functions that sit inside. And if I hit F8, it moves past the debugger. In the next module, we're going to add a form here instead of merely displaying the slug. But next, let's talk about implementing prompts using React Router.

Prompt on Transition

Sometimes you'll want to prompt the user before you allow them to navigate away. A good example is a form that has information entered. Here, when isBlocking is true, then the Prompt

component will display the message that I've specified here to the user. IsBlocking is a value and state that we can set to true when the user types in the form, and we can set it back to false when the user submits the form. And to clarify, the name of the variable that you store in state is up to you, but when accepts a Boolean that you can control.

Demo: Prompt on Transition

Let's remove the debugger that we added in a previous clip since we no longer need that. And to try out Prompt, let's import Prompt here on the ManageCoursePage. Then down here, below the h2, we can call the Prompt component, and we can say when it should display. I'll just hard code this to true so that it displays when we first load the page. We can see this work. And then I can declare what message I want to say here. I'll say, Are you sure you want to leave? And close the component. Now is we come over to the Manage Course page and click on About, it says, hey, Are you sure you want to leave? If it hit Cancel, then nothing happens. If I hit OK, then it proceeds. Now this was obviously a silly example, so I'm going to take this out now that we've seen how it works. But you can imagine how the Prompt component is useful when someone has a form filled out that they haven't saved. Alright, let's close out this module by reviewing the React Router features that we've used so far.

Summary

So that's React Router in a nutshell. As you can see, React Router is a low friction way to set up declarative client-side routing when working with React, so it's easy to envision how your routing will work by looking at the code. You wrap your app with the desired Router component. We chose to use the BrowserRouter. We declared our routes via the Route component. The Link and NavLink components give you anchors that React Router controls. You can handle redirects via the Redirect component or via the history object that's passed into each route. We used the Switch component to display a 404 page when no routes matched. And you can prompt the user on page transitions via the Prompt component. In the next module, let's turn our attention to a critical feature for almost every web app. Let's explore how to handle forms in React.

Forms

Intro

So far, we've learned how to create reusable components with React. We've seen how to handle client-side routing using React Router as well. Now it's time to enable writing and editing course data using forms. As you'll see, React's virtual DOM makes working with forms a bit different than what you might be used to. We've been doing forms for years, so you might think that forms should be easy. Well, if you've built forms for long, you've likely learned that actually there's a lot of pieces to get right. And watch out because React has some unique attributes that might surprise you when you start creating forms with it. In this module, we're going to implement a form to handle adding and editing course data. That sounds simple, but we're going to cover a long list of concerns including validation, redirects, reusable inputs, user notifications, saving, and population on load. All right, let's get started.

Create Add Course Button

To begin adding support for creating courses, let's create an Add Course button on the Courses page. This will navigate to the Manage Course page. Let's use React Router's Link component to do so. So import Link from react-router-dom. Then we can call the Link component down here below the h2. Let's give this Link a className of btn and btn-primary. These are classes that come with Bootstrap. Then set the to property to /course. We'll configure this new route in a moment. For the text, I'll say Add Course. If you're not running your app, be sure to do so. If I come over to the browser and click Add Course, it throws a 404. Why? Well, because we haven't declared a route for /course yet. Let's open up App.js and do that. We can copy the existing ManageCoursePage route, but we need to take out the expected slug here. We'll still call the same component since we're going to load the Manage Course page for either of these routes. Notice that I've placed the plain course route below the slug route. This order is important because if I place this new route above the edit course route which contains the slug, then the route with the slug will never match. So it's important to place more specific routes on top of less specific routes. Now if I check the browser and click the Add Course button, I'm taken to the Manage Course page as expected. In the next clip, let's start setting up our first form.

Create Course Form

Next, let's create a form for adding and editing courses. We can add our form and ManageCoursePage.js, but let's put it in a separate file instead. This way, we have a smart component that handles state and logic and a dumb child component that just handles markup. So right-click and create a new file called CourseForm.js. To save some typing, I'm going to paste

this in. You can grab this file from the before folder for this module. This is pretty standard markup, nothing special. Also, we're going to eliminate some of this code in a moment by creating some reusable components, so I don't want you to waste your time typing all of this. This code contains some extra markup and Bootstrap classes so that it looks nice, but there's nothing too interesting here. There's an input for course title. A drop-down for author with some hard-coded author data, an input for category, and a Submit button at the bottom. Now let's reference this CourseForm on the ManageCoursePage. First, import the form. Then we can replace line 8 with a reference to the form. Looking good so far. The form should display in the browser now. However, if I try typing in the fields, nothing happens. To understand why, let's talk about controlled components next.

Controlled Components

A word of warning. One of the oddities that you'll run into when creating forms with React is that any input with a value is called a controlled component. This means that its value is controlled by React. So the element's value will always match the value of the assigned prop. Since React automatically redraws the UI as data changes, if the data doesn't actually change, the UI won't reflect the change. This may create some surprising behavior if you're not familiar with the concept of controlled components. Because if you don't write a change handler for your input, then any keystrokes that you make in the input field will be immediately lost. If the input doesn't set a value or it sets it to null, then the input is uncontrolled, so the input will continue to operate as you've always expected. That said, you'll typically want to work with controlled components. Let's check out an example.

Form State and Change Handlers

When I try to type in one of these fields, nothing happens. Can you hear my keyboard clicking? But I don't see anything. Even though I've tried typing, my input doesn't seem to register, and that's because we declared a value prop on each input, so these are controlled components. Their values are being controlled by React. Since we haven't attached any change handlers to these inputs yet, there's no way for their state to change. So even though we're hitting keys, the value is immediately lost. React ignores it. So to solve this problem, we need to declare change handlers on our inputs. So to solve this problem, we need to do two things. We need to declare change handlers on our inputs, and we also need to declare some state that will hold the state for each one of our inputs. Now, at this point, we could come over to ManageCoursePage and convert it

into a class component and add state, but this module is about React Hooks, and with React Hooks, we can handle state within function components. So let's put the useState Hook to use. First, let's go to the top and import useState. The useState Hook will allow us to handle state in this function component, and the state that we want to manage is a course. So let's declare some state to hold course data. We'll declare it in a const, we'll call that state course, we'll declare a setter called setCourse, we'll call useState, and we can initialize this state with an empty course. So we can set the id to null, the slug to an empty string, the title to an empty string, authorId to null, and category to an empty string. The call to useState returns an array of two items. The first item is what we want to store in state, and the second item is the setter. We want to store course data, so we'll call the variable course, and the setter is going to be called setCourse. I know this syntax looks weird at first, but you'll get used to it quickly. This is using array destructuring, which is a feature built into JavaScript. UseState accepts a default value, so we're setting the default value to an empty course object. We now have a spot to store our form state, so let's update the call to CourseForm to pass it down. I'll save my changes in ManageCoursePage, and then let's go to manage CourseForm and put this state that we're passing down to use. Come down to the value for the title, and we'll set it to props.course .title, set the author ID to props.course .authorId, and set the value for category to props.course .category. Note that with React you declare the value of a select as a prop rather than setting the selected attribute on the option. This is another one of those rare differences between JSX and HTML. And up here on the value, I'm going to make one other tweak. We could leave this code as is, but if we come over here into the browser, we can see that we get a warning from React about setting a value to null for our select. Instead, we can set it to an empty string by jumping back over here and saying if the authorId is null, go ahead and set the value to an empty string. So this way we can still initialize the state and null within the ManageCoursePage component, but anytime that it is null when we're rendering it, we can render it as an empty string using JavaScript's logical or operator. So now if we jump back over to the browser, we still can't type into the fields, and in fact, we can see why because React is pointing out that we've provided value props without the change handlers. So the onChange handler is the next logical piece to tackle. Scrolling to the top, let's first add a change handler for the title input. Since ManageCoursePage holds the state, it will hold the change handler. Let's add a function called handleTitleChange. It will receive an event automatically passed over from the browser. I'm going to place a debugger inside just so we can see this function being called. We need to pass it down to CourseForm. We can call the prop that we passed down onTitleChange, and we will call handleTitleChange. Now, to clarify, these names can be whatever you like, but I tend to use the convention of calling event handlers with a name that starts with the word handle. And when passing them down to components, I tend to use the on prefix. Now let's jump over to

the CourseForm and put this to use on our input. So we can say onChange= props.onTitleChange because that was the name of the prop that we declared over here. Notice that I'm passing it down as onTitleChange, so that's why we say props.onTitleChange here on line 12. Let's jump over to the browser and try this out. Now if I type the letter D, it hits our breakpoint. The browser automatically sends an event to our change handler. And we can inspect it since we've hit a debugger. If I look under target and scroll down to value, I should be able to see the value that I just typed. Yes, there's a lot of options here. If I click on the ellipses there, I can see the letter d. So I know the title input now has a value of d by inspecting the event. And if I scroll up and look at the name property on target, I should be able to see the name of the input right here. There it is. So I know that the title component now has a value of d based on this event. We now have enough information to update state. So we can come over into ManageCoursePage and provide our implementation for handleTitleChange. Now, you might be tempted to do something like this: course.title = event.target .title. However, in React, we should treat state as immutable. We don't change state directly. Instead, we call the setter, which we called setCourse up above. To avoid mutating state, let's create a copy of the course object. So I'll create a constant called updatedCourse and set it to the spread of course. So I'm using the spread operator to create a copy of the course object. Now we can safely manipulate the copy. We can set the updatedCourse.title equal to event.target .value. However, there's a simpler way to do this. The object spread operator allows us to declare other properties that we'd like to set in the copy within a single statement. See, we can declare other properties right here. What we want to do is set the title equal to the value passed in on event.target .value. So now we can omit this second line. So this says copy the course object and set the title property on the copy to the value passed in on the event. Now we can pass this updated course to the setCourse function. We should now be able to type in the input and see our results. Let's give it a shot. You might have to refresh the screen or hit F8 to get past that breakpoint. There we go. I can type in this input. Now, we do still have warnings down here from React because we haven't provided change handlers for our other two inputs. Now, you might be thinking yuck, so I have to come over here and declare a change handler for every single input? What if my form has a bunch of inputs? Well, don't worry. We can declare a single change handler, so let's do that in the next clip.

