Building a Full Stack App with React and Express

by Daniel Stern

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi everyone. My name is Daniel Stern, and welcome to Building a Full Stack Application with React and Express. I'm one of the world's leading web development experts and publisher of countdown JavaScript libraries, tutorials, and courses like this one. Did you know that a full stack application, like the one we're making in this course, has everything you need to create a functional ecommerce site, complete with customized user interface and secure formulas and storage? In this course, we'll be building a full stack application from the ground up, including a back end server, a front end web app, and a secure database. Some of the major topics, we'll be covering are making fast and maintainable React components, managing a complex and expanding state with Redux, creating a persistent data storage environment with Express and MongoDB, and using Redux Saga and forms to save crucial user data. By the end of this course, you'll be able to build many kinds of sophisticated websites from scratch or integrate into any business team currently using these technologies. Before beginning this course, you should be familiar with JavaScript, HTML, and React. From here, you should feel comfortable diving into advanced full stack topics with courses on advanced web security, React components and their lifecycles, and creating dynamic servers with Express. I hope you'll join me on this exciting journey

to learn full stack web development with the Building a Full Stack Application with React and Express course at Pluralsight.

Structure of Full Stack Applications

Course Roadmap

Hello, I'm Daniel Stern, also known as the Code Whisperer, and welcome to Building a Full Stack Application with React and Express. We've got a lot to cover, so let's get started with this module, which discusses the structure of full stack applications. Of course, whenever you leave for a trip, it's good to have some idea of where you're going. So let's have a look at the roadmap for this course. First, we're going to understand what full stack applications are, what they do, and how they help us solve our business needs. We'll learn about the differences of the two components of a full stack application, the front and back end, and see what unique tasks they each solve. Once we understand the nature of a full stack application, we'll actually build one, starting with a front end that we're going to build out of React and Redux. Then, we'll build a back end with Express and MongoDB and connect the front end and the back end together. Finally, to complete our real-world scenario, we'll deploy the application to the net. In the next clip, we'll look at a scenario for building a full stack application.

Building Full Stack Applications: A Scenario

So if you are like me, the first question that you asked about full stack applications is how can they help me accomplish my business goals or the business goals of my company? How can this structure of application translate your hard work into an improved bottom line? This is a developer named, well, Dev. Dev's boss, Mr. C. Eeyo, asks Dev, the computer programmer, to help him solve a business problem. It seems the company has a great new product, and it will be sold on a brand new website. Dev, says Mr. Eeyo, can you build this website? The website needs to have a simple, clear interface that makes it fun and easy for your customers to buy your product. But the website also needs to validate credit cards and store confidential user information. Scenarios like these are fairly common in the business world, but why do businesses need full stack applications to solve these problems? Firstly, in the modern age, users expect a very fast,

good experience. This means that if you focus exclusively on a back end or a data component while completely disregarding the front end, users will simply not want to use your site and will go to the website of a competitor that has a pleasing website to use. Additionally, users have become very used to the data that they put on websites or apps still being there. I like to call this data persistence. Not only are users quite unhappy if their content, let's see, like an article that they wrote or a picture they took disappears. But there's lots of information that you're interested in storing too, like data about the user. That needs to persist so that you can use it later. Additionally, unless you've offloaded all of the payment processing to some third party, processing the payments of your users and managing their data is critical to actually generating the revenue that your company needs. In the next clip, we'll do a deep dive into what full stack applications are and how they help us solve these problems.

What Are Full Stack Applications?

What are full stack applications? A full stack application can be defined as follows: an application that can be viewed in a web browser with an accompanying means of persisting data and working with private information that exists on a server. Let's look deeper at these two components of a full stack application, the front end and the back end. The part of your application that your customer sees and interacts with is called the front end. You might also hear it referred to as the client. When you break it down, almost all websites come down to a bunch of forms and buttons. Users generally use forms to input user- created content or, let's say, their shipping address for you to ship them your product. Some websites may use a lot of style to perhaps mask the fact that the website is just a form. But when you get down to it, almost all websites that are business- oriented take the form of a call position of pages, buttons, and forms. The front end is the part of your application that's concerned with things like user experience, polish, speed, design. Though the benefits may seem intangible, we're all aware that a website that is clean and polished and has a consistent good style creates confidence in whatever organization has that website, thus improving your chances a customer may choose to part with their hard-earned money to obtain your product or service. Front end applications also have a unique quality called reactivity, or you might also have heard it as in the context of a reactive application or even reactive CSS. The idea is that this website will look different if you access it on your computer or your phone. This is a very common feature with websites nowadays. But it's important to note that with a few exceptions, this is a concern of the front end. The back end server isn't too concerned what device exactly the end user is using. In addition, front ends are very consistently made up of the languages JavaScript, HTML, and CSS. That's because web

browsers typically only universally understand these three languages. It's not like building a Windows application where you have a choice of quite a few different languages to use. Basically, if you want a website to work, it has to be written in JavaScript. So you're probably thinking the client sounds pretty great. Why do we need a back end at all? What are the limitations to the front end, the browser? So the client has problems persisting data. In other words, your web browser is not very good and making sure that the data you put into it lasts forever. There are tricks you can use like local storage to make it seem like your data is being saved somewhere. But at any time, the user could, for example, clear their cache, and the data would be gone. Additionally, as we'll see, it's not possible to hide things from a determined user once you send them to their web browser. That means that any passwords or secret formulas that are sent in a JavaScript bundle to an end user can be found with relative ease by a determined and skill attacker. Finally, you don't have any control over the hardware that your user is actually going to use to access your website. Now let's say your website is some kind of mortgage calculator, for example, and it uses a lot of math to figure out what the rate of payment might be for a mortgage. This math might be easy for a computer to do, but so hard for a phone to do that it just crashes when it tries to use it. Thus, back ends can provide additional processing power to help our customers and the users of our websites do what they're trying to do. In the next clip, we'll talk about the back end.

Understanding the Server

In the famous movie, The Wizard of Oz, the protagonists are treated to an amazing show of light and sound and magic that seem to come from this incredibly powerful being called The Wizard. However, when they pull a curtain back, they can see that the so- called Wizard is a very ordinary individual doing very rudimentary tasks and that there's nothing exciting or magical about it. The Wizard is like the back end of our application, quietly pulling the strings and making sure everything goes along smoothly. But the end user pays no attention to the man behind the curtain or the back end. So what is the back end, which we can also call the server? So the server persists data by storing it in what we call databases. Data that's in a database can be considered safe, especially if the data is backed up. In addition, the server can be used to conceal secret information. Perhaps you have a secret code that you use with one of your vendors to confirm that it's you making the request. Who knows the trouble that could happen if this code got into the wrong hands? But as we'll see, we can design our applications in such a way that all of the sensitive information is in the back end, not accessible to the end user. Lastly, as I mentioned, web browsers are quite restricted in the communication they can do with other websites. If you're on

website A, and it makes a request to website B, there's many features that exist by default to prevent this action due to the security concerns that automatically arise. Thus, for various kinds of coordinating with third parties, the back end is usually the place to store this code. So on the flipside, if a server can persist data and do calculations and talk to third parties, why do we even need a client? So an application without a client or a front end is essentially a Bash script or a terminal if you do that kind of low-level programming. If you don't have technical knowledge, they're basically impossible to use. And if your product can only be used by people with technical knowledge, then that's a huge base of customers, which you are potentially ignoring. However, web browsers are designed to have images and animations and styles. All these can be used not just to create a fast and fluid checkout experience, but a positive impression of your organization. So as you can see, even though its use is simple, we simply couldn't get along without a front end. Finally, what makes up a back end? I know, I know is everything's made up of other parts, and this has two parts. But a back end has two distinctive parts to it. A back end first contains a database, and a database is a place for data to go. When databases are working correctly, they're extremely boring and unexciting. Basically, a good database does not hit you with any surprises. It just works in a really predictable fashion. However, a back end also has a server. The server, unlike a database, can actually contain an execute computer code. So you can put, for example, secret business logic on the server. This logic might make a calculation and then store the result of that calculation in a database. When we build the application, we will have a distinct server and database component. In the next clip, we'll review what comprises a full stack application.

Composition of Full Stack Applications

Let's review the composition of a full stack application. On the one hand, you have the front end or the client. The client is everything that the user sees or interacts with. They are responsible for having a smooth experience, and you build it with HTML, CSS, JavaScript, and frameworks, such as React. The back end, on the other hand, contains all the features that are basically invisible to the end user. They don't even know they're there. The back end is not responsible for fancy animations. Instead, it's responsible for high reliability, data persistence, fast responses, and good uptime, which is the percentage of your time that your website actually works. Unlike a front end, you can build a back end with many different languages, including PHP, C#, Java, etc. In this course, we'll be using JavaScript to write the back end of our application. Is there a good reason why we want to use JavaScript as well for the back end as we did for the front end? In the next clip, we'll discuss this very important question.