Declaring State Change Handlers

Creating a separate change handler for each form input would get annoying fast. Thankfully, there's a simple pattern for creating a simple change handler for each form. Imagine that I was going to create a change handler for the category input. I could copy this handleTitleChange

function and end up changing just one line inside of it. I would end up setting category here instead of title. But otherwise, these two functions would be identical, two different names setting two different properties. Now remember, in the previous clip, we looked at the event object, and one of the other pieces of data being returned on that event object is the name property. I'm setting the name property on the input so that it corresponds to the property that I want to set. For example, over here on CourseForm, I've set the name to title because that corresponds to the property on the course object, course.title. So by this convention, if the title input has a name of title, then it's assumed that its value should be stored in course.title. With this convention of passing a useful name property, we can declare a single change handler. So let's come back over to ManageCoursePage. I'm going to delete this copy here that I showed just as an example, and let's rename handleTitleChange to handleChange. Then, over here, where we're calling title, instead of hard coding this property, we can use JavaScript's computed property syntax, which looks like this: event.target .name. You can read this as hey, JavaScript, set a property on this object based on the value of this variable. So if event.target .name is equal to title, then this will end up setting the title property. When you see these square brackets, you might think array, but in this context, the syntax serves a different purpose. This is called a computed property. It allows us to set a property based on a variable. So let's scroll down here and update the value that we're passing down. Now we're going to be passing down an onChange property and setting it to handleChange. And when I save, Prettier puts things on a separate line. Let's jump over to CourseForm and update our call here as well. We can change this to say onChange. And I'm going to copy this line. And I actually already had an onChange here in my code, but yours probably doesn't, so you'll need to add an onChange right here for the author and then come down here and also apply an onChange prop for the category. We should now be able to change values for all inputs. We do see that the errors are gone, so that's a good sign. Go ahead and close the console. And looks like I can change the author and type in Category as well. Excellent. There's one other tweak that we can consider. If we come back over to ManageCoursePage, the handleChange function could be more concise. We could destructure the target property off of the event like this. And this would shorten these calls down here below because we wouldn't have to say event dot in front of each one. And if you find this destructuring confusing, recognize that destructuring is really a shorthand for me doing this. I could say const target is equal to event.target right here. So this line 14 is equivalent to what I'm doing right here. I'm effectively inlining this particular code and using destructuring. Now of course, on 14 I could be more concise and use destructuring here. So effectively, line 14 is the same as what I'm doing up here for the first argument. I've just moved this destructuring statement in line. We could also inline this updated course code within the call to setCourse. So we could go take this code, cut it, and paste

it in here. And remove the semicolon and remove the const at the top. If you find this code confusing or harder to read, that's fine. Feel free to keep the old style. I admit that destructuring takes a bit of getting used to, but it's a powerful feature that's worth learning. Now, looking back at our CourseForm, it currently contains some competitive markup, so in the next clip, let's create a reusable TextInput component.

Creating Reusable Inputs

There's some obvious duplication in the CourseForm component. I had to manually keep the label and the input in sync. I had to provide the same class name to each input so that Bootstrap will style it properly. And this isn't even all the markup that we're likely to want. We might want to declare a placeholder text, as well as a consistent spot for displaying inline error messages too. Add all this up, and it's clear what we need is a reusable component for text inputs. So let's create a TextInput component that can handle all this complexity in a single spot. To get started, let's create a TextInput component over here under the common folder, and I'll call it TextInput.js. To get started, let's add the usual boilerplate for creating a new React component. Then let's go copy the title input from the CourseForm component and use it as a starting point. I'm going to return the markup within parentheses. Now let's change the hard-coded pieces that are specific to the title input so that this TextInput becomes truly reusable. HtmlFor should use props.ed. For label, we'll use props.label. For the ID, again we'll use props.id. Set the name to props.name, and set the value to props.value. We may also want to display validation errors below the input, so let's add some code for that. Since I'm going to write a little bit of JavaScript here in JSX, I need to add curly braces, and then I can say if there's an error passed in on props, then I want to display a div that displays that error. I'm going to set a className equal to alert and alert-danger. Again, these are classes that come with Bootstrap. We'll display the error if it occurs within this div. I'm using JavaScript's logical and operator here. JavaScript's logical and runs the code on the right if the condition on the left is true. So this says if there's an error, render this div. Now with Bootstrap, if the input is in an error state, then we should set a has-error class on the form group wrapper. So let me show you a way to handle dynamic classes in this component. Declare let wrapper class, and we'll start out with a default value of form-group. But if props.error is set and the error isn't an empty string, then what we want to do is add another class to this string. Notice that I'm going to start with a leading space so that I have a space-delimited list of classes. Now I can apply this calculated class that we just determined up above right here. Bootstrap's has-error class will add a red line around the input to draw attention to it anytime there's an error. Again, the has-error class is a class provided by Bootstrap. So what we're really doing here is

concatenating strings so that we can dynamically add an extra class name. There are some slicker ways to get this done such as the classnames npm package, but I'm going to go with this simple approach here. So now if an error is passed in on props, the input will be styled in red, and the error will be displayed below the input and styled using Bootstrap. Great! So now our configuration for text inputs sits in this one place. This will enforce consistency and help speed development as our application grows. So this doesn't just make it easier to create a form, it also enforces the conventions that we've decided upon. There's one important piece missing. Anytime that we create a reusable component like this that we expect to be used by a lot of different people, then it's especially important to define PropTypes so that people understand what data to pass down and also so that we get warnings if we forget to pass the expected data down. So let's go up to the top of the file and import PropTypes. Then jump down to the bottom, and we can make our declaration. I'll just paste it in since this shouldn't be anything new to you. I'm going to require the ID, the name, the label, and onChange handler, and we'll make the value optional and the error optional. It's also a good idea to declare a default prop for our error. I briefly introduced default props in a previous module, but this is a great place to put one to use. So we'll say TextInput.defaultProps, and then we'll say that if somebody doesn't pass in an error that it should default to an empty string. Now that we've declared this default prop, it means we can come back up here to the top and simplify our code a bit. We no longer need to check for whether props.error exists because we know it will. Now we can simply check if the length is greater than 0. Now that we've finished creating our TextInput component, we can go back to the CourseForm and put this to use in the next clip.

Consuming Reusable Inputs

We've created a reusable TextInput component. Now let's go back to the CourseForm and put the TextInput to use. First, add our import. And then here comes the striking part. We can significantly simplify our code. We can delete the div, the label, the other wrapper div, change our code here to say TextInput. We can remove the type property, as well as the className and these extra divs down here. And the only prop that we need to add is the label prop, which we'll set to Title. Ah, I like that. Much simpler. Let's jump down to our other TextInput and do the same. Remove the div. Remove the label. Remove the other div. Remove these divs down below. Remove the className, and remove the type. Call TextInput instead, and, again, let's specify the label as Category. Let's check the browser and make sure we can still type in our fields. Great! Everything's still working just fine. Let's look at some of the things that have just changed. Our code is now more concise. We no longer need separate labels and inputs because the input tag

alone handles both of those. We're making the assumption that if you're using the input tag, you're going to need a label to go with that input. Before, I had to define the form-control class for each of my inputs. I needed to remember this, or they wouldn't be styled properly. We no longer have to define that because the form-control class is now defined within the TextInput component. And our new TextInput also puts the necessary div wrappers and error styles around each one of these inputs, so this will help us in a moment when we wire up validation. We could use this same pattern to create a reusable select input, but I'll leave that as an optional exercise for you. In the next clip, let's wire up the Save button so that our form actually works.

Saving Data

To get the Save button to work, let's go over to the ManageCoursePage. We can see that right now we're passing over a course and an onChange event, but we also need to pass over some way for the CourseForm to save, so let's declare a function that will save courses. I'm going to call it handleSubmit since it will handle the form submit. Again, it will receive an event automatically from the browser. We want to handle this form submission on the client side, so the first thing we want to do is call event.preventDefault. This will keep the form from posting back to the server. The next question is, how are we actually going to save the course? We'll call the API that we set up in the Environment Setup module. Let's go look at that courseApi file again. When we call this saveCourse function down here, it will make an HTTP call to our local API that's being hosted via JSON Server. JSON Server will write the data to the file system to mimic a data base, so even if we refresh, we're going to see that new data. So first, let's import a reference to the courseApi over here on our ManageCoursePage. I'm going to use a wildcard import here, which will allow us to reference any functions that are being exported from this file. Now let's go down to the handleSubmit function and put this to use. We're going to call courseApi.saveCourse, and we'll pass it the course that's held in state. All the functions in the courseApi file return of promise, so I could declare a .then here to specify some code that I want to run after the saveCourse is completed, but for now I'll just leave that out, and we'll just make a call to saveCourse. Let's pass this new function down as a prop to CourseForm. We'll say onSubmit should be set to handleSubmit. Prettier puts those on separate lines, but notice how I'm continuing to follow that pattern of the prop that I passed down starts with on, and the event handler starts with the word handle. These are just my conventions, so feel free to name things as you like. Now let's open up CourseForm and put this to use. Let's use our submit handler that we just declared right here on the form tag. We'll say onSubmit is equal to props.onSubmit. And note that I'm putting this on the form tag rather than the Submit button deliberately. This is better for accessibility because you

can either click the Save button to submit, or if the user hits the Enter key, then the form will submit as well. Now that we've wired this up, let's go over to the browser and see if it works. We'll enter some data and then hit Save. I didn't get any kind of feedback at this point, which isn't very helpful. But if I click on Courses and then scroll down, ha, ha, look at that. My save worked. And if we jump back over here and open up db.json, scroll down to the bottom, and we can see that new record written to db.json. This means if I come over here and I reload the page, I should still see the data there. And I do. Nice. So this data will persist at least until I stop the application and restart it because remember the data base gets reset each time we run npm start. We're in pretty good shape, but we do have some tweaking here to make this experience more polished for the user. It would be friendly if we redirected to the list of courses after clicking Save. So in the next clip, let's see how we can use React Router to redirect the user over to the list of courses after the save completes.

Programmatic Redirects with React Router

The Save button is working, but the experience isn't very good right now. You hit Save, and you don't get any kind of feedback. You continue looking at the same page. So let's set up a transition using React Router. As I mentioned earlier, this API call returns a promise, so we can call .then to handle the response. The question is, what do we want to do after save? I think it makes sense to redirect the user to the Course page. I showed how to use React Router's Redirect component in a previous module, but this time I'm going to show you a different approach. Since this component was loaded via React Router's Route component, we have access to React Router's history object on props, so we can programmatically redirect the user here after the save is completed. I can do so by saying props.history .push and then tell it where I want to redirect to. I want to send it to /courses. So now when this function is called, it will save the course and then redirect to the Courses page. So now when this function is called, it will save the course and then redirect to the Courses page. Let's try it out. Come over here, add a course, hit Save, and hey, there we go. And we end up back on the Courses page, and there's that new course that I just entered. However, this could still be nicer. We're not getting any kind of confirmation that it was saved. I had to scroll down here and look for it to know that my save worked. So in the next clip, let's notify the user when a save is completed.

Notifications via a Third Party Component

We're not notifying the user when the save is completed, so this is a good chance to try using a React component that's available online. Let's use an open-source React component called React-toastify. Open up the terminal. I'm going to hit Ctrl+C to stop my application, and then I'm going to run npm install react-toastify@ 5.1 .1. Now let's configure React-toastify for use throughout our app. First, open up App.js, and jump to the top. I'm going to import the ToastContainer from react-toastify. I'm also going to import some CSS that comes with this project, which is under react-toastify/dist/ ReactTostify.css. Now we can place the ToastContainer anywhere in render. I'm going to go ahead and place it up above the header. We can configure React-toastify globally by specifying a few props right here. I'm going to set autoClose equal to 3000, which means it will auto close after 3 seconds, and I'm going to tell it to hide the progress bar. The props that React-toastify accepts are documented in the project's README. Also note that since this is a Boolean, I can omit explicitly setting true. The existence of this property infers truthiness. This line configures React-toastify with some default settings. And now that we've configured React-toastify, we can call it anywhere in our app. Let's save our changes here and jump over to ManageCoursePage. At the top of the page, let's import toast from react-toastify. Then, down here, when the save is complete, we can call toast.success and tell it what message we want to display, which will be Course saved. That should do the trick. Let's jump down into the terminal and run npm start again. Let's try adding a course. Hey, look at that. Have a nice toast display, shows for 3 seconds, and then disappears. Our app is feeling quite polished now, but there's a significant concern left. What about validation? Let's handle that next.

Input Validation and PropTypes

We haven't configured any client-side validation yet. So if I come over here, click on Add Course, and then click on Save on an empty form, I have to wait for the server to respond, and it throws an error. It would be more friendly to the user if we do client-side validation so that the application ensures that the user has entered valid data before it even tries saving a new course on the server. To handle that, let's jump over to ManageCoursePage. Scroll to the top, and let's add another piece of state. Let's store errors in state as an object. So I'll say const errors and setErrors are equal to useState, and I will initialize state to an empty object. Next, let's add a new function above handleSubmit and call it formIsValid. In here, I'm going to use a pattern that I enjoy for handling validation. I'm going to begin by declaring a local variable called errors, and notice that I'm prefixing this name with an underscore. This helps avoid a name collision with the errors in state that I've declared up on line 7. I'm deliberately storing errors as an object rather than an array because it makes referencing errors easier in our form. You'll see how this works in a

moment. Now I'm ready to begin handling my validation. I'm going to say if there isn't a course.title, then I'm going to set errors.title equal to Title is required. I'm going to copy this line because we're going to add validation for the other two fields as well. So I'll check for author I'd, as well as for category. Now we've set our local errors object accordingly if any of the fields didn't have a value. So we can call setErrors and pass it our local errors variable. Finally, the idea here is for this form is valid function to return a Boolean that determines whether the form passed validation. So what we want to say is that the form is valid if the errors object has no properties. To do that, we can use a built-in JavaScript feature called Object.keys. Object.keys returns an array of an object's keys. So we want an array of the keys off of the errors object that we declared up above, and if the length of that array is 0, that means that the errors object has no properties. Now that we've declared our validation function, we can put it to use down here in handleSubmit. I'll use it on the second line, and I'll say if the form is not valid, then I can return early. There's nothing more to do here. Our console is getting cleaner now, but it points out the next step that we need to do here. We're not actually using the errors that we've declared in state yet. Let's pass those down to our form. So we can add those as another prop right here. This way our CourseForm will have access to the errors that we're now tracking in state. So let's jump over to the CourseForm and pass the relevant errors down to each one of our fields. And this is where you see why I've chosen to store errors as an object rather than an array, because I can say the error for this input is props.errors .title. If I had stored errors as an array, then it would've been a little more of a hassle to get the appropriate error for each one of these inputs. For our second input, we need to add some code to display the error, so I'll do that right down here. Say props.errors .author. If that is populated, then we want to show a div with a className equal to alert and alert-danger, and it should contain props.errors .author. Is it author or authorId? Let me check. If I scroll back up here, it's authorId. So I have a little typo here. I need to say errors.authorId like this, and I'll close the div. So this is the same error display pattern that we used on our TextInput. We just had to implement it manually here because we haven't created a reusable SelectInput. I'm going to leave that as an exercise for you if you'd like. Let's come down here and put our last error to use to display props.errors .category for our last input. That should do the trick. Let's jump over to the browser and see if it works. If I hit Save, ah, look at that. Validation for all three fields. And if I fill out the form, I can still save as expected. Excellent! Our form is coming together nicely, but we should be more explicit about the props that we expect to receive on our CourseForm. Let's declare a PropType so that other developers are clear what data is expected. And again, I'll just paste these in. So we're requiring a course onSubmit, onChange, and an errors object. You can see I have red squigglies because I need to come to the top and import PropTypes. Now if we forget to pass the expected props into CourseForm in the

future, we'll receive a runtime warning in the browser during development. Now this might seem strange for CourseForm since we're not really expected to reuse this component; however, remember that PropTypes are also useful because they help document our component's expectations. So these are useful for developers in the future so that they can quickly see what this component expects. We now have Add Courses working well. In the next clip, let's learn how to populate the form via the URL so that we can edit courses too.

Populate Form via the URL

We now have the Add Course functionality working well, but you've likely noticed one obvious thing that's missing is the ability to edit courses. If I click on an existing course, the form doesn't populate. This is actually very close to working right now. We need to update the ManageCoursePage to populate this form when it sees the course slug up in the URL. To make this happen, we need to call the API to request the course when the component mounts. If we were in a class component, we'd make the call in componentDidMount's lifecycle method, but since we're in a function component, we need to use a React Hook instead. The Hook that we can use is useEffect. So let's come up here and import useEffect. Then we can declare the implementation down here in our component. I'll begin by calling useEffect, and remember that we pass a function to useEffect. Inside this function, we can declare the code that we want to run when this component loads. First, let's read the slug from the URL like we were before. So we'll declare a constant to hold the slug and read it from props.match .params .slug. So this will be pulled from the path /courses/:slug. So slug now has a reference to the URL's second segment. And again, this property exists because over here in App.js we declared a placeholder within the route for this page. For example, if I change this placeholder to id, I'd have to change my code in ManageCoursePage to say params.Id instead. Now let's use this slug to request a course. If we open the courseApi file, we can see that there's a handy function in here that I've already declared called getCourseBySlug. It accepts a slug and returns a course. To understand how this works, we can request a course by slug from our mock API using a query string. If we jump over to the browser, as long as your application's running, you should be able to request localhost:3001/courses and then put in a slug. Now this slug doesn't exist, but if I put in a valid slug, then it returns the course based on that slug. This feature is built into JSON Server, so this function will keep our React component nice and simple. I prefer to keep code for making API calls centralized over here in the API folder. This way, I can keep my API call code consistent, and my React components don't contain any API-related complexity. So our React component will call this function and get a course back. Let's jump over to ManageCoursePage and do that. If a slug

is in the URL, then we want to request that course by slug. So we can say courseAPI.getCourseBySlug and pass it the slug that we just read from the URL. The getCourseBySlug function returns a promise, so we can handle the promise to get the result by chaining a .then on the end. This method will return a course, and what we'll do with the response is call setCourse and provide it the course. However, again, notice how we have a naming conflict here because this course represents the response that we're receiving, but it has the same name as the course up on line 8. So I'm going to use the same technique that I used earlier of putting an underscore on our local variable. So we will call setCourse and set it to this local course that we receive from the API. Now I'm using an inline arrow function here to keep things concise, but if you find it easier to read, you can declare this on a separate line. And that would look like this. Now I personally declare the single inline because it's a little less typing, so I'm going to undo that, but feel free to set it up whichever way you find clearest. Next, we need to declare our dependency array. We want this to run just one time, so you might thing that we'd come in here and put in an empty array like we did before. But notice that when I put in an empty array I get an ESLint warning. That's because we have a missing dependency. We're looking at props.match .params .slug. So since we are monitoring this value, we need to tell useEffect to keep an eye on it. So we need to say props.match .params .slug here since if this value changes, useEffect should logically rerun. Remember, by default, useEffect will rerun every time React re-renders, so it's important to declare a dependency array. If any of the dependencies listed in the dependency array change, then the effect will rerun. Since we're using values from outside of useEffect, in this case props.match .params .slug, it's important that we list it in the dependency array. So this is saying if the slug in the URL changes, we should rerun this effect. And notice that the ESLint warning has now went away. Again, it's very important to remember to declare the dependency array on useEffect. Without it, the useEffect will run every time that any state or props change, and that's likely not the behavior that you want. Let's jump to the browser and see if this works. Hey, look at that. Loaded right up. So I should be able to come back over to courses, I can click on a course, and it populates the form successfully. And if I change some data, maybe put the number 2 here and hit Save, that's reflected as well. If I come over here and look in db.json, I should also see that data is stored right here. There's the update that we just made. The app is coming together nicely. There's certainly still some room for improvement though. It would be nice if we showed the author's name here, but I'm going to save that as a challenge for you at the end of the course. For now though, let's wrap up this module with a summary before we shift our focus to Flux.