JavaScript and Full Stack Applications

So while the front end of an application is almost always written or at least boils down to JavaScript, the back end can be written in various different languages. My preference, and the language I've chosen to make this course in, is also JavaScript. What are the advantages of JavaScript versus a different language for your back end? So when you have JavaScript on both the front end and the back end, you can hire developers more flexibly. If a developer says that they know JavaScript, then you can say well, you can come in as a front end developer, but if we have some extra back end work to do, you can do it as well. In addition, you can actually share code between the front and the back end. For example, if you have a useful function that perhaps translates one part of your business logic to another, you don't need to have and maintain two versions of this function for your front end and your back end. The man who's tried to do this in an application saying we had a PHP back end can attest to how difficult and frustrating and time-consuming this can be. Finally, if our website is written with, say, React, and React is written in JavaScript, then if our server is also written in JavaScript, it should in theory and can work with React. This allows for some very cool things, primarily what gives our websites the ability to prerender pages. It's also easier for servers to assist with calculations since the data format, let's say JSON, will be consistent between both the front and back end. Ah, but I wouldn't be being fair to you if I omitted the limitations of using a JavaScript back end. It turns out there are some problems you should be aware of. First of all, JavaScript is kind of slow. You might say its processing is sluggish, and its math capabilities are greatly limited, virtually non-existent unfortunately. That means that if your application really, really does a lot of hard math, you might find that it actually goes thousands of times slower in JavaScript than in a language that's a bit more designed for that kind of stuff, like C#. In addition, languages all have their own unique ecosystems of libraries. Some languages have a larger selection of certain kinds of libraries than another. A good example is how many of the most popular data science libraries are all written in Python, which makes sense because Python is pretty cool and it has some beefy math capabilities. Additionally, Java has a lot of very interesting libraries for working with databases, such as Elasticsearch and so forth. Finally, compared to a language like PHP, it can be a little more difficult to deploy a JavaScript application. The reason for this is largely that PHP- based applications have existed for so long. Many internet service hosts, like the one that might host your website, support PHP by default. You can just throw a .php file in, and it will work. Not so for Node applications. Typically, you need a more sophisticated tool, like Heroku, to actually get your JavaScript application on the web. However, we have a module where we'll be doing that with our application. So by the end of this course, you'll be fully aware of how that is done.

Security Considerations

You don't have to look very hard to find another story in the newspaper about a company that's gotten in big trouble because of a data breach or a privacy violation or the security of their website failing in such a way that betrays the end user. The first and primary concern that you must be aware of is that certain code should not be visible to the end user. For example, you might have a line of code like this somewhere in your application. In fact, most applications need to have secrets like this for them to identify themselves to your third party. If the end user could see this, they could identify themselves as you to your vendors. Additionally, you may have secret formulas, like the one here indicated, that maybe cost you a lot of time and money to develop. Maybe if your competition got their hands on this formula, it would represent a major advantage. Thus, must of what we do security-wise is designed to keep data like this out of the hands of the end user. Most won't even know it's there. But if any of the users happened to wish to compromise your website, they could use these tools to do it. I'd just like to share a disclaimer here that I'm not a security expert. We'll be building a full stack application, and there will be some token security measures in. but please remember that I'm a full stack expert, not a security expert. Before building security procedures to protect your actual real data, please consult the relevant courses that we have here. For example, Pluralsight's resident author, Troy Hunt, is one of the most respected authorities on security in the world. So please take my advice when it comes to making websites that look great and work fast and do cool stuff. But when it comes to making websites that are secure and impregnable to attackers, please take the advice of an expert in that field. You can view some of Troy Hunt's courses at the link below.

Recap: Client / Server Workflow

We now understand that full stack applications comprise both a client and the server. But how do the client and the server work together? Here's an example of a typical client- server workflow. It starts with your end user loading up your website. They need a desired piece of data. The user clicks the button in their client that asked for the data, and an HTTP request is sent from their browser to your server. And I use the words their and your specifically because being their browser, they have access to any secret data to send inside. They really do have control. But it's your server. They don't have control over the server. The server authenticates the request through an opaque process, in other words a process does not know exactly. One way or another, the server confirms that whoever's asking for the data ought to be able to get it, and the server makes a request to the database. The database then responds by sending the server whatever data it asked for. This whole process is secure. So without the necessary credentials, you can't get

data, data that doesn't belong to you. All the while, this HTTP request has still been going on. Once the server has the data, it sends the response to that HTTP request usually containing a payload of the desired data. Then on the client, frameworks, like React, take this data model and render it into components. Components look like pages and forms to the end user, and generally speaking, their result is that they're happy because the data that they wanted is now being shown to them on the screen. So how can we use JavaScript to combine these disparate client and server elements? As we've learned, almost all front end applications are written with JavaScript. So if we're using JavaScript, we've got the front end basically covered. It's possible to write the back end in a different language as PHP. But in the option of this author, it's not optimal. Modern tools, such as Node, allow the front end and the back end to be written in the same language. And unlike other applications, a Node-based full stack application can be approached either as two applications or, more interestingly, as a single application. In the next clip, we'll take a look at the finished app.

A Look at the Demo Application

We'll now have a look at the final application. We'll be building this application over the course of the next few modules. First, we'll have a preview of the front end. For the front end, React and Redux are used to display components. The front end itself consist mostly of forms that reflect a user's unique data, and it uses routing to determine what components to display. Let's have a look by opening it up in my browser. So here I am. I'm accessing this website to localhost 8080. I'll be providing instructions on how you can run the application on your own computer during the next chapter. The application starts with a login page, which is a form consisting of a few inputs and a button. Pressing the Login button validates the information with the server. And when it's validated, it loads this page up, which shows all the users' activities, which need to be done. Clicking on any of these to do items takes us to a separate view, which is called the task detail. We can edit the task and even add comments to it. The changes have persisted. There's not a whole lot going on on the front end. As I said several times, it really is just a bunch of forms that let the end user do what they need to do. Now how about our application's back end? Well for starters, it's going to be difficult to look at or see the back end since, unlike the front end, it doesn't use components to display what it is or what it's doing. Our back end consists of a MongoDB component, which stores data persistently in a database. In our production scenario, Express is there to serve a static HTML page, which then runs our client application. Finally, we're using a REST API to communicate with the back end. As I mentioned before, back ends often take the form of Bash scripts or terminals. In the case of our application's back end, to make it

run, we have to go into the terminal, like I have open, and I'm going to run the command npm start, which, in our application, gets it started. As you can see, upon running start, a whole lot of code just appears on-screen. The application is opening here by default. But if we close and go back to our server, we can see it's basically doing a lot of stuff on the technical level. It's compiling our JavaScript and so forth. If we go back to our front end and, let's say, log in, and let's update the title of this article, you can see that the server has updated or rather the server has responded to this. Let's see that in real time. I'll open up this and start doing changes. Here the server, it's basically doing a console log whenever this particular route, or the route that edits the title of a task, is invoked. In addition, we have a database element. I'm going to open up a tool here. This tool is called Robo 3T, which I use just to view what's going on on my local database. I have a database here called organizer. And you can see using Robo 3T, this lets us actually see what's going on with the data of our application. Our application stores data in four buckets, one for comments, one for groups, one for tasks, and one for users. If we use Robo 3T to look inside our tasks bucket, we can see we get a database entry that represents each one of our tasks. So this is how the data is represented on the database, the database being the part of our application where data can persist for a very long time, even once the application is closed. And that's the application that we'll be building. In the next clip, we'll discuss what's coming up in the future modules.

Coming up Next

What's coming up in the next module? Well if my guess is correct, by now you're probably really excited to get in there and actually build a full stack application. That's what we're going to do starting with the next module where we build the front end of the application. It's going to be built with Webpack and Babel to start, and that allows us to write our application in ES6 and JSX. Those are both JavaScript derivatives that essentially become JavaScript once they're processed by Babel. But they give us some additional syntax, which makes our code clean. We're going to use Redux to create a store and update and manage the local state. This lets us organize the data on the user's client, though we will still not be able to persist it. We're going to build a number of React components that, when assembled, will make up the website that you just saw. And finally, we'll use Bootstrap to add some style since presentation is very important. Stay tuned for the next module, Creating a View Layer with React and Redux.

Configuring the Development Environment with Webpack and Babel

View Layer Goals, Limitations and Roadmap

Hello and welcome back. In this module entitled Creating a View Layer with React and Redux, we'll be creating a front end for our application, something the user can interact with. So in this module, we'll be building only a front end. We'll be leaving the back end component completely unbuilt for now. So what are the limitations and goals in this module? So we want to create a front end that's fast, that is something the user can use, and something that we can maintain without too much effort. On the other hand, without a back end, we won't be able to persist data. So for now, any changes that the user makes will disappear when they refresh the browser. Here's the roadmap of the path ahead. We'll start by setting up Webpack. Webpack will allow us to compile ES6 and JSX, and we'll turn that into JavaScript, which can be used in any browser. We'll create the application state and use Redux to manage it. It won't yet reside on a database though. We'll create a dashboard, which is kind of the main component of the application. And we'll also create a task detail display, which is a form that a user can use to update a task. Before we start, here are the prerequisites you'll have to have taken care of. You'll need to have a text editor. The two I recommend is either WebStorm or Atom. I use WebStorm, though it does cost a bit of money, and that will be the application I'm using now. So if paying $100 or $200 for a text editor isn't a big deal for you, I recommend WebStorm. Or if it is, then just grab Atom for free. You're going to have to have Node.js installed on your computer, and you will need to have admin privileges for this to work. If you don't yet have Node.js, you can go to the URL visible on the screen now and follow the instructions to download it for whatever system you're using whether it's Mac, PC, or even Linux. Finally, you have a web browser, Chrome or Firefox specifically since we'll be using their developer tools to help us develop. So if you have all of this taken care of, you should be good to go as far as developing the application goes. What about troubleshooting? So the finished application code is available at the URL visible on-screen. So all the finished application files are already available on GitHub. So if I'm going a little fast and you want to take some time to look at a file I've made, you can go here. Or even more importantly, if you run into some error you can't fix, you can just copy and paste the final code that's found at this Git repository. Also, don't forget to give it a star if it's something that you want to be updated on.

Additionally, probably the most common form of error that comes up is version-number related. So make sure that you type the full version numbers when you install things, just like I do. And to emphasize the point I made earlier, if you're at all confused, just look at the finished application file. In the next clip, we'll be jumping into our text editor and setting up Webpack.