Summary

Let's wrap up by considering what we learned by building our first forms in React. We saw that any input that has a value is a controlled component, and if it's a controlled component, you need to create a change handler or the user's input won't register. Most of the time you'll want to work with controlled components, but any components that don't set a value are uncontrolled and thus operate like normal inputs. We created a low-level reusable TextInput component that abstracted away a native HTML input. This helps us enforce consistency in the UI, and it also avoids having to deal with the extra markup required by Bootstrap in multiple places. We saved our course data by making calls to the courseApi in the controller view, and we redirected to the Course page after saving a course using React Router's Redirect. We saw how to use an open-source React component. We downloaded React-toasify via npm to display notifications to the user without having to create our own notification component, and we added validation to our form to ensure that we get valid input. We used PropTypes to ensure all callers are clearly aware of the props that must be set. We saw how to use the Prompt component to prompt users before navigating away. Finally, we pulled a parameter from the URL to populate the form. We saw how React Router passed data down to the component via props. This allowed us to edit existing courses too. We now have a robust process for building forms. However, as your app gets larger, making data calls from your React components can start to create maintainability issues, and that's where the final capstone module of this course comes in. In the final two modules, we'll explore how to implement unidirectional data flows with Flux.

Flux

Intro

So far, we've learned how to create reusable components with React, and we've seen how to handle client-side routing with React Router. Now it's time to handle data flows throughout our app using Flux. Flux is a state management pattern that was popularized by Facebook. In this module, we'll explore Flux in detail, including actions for handling events, stores that hold app state, and a dispatcher that acts as the central hub.

What Is Flux?

Before we dive in, I want to clarify that the rest of this course focuses on Facebook's Flux implementation. Flux is Facebook's name for an architectural pattern that has unidirectional data flows and a centralized dispatcher. However, since Flux was introduced, many other developers have created alternative open-source implementations. We're going to discuss Facebook's Flux implementation because it's battle-tested in production on one of the highest traffic sites in the world, Facebook.com. It's also the inspiration for these other implementations that you see here. That said, you might see some things that you don't like about Flux. There's admittedly some boilerplate and plumbing code necessary. But I will say that once you get used to the pattern, it's easy to understand and quite scalable. Regardless, the great news is that if you decide that Facebook's Flux isn't for you, there are other options out there that use this same architectural pattern that you can consider. Redux is easily the most popular option on this list, and I recover Redux in detail in my Building Apps with React and Redux course. It's an excellent follow-up to this course. Once you've learned React and Flux, it's quite easy to pick up Redux. Earlier, I joked that they call it React for a reason. The name Flux also makes sense because Flux deals with actions and data changes in our application. Have you ever built a large client-side app that requires interactions among multiple views and models? As your app grows, you may find the traditional MVC pattern devolves into models and views that end up interacting with one another. You may find that you need to pass data between your view models. Doing so makes your app difficult to think about and thus unpredictable and tricky to debug. And this leads to the core sales pitch for Flux. Facebook ran into these issues and chose to utilize unidirectional data flows as a solution. Now, Flux is not a framework. It's really just the name for a pattern of unidirectional data flows, and this is why you see so many alternative implementations of this pattern with slight tweaks. With Flux, all updates to your app's state occur via a centralized dispatcher that, not surprisingly, dispatches data to the application stores. This pattern enforces clear, predictable, unidirectional data flows. Unlike popular two-way binding libraries, data flows in one direction with Flux. Now, a unidirectional data flow sounds complicated, but it just means the data flows in one direction. This is easier to understand by contrasting unidirectional data flows with two-way binding. If you've used older versions of Angular, Ember, or Knockout, you might be familiar with two-way binding. When you change the value of a text box, the corresponding data behind the scenes is updated automatically. It's quite handy and simple to understand. In contrast, with Flux, an action occurs. Then the dispatcher notifies any stores that have registered that that action has occurred. When the store changes, any React components that are listening to that data re-render to show an updated UI. When the user interacts with a UI element, a new action occurs, and this unidirectional flow starts all over. This is called unidirectional because updates flow in one direction, from the action, through the dispatcher, to the store, and ultimately to your React

components. Unlike two-way binding, the view doesn't directly update state. Instead, it fires off actions, which ultimately update state. Unidirectional data flows make your app easier to reason about because the results of a given action are easy and predictable to trace. Admittedly, unidirectional data flow does require some extra concepts and code, but at the benefit of clarity, testability, scalability, and predictability. So you can think of the trade-off like this: Two-way binding is simpler conceptually and requires writing less code. Unidirectional data flow pays off as your application grows because it's more explicit and makes it easy to update multiple stores when a given action occurs.

Three Core Flux Concepts

Flux may seem complicated at first, but to understand Flux, you really just need to learn three new concepts that we saw on the previous slide: actions, a dispatcher, and stores. Actions describe user interactions that occur in your React components. So for an example of an action, a user may click a Delete button. Actions are handled by a centralized dispatcher. The dispatcher is a singleton registry, and that sounds more complicated than it is. It's really just a centralized list of callbacks, so the dispatcher ends up making calls to stores. So, in summary, the dispatcher notifies all the stores that care about some action that just occurred. Stores hold your application's data. When the data in your Flux store is updated, your React components re-render to reflect the new data. This is a unidirectional flow. In other words, the UI never updates the data directly. Updates to the UI are ultimately rendered because of changes to the store, and the store can only be changed by dispatching an action. Now I know this may feel a little confusing at first. I know it did for me. So let's talk about each of these pieces a little bit more.

Actions

Actions encapsulate specific events that occur within your app. An action might be save user, delete item, et cetera. The dispatcher exposes a method that allows us to trigger a dispatch to the stores and to include a payload of data, which is called an action. Action creators are dispatcher helper methods. They describe all the actions that are possible in the application. Actions are passed to the dispatcher. You can think of action creators as dispatcher helper methods. They describe the actions that are possible in your app. Now actions are typically triggered in two places. First, when a user interacts with the user interface, the view will call the appropriate action. Typically, action creator functions are grouped together in files that have related actions. For example, a courseActions file might contain a list of action creators like loadCourses,

saveCourse, deleteCourse, and so on. Action creator methods add a type that is stored in a constants file. This type is used by the dispatcher to properly handle the action and pass updates to related stores. The second place that actions are triggered is from the server, such as on page load or when errors occur during calls to the server. Action payloads have a common structure of a type and some data. This is an example of an action that occurs when a user is saved. As you can see, it has a type of USER_SAVED, which is sent as a string, although it's often a good idea to create a constant to hold this string to avoid typos. It also contains data that consists of the user's first and last name. So this action payload sounds complicated, but it's really just an object. The data that you send here will vary depending on the action. Also, the payload doesn't need to be under a property called data. For example, here I've renamed the payload to be under a property of user. You can decide how to send your data. It's totally up to you. Only the type property is required.

Dispatcher

All data flows through the dispatcher. Think of it like a central hub. The dispatcher is a singleton, so there's only one dispatcher per app. The dispatcher, not surprisingly, dispatches stuff. So what does it dispatch? Actions. Stores register with the dispatcher so that they can be notified when data changes. So the dispatcher simply holds a list of callbacks. The dispatcher invokes the callbacks that have been registered with it, and it broadcasts the payload that it receives from the action. This effectively dispatches the actions to the relevant stores. This centralized dispatcher design creates a single point where multiple stores can request updates when a given action happens. This makes the application's data flow predictable. Each action updates specific stores based on the callbacks that are registered with the dispatcher. So basically, the dispatcher distributes actions to stores. The dispatcher exists to help get actions that you dispatch from your React components to your Flux stores. While we're discussing dispatchers, I should also mention constants. With Flux, it's helpful to create a constants file. This file helps keep things organized by defining constants that are used throughout your app in a centralized spot. And as you'll see, it also serves as a handy high-level view of all the actions that your app supports.

Stores

Stores are the place where our app's data is stored. Stores hold application state, logic, and data retrieval methods. However, a store is not a model like you'd think about in traditional MVC. Instead, a store contains models. This will make more sense in a moment when we look at an

actual example. Your app can have a single store or many, but as your app gets larger, you may find it useful to create multiple stores. This helps keep related data grouped together in well-named stores. With Flux, you can rest assured that the proper stores will be updated for each event that occurs in your user interface. Stores get updated because they have callbacks that are registered with the dispatcher. Remember, the Flux dispatcher keeps track of all the stores that would like to be notified when a specific action occurs in your app. Only your stores are allowed to register dispatcher callbacks. Your React components themselves should never be registered with the dispatcher. And as you'll see, Flux stores utilized Node's EventEmitter. This allows our stores to both listen to and broadcast events. It also allows our React components to update based on these events. Our React components listen to our stores. Stores emit changes using Node's EventEmitter, so this is how our React components find out that our application state has changed. So the React components that use a store's data are redrawn when the state changes. In summary, stores are just like they sound, a centralized place where the app's state and logic is stored. Stores have no direct setter methods. Instead, they only accept updates via callbacks that are registered with the dispatcher. So these callbacks say, hey dispatcher, when an action occurs, let me know about it. And this is a key concept. The store is the only part of Flux that can update data. The action that we dispatch doesn't know how to add or remove items. Only the store knows how. When stores are updated, they emit a change event so that the relevant React components know that a change to a store has occurred. Every Flux store has a common structure. These functions are the core interface that you always implement. You extend EventEmitter so that your store can emit events to tell your React components that the data has changed. You expose methods for adding and removing change listeners as well. This lets you easily say I want this react component to know when this store changes. And finally, you need a method to actually emit the change. Now remember, there may be multiple stores in a complex app. For example, you may decide to store your user, address, and product data in separate stores. This is a great example of where Redux's flow can really shine. In this case, the dispatcher will send the payload to all three of the stores so that they're aware of any actions that have occurred which might update the store's data. And remember, the dispatcher always dispatches the same payload. The payload has a type and some data. The type tells the store what action just occurred so that it can respond accordingly. As I mentioned earlier, the Flux pattern admittedly requires extra code. You have to register callbacks with the dispatcher, which is a bit of extra work. But as your app grows, this explicit one-way data flow is really helpful. As the Flux docs point out, "As your app grows, the dispatcher becomes more vital as it can be used to manage dependencies between the stores by invoking the registered callbacks in a specific order. Stores can declaratively wait for other stores to finish updating and then update themselves

accordingly. " This quote is a reference to the wait for API that is part of Flux. This API allows you to specify the order that your stores are updated for a given action. This is useful when there are dependencies between your stores. Next, let's talk about controller views.