Setting up Webpack to Compile Our Application

We will start by setting Webpack up to compile our application. This is a key step since the language we'll be writing our application in can't be used by a web browser just yet. So Webpack uses another library called Babel, which we'll also be installing. It uses this library to convert JSX and ES6 into JS files. One thing Webpack does which Babel can't do on its own though is it bundles sets of files that are connected to one another by using import statements. We'll be using imports throughout our application instead of, for example, require, which you might see in other Node applications. And Webpack allows all these disparate files to be connected into one JavaScript file. Finally, Webpack has another tool, which we'll be using, called webpack-dev-server, which will allow us to develop our application in a fast, convenient way. So without further ado, let's jump into the demo. Step 1 will be installing Webpack, Babel, and a few other libraries we'll need to bundle and transpile the code. So here I am in my text editor. Isn't this exciting? Like many Node projects, it will begin by a few commands into our command line. So I'll open up my terminal, and with WebStorm, the terminal is available right inside the program. However, if you're using something else, you might have to open one in a folder. But in any case, I've created an empty project here, just given it a name, and now I have a terminal open inside. So the very first thing we'll need to do is create a package.json file. This tells various tools, such as NPM, what our package and our application is really about. This is simple, so we'll just say npm init --yes. This process creates a file called package.json. Right now it's pretty empty. It just has an automatic name for our application. However, when we install dependencies, this will start filling up and it kind of is a record of the dependencies we have. First, let's install Webpack. So we'll say npm install --save-dev webpack@ 4.17 .2. So first of all, the save-dev part indicates that we want this to be added to the dev dependencies part of our application. Most importantly, this allows us to use these packages when we use the command line, like we'll be doing in a few moments. So we'll install Webpack like so. And, of course, the version number is important since different versions may work similarly, but lead to subtle and difficult-to-fix bugs. The best way to avoid these bugs is just to use the very same version as was installed. So once we've installed that, you can see that the package.json file has been updated with devDependencies, so our Webpack is there. And even if we opened our node_modules folder, we can see that a whole bunch of packages have

been installed. These are all necessary to get Webpack working. That's looking good. Now let's add a gitignore file so it stops indexing our node_modules and say New, File. I'll call it .gitignore, and I'll just say node_modules. And you can see my node_modules has been grayed out, and it's no longer going to try to index that whole directory. So back in the world of dependencies, we're going to need to install a couple more dependencies. First we'll install those related to Webpack. We'll say npm install --save-dev webpack- cli@ 3.1 .2, and this allows us to run Webpack from the command line as will be a preferred way of doing it, and we'll also install a Webpack DevServer. So we'll say webpack-dev-server@ 3.1 .7. You can install as many packages as you want this way. Just separate each one with a space. And that should do it for the Webpack family, so let's press Enter and finish those installations. That's looking good, and we'll be making a webpack.config file shortly. Now the other installs that we're going to need is our Babel installs. So these libraries actually will compile our ES6 into JavaScript. So we'll say npm install --save-dev, and we'll say @babel/core@ 7.0 .0. You'll notice that babel/core begins with an @. This is just a unique convention where some npm packages can begin with @. And that indicates that the first half of Babel is the publisher. So we'll install that, and that looks good. It's been added there. I'm going to press up in my terminal to bring me the same code I just entered, and let's install the other Babel dependencies we'll need. We'll need @babel/node@ 7.0 .0, and this will allow us to run, to sort of compile Babel within the command line. We'll go with @babel/preset-env also @ 7.0 .0. Present-env allows us to compile ES6. We'll install babel/preset-react@ 7.0 .0. This allows us to compile React. And I'll install @babel/register also @ 7.0 .0. And babel/register, it's a weird one that just sort of needs to be in your package for many Babel features to work. It's sort of like it checks if it's there. Now let's install those. So those are our dependencies. Now if you got a little bit lost, don't worry because, like I said, you can just check out the GitHub, and here is the express-react-fullstack package.json is all the final versions of the libraries. So if you want, you can really just copy this whole dev dependencies block, paste it in, and then type npm install. If you just type npm install, it'll read your package.json and install all the libraries that are needed to fill in what's required. That should work for now. Next, let's create a babelrc file and define how we should handle our JSX and our ES6. So we'll make a new file, and we'll call it .babelrc. This is a JSON file that Babel automatically checks for to determine how it should run. So we'll start with some curly brackets, and we'll have a presets property with an array. And the first element of the array will be another array with the first element being @babel/preset-env, and, after that, we'll put an object. This object represents our configuration for preset-env, and we have to give it a particular option called target, and that's another object, and that object has a property called node. This simply affects what kinds of ES6 is transpiled. If a target is an old target, then it may not automatically transpile a newer feature. We want newer features, so we'll say targets node, and we'll make that

equal to current. So that'll be for our ES6 compilation. And now we'll set up our React compilation by adding a string after that array and say @babel/, and we'll say preset-react. So that's all there is to it for Babel. This file's telling Babel that whenever it's called upon, and Webpack will be calling it, to use these libraries to transform our files. And these libraries know what to do specifically to turn, say, a JSX file into a JavaScript file in the case of babel/preset-react. Alright, next we need to create a webpack.config .js file, and this will indicate how our application should be bundled. So we'll create a new file, and we'll call it webpack.config .js. We'll begin by requiring a dependency path. We'll say const path = require path. Now you may notice that I said we wouldn't be using require statements, but rather import statements. This file is the one file that we're actually going to use require. Since Webpack itself is going to be compiling ES6, it gets tricky when you also want to write your Webpack configuration in ES6. So for the case of stability, we're just going to use statements like require just for this one file. So what we'll do for a webpack.config is we'll define an object for module.exports. And this object will essentially be our configuration. It's what Webpack is going to check when we run it. So we'll start with mode, and mode can either be development or production. And while we're developing, it will be set to development. We'll define entry, and entry is the file that we will start by using. You know what, let's go ahead and make that file while we have the chance. So we'll make a new file, and I'll call it src/app/ index.js. While we have it open, let's put our Hello World. I'll say console.log Hello world. That's all for now. Back to webpack.config. So this is the going to be the first file, kind of the root file of our application. So we'll say path.resolve, and we'll start with __dirname, which is a special property, which is always the directory the file has been run from, then the src folder, then the app folder. And files called index.js have a special designation where we'll automatically look for that file in the path. In other words, we don't have to say index.js at the end of this entry. It's smart enough to figure out that's the file we want. Next, we'll set up our output. So this is the definition of the one file that our application's going to become. We'll define a path, and we'll want it to go to a different path. We'll make a directory called dist, which is for our built finished files. We'll call the file bundle.js with the filename property, and we'll define a publicPath, which is where our code expects to find this file, and we'll just give it a forward slash, indicating the root. Let's test what we have so far. We'll go into package.json and add something to the scripts. Scripts are something that we can trigger through the command line. I'll make one called start, and all I'll say for now is webpack. So if we open our terminal and type npm start, it will run Webpack. We might get some errors, but let's see what happens, npm start. So we didn't get any errors, I guess because our file is just so simple, there wasn't really anything to trip up on. And you'll notice now a dist folder has been created. Inside the dist folder is a file called bundle.js, which just has a whole lot of code that Webpack adds to it to make it work in all browsers. But if

we search for a chunk of our original code, I'll say world, you can see deep down buried in there is our application code. This file gets a little bit big, so let's go to gitignore now, and we'll add dist to our ignored file. So that worked. There's a bit more to be done. Let's add resolve to webpack.config. And it has a property called extensions, and this is just an array of the extensions that we want Webpack to process. In this case, we'll say .js for JavaScript and .jsx, which is the kind of file React uses. Next, we'll configure our DevServer. So we'll add a property called devServer, and we only need to set one setting called historyApiFallback. This is setting we have to enable if we want to use React Router in a later module, like is our intent. Finally, we have the module prop. And so the module is really where we describe how we want our app to be compiled. We'll add a rules property to this object, and that will be an array, and the array will contain one object. We'll add a test property, which is the regex that application will use to determine if a file should or should not be compiled. So we'll say /\ .jsx? /. So that means any file with .js or maybe x at the end, which means js or jsx, matches the test. And we'll define a loader that we want to use for that file as babel-loader. And that's our webpack.config file. So we're almost done with the lengthy setup process. As I mentioned, full stack applications with Node involves a bit more setup, and they're also a bit more challenging to deploy. By the end of the day, you'll find that they're also quite powerful. So we have to create some steps. We've already made our index.js, and it just says Hello world. Now let's make our index.html file in which we'll have to follow a few simple conventions. So I'll make a new file, and I'll put it right at the root and call it index.html. I'll add a head tag and a title tag, and I'll give my application a title. I'll say My Application. You can name your application whatever you'd like. Next, we'll give the body tag. And inside that body, we'll have a div with the ID of app. We'll use that later. That's where our React application is going to go. Finally, we'll have a script tag, and the source of that script will be / bundle.js, which, as you'll note, corresponds to the name of our script and our public path in our webpack.config. That's all we'll need to do for now. So we have our index.html and index.js set up. Last but not least, our application should now say Hello world in JS and HTML. Does it? So we're going to define another script that lets us launch our application. We'll call that script dev, and all we'll say is webpack-dev-server, which will read our webpack.config to figure out exactly what to do. And we'll also say --open, which, if we're lucky, will pop open a browser for us when we run this. Now let's give this a shot and see what happens. We'll say npm run dev. If the script isn't start or test, we have to have the word run before. You can just say npm dev. And we have an error, but that's okay because we can take this opportunity to learn how to work through errors. It says ERROR module can't be found babel-loader. Well, it looks like in our case we forgot to install babel-loader among the many packages we need, and we can fix that now. So we'll press Ctrl+C to cancel this process and install babel-loader. Babel-loader puzzlingly does not need an @

like the other Babels. We'll just say npm install --save dev babel-loader@ 8.0 .2, and there is it in our package.json, which is looking up to code. Let's try saying npm run dev again. No errors this time. So here it's open to localhost 8080. I can see the title from my file, which is a good sign. I can open the Dev tools, in my case, by pressing F12. And you can see, here's our Hello world line as promised. If you're running Mac, it may be a different command, for example command option i may help you get your terminal open. And I do believe I promised that the HTML would also say Hello world, so let's go back into our script. Into index.html, into the body, and let's just add h1 to say Hello world Save that and refresh the page, and there we go. So that's a lot of work for Hello world, but we've set up a really stable scaffolding that's going to come in handy again and again for our application. In the next clip, we'll be setting Redux up, which is very cool.