Controller Views

We discussed the concept of controller views a bit in the last module, but I want to mention it again here because it has implications when working with Flux. React has the concept of a controller view, and it's the top-level component that often ends up composing child components. Controller views control the data that flows down to all child components, so the name controller view makes a lot of sense. I mention controller views here because they're the components that should interact with your stores. When a store emits an update, the controller view should receive the update and pass the updated data down to its children. Child components are automatically updated via their props. Remember, controller views hold data in state and send data down to children as props. In React, you can nest components as deep as you like but it's recommended to have a single top-level controller view that interacts with the store, holds date in state, and passes the necessary data down to children. Sometimes it can be helpful to pass a single object down to children via properties so that you don't have to change code in multiple places as new properties are added to data objects in the future. Although it's possible to nest controller views, it's not recommended because it can cause multiple data updates and cause React's render method to get called multiple times, which can hurt performance. So I recommend having a single top-level controller view per page or a single controller view for each major portion of the page. This means all child components simply receive data from the parent via props. This keeps them dumb and focused solely on presentation. This is how we wired up our components in the previous module that introduced React.

Flux Flow and a Chat With Flux

Let's consider a granular example. The flow begins with an action occurring. Imagine that I click the Save User button in the user interface. This generates an action. When an action occurs, the action sends a payload to the dispatcher. It's an object with a type property and some sort of data. Here's an example of a payload that might be sent by the action if I clicked Save User. The dispatcher checks its list of registered callbacks to determine which stores should receive the payload. Remember, stores register with the dispatcher to say, hey dispatcher, when this action

occurs, I want to know about it. Since the user clicked the Save User button, the payload's type was USER_SAVED. So the dispatcher looks for any registered callbacks that care about users being saved. The dispatcher sends the payload that it received to all the stores that have registered callbacks for that action. The store updates its internal storage based on the payload that it received. In the case of saving a user, the store will update the user's data. If a new user was created, the user's key would now be reflected in the data set. Once the store is done processing the update, it emits a change event. Emitting the change event notifies React that data has changed, so now React knows that it needs to re-render the UI to reflect that new data. Of course, any new activities in the UI may generate a new action and cause this unidirectional flow to start over again. And although I didn't mention an earlier, actions commonly make calls to web APIs to get and receive data as well. And that's it. Okay, saying that's it makes it sound simple. I know it likely doesn't feel that way yet. There's a lot of pieces here which can feel intimidating at first, but trust me, the clarity of being explicit and having all data flow in one direction makes even very complicated apps highly predictable. This unidirectional flow requires more typing, but once you understand these concepts, it creates code that's easy to navigate and reason about because it's very explicit. Let me close out this module with a unique way of thinking about Flux, like it's a group of people talking to each other. We just went over the core players in a Flux application: actions, the dispatcher, stores, and React. That's a lot of concepts, so it's easy to get confused at first. So I find it helpful to think about actions, dispatchers, and stores as these three different people with different roles who interact with each other. Here's an example conversation that I like to play through my head. React says, Hey CourseAction, someone clicked this Save Course button. The action says, Oh, thanks React. I registered an action creator with the dispatcher, so the dispatcher should take care of notifying all the stores that care. The dispatcher says, Let me see who cares about a course being saved. Ah, looks like the course store has registered a callback with me, so I'll let her know. The store says, Hi dispatcher! Thanks for the update. I'll update my data with the payload that you sent. Then I'll emit an event for the React components that happen to care. Finally, React says, Ooo! Shiny new data from the store! I'll update the UI to reflect this. That's the way these four players interact with each other. Let's close out by looking at the full Flux API.

Flux API

As I said, Flux is really just a design pattern, so once you understand the philosophy of the pattern, there's not much actual code to review in Facebook's Flux. The dispatcher is a singleton that registers callbacks. These callbacks are called when actions happen in your application. So

the core API for Facebook's Flux contains only five functions, all of which revolve around the idea of registering callbacks and calling them when actions occur. Let's look at these five functions. The first is register. The register function accepts a callback. This is how you tell the dispatcher when this action happens, I'd like you to call this callback. Unregister, not surprisingly, removes a registration from the dispatcher, so this is how you tell a dispatcher to stop calling a callback when an action occurs. WaitFor is a way to run callbacks in a specific order. The idea here is if multiple callbacks are registered with Flux at the same time, then you may want them to be called in a certain order. This API allows you to specify the callbacks that should be completed before continuing execution of the current callback. The registered callback that you'd like to wait for is specified by passing the dispatch token of the callback that you'd like to wait for to the waitFor function. Dispatch actually dispatches the payload to all registered callbacks. This is how an action tells the dispatcher that something happened. So basically, this is how the action says, Hey dispatcher, go tell all the stores about this thing that just occurred. The payload is an object with two properties, an action type and the data itself. Finally, isDispatching is a Boolean that is true when the dispatcher is busy dispatching things. When you hear the term centralized dispatcher, you might imagine a traditional publish-subscribe pattern. Flux's pattern is similar, but it differs in two key ways. First, callbacks are not subscribed to particular events. Every payload is dispatched to every registered callback. And second, callbacks can be deferred in whole or in part until other callbacks have been executed. This is useful when you need to ensure that one data store has been updated before another is updated. Implementing this behavior involves using the waitFor API that I showed on the previous clip. And that's the Flux API. Before we jump into code in the next module, let's close out this module with a summary.

Summary

In summary, Flux is a pattern for unidirectional data flows that's composed of three core pieces: actions, which encapsulate events, such as a user clicking on a button; stores, which hold application state, and you can have one store if your app is simple or multiple stores if you find that it helps; and a dispatcher, which is the centralized hub. It's a singleton, so there's only one per app. The dispatcher holds references to callbacks so that it knows what stores to notify as actions occur. Once the store is updated, any subscribed React components are notified when the dispatcher emits an event. Flux has many alternative implementations that are similar to what we just discussed, but the most popular is Redux, which I explore in Building Apps with React and Redux. For our final module, let's put this knowledge to use. We're going to use Flux to handle data flows in our course management application.

Flux Demos

Intro

Welcome to the final capstone module for the course. In the last module, we learned about Flux. We discussed the core concepts in Flux including actions, dispatchers, and stores, but that's enough concepts because it's time for some action. Let's get back into the IDE and update our app to put Flux to use. This entire module is coding. We're going to build out actions for creating, updating, and deleting courses, emit those actions via the dispatcher, and create a course store to handle data changes and store course data. And we'll attach React to the Flux store so that it's notified and re-renders when the store changes.

Dispatcher

To get started with Flux, let's create a file directly under src called appDispatcher.js. Inside, let's begin by importing the Dispatcher from the Flux package. Then we can create an instance of it, and finally, export it. Simple enough. Now we have a single dispatcher that the rest of the app can use. It's a singleton. So, in other words, there's only one dispatcher per application. The dispatcher is the central hub for the app. This dispatcher will hold a list of callbacks, and all our app's actions will be dispatched via this dispatcher. The stores will register with this dispatcher to say that they'd like to be informed when actions occur. As you can see, I'm making no changes to Facebook's Flux dispatcher. I'm just referencing it and instantiating it. Now that we have the dispatcher covered, in the next clip, let's look into actions.

Actions

We've implemented the dispatcher, so let's move our attention to actions. To get started with actions, let's create a folder called actions over here under src. I find it helpful to create a file for related actions. This file will contain actions that are related to courses, so let's create a new file that's called courseActions.js. Note that the file name starts with a lowercase character here since it's not a React component, and it doesn't contain an object that you instantiate. First, let's import the dispatcher. Let's implement our first action for saving a course. To do that, we'll export a function called saveCourse, and it will accept a course. Nothing too interesting so far. This is a plain function. Now, think about this. What is the first step in saving a course? Well, that's calling our API, so let's jump back to the top and import our API. I'm going to use a wildcard import

again. The nice thing about a wildcard import is we can reference multiple values out of this file, and we don't have to maintain an ongoing list of imports up here on line 2. Now we can call courseApi to save a course and pass it the course that's passed into this function. Remember, this function returns a promise, so we can use .then to handle the response. What we'll get back is a saved course. At this point in the code, the course would've just been saved, so we can put Flux to use and dispatch an action. We can call the dispatch function on the dispatcher, so say dispatcher.dispatch. The dispatcher expects us to pass an action. Now, remember, an action is an object with an actionType property, so inside here we can declare that action. We'll set the actionType equal to CREATE_COURSE. I'm going to put this in all caps just to emphasize that it is a constant. But to clarify, Flux doesn't require this to be in all caps. Now, the rest of our action can have as many properties as we want. ActionType is the only required property. We want to pass the savedCourse as part of the action, so let's add that as a second property to this action. I'll set the course equal to savedCourse. Let me clarify some terminology here. This entire function is our action creator. What we've placed right here in the call to dispatch is called the action. So the name makes sense. This function creates an action. That's why it's called an action creator. I've declared the actionType as a magic string. Now this will work, but the problem with using a magic string is I have to carefully type it the exact same way when I declare my store, which will handle the action. So instead, it's recommended to create a constants file. The constants file will contain a list of all the actionTypes that we use in our system. So let's create a new file over here under actions, and we'll call it actionTypes.js. This file will export a list of our app's actionTypes. Let's add our first actionType to the file. Export default, and we'll declare this as an object. The first item will be CREATE_COURSE, and we'll set it to a string that has the same value, CREATE_COURSE. Now this might seem silly, but this is a more strongly typed way for us to have a list of actionTypes that we use in our system. By declaring this, we avoid typos, and we get autocomplete support. As we add other actions to our app, we'll add our other actionTypes right here. Another nice thing about this file is that anyone new to our app can look at our actionTypes file to see a list of different actionTypes that happen in our application. Now that we've declared this actionType constant, we can put it to use over here in our courseActions file. Let's import actionTypes. Then, down here, instead of having a hard-coded string, we can call actionTypes, and notice how I now get autocomplete support, so I can't make a typo anymore. Let's make one final tweak. It's a good idea to return the promise that's generated right here from calling courseApi.saveCourse. This way the caller will be notified when the promise resolves. For instance, we could display a loading message on the course form while the save is in progress and hide it when the saveCourse API call returns here. We could also notify the user on the course form if the API call fails. To keep things simple, I'm going to leave these details out of this course,

but to see an example of how to handle loading and error states, check out the more advanced React and Redux course next. The patterns that I use in the Redux course apply to Flux as well since their general patterns for handling async errors, confirmations, and loading states. But for this course, we'll keep it simple and focus on the Flux fundamentals. We've now created our first action creator, so think of the action creator function as a handy helper that wraps our actions. It makes it easy to create these actions and have them dispatched through the dispatcher. So, in summary, right here, this is saying, Hey dispatcher, go tell all the stores that a course was just created. We haven't created any stores yet, but effectively, that's what this call says. So when this code runs, any stores that care about this action will be notified, and they can update themselves accordingly using this payload. In the next clip, we can move on to exploring stores. Stores will take the data that we're dispatching here and put it to use.