Implementing React Components and Redux State

Managing Application State with Redux

In this clip, we'll be adding Redux to our application. So what is Redux? So Redux manages the data that underlies your application. When you're using Redux, you can always easily get the state of the application or the data. However, it's hard to change the data. You have to use something called actions. At the end of the day, we'll use another library called React-Redux to connect Redux to React, which kind of makes our application automatically mirror its state. Now we don't have time to talk all about Redux, so for a full talk about Redux, please consider the following course. It's called Flux Redux Mastering. It's available on Pluralsight, and it's by me. So here's the demo. First, we're going to create a default application state as a JSON file. So here we are in WebStorm. I'll make a new file inside the src, and I'll call it server/ defaultState.js. Right now, there is no server, but this is where the file's eventually going to live, so we'll just put it here. Also note how we'll be requiring the server file in our front end files, an excellent example of what can happen when both are written in JavaScript. So I'll say export const defaultState, and that will be an object. So this state kind of going to define everything that our application needs to be unique from one user to another. I'll start by typing in one of each of the various categories of data. First, let's have a users collection, and the users will have an id identifying themselves. This one will

have U1. They'll have a name. We'll call them Dev. And we'll add a password later. We also have a groups group, which is something that the tasks that are to be completed belong in. So we'll have a group called To Do. It will have its own id, which is just a unique string, and an owner, which corresponds to a user. So I'll just it an owner U1. That's what our groups look like. Next, let's add a tasks collection. So a task would have a name, call it Do tests, an id, and you'll notice that everything has its own unique ID called T1, the group that it belongs to, which has to correspond to a group we have, we'll say G1, a user or let's say an owner, which has to correspond to a user, let's say U1, and a Boolean, isComplete. We'll say false. Finally, we'll have a comments collection, which are little bits of interaction that users can add to tasks. So each comment will have an owner, an ID, we'll say C1, and the owner is U1. And we'll correspond to a task, and let's have that correspond to our existing T1 task, and content, which is what the task is. So that's what our state looks like. Now I'm just going to go ahead and fill this in with a bunch of additional groups and tasks of the same structure. In interest of time, I won't show myself typing that all in and bam. So I've added an additional user, a couple of extra groups, a couple of extra tasks. And you can name your tasks and groups whatever you want, but you just have to remember that the individual IDs match up to something that exists. If you'd like, you can copy these items from the finished code here, but watch out for the passwordHash since we haven't brought that in. Just delete that line if you're going to copy and paste this whole file and this top line here since that won't quite work just yet. Next, we need to add a basic Redux store. This will be the thing that provides the state to the application. So I'll open up that big old terminal, and in my one terminal window, I'm running the DevServer. So I'll open up a second terminal window to do some more installs. So we'll say npm install --save. Now unlike our Webpack, these Redux libraries are actually part of the application that's going to be sent to the end user. Hence, we don't save dev, we save, indicating the difference I just mentioned. So we'll install redux@ 4.0 .0, and that should be quite sufficient for now. So we'll create a new file to hold our Redux store. Inside src/app, I'll create a new folder and file. I'll call it store/ index.js. So we're going to make a very simple store. First we'll start by saying import from redux, and we're just going to import the function createStore. We're going to be building on this in the future, but this is all we need for now. So we'll export const and call it the store, and that will be equal to calling createStore with a couple of arguments. The first argument is the reducer, and it's a special function that takes an argument called state and an action and always returns a new state. We're going to put a switch statement here, but for now, we'll just say whatever happens in this reducer, just return the state. Keep returning that same old object. Now let's take this opportunity to grab our default state from our server directory. So we'll import from, we'll say,.. as in up one directory,.. as in up one more, server/defaultState import defaultState, and we'll say that the state in the reducer is equal, by default, to this defaultState.

And so we've created a very basic Redux store that only works with this one state. Is that enough? Let's find out. Let's go to index.js, and we'll now import our store. We'll say import from store, and we'll just import the object called store, and we'll just use the log to see, we'll console.log store.getState, which should give us the state of our application. Let's save this. Now I still have DevServer running, so this has conveniently updated my application without me needing to restart. If I open my terminal up, you can see it's now logging the object. If we have a look at it, here it is, our application's full state. So we now have Redux, and it's all set up to pass the state object to whatever we need. And, in this case, it will be React components, and that's coming right up. Finally, I'd like to note that we're eventually going to have to change the state to make it a truly interactive application, but we'll add those changes after we've created a form since they may not make much sense without the relevant user inputs to give them context. In the next clip, we'll be adding our first React component, a dashboard component.

Adding a Dashboard Component

In this clip, we'll be creating a dashboard component, which will act somewhat like a home page for the user. So we'll add a React component that will call the dashboard, and this will act as a sort of home page. The dashboard will basically take the application state from our Redux or, in the future, in the database and translate this data into components that are user friendly. Now as you may have expected, first we need to install some dependencies. I'll say npm install --save, and, of course, we'll need React. So we'll say react@ 16.4 .2. And we'll also need react-dom, which is what allows us to turn our JSX into actual HTML, and we'll install that @ 16.5 .0. And finally, we'll want React Redux. We'll say react-redux@ 5.0 .7 and install all that. So let's create a new folder and file. inside app, we'll make a file called components/ Dashboard.jsx, indicating that it's a React file. So let's make this component. At the top of all React components, we want to import React. Say import React from react. And we'll export a const Dashboard, which will look like this. It's going to take the form of a function, which returns a JSX tag. You'll notice that here I've used round brackets instead of curly brackets, which indicates that this is not a function, but an object to be returned. The arguments passed to the dashboard will represent the parts of the state that the application is interested in. So we'll just go ahead and just pass the groups. Just for clarity, let's also give this a title here and just say, I don't know, h2 Dashboard. So we actually need to tell the application somewhere to render this dashboard. So we'll go to index.js. Let's get rid of these two console.logs. We don't quite need them anymore. We will need to require react and react-dom though in this file, as well as the dashboard we just made. So we'll say import_React from react, import ReactDOM_from react-dom, and we'll import the dashboard, import from

components/Dashboard, and we'll just import the thing called Dashboard. Now how we actually take our application off is we say ReactDOM.render, and the first argument is the actual component to render, which, for now, will be Dashboard. Oh, and a single error here. I forgot to name this file .jsx, but that's what we'll want. So let's rename our index.js to index.jsx. Gets rid of that error. And the second argument here is where it's being bootstrapped to. You'll remember we created a div with the ID of app in our index, so let's use that, document.getElementById app. So if this all worked to plan, we should see our dashboard appearing in our application. Let's have a look. And we've gotten a weird error, Cannot find module webpack-dev- server/client/ index.js. Ah yes, and there's the problem. It turns out to get this working, we're going to have to go back into our WebStorm and restart. We'll do that now. So here's my--- So I'll press Ctrl+C to end this process, and I'll just press up to get my npm run dev command again and press Enter. And there we go. Our Dashboard component is loaded. So we're now going to use React Redux to connect the dashboard to our Redux store. So we're going to take all this in this index and compose it into a different higher up component. So I'll make a new component and call it Main.jsx. And it's kind of the parent component. It's even be the parent of our dashboard. We'll start with a few import imports like React. We'll also need to import Provider from react-redux. Finally, we'll import the store that we made. So we'll export a Main component. We'll say export const Main, and it will start with a Provider element. So the Provider is an element, which takes a store as a prop, and any connected components inside this provider will have access to this store. So inside the Provider, we'll just put a div. And for now, I'll just put a placeholder. We'll say Dashboard Goes Here. Go to index.jsx and change this so that the Main component is loaded in. Change all that up, and we can see our application is still looking good. It says Dashboard Goes Here. Now let's bring in the actual dashboard. So inside the Dashboard file, it has the ability to be passed these groups. But to know how to get groups, we have to connect it. We do so with a function that will call mapStateToProps. And this is an argument, which takes state, and whatever it returns becomes the props of dashboard. So we'll return a new object, and we're only interested in groups. So we'll say groups is state.groups. And now if we combine this React component with mapStateToProps, we can get a connected component. So we'll export a ConnectedDashboard, and that will be equal to a react-redux component. We'll need to import something at the top here, so let's import curly brackets from react-redux, and we'll import the method called connect. Now we'll call connect, pass it mapStateToProps. And then after that in a second tricky function brackets, we'll pass the actual item, the dashboard. And now we'll go to Main, and we'll import this connected dashboard. And instead of Dashboard Goes Here, we'll say ConnectedDashboard in a tag. Having a peek at the application, it looks good. But let's have it actually display what our groups are. So in React, when we want to work with a repeating element like these groups, we use a map. So

inside a single set of curly brackets, we'll say groups.map. And this will be a function that takes each individual group and returns a JSX object, so we'll call that a div there. And why don't we just start off by saying group.name. That looks pretty cool. Let's see if it works. Nice! So you can see here in our application, we're now showing a list of our three groups. These correspond to the data in the default state, and you could even change this if you wanted to see your app update on the fly. But we get into our first tricky hurdle. We have these groups, and these groups contain tasks. How can we list the tasks inside each group? We're going to need to create another component. So we'll create a new file called TaskList.jsx. We need react and connect, so we'll just copy these imports from the top of Dashboard. So we're going to define our component, and it will have a list of tasks. And for now, we'll just return a div, and then we'll map the tasks to their name. But how do we know what tasks to include? That's where mapStateToProps comes in. So the first argument, as we've seen, is state. But the second argument is actually the props of the component called ownProps. So we'll let groupID be equal to ownProps.id. So we'll pass this element an ID property, and it will figure out what the tags are that it's interested in. So we'll return the state. It'll have a name, which is just going to be equal to its ownProps.name, an id, which is going to be equal to ownProps.id, and we can just shorten that to groupID, and tasks. And for tasks, we'll look at state.tasks and filter them so that only tasks where the group is equal to our groupID. And finally, we'll export a ConnectedTaskList, and we'll call connect, pass it mapStateToProps. And then in another set of round brackets, we'll pass it the TaskList. Now inside the Dashboard, we'll import our new ConnectedTaskList. So we'll say import from TaskList, and we'll import that ConnectedTaskList. Now for each one of these groups, instead of the div, we'll just put directly in a ConnectedTaskList, and we'll say id equals group.id and name is equal to group.name. So that's quite a few steps at once. Let's see how our application looks. Nice! So now our application is showing all the tasks. Let's group the tasks by their group names. So what we'll do here is we're passing name here during mapStateToProps. That means we can access it here in these objects that are passed to the actual React component. So we'll say name, and then we'll just say h3 and give it the name. And in all React components, they all need to be within one greater tag. So let's put all that within a div. Very cool! Doesn't that just looking outstanding? So now the user is able to see their dashboard, which ostensibly contains a list of activities that they're interested in. Before we continue, let's go ahead and save our work. So we're going to use Git to kind of keep track of our work, and we're going to commit every now and then. To get started, we'll git init. It'll create a new Git repository. You'll want to make sure that you've created this file, gitignore, and, it contains node_modules, or else you'll get a real big Git repository. I'll say git status to make sure everything's looking good. It has a short list of untracked files. And I'll say git add -A., which means add everything. These IDF files are simply something that WebStorm

has created. I'll take them out later. And I'll say git commit -m, "Working on the app, " and our work is now saved. In the next clip, we'll be adding routing and navigation. Very cool.

Routing and Navigation

In this clip, we'll be adding routing and navigation. So what is routing and navigation? So routing's a term for websites when how the application looks or acts is affected by what the URL is in the navigation bar on the top of the screen. We're going to use a library called react- router, which determines which component to display based on this URL. By using routing cleverly, you can store a lot of information in the URL. This allows users to share information with each other or with, for example, your technical support staff just by using their URL. So let's jump into this demo. First, we're going to add a Main component, and the Main component will be affected by what the URL is. Let's get right into it. So here we are in Main.jsx. We already have the starter of a Main component, but we need to add some routing capabilities. So I'll open up my terminal, and I'll install react-router-dom, which is a subset of react-router that is used in the browser, and we'll say @ 4.3 .1. Alright. So at the top, we'll import a couple of things from react-router-dom. Oops, and I made a little mistake there. I forgot to say save after installing react-router-dom. This could create problems when you try to upload it to a deployment site. So let's just retype this and say --save. That's better. So we'll import two things. We'll import Router, which is the parent component that all routes have to be inside, and we'll also bring in Route, which is a component that looks different or displays depending on what the URL is. So we'll now wrap all our components inside our Router component. We're going to add a history property, and we'll leave it blank for just a moment, and we'll finish wrapping all of these inside our Router. Now to use history, we're going to want to install an npm library called history. So open the terminal again, and we'll say npm install --save history@ 4.7 .2. Now I'll create a new file in the store called history.js. So we'll import something from history, and we'll import a method called createBrowserHistory. This will create an object that basically React Router can use to determine what it is and what it was in the past. And now we'll just export a constant called history, and it will be createBrowserHistory. So now this file exports a history instance that all our other files can use. Let's go back to Main.jsx, and we'll import this new history file. So we'll say import history from store/history, and we'll pass history to this Router. So let's update it so that ConnectedDashboard only appears when the URL says dashboard. Let's gray this out, and we'll add a Route component. First, we'll add the exact attribute, which says that this route should display only if the path matches exactly. We'll say what we want the path to be, which, in this case, will be /dashboard. And we'll add a render function, which is a function that will determine

what the component is. So we'll be coming back to add some stuff to it later, but let's just add a function, and it will return a React component. We'll just make it our ConnectedDashboard. Change the indentation on that, and we'll save that. Now let's have a look at our application. Cool. So now when we go to our application, you can see our Hello World message is still there, but our dashboard is conspicuously absent. However, if we type /dashboard and make that the URL, here is our dashboard. So it's now linked to the URL. This will become more useful as we have more than one route in the future. That went smoothly. Now let's have a navigation component to be at the top of the application at all times. So in components, I'll make a new file called Navigation, and I'll give it a JSX extension. Let's do a couple of imports. We'll need connect from react-redux, as well as link from react-router-dom. We'll also need react. Cool. So let's define a DOM component, one that's going to be connected to the rest of the application via the connect method. We'll say const Navigation. We'll leave the parameters blank for now, and we'll just return a React component. We'll put it the div. and now inside, we'll put a Link component. So a link is like an a tag, an anchor tag, except instead of an href, it has a to property. And here we'll specify what we want the URL to be. I'll say dashboard. And we can put anything we want inside this link, and whatever is inside the link if it's clicked will go there. So I'll just have a h1, and I'll give my application a name, My Application. We don't need this Hello World in index anymore, so we'll get rid of that. Now let's create a ConnectedNavigation component. I'll just export const ConnectedNavigation, and we'll call connect. For mapStateToProps, we'll just put in a shorthand. We'll just pass it the state directly and tighten that up later and invoke this on Navigation. Now we can go to Main.jsx and import the ConnectedNavigation. Let's add that to the top. Now let's have a look at our application. Nice! So now we have a navigation that's always present. If we go to some URL that doesn't make sense, like this one we haven't made yet, we can click the navigation, and it'll automatically take us to Dashboard where the expected component will render. So far, we only have one route. But in some upcoming clips, we're going to be adding additional routes. Stick with us because it'll be great.

Adding New Tasks - Lecture

In this clip, we'll allow the user to make their first impact on the state of the application by adding a new task. So we're going to update our reducer in our store in such a way that the correct type of action will make it add a new task to its list. However, we immediately run into a problem. Our new task needs to have some kind of random or at least incremental ID. But reducers cannot have any randomness in them. Therefore, we have to use a saga or a thunk. Luckily, since our components are connected, once we solve the problem of updating the state, the way the

components look should change on their own. Now I've chosen to use sagas to solve this problem. Other developers may prefer thunks. But in my opinion, sagas are better for this kind of data transformation. So a saga is something that runs continuously in the background of your application. Sagas listen for action and respond by generating side effects, which is anything that happens outside your application or anything that is not a pure function, like randomness. And sagas are denoted by a function* syntax that we'll be using, which is not found in very many other situations. So all sagas, as we'll see, are generators. Briefly, a generator is a kind of JavaScript function. Standard functions return one value right away. However, generators can return any number of values, 3, 4, 5, 10, any number. And generators can return values later, not just right away. This is, in essence, the difference between a generator function and a basic function. Here's an example of a generator function. The function* tells us it's a generator function. The code inside is normal except for two things. A while true loop, which would normally cause a crash, is acceptable inside a generator function as long as it has the yield keyword. The yield keyword tells it to return a certain value, in this case meaning, and then wait until the generator's invoked again. In our application, Redux Saga will be invoking these generators for us. That's a lot of knowledge to impart in such a short time. I have a full-length course on Redux Saga, which will fill in a lot of the blanks regarding the previous fast explanation.

Adding New Tasks - Demo

Let's jump into the demo. So we're going to create a saga to generate a random task ID, and we'll create a corresponding action for that saga. We'll be creating a mock saga. So it's not actually going to interact with the server yet, but it will create a good basis for us to add that interaction in the near future. So here we are inside the TaskList. We're going to add a button to the bottom of this TaskList that creates a new task. Let's do that presently. We'll say button onClick, and we'll give it some React code here. We'll just call a method, and we'll call the method createNewTask. We want createNewTask to have the ID of the group in question. So we'll pass it id, and then here at the top, we'll add id. You'll notice that id is being passed in mapStateToProps. However, there is a problem, which is that createNewTask is not being passed anywhere. We don't add createNewTask as a property to this mapStateToProps. We actually need a separate function called mapDispatchToProps. And it gets two arguments, dispatch and ownProps. So we'll return an object, and we'll give that object a property called createNewTask, take an id, and for now, I'll just give it a console.log. Console.log Creating new task id. Now as a second argument to connect, we pass mapDispatchToProps, which should provide our component to access to this function. Last, let's add createNewTask here to the destructured arguments of TaskList,

createNewTask. So createNewTask is being created here, passed as an argument to TaskList here, and used here. Let's see if that works. One or two quick fixes. Let's give this button some text. I'll say Add New. And also above this Tasks map, React really likes it if you give everything inside a map a key, which is unique and helps React work performantly. So I'll just say key= and just pass it the task.id. That should do. We'll save that. Now here we are in our application, and we can see here is the Add New button. And while we're at it, let's also go to Dashboard and add a key to this ConnectedTaskList. Say key= group.id. Now if we open up the application and we click Add New, we now see that we have our Creating new task for the appropriate group. Very nice. Now we're going to create a new file called store/mutations, which will be a template for all the changes to our application state we might want to do. We'll start with a CREATE_TASK and REQUEST_TASK_CREATION mutation. So we'll say export const and, in capital letters, REQUEST_TASK_CREATION, and we'll just make that equal to a string of the same name. We'll do the same for CREATE_TASK. Now those are constants. Let's create methods that automatically create objects to do these mutations. We'll start with requestTaskCreation, and that will take a groupID as an argument. And we'll give it a type of REQUEST_TASK_CREATION and the groupID. And we'll also add the CREATE_TASK. So CREATE_TASK will be dispatched by the saga once it's finished creating this object complete with its own random ID. It'll take three arguments, taskID, groupID, and ownerID, and it will simply return a new object with type:CREATE_TASK and those three values as properties, taskID, groupID, ownerID. Now back in the TaskList, we'll import requestTaskCreation. And now here under our console.log, we'll call dispatch and dispatch a requestTaskCreation mutation with the ID that's been provided. Now let's go back to our store. To help figure out what's going on, we want to add some logging to our store's action. So let's open up our terminal, and we'll do a few installs. We'll npm install --save, and we'll grab redux-logger@ 3.0 .6 and redux-saga@ 0.16 .2. So we'll import createLogger from redux-logger. And for createLogger to work, we have to add a second argument to createStore. We also want to get a second import from Redux called applyMiddleware, like that. Now the second argument will invoke applyMiddleware, and the first argument will be createLogger. Now we'll save that. Now whenever we dispatch an action, we'll see it in our terminal. I'll click Add New, and you can see here is the nature of the action. Usually, actions change the state of the application. However, for actions that require any kind of randomness, like this TASK_CREATION, we need some kind of intermediary, in other words a saga. So let's add a saga to deal with this unusual request. So we'll make a new file in store called sagas.mock .js. All of our real sagas will be communicating with the server. But until we have that, we'll use this mocks that just do it on their own. So first we need to import some stuff from redux-saga or redux- saga/effects. We'll import take, put, and select, and we'll see what all of these three do in a minute. We'll also import