Stores: Change Listeners

We've set up our first action. We've created an actionTypes constants file, and we've implemented a centralized dispatcher using Facebook's Flux dispatcher. Now it's time to create our store. As we covered in the slides, you can have one store per app, or you can create multiple stores as your app grows. For now, our app is solely working with course data, but this app might manage author data too in the future, so for the first store, let's give it a specific name. Let's create a courseStore. To do that, in the src directory, let's create a new folder and call it stores. Inside, let's add a new file call courseStore.js. Inside, let's declare our store as a class called courseStore. Our store needs to emit events each time a change occurs, so let's extend a base class that gives us this behavior. Node includes a class called EventEmitter that will allow us to emit events, so let's import it. Again, since this is built into Node, we don't have to install it. Now that we've imported it though, we can use it to extend our class's behavior using the extends keyword. Now our class will have access to all of EventEmitter's capabilities. The Node docs outline the methods provided on EventEmitter. We're going to call three things from Node's EventEmitter in each of our Flux stores. Let's review each. We'll call on. On adds a listener to the array of listeners. We'll call removeListener. This removes a listener. And we'll call emit. Emit calls each registered listener. Every store needs to provide a way for our React components to interact with the store, so we're going to call these three methods that are provided by EventEmitter over here inside our Flux store. By convention, each Flux store should have three functions that call these EventEmitter functions: addChangeListener, which wraps EventEmitter's on method; removeChangeListener, which wraps EventEmitter's removeListener method; and emitChange, which wraps the emit function. Next, let's declare each. We'll start with addChangeListener. It will

accept a callback. And inside we will call this.on and pass it the event that we want to watch for, which is change. When a change occurs, we will call the callback. So with this function, when a change occurs in our store, we'll call the callback provided. Remember, the on method we're calling here is provided by EventEmitter, the base class that we're extending. So this method will allow React components to subscribe to our store so that they're notified when changes occur. In summary, this function will be useful for our React components to say, hey, I'd like to know when this store changes. Whatever callback gets passed in is going to get called anytime that things change in our store. Next, let's do the opposite and implement removeChangeListener. Now let's do the opposite and implement removeChangeListener. This is the mirror image of addChangeListener. So we'll call it removeChangeListener, and it will accept a callback. We're going to call this.removeListener and then pass it change for the event and the callback as the second argument. This function will allow our React components to unsubscribe from the store. So now we've defined a way to add change listeners and remove change listeners. These functions will allow our react components to subscribe and unsubscribe from the store. Next, we need a method that will emit a change event, so let's come down here and declare emitChange. This method will call this.emit So this calls the emit function that's provided by EventEmitter. Notice that in all three methods we're passing a hard-coded string for the event type, which we set to the word change. To avoid repeating strings, let's declare a constant up here at the top. I'll declare a const CHANGE_EVENT and set it to the string of change. Now I can update my call sites down below to no longer use a magic string. Notice that as I change this I get autocomplete support in each one of these spots. So now I can't make a typo. These three methods are utilizing nodes built in EventEmitter class's functionality. We're just defining some handy helper methods here because our React components care specifically about the change event occurring within our stores. Now that we've declared the class, let's create an instance of the store class down here below. And I just realized I should be more consistent on my casing. Traditionally, this should be an uppercase C, so I'm going to change that to say CourseStore. Since we're instantiating it, the Pascal case is the common convention. Now that we've instantiated the store, on the last line, let's export it. Since it's the only thing exported, we'll export it as the default. So we've now defined our store, but there is a piece missing. We haven't registered with the dispatcher so that the store will emit when actions occur. In the next clip, let's set up the store registration.

Stores: Registration

In the previous clip, we set up three core functions that are part of every Flux store: addChangeListener, removeChangeListener, and emitChange. But there's another piece that's

part of every store. We need to register the store with the dispatcher so that the store is notified when actions occur. First, let's import the Dispatcher. The dispatcher registration is typically defined below the store itself since it's a concern that's private to the store and thus not part of the public API, so let's jump down here. We can register the dispatcher right above the export here. We'll begin by calling Dispatcher.register. The register method accepts a function that takes an action, so I'm going to use an arrow function. This function is going to get called anytime that an action is dispatched. And by that, I mean any. This is a place where Flux's dispatcher implementation differs from traditional publish-subscribe patterns. Every store that registers with the dispatcher is notified of every single action. So here, we're going to need to add some logic that switches based on the actionType that's being passed. Let's declare our switch statement, and we're going to switch based on the actionType that's passed in. Within this switch statement, we'll have a case statement for each one of the actionTypes that we want this store to handle. The only actionType that we've declared so far is saveCourse. Before I move on, I want to stop for a moment and discuss this structure. We now have all the boilerplate code set up for our store. Yes, I just said boilerplate. We still haven't added any code that's specific to this store yet. So what you're seeing here is the basic shell of any store that you create. Now you might be thinking, wow, that's a lot of boilerplate. And you're right, but remember this: If you create a large app with multiple stores, you can, of course, use various design patterns to eliminate the boilerplate you see here. I'm deliberately showing you the full store implementation here to ensure that you're clear about what stores are and how to create them. That said, Redux is a popular Flux implementation to consider. It avoids some of this configuration code. The great news is by understanding actions, the dispatcher, and stores that we're creating here, it's easier for you to pick up Redux if you like. Redux builds upon these same principles. Oh, and everything that we've done here, we only have to do once per store. Adding more features to our store will be much quicker as you're going to see in a later clip. Next up, let's configure the store's private storage.

Stores: Private Storage

Our store isn't very useful yet since it's not storing anything. Let's fix that. Since this is the courseStore, it's going to store course data. Let's jump up to the top and declare a private array. We'll say let _courses equal an empty array. This is enforced as private too since it's not exported outside of this module. Only the CourseStore is down below. And this is a good thing because it means that no one can mess with the data in the store unless they interact via the public API, which are the functions that we're going to declare that sit within the courseStore. Each time a courseStore is created, the CREATE_COURSE action will be dispatched. It will include the new

course as part of the payload. So let's take the new course that will be passed with the action and push it into our private array of courses down here where we register the dispatcher. First, we need to import actionTypes. Then let's add our first case statement down here in the switch. We're going to check for the CREATE_COURSE actionType. So we'll say case actionTypes.CREATE_COURSE, and notice again how I get autocomplete support right here. And I'll add the break, which is important for each case statement. To add the new course to the Flux store, we can push it into the private array of courses. So we'll call _courses.push and pass it action.course. We're going to add support for updating courses with a separate actionType later. Let me clarify where action.course came from. If I open up courseActions, remember, this was the payload that it dispatched. It dispatches an actionType and a course. So we know we can get the course off that payload, and that's what we're doing right over here, action.course. So these are effectively glued together through the dispatcher. This action is dispatched right here on line 8 via diapatcher.dispatch, and then it's caught over here on the other side via dispatcher.register. So now we've written code that will store a new course within our store's private courses variable. The other thing that we need to do is emit a change, so we can call courseStore.emitChange. We'll do that right below the call to push. We'll stay store.emitChange. So this is calling the emitChange function that we defined up here above. This is important, so pay attention. Anytime that the store changes, we need to call emitChange. We just changed the store by pushing a new course to the array. By emitting a change, any React components that have registered with the store will be notified, so they'll know that they need to update the UI accordingly. More specifically, any stores that ever called addChangeListener will then be notified anytime that I call emitChange. Now, you can see that we have these green underlines down here. That's because the linter is unhappy. If I hover over, you can see that it expects me to implement a default case. So let's do that. For a default case, we have nothing to do, and this may seem strange to you, but remember, we may create multiple stores, and every store's dispatcher receives every action. So if none of the cases above match, then there's nothing to do here. In other words, some action just occurred that some other store cares about. This is easier to understand when there are multiple stores. Okay, one final piece, and we'll have our initial build of the courseStore completed. We need to expose the course's data. Remember, right now the course array is private. I deliberately created a private variable up here at the top to keep people from messing with the data directly. Let's create a getCourses method in the courseStore. I'll define it right here below emitChange. The implementation will be very simple. I will return the private courses array. There's one other function that would likely be useful, and that's getCourseBySlug. It will receive a slug, and then we need to write some code that would look through the array of courses to find the corresponding course that has that slug. To do that, we can use JavaScript's find method. So we can say

_courses.find. And find accepts a predicate. A predicate is a fancy word for a function that returns a Boolean. We're going to look through this list of courses, and we want to find the one that has a slug that's equal to the slug that's passed in. Now note what I'm doing here. I'm declaring handy functions that return a portion of the Flux store. You can add as many of these as you like. You can imagine how other parts of the app might want the courses sorted by title or filtered by category, so you could add other functions right here. This way, the rest of the app can easily request the data that it needs. I like to think of Flux stores a bit like a table in a relational database. Like a table, they hold data, and you can think of these functions a bit like views or stored procedures in a SQL database. They return a specified subset of data. Now that we've finished creating our store, let's move back to the ManageCoursePage in the next clip and put the store to use.