all our mutations from our mutation files, put * _as mutation from mutations. We also want a library to generate random strings. So let's open our terminal and install uuid. Again install --save uuid. Very cool. So it will import that. So we'll just say import uuid from uuid. Great. Now let's create a saga to create this task. We'll export function* taskCreationSaga. And we'll have a while true loop. So what we'll do is we'll yield take mutations.REQUEST_TASK_CREATION. That means when it gets to take, it wills top until the specified action is dispatched. We can also get properties from this action, so we'll say const groupID is equal to yield take mutations.REQUEST_TASK_CREATION. And we'll just add a console.log here and say Got group ID, group ID. Now we're going to have to run this saga for it to work. So we'll go back to our store index. We'll import from redux-saga the middleware we need, and that's called createSagaMiddleware, and it should be imported without curly brackets, just createSagaMiddleware like that. Now we'll say const sagaMiddleware equals createSagaMiddleware. And we have to do that at the top because we'll be using it in two places. First, we'll make it as a second argument to this applyMiddleware call. Now we'll just tell it to run all the sagas. We'll import all the sagas, import * as sagas from sagas.mock, and we'll say for let saga in sagas, and we'll just tell the sagaMiddleware to run that saga. Now if we look at our application, we can see that every time we dispatch this Add New event, our saga is triggering. We can see this see this Got group. But we're still not seeing any new tasks. Let's deal with that now. So here in our saga, we have our groupID. We're going to need an ownerID and a taskID. For the ownerID, since we haven't implemented logging on yet, we'll just default that to U1, the user ID of our Dev user. And we need a random task ID, so we can say const taskID equals uuid. Now that we've done the random part, we have to put out a new mutation, the CREATE_TASK mutation, which can be responded to by reducer without the reducer needing to be random in any way. o we'll yield put, and put means whatever action we pass to it, send that action out into the store. Say mutations.createTask, and it has three arguments, the taskID, the groupID, and the ownerID. Now we can finally update our store. So let's go to the store here, and here's our reducer. As you can see, it only has one clause where it returns state. But it can do various things depending on what the action is. Redux provides a utility called combineReducers, which we can use to create a reducer that deals with each collection in our state differently. So we'll import combineReducers. And instead of this reducer here, we'll just cut this away, I'll call combineReducers and pass it an object. Now the name of each property of this object corresponds to a collection. Right now, we're interested in tasks. So task will be a method. The first argument is the current state of task will start out as defaultState.tasks. And the second argument is the action. Now we'll put a switch statement for action.type. We need to check for the type of our new mutation, so let's import * as mutations from mutations. So we'll say case_mutations.CREATE_TASK, and let's just put a log

here. We'll say console.log action. Save that. Looks like we need to fix up one or two things. First, let's add a. / to this, so it refers to the local file mutation. Next, we need to add a default case to each of these tasks. So if the action type isn't as expected, we'll just return task. Now we actually need to add a property for each one of our collections that we're using. So we'll say comment, and we'll just say comments equals defaultState.comment, and we'll return comments. And we'll do the same thing for groups and users. I'll just copy and paste this and change the appropriate words and users. Nice. Now if we have a look at our application and we click Add New, you can see the CREATE_TASK action has been dispatched. here are the properties of the action. Of course, our next state hasn't changed because we haven't yet updated our tasks reducer. Let's do that now. So here in the CREATE_TASK handler, let's get rid of this log, and what we're going to do is return a new array. The array is going to start with everything in tasks. So we'll say ...tasks, so everything in there, put in there. And then we'll add one new task, an object. Its ID will be action.taskID. Its name will be New Task or something similar to that. Its group will be action.groupID. Its owner will be action.ownderID. And we'll give it an isComplete property of false. We'll save that. Alright, here's the big moment. Let's go to our application and see if it works. Alright. So if we click Add New, voila. Our application is updating. A new task is created and added to these different groups, and these changes are confirmed in our logs on the right. We can add as many new ones, and they each have an ID. But of course, if we refresh the page, these changes don't persist since this application does not yet have a persisted component. Now that we have a good understanding of editing the state, in the next clip, we'll be creating a task details page, which will allow us to change very many things about the tasks.

Implementing Task Details Route Part 1 - Displaying Data

The stage is now set for us to add our most complex route, the task details route. So you might have noticed that we were using a mock file there during development. When you use a .mock extension, it indicates that the business logic is not really the logic that will be implemented in the production application. We use this to reduce the complexity of our development process by, say, having a file that can function just fine with the server element not yet created. And mocks are commonly used in certain testing frameworks, such as Jest that you may be familiar with. We're going to add a route, which displays the details of a single task. And the route will have forms and buttons that allow the user to change the data. We're going to use our existing router implementation to determine which tasks should be viewed and edited. Later we'll add mutations that go along with this form. So I'll create a new file called components/TaskDetail. This is going to be a challenging one, but we'll learn a lot about React and Redux while we're at it. So we'll

import, at the top, React from react, and we'll also want connect from react-redux. So we'll define a React component called TaskDetail. For now we'll be interested in TaskDetail being passed an id property, a comments property, the task itself, and the isComplete property. And we'll return a div, and for that, we'll just say Task Detail. We'll create a mapStateToProps function and leave it empty for now, so I'll just say state, state. And then we'll export a ConnectedTaskDetail. So we'll connect mapStateToProps and pass it TaskDetail. Now let's go all the way back to our Main component. Here's where our routes are set up, and we'll need to add a route for our TaskDetail. So we'll import here at the top right under ConnectedNavigation. We'll import ConnectedTaskDetail, and we'll add a route for it. So we'll say Route exact path equals /task/:id, and we'll add the render method. Now the render method gets an argument, and that argument has a property called match. We're going to need that match property as our own props in order to determine which task it is that we're looking at. So it will return a ConnectedTaskDetail where the match property is equal to match. I'll give this some round brackets, right around ConnectedTaskDetail, and close the Route tag. So here inside our TaskList, we'll modify what comes out for every task. So let's remove the task.id from actually being printed. So we'll wrap all this in a link. We'll need to import link from react-router-dom. And we'll give this link a property. We'll say Link to, and we'll pass a string, /task/$ curly bracket task.id. So this here is a template string, and this interpolation will tell us to actually go to the /route of the task.id. And one more tricky thing. React would like this key to be on the highest-level element, the Link. Let's fix that. Cool. So now if we click any of these tasks, it takes us to a unique URL with that task's ID at the end of it. Why, we can even create our own task and see our uniquely generated uuid here at the top. Very cool. Now let's update the TaskDetail to reflect the relevant information regarding the task. So we'll just have an h2 tag and say task.name. We'll have a button, which we'll configure later, which will just say Complete / Reopen Task. We'll add a select tag, and we'll add groups here to the arguments of TaskDetail. And in our select, we'll just say groups.map, and it'll be an option for each group with the key equal to group.id, a value equal to group.id, and a contents equal to group.name. And we'll have a link back to the dashboard, so we'll import link from react-router-dom. We'll say Link to the dashboard, and we'll have a button, and it'll say Done. Now before this works, we have to update mapStateToProps. So we'll update this to be an argument with two, state and ownProps. Now we're going to need a whole bunch of different properties that we have to cleverly assemble from the state. We'll get the id of the task, which will be ownProps.match .params .id. We'll need the task, which we will use by going state.tasks .find and finding the task with the appropriate ID. And we'll let the groups equal state.groups. And we'll return an object containing the id, the task, the groups, and we'll add a shorthand for isComplete. We'll just say isComplete is equal to task.isComplete and make sure isComplete is spelled right,

better than I spelled it. Let's see how that looks. So it's starting to look right, but it still looks a little bit cramped. Let's update it. First, let's put this button in a div and this select in a div and this link in a div. That's pretty smart. And now let's actually just update this task.name. So instead of an h2, it's an input with a value equal to task.name. And we'll put that in the div as well. And now our task route is the displaying the name of the task, as well as a Complete / Reopen Task button and the drop-down where we can change what the task is. Of course, none of these actually do anything until we hook up the correct dispatches and mutations, which we'll do in the next clip.