Stores: Interactions

We've wired up the unidirectional flow for saving a course using actions, a dispatcher, and our first store called CourseStore. Now let's update our ManageCoursePage to put the Flux store to use. First, we don't need to call the API directly anymore since the courseStore holds the full list of courses. For the ManageCoursePage, instead, we need to get a single course by slug, so we'll be able to call the getCourseBySlug method that we created earlier in the courseStore. So let's remove some code that we no longer need. We can remove this courseApi import, and instead, what we want to do is import the courseStore. Now let's jump down to useEffect. In useEffect, we were calling the API, but instead, now we can query the store. We're still going to call setCourse like we did before, but instead of passing it the results of making an API call, we're going to query the courseStore and request getCourseBySlug and pass it the slug from the URL. Remember, getCourseBySlug returns a course with the specified slug from the Flux courseStore. We can pass the result of calling that function to the setCourse function. Next, let's jump down to the handleSubmit function. In the handleSubmit function, we're calling courseApi.saveCourse. Now, instead of calling the API, we should call the createCourse Flux action. So let's jump back to the top first and add another import. What we need to import is the courseActions. I'm going to use a wildcard import for this. And now we can jump back down to the handleSubmit function and put this to use. Instead of saying courseApi.saveCourse, we can say courseActions. We're going to pass the same parameter, which is the course that's within state. That's the only change that we need to make to the manage course state. Now we're using actions and our courseStore instead of directly calling the API. Now, that said, we're not done migrating the entire application to Flux, but we should still be able to insert a course using the saveCourse action, and it should work the

same as before, so let's try it out. If you're not running the app, make sure you start it. I'm going to click on Add Course and enter some data. Scroll down, and hey, it still works. The new course has saved successfully, so great, it's still saving. However, the course page should be updated to utilize our Flux store as well. Right now it's calling the courseApi every time on page load. So let's update the CoursesPage component to utilize our courseStore. First, we can remove this reference to the courseApi and instead add a new import for the courseStore. Then, down here in useEffect, we're currently getting all courses from the API to set the initial state. Instead, we can refactor this a bit. We can call setCourses and pass it the result of calling courseStore.getCourses. So this will either set the array of courses to an empty array if the Flux store is empty or to the full list of courses if the Flux store already contains courses. That's the only change that we need to make on the CoursePage. Pretty simple. Let's jump back to the browser and make sure that we can still add a course. Of course, the first thing that you'll notice is our list of courses is now empty. That's because we're not populating the Flux store with the list of courses from the API yet. We're going to address that in a moment. But for now, let's just make sure that we can still add a course to the Flux store. Great, we can still create courses just fine, but we're getting these courses from the courseStore instead of making a call to the API. In fact, let's open up the DevTools so I can show how this works. If I click on the Network tab, notice that as I navigate between Home and Courses no API calls occur because we are now getting this data from the Flux store instead of making an API call on every single page load. So now we have unidirectional data flow for saving courses working with Flux. In the next clip, let's fix the initialization of the Courses page by using Flux actions to initialize the app.

Stores: Initialization

In the previous clip, we converted to using Flux; however, the list of courses isn't displaying on initial load anymore. Why? Because the Courses page was calling the API to get the course data on page load previously. But now thart we've moved to Flux, we need to tell Flux to load the course's API. We can do this by adding a LOAD_COURSES action to the courseStore. Before we do, I want to ask you to pay attention. Adding our first feature to Flux was admittedly quite a bit of work, but now that we have our courseStore set up, note that it will take us much less work to add this second feature. Let's begin. First, open up actionTypes.js. Let's create a second action called LOAD_COURSES. Now, open up courseActions. Let's add this new actionType. We're going to export a new function that's called loadCourses. The body of this function is going to be very similar to the saveCourse action creator except we're going to make a different API call and dispatch a different action. To show the differences, let's copy the body of saveCourse and paste

it down here. Now we need to make a few tweaks. I'm going to remove the comment. I'm going to rename saveCourse to getCourses. It won't take a parameter, and what we'll receive back is an array of courses. Then the action that we want to dispatch here is going to be actionTypes.LOAD_COURSES instead. And instead of passing along a single course, we're going to pass along the courses that we just received back from the API. And on line 19, since the left and right-hand side match, we can use the object shorthand syntax and omit that right-hand side. But to avoid any confusion, I'll go ahead and leave this as is. Next, let's update the courseStore to handle this new actionType. We'll want to come down here to the bottom where we have our case statements, and we need to add a new case. This case will be fore actionTypes.LOAD_COURSES. In this case, we want to set the private array of courses to action.courses. And again, be sure to call store.emitChange. This is very important. You don't want to forget this one. Finally, I'll put in the break. Now we can put this new Flux action to use in our app. We have a choice to make. When do we want to load all the courses? There are a couple of options. We could load the courses when needed, or we could load courses immediately when the app loads regardless of which page the user is on. The benefit of approach number one is that you don't request courses until you actually need them, but the downside is it's a bit more complex because you need to check if courses have been loaded on each page that needs course data. If your app needs the same data on many pages, then it may be best to load the data the moment that the app loads regardless of page. For our app, let's go ahead and use option one. This way I can show you how to handle lazy loading data. First, let's update the CoursesPage. Our goal is to subscribe to the Flux store, and if courses haven't already been loaded, we need to call the loadCourses action when this page mounts. To subscribe to the Flux store, we need to call addChangeListener when this component mounts. We can do that down here in the useEffect Hook. So we'll call courseStore.addChangeListner. And addChangeListener accepts a function that it will call when the store changes. Let's declare that function below and call it onChange. So I'm going to pass onChange as an argument to addChangeListener. Now let's go down here and declare it. So what should we do inside the onChange function? Well, this component displays a list of courses, so anytime that the courseStore changes, we want to get the list of courses and update state. We're doing exactly that up here in useEffect where we call setCourses. So let's move this line down here into the onChange. Our component is now connected to the Flux store, but remember the Flux courseStore is initialized to an empty array, so we need to request the list of courses if this page is being loaded for the first time. So let's call the loadCourses action that we recently added to the courseStore. First, we need to import it up here, so we'll import loadCourses from the actions file. Then we can use this down here in useEffect. I'm going to check if the courseStore has any courses by saying courseStore.getCourses () .length === 0. So if

there are no courses in the courseStore, then let's call loadCourses. Since our component is connected to the Flux store, when courses are added to the store, the onChange function below will be called. It will request the courses from the store and then add them to this component state using setCourses. There's one other Flux-related concern to handle here. When you add a change listener on mount, you should also clean it up when the component unmounts. With useEffect, you declare the code that you want to run on mount by returning a function. So let's return a function that calls removeChangeListener from our useEffect call. I'm going to declare it as an arrow function, and we'll call courseStore.removeChangeListner and pass it, again, onChange as an argument. So this will clean up on unmount. This cleanup function will be called when our component unmounts. In other words, when we navigate to a different page. That should do the trick. Let's try it out. The list of courses is empty. That's because we're initializing the list of courses to an empty array. So back over on the CoursesPage, up here, instead, we need to initialize our state to use the data in the Flux store. So, remove this empty array, and instead, let's initialize it to courseStore.getCourses. So now, each time the page is loaded, courses will be initialized to whatever list of courses are in the Flux store. Now we should be able to navigate over to Home and back to Courses, and notice that it loads successfully even when we navigate back. If we open up the Network tab, we should find that we can navigate between Home and Courses, and notice how no new network requests are created because we already have the data in the Flux store. So our app is more efficient now as well. We now have the Courses page and Add Course function working with Flux, but we haven't implemented UPDATE_COURSE yet. So, in the next clip, let's get UPDATE_COURSE working with Flux.

Update Course Flow with Flux

In the last few clips, we've gotten Flux working to populate the Courses page and to create a new course; however, in the process, we broke editing courses. To fix the edit course feature, we need to add support for updating courses to Flux. First, open up actionTypes and add a new actionType for UPDATE_COURSE. Then open up courseActions. Currently, the saveCourse function always dispatches the CREATE_COURSE actionType. We need to tweak this code. It should dispatch UPDATE_COURSE if it's working with an existing course. So how do we know whether to call UPDATE_COURSE or CREATE_COURSE? Simple. If the course has an ID, then we know that we're updating an existing course. So let's tweak the actionTypes that gets passed up here in saveCourse. So we will look at the course ID, and if there is one, then we want to call actionTypes.UPDATE_COURSE ; otherwise, we want to call all actionTypes.CREATE_COURSE. We're now dispatching a different actionType for course updates, so let's enhance the

courseStore to handle it. We can jump back down here to the bottom because we need another case statement in here. The case statement we need is going to be case actionTypes.UPDATE_COURSE. The question is, how do we want to handle updating a course in the array of courses? One way we can do it is to map over the array of courses, and when we find one that was just updated, we can replace it in the array. Remember, the map method is built into JavaScript's array prototype. Map allows us to create a new array by iterating over an existing array. We specify what value to put in the new array for each element currently in the array. So let's make that happen. We're going to update the private array of courses, and we're going to do that by mapping over the existing array. And for each course we're going to say, hey, is the course ID equal to the action's course ID? And if it is, then that's the one that we want to replace. So we will replace it with the course on the action; otherwise, we'll just leave it as is. So map will iterate over each one of the courses, and for each course in the array, it will say, is this the course that we're looking to replace? And it does that by looking at the ID on the action and comparing it to the ID on the course that we're currently looking at. If it's the course we want to replace, then we replace it; otherwise, we do nothing. We return the course as is. Now our linter's reminding us that we need to put in a break here. And also, don't forget store.emitChange. Let's hit Save and then check it out in the browser. We should be able to select a course. I'm going to put a 2 on the end of this. Hit Save. There we go. Our edits work successfully too. Next, let's update the ManageCoursePage to work with Flux too.

Adding Store Listeners

There's another bug lurking that you might not have noticed yet. If we look at the ManageCoursePage, it's requesting data from the Flux store down here in useEffect. However, if I click on one of these existing courses and then I come over here and reload this page directly, notice that it fails. And it fails because our component isn't properly connected to the Flux courseStore yet. When the page is loaded directly, it requests the course by slug, but there's no code in here to populate the Flux store with the array of courses first. To fix this, we can use a similar pattern as the CoursesPage. Let's store the list of courses from the Flux store in state. So we can come up here and declare another useState declaration. We'll store courses in an array called courses and have a setter called setCourses. We'll call useState, and we'll initialize the state by calling courseStore.getCoruses. So now the courses array will be initialized to the list of courses in the Flux store. Here's the plan. If this page is loaded directly and there are no courses in the Flux store yet, then we need to ask the Flux store to load all the courses, and then we can save those courses in this array on line 9. Now let's jump down to useEffect and properly connect

to the Flux store. So we need to call courseStore.addChangeListener and pass it onChange, which we will implement in a moment. But before we do, let's come down here and also implement our code to run to remove the change listener. Remember, the function that we return from useEffect will be run when the component is unmounted. So here we want to call courseStore.removeChangeListener, and again, we'll pass it onChange. Now let's declare the onChange function. I'll say function onChange. Inside, we need to request the list of courses from the courseStore and store it in local state using the setCourses function that we declared up above. So we'll call setCourses and set it to the value that we get from courseStore.getCourses. So now, anytime the Flux store changes we will update our local array of courses with the data from the Flux store. Now the problem is if the user loads the page directly, courses may not be loaded in the Flux store yet, so up here in useEffect we should check for that. So what we want to say is if courses.length is equal to 0, then we should request courses, so let's say courseActions.loadCourses. Then this check can be separate. We will say else if there's a slug in the URL, go ahead and call setCourse, and get that course from the courseStore. Now notice that ESLint points out down here that our dependency array is no longer sufficient because we are now calling courses.length. If I hit Save, then that dependency array is automatically fixed and now lists both courses.length and the params.slug. So let's talk through this. We call courseStore.addChangeListener to say I want to run the onChange function that we've declared down below when the courseStore changes, and we clean up down here by returning a function. On page load we check if there are any courses in state yet, and if there aren't, then we call courseActions.loadCourses. When that call is complete, the onChange function will be called and update the array of courses. Since courses.length is listed in our dependency array, useEffect will run again. Remember, the dependency array basically says if anything in this array changes, rerun this effect. The second time that useEffect runs, courses.length won't be 0. So if there's a slug in the URL, it will ask the store for the relevant course, and that course will now be there because the array of courses has already been populated in the Flux store. So now we should have the Manage Course page working. If I come over here and refresh, notice that now it loads just fine. I can come over here. I can view a course, and I can refresh, and load a deep link to a particular course directly. We're in good shape. We have create, update, and add working properly using Flux. The one obvious missing piece is delete. Let's close out the course by implementing course delete functionality.

Delete Course via Flux

To complete our Flux implementation, let's add support for deleting courses. We can add a Delete button here on the Courses page. To do that, jump over to the code, and let's open up CourseList.js. By now, you have all the knowledge that you need to try this on your own, so consider pausing this video and trying this first before you watch this clip. So if you want to try this on your own, pause the video now. All right, ready to move forward? Let's implement delete functionality. To support course delete, let's begin by opening up actionTypes.js. We need to add DELETE_COURSE right here so that we have a new actionType to handle this. Then open up courseActions. We need to implement the DELETE_COURSE action. This action is going to call the API and then dispatch an action with a type of DELETE_COURSE, and it will pass the deleted course's ID along for the ride. This will look quite similar to our loadCourses function, so I'm going to copy it and paste it again. This time it will be called deleteCourse. Remove the s. Method that we want to call here is deleteCourse, and it will accept a course ID. We also need to accept the course ID on the function itself. It won't return anything, so we'll have an empty return here. For our dispatch, the actionType will be DELETE_COURSE, And we can go ahead and pass the ID along in the action as well. Next up, we need to enhance the courseStore to handle this new actionType. So let's go to the courseStore, scroll down here. It's for actionTypes.DELETE_COURSE. So how do we delete a course? Well, I suggest using JavaScript's filter function. So we will say _courses is equal to _courses.filter. And filter works a lot like find. Again, we pass it a predicate that determines what records we want to filter out. So we will say for each one of the courses, if the course.id is not equal to the action.id, then it should be filtered out. Now there is a little tweak we need to make here. The action.id may be a string when it's passed over, so for safety, we should call parseInt to parse this as a number. And it's a good idea to pass a radix for the second argument to parseInt. We're dealing with Base-10 numbers here, so I'll put a 10 in as the second argument to parseInt. So this code says iterate over all of the courses and filter out any course that has the course ID that was just deleted. So this will give us a new array of courses with one less course in it. We need to be sure to call store.emitChange, and don't forget the break. One other interesting point here. If you want to show a preloader in your app, one easy way to do that is to have a separate COURSE_DELETED action that fires as well. So, for instance, you could immediately fire your DELETE_COURSE action, and then once the async delete call completed, you could fire off a COURSE_DELETED action. Note the past tense. That way, all of your UI would be aware that there were an asynchronous call currently in process. And once it received the COURSE_DELETED actionType, it would know that it should hide any preloaders and show any final confirmations. You can see that approach in Building apps with React and Redux. Now that we have Flux configured to handle deletion, we're ready to implement the UI. Let's open up CoursesPage, and up here we need to import a second action.

We want to import the deleteCourse action now. Then we can pass it to the CourseList via props right here. Let's call the prop deleteCourse, and we'll pass down deleteCourse. Finally, we need to add the Delete button to the course list. But before we do, you might be wondering, hey, why pass the deleteCourse down to CourseList? Why not just import deleteCourse on the CourseList component? Well, that's absolutely fine too, and it's a bit simpler as well, but it would make the CourseList component a little less flexible since it would couple the CourseList to Flux. Part of this decision comes down to how likely it is for the CourseList component to be reused and in what way. With our current approach, the CourseList component that you see here remains just JSX, so it's highly reusable. It accepts all of its dependencies via props. I like this clear separation of responsibilities. Effectively, it keeps this component dumb. Next, let's add our Delete button to this CourseList component. I'm going to add an extra column to hold that button right here. I'll just put a non-breaking space inside. And then down here we can place our button. I'm going to give it a className equal to btn and btn-outline-danger. These are classes from Bootstrap. I'm also going to set the onClick handler because in this case we need to call deleteCourse and pass it over the relevant course ID. So I'm going to declare an arrow function here and call props.deleteCourse and pass it the course.id. Now let's close the button tag and put the word Delete inside. I'm going to hit Save, and then this will get a little bit easier to read. So notice that I'm using an arrow function here because it allows me to specify the value that I want to pass over to deleteCourse when this particular button is clicked. Now that we've added another PropType to this component, it's also a good idea to come down here and declare that as part of PropTypes. So it's not just courses anymore. We also are passing in deleteCourse, and that is a PropTypes.func, and it is required. That should do the trick. Time to try this out. Let's jump over to the browser. Hey, there's our Delete buttons. And when I click Delete, ah, it goes away. The course disappears. There they go. But please don't delete this course. It might create some kind of a black hole. So great, our deletion is working, and our listeners are listening to the changes on the store. We could also add a toast to display a delete confirmation, but since you've already seen how to add a toast, I'll leave that as an exercise for you. And to prove that the delete is working, we can also jump over to our code, look at db.json, and we should see that there are fewer records here in db.json. And that is the case. We can see that I deleted the first few. So 4 is the first ID now in the database. Great! Our deletion is working, and our listeners are listening to the changes on the store. We could also add a toast to display a delete confirmation, but since you've already seen how to add a toast, I'll leave that as an exercise for you. In our final exercise, let's set breakpoints on each of the steps in the Flux flow so that we can watch our data move through the app.

Summary: Stepping through Flux

To show the entire flow of Flux, let's set breakpoints at each of the steps so that we can watch our data flow through the application. Let's watch the delete flow in action. Right now onClick this gets called, so I'm going to make this two lines so that I can set a debugger inside of it. I had to put this on two lines just so I could add the debugger statement in. But this will work the same way; otherwise, it will just break at this point in the browser. For the next step in the flow, let's open courseActions. We're obviously calling deleteCourse, so let's set a breakpoint right here as well. We expect this to get called second. In here, we're dispatching an action, so that should get handled by our store. Let's jump over to the courseStore. We'd expect that action to get handled right here. So let's add a debugger within this particular case statement. When the store changes, our listener and CoursesPage should be notified. So let's go over to CoursesPage, and we should see that this change listener will get fired, which means that we expect onChange to get called after that deletion occurs. And this is why we see that new list of courses reflected without the one course that we just deleted. Now let's try this out in the browser. Be sure to open up the DevTools so that the breakpoints will apply. I'm going to pull this down just a little. And I will click Delete. We hit our first breakpoint, which is right here before we call props.deleteCourse. Hit F8, and this will move us to our next breakpoint. Now we're over in courseActions.js. We can see that the ID got passed along to deleteCourse, and we're about to dispatch an action. That actionType is DELETE_COURSE, and that ID will be passed along in the payload. I'll hit F8, and next we land over in the courseStore. We're within the case for actionTypes.DELETE_COURSE as expected because, as we can see, the actionType is set to DELETE_COURSE, and the payload contains an ID with the number of 4. We're going to filter out that course and then update our private list of courses on line 36. It will emit the change, which will cause React to re-render. Finally, we land in our onChange handler, and this gets called because we called courseStore.addChangeListener up on line 11. That was our way of saying tell me anytime the courseStore changes. So on line 18, we request that array of courses, and we set it to our local state. So when I hit F8, we see that React re-renders, and that course that we just deleted is no longer visible. We just stepped through the complete unidirectional data flow that's implemented via the Flux pattern. I suggest removing the debuggers from your code now since you don't need them anymore. To wrap up the course, I want to provide you a few ideas for enhancing this app. This way you can put your new React and Flux skills to use to enhance this app on your own.

Final Challenges

Congratulations! If you've made it this far, then you're fully prepared for my final challenges. There are a variety of additional features that you can add to our app to help practice your new-found skills. You can show a 404 page when the URL contains an invalid course slug, display the author's name on the course's page instead of the ID. As a hint, you're going to need to call the author API, which I've also provided in the mock API. Along the way, you can create a separate author store if you prefer. Populate the author drop-down on the Manage Course page with data from the API. Note that it's currently using hard-coded data. As a hint, if you load localhost:3001/authors, you'll see a list of authors. Create a reusable drop-down component, and replace the existing author drop-down on the course management page with this reusable component. You can use the TextInput that we created as an inspiration for this task. Add support for managing author data just like we do course data. Again, the author endpoint is at localhost:3001/authors, and it works the same as the course's endpoint. So it supports creating, updating, and deleting data too. I've ordered these roughly from easiest to most challenging. If you're looking for the next logical step, check out Building Apps with React and Redux. You should find that Redux is easy to learn since it's heavily inspired by Flux. This is a more advanced version of this course that uses Redux instead of Flux, and it also explores more advanced concepts such as immutable data structures, reducers, handling loading and error states, building a custom development environment, the Redux DevTools, automated testing, and much more. All right, that's a wrap. I hope that you find working in React as fun as I do. Thanks 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	★★★★★ (1374)
My rating	★★★★★

Duration	5h 11m
Updated	19 Jun 2019

Share course