Implementing Task Details Route Part 2 - Mutating Data

In this clip, we'll be augmenting our task details route so that it does not only display data, but also changes the data. So we'll add methods, which dispatch actions when the various form elements are interacted with. And we'll add clauses to the Redux reducer, which causes the state to change when the relevant action is dispatched. So first, let's give some life to this complete or reopen task button. First, let's change the text. So we'll put a statement here, and we'll say isComplete? If it's complete, it'll say Reopen. If it's not complete, it will say Complete. Now let's scroll down and create a mapDispatchToProps method. Let's do that below mapStateToProps as is the convention, mapDispatchToProps. So it gets two arguments, dispatch and ownProps. We'll start by defining an ID, and that will be equal to ownProps.match .params .id. We'll be needing that later. Now I'll return an object. This object will have a few properties, but will start with a setTaskCompletion property with two arguments, id and isComplete. We'll also need a corresponding mutation before we can proceed. So let's go into our mutations file, and let's create a few constants. I'll create a constant for SET_TASK_COMPLETE, and now I'll quickly do the same thing for SET_TASK_GROUP and SET_TASK_NAME. Now let's make some action creators. We'll make one called setTaskCompletion, which will get two arguments, id and isComplete, and it will have a type of SET_TASK_COMPLETE, a taskID equal to the first argument, id, and an isComplete property equal to the isComplete value that was passed. Now I'll copy and paste this and just modify it slightly so there's one for SET_TASK_NAME and SET_TASK_GROUP. So that's SET_TASK_NAME and SET_TASK_GROUP. So now we have three mutations. Let's go back to TaskDetail, and we'll import * as mutations from store/mutations. And now here, in setTaskCompletion, we'll dispatch mutations.setTaskCompletion, passing in the id and the isComplete value. And there's actually one thing we need to do real quickly to make that work. We'll take our setTaskCompletion method that we've made and pass it as an argument to TaskDetail. Now we'll add an onClick to this button, onClick, and it will be a method, and it will call setTaskCompletion, passing in the ID of the task and the opposite of whatever isComplete value

was passed. Let's make sure that mapDispatchToProps is here being passed after mapStateToProps, and let's have a look at the effect on our app. Great. So you can see now that an action is being dispatched with the correct taskID and setting the isComplete value to something else. Notice this error at the top. This is expected since we have this input here with no way to change it. This error is saying that if we try to change this input, nothing will happen. In my opinion, it's really kind of more of a warning, but that's how it goes. Now let's update our reducer to be able to handle this SET_TASK_COMPLETE. It should be pretty simple. Go to store/index. So we'll add another case for mutations.SET_TASK_COMPLETE, and we'll return tasks.map. And for each task, we're either going to return the task itself or a modified version of it. We'll say return, and we'll have a ternary here. So we'll say if the task.id is the same as the action.taskID, we'll return either this object or the task itself. In the case of this object, it will have all the properties of task ...task followed by an isComplete property equal to the action.isComplete property. So basically what we're saying is if the task doesn't match the ID, leave it alone. And if it does match the ID, give it a new isComplete property. Alright, let's see if that works. Save that. Cool! So now when we click Complete or Reopen, the button changes. And if you wish, you could reflect this on the dashboard by having these indicate if they're complete or not. So, as you can see, the completion status for Meet with CTO persists even if I go back to the home page. But, of course, if I refresh, it goes back the way it was originally. Now let's wrap up this massive and very dense module by adding mutations for these other two elements, the input and the group drop-down. Now back in our TaskDetail, let's update mapDispatchToProps. We'll add two more, setTaskGroup and setTaskName. In the case of TaskGroup, it's going to take an argument e, as in event. We'll then dispatch a mutations.setTaskGroup mutation, passing in the id and the e.target .value, which will be the group that's selected. While we're at it, let's make setTaskName as well, which looks exactly the same and is the same really, except it's a setTaskName mutation. Not too bad. Add a comma there. Now you're probably thinking, we could do this all with one simple mutation, setTask, and that certainly is a valid alternate way of doing it. I prefer to set things out in the longhand like this as this tends to facilitate better maintenance by large teams or perhaps someone who's never seen this application before being brought in to update it. But, of course, this could be done in a less verbose, more condensed way. So we have our two dispatches. Let's write those in here in the arguments that are provided, setTaskName, setTaskGroup, and let's hook them up to the appropriate value. We'll update the input, and all we'll say is onChange setTaskName, just like that without any brackets. And for the select, we'll say onChange setTaskGroup. Ah, if only they could all be so clearly written. Now let's see what happens when we interact with our form. So as you can see, interacting with these two forms causes actions to be dispatched on our logger, but nothing changes yet. The reason is that we have to update our

reducer to deal with this interesting variety of new actions. That should be easy. So let's go to store/index, which is where we're currently keeping our reducer. By now, you've probably figured out what this is going to look like. We'll create a new case for our two new mutations that we've just set up. We'll start with TASK_NAME. And we're just going to copy this code from above since it does the same kind of thing. We just have to change one element of it. So we'll say if the task matches the task in the action, copy the task. But instead of isComplete, we're just going to go with name, and we'll say action.name. And let's add one more for group, task. And the group will be action.groupID. Don't forget that ID if you've been writing your application the exact same as mine. And we'll save that, and maybe it'll work. Let's see. So here's the application. First, let's test the group changing. This was in To Do. Let's move it to Done. Ah, there it is. And, I don't know, let's take Compile ES6 and move it to Doing. Hmm. It looks like there's a little problem. It always seems to show To Do by default. Let's fix that. So to fix that up, let's just give our select a value of task.group. Very nice. Now check our name changing. Let's change it to ES7, why not? And bingo, it's updated. So we're now updating our application state with actions. That's super, super cool. That's as far as we're going to take it in this module. In an upcoming module, we're going to take this and add persistence with a back end that we're going to develop soon.

Summary

Well we've come to the end of building the front end of our application, at least for now. Of course, we still need to add some styles and hook it up to the back end, and I'd be lying if I said building that app we just built wasn't a bit more challenging than I thought it would be. But let's review what we learned just about the front end. So we've learned about Webpack, and we've learned how useful it is because we can write our application in ES6 and JSX, which are powerful subsets of JavaScript. We've learned that Redux is reliable and convenient and a good place to keep our application state. We've used a few React forms and noted that React components often contain these forms, and the user interacts with them to change the state of the application. Finally, we've seen that by using React- Redux, React components can automatically reflect Redux data, allowing us just to focus on cleanly updating the state. Coming up in the next module, we'll set up a server with Express, and we'll install MongoDB and configure it to work with Express, and we'll create a simple REST API that will let us persist data on our server using this MongoDB. It'll be a few modules before we revisit our front end components, so go ahead and commit your changes, and I'll see you in the next module.

Creating Persistent Data Storage with Node, Express, and MongoDB

Introduction

Welcome. In this module, we'll be using Node, Express, and MongoDB to create an implement of our application that stores data indefinitely. First, we'll install MongoDB, and then we'll get it started locally. We'll write a script that initializes the database with the default state we've already prepared. Finally, we'll use Express to create several routes that can modify these contents of the database. We will be adding authentication, but this will take place in a later module. Let's get started. In the next clip, we'll install MongoDB.

Installing MongoDB

In this clip, we'll get MongoDB installed and running on our workstation. So in case you're not familiar, let's do a quick refresher on what MongoDB is. So basically, it's a database, which means it's a place that you put your data where your data stays persistently. It's a non-relational kind of database, which basically means it's less structured than relational databases like SQL. MongoDB uses JSON by default to communicate. This makes it useful to use with JavaScript since JavaScript also uses JSON. At the end of the day, you can choose if you want a relational database like SQL or a non-relational database like Mongo for your application, but in this case, we're using Mongo since it's fast and effective. In this demo, we'll start by installing MongoDB. To install MongoDB, I'm going to go to the URL available at the top of the screen, mongodb.com /download-center. I'll just click this Server button, and the Download button here will download a version that's appropriate for my computer. I've gone ahead and downloaded the file and followed the installation instructions by default. Now I'll use the command line to start MongoDB. So here I am in my MongoDB folder, which I have chosen as E/Data, but yours might be in a different location. We'll open up bin, short for binary, and the file of interest here is called mongod, or Mongo driver. So I'll open up my Git Bash, and on Windows to run this particular file, I need to type cmd first, which turns my Git Bash into a more faithful representation of the Windows command terminal. Now that I'm in cmd, I can type mongod.exe, and you should get a wall of text like this indicating that Mongo is running. You shouldn't see any major errors on this. If

you see this whole page, ending with waiting for connections on port 27017, you've installed MongoDB. Now, for deployment, a separate production version of Mongo will be used, but our deployment partner will handle that for us. For now, we have MongoDB running on our computer, which means we can start persisting data.

Initializing the Database

In this clip, we'll be initializing the database. First, we'll install a tool for verifying the contents of the database. I'm going to be installing a tool called Robo 3T, but we're not going to be using this tool for anything other than just making sure our database looks like we expect it to, so you can really install any tool that works for this purpose. So here I am at robomongo.org, which is the former name of Robo 3T. I'll click this link to download Robo 3T, and in a few minutes it will be installed. I have now completed the installation of Robo 3T and have the application open on my computer. I'll go ahead and create a new connection, and I'll just save it with all the default properties. Then I'll press Connect, and here is our local MongoDB data. These names on the left are of previous projects of mine. Now let's initialize our database. So what we're going to do is create a Node script, which is basically going to populate our database with the default state we've already made. So here I am in my application. First, we're going to need to install the MongoDB Node driver, so I'll open up my Terminal and say npm install --save-dev mongodb@ 3.1 .10. That's all looking good. Now first we need to connect to our DB before we can initialize the state, so I'll make a new file in src\server called connect-db. So first, we're going to import MongoClient from our new MongoDB library. Now we need to define our URL. We'll use a different URL for production, but locally we'll say our URL is a string mongodb://localhost:27017, which is that URL we saw earlier, forward slash, followed by the name of the collection. This collection can be any name. I'll call it myorganizer. Finally, we'll have a variable that represents our database connection. This is so that we don't have to reconnect every time we want to use it. We can just reuse this existing connection. We'll call it db, and it'll be equal to null. Now let's write the function that actually connects. We'll call it connectDB. It'll be async, and we'll export it. So we'll say let client, which refers to the connection, be equal to will await MongoClient.connect. The first argument we pass, connect, is the URL, which will just be our URL variable from before. The second is arguments, and in this case, we just have to use one argument, useNewUrlParser. Make that true. We're not actually going to be using these URL parsing features very extensively, but this will prevent an error from coming up later. Now we'll say our db variable that we've defined above will be equal to client.db. We'll add a console.info here just saying we Got the DB, and we'll return the db. Finally, so that it can be cached, we'll add at the top, we'll say if database is

defined, simply return it. So on this line of code here we're defining the database, but this line of code will never run again as once it's defined, it will just use this existing variable. Let's go to package.json and run a script to test this file. I'll write a script called test-connect, and it will be babel-node src/server/connect-db. And if I open my Terminal and I run my new script, nothing happens because connect-db merely defines a function and doesn't call it, but we can just invoke connectDB here at the bottom so that it runs, and we'll test our script. And you can see it logs this long list of properties, which represents our database. Looking good so far. Let's stop this, get rid of this connectDB call here, and we'll write a new script called initialize-db. So we'll be importing the defaultState from this directory, and we'll also import connect-db that we just wrote. Now we'll define an async function called initializeDB. First, we'll get our database connection by awaiting connectDB. Say let db = await connectDB. Now we'll say for (let collectionName in defaultState). So if we look at our defaultState, those collection names would be users, groups, tasks, etc. So we'll define a collection by saying db.collection, and we'll give it the name of the collection in our defaultState. Then we'll use our first MongoDB command, collection.insertMany. Collection.insertMany works pretty darn intuitively. All we have to do is pass it an array, and that array will hopefully be inserted into the database if everything is valid. So we'll just say defaultState, square brackets, collectionName. Finally, we'll invoke our initializeDB call. Cool. Now let's go to package.json. We don't need test-connect anymore, so I'll delete this and write a new one. We'll just call it initialize, and we'll say babel-node src/server/initialize-db. Now I'll open my Terminal, and I'll say npm run initialize. So I'll get the same console log as I did, this is our connection, but let's go to Robo 3T. So if we go to Robo 3T, we can see a new collection has been created called myorganizer. If I go inside, the collections I've defined, groups, tasks, users, and comments, have been added. If I have a peek at these tasks by double-clicking, I can see all the tasks are here. Unlike our defaultState.js, this data is persistent. It can be migrated from different databases and is generally a much smarter place to store your important client information than just in your Node memory. So take a moment to appreciate your newly initialized state, and in the next clip, we'll continue integrating.

Creating a Server and Updating Tasks - Part 1

In this clip, we'll be creating a server. So why do we need a server? As we'll recall, first of all, servers can hide confidential logic from the end user that a client cannot. Our server provides our client a consistent way of working with our database that we just set up. And finally, it's useful for serving the HTML and finished JavaScript files of our final application when we get to production. So, for our server, we'll be using Express, but what is Express? So Express doesn't really create

anything new, but wraps the existing HTTP toolset that is offered by Node in a more convenient package. Basically, we'll be going through the standard process of creating a server and then listening to a particular port. We'll use functional JavaScript to define the way that these requests should be interpreted and what data should be sent in response. Looks like it's demo time. First, we're going to create an Express server and listen for HTTP requests. First, we'll install Express. We'll say npm install --save-dev express@ 4.16 .3. And while we're at it, we'll also install cors@ 2.8 .4, which is an Express plugin that will let us do cross-origin resource security, as well as body-parser@ 1.18 .3, which is another Express plugin which will let us use POST requests. Let's press Enter to install these. Nice. Now we'll create a new file called server\ server.js. So we'll do a few imports. We'll import express from 'express' and do the same for cors and bodyParser. We'll also define our port, and the port can be any number we want for this time. I like to pick one that's usually not taken, like 7777. Of course, you might see 8080 or 3000. You can really choose any port you like at this point. Now we'll create a new Express instance by saying let app = express. Then we can say app.listen, provide the port, and we can also add a console statement that will be displayed once it's listening. So we can just say console.log ("Server listening on port", port). Let's write a simple line of code to test it. We'll say app.get, forward slash, followed by an argument, which is a method, the first argument being req and the second argument being res. Req refers to the request parameters, and res refers to the response. When we're using Express, when we want to respond to something, we say res.send, and we can send whatever we want. For example, let's say Hello world!!!!! Let's give our script a shot. We'll go to package and make a new script called server, and we'll run it by saying babel-node src/server/server. And we can run it, npm run server. So I got an interesting error. It said the address 7777 is already in use. I can tell this is because I've already used that address setting up a similar example earlier. But if you get this error, this is an indication that you need to pick a different port. In this case, I can just go to server and change my specified port to something different, maybe 8888. And I'll try running it again, and now it's listening on port 8888. If I go to localhost:8888, I see my message Hello world!!!!! I find this actually so cool that with just a simple line of code we can create this Express server. It's really doing everything we might do with Apache or NGINX, but now we have the opportunity to describe everything we want functionally using JavaScript. So cool. But this is just a simple example. Let's write our real application. So I'll get rid of app.get. First, I'll say app.use. Now whatever I pass here is going to be considered a plugin of the application. We'll use cors invoked with no arguments, which just means to create very lax CORS requirements. Then we'll require bodyParser.urlencoded. I'll pass it the option extended equals true, and we'll also use bodyParser.json. These two last ones will let us use POST requests, which are similar to GET requests, but more powerful, though harder to use. First, we'll create our route for adding new

tasks. We'll say app.post /task/new, and a second argument will be an async function with the arguments req and res. In this case, the task details are going to come to us from req.body, so we'll say let task = req.body .task, body being whatever the data that the requester passes in with an HTTP request. Now, what I'm going to do is I'm going to write a separate function that actually communicates with the database. This is because POST requests are hard to test. If we wanted to verify that our logic was working and had to make a POST request every time, that would be pretty time-consuming. So I'm going to write a file right here. I'm going to say export const addNewTask = async. It's going to be a method with an argument called task. Let's import our DB connector by saying import from connect-db, which is import connectDB. At the top of addNewTask, we'll say let db = await connectDB. We'll access the tasks collection by saying let collection = db.collection and pass in the string tasks, and we'll await collection.insertOne, passing it the task. Now of course you'll note that the task could have any structure, even a variety of properties that is totally different than any previous task. This is both a strength and weakness of MongoDB. So we have this method now in task/new. We'll just call an await addNewTask. We'll say await addNewTask, pass it in the task, and then we'll say res.send nothing, but we'll also say res.status (200). Two hundred is an OK status, so when you send a status of 200 followed by nothing, it simply tells the requestor that your request was okay. Now we're not actually going to be able to use this until the next module where we integrate it with our front end, so let's write a little spec to make sure addNewTask works as expected. We'll say New, File, and we'll call it server.spec .js. We'll import just addNewTask from our server file, and we'll call addNewTask, passing it in just a simple task. Just call it My task. Give it a simple ID. Now I'll go to package.json and make a new script, server-test, and we'll say babel-node src/server/ server.spec. Now if I open up my Terminal and run server-test, I don't get any error at all. Let's go to Robo 3T. So if I go to myorganizer, Collections, tasks, sure enough, the new task I've added is right here at the bottom. We've now verified that our function that adds tasks works, and we have a server route which should call that function every time the front end invokes it.

Creating a Server and Updating Tasks - Part 2

So we've added route for creating a new task. Let's now add a route to update a task. So here in server, right below addNewTask, I'll add a new const updateTask be an async function called task. Basically, to use this, you pass in the ID along with one or more of any of the properties you'd like to change of the task. So you might want to update its group, putting it from to do to done, or its isComplete status, or even its name. So we'll destructure task into these properties. We'll connect the DB like above, let db = await connectDB, and we'll also get the collection. We'll say let

collection = db.collection of tasks. Let's start with group. If group has been defined, that means that the requestor wants to change the group this task belongs to, so we'll say await collection.updateOne. The first argument being the properties to match. In this case, we'll put id in curly brackets, meaning find the object with the matching property ID, and there should be just one. And the second argument, first we'll use a special keyword called $set, which means that whatever object is passed as the set property will be changed in the record. So we'll pass it an object with a group property that will be equal to our group. Now let's do the same for name and isComplete. I'll copy this and paste it for name. And of course, if you are suitably creative, you could combine these different if statements into one. But briefer is not necessarily more clear, so let's just keep it like this. Finally, we'll also add a clause for isComplete. And since isComplete being undefined would in some cases be equal to isComplete being false, we'll say if isComplete does not equal exactly undefined, then we'll update the collection with the appropriate ID, and we'll set isCompelte. Very cool. Now let's update our server. Our task update looks similar to task/new, so I'll just copy and paste this, change this URL to update, and we'll just call updateTask instead of addNewTask. Very cool. I'd also like to point out that the way the server is written out, even if there was some kind of error, for example, if they couldn't find something with a similar ID, there's no handling for that. All applications need to have some kind of handling so that your devs know where to start troubleshooting. But since we have limited time, I'll leave that up to you. To end things off, let's just test our update method. So I'll go to server.spec. We called addNewTask here. Let's actually wrap this all in an async function. Say async, and we'll say await addNewTask. Just say yes, async function. Needs a name. Async function myFunc. So we'll await addNewTask. I'm just going to change this id here and give it a 6 at the end, and we'll also import updateTask. So after we've waited for that, we'll await updateTask, we'll pass the ID that we just used, and we'll give it a new name. And we'll end it by just kind of invoking this function with two sets of round brackets. Now if I try to run server-test again, I'll say npm run server-test, great! So back in Robo 3T, if I go to tasks, I can see here the last task on the list is My task - UPDATED!!!!! So we can confirm that our server interaction with our database works. Now all we have to do is hook our client to these cool capabilities. That sure was exciting. In the next clip, we'll wrap up this module.

Summary

Oh my goodness, folks. Creating an Express server is just a whole lot of excitement for one day. Let's wrap up this module so we can take a 2-minute breather. So we learned that MongoDB is our tool of choice for persisting data in the form of a database. Our goal here is to make the data last indefinitely. We learned that we can use Express routes to allow certain types of interaction

with our DB, for example, creating tasks, while limiting access to others, for example, deleting tasks, which we created no interface for doing, so our client is unable to do that kind of action. Finally, we saw that the purpose of our Express routes is to respond to HTTP requests, for example, POST requests, and respond with data or with changes to data. Coming up in the next module, we're going to integrate the front end with the back end so that the changes that the user makes will persist. We're going to do this by updating our sagas to make HTTP requests, and to do so, we'll be using a cool little library called Axios. Finally, we'll update our front-end application so that instead of getting the default state from a hard-coded value, it will make a request to the server. Going to be really fun. See you there.

Integrating React View Layers with Persistent Data

Introduction

Hello, and welcome to this module entitled Integrating React View Layers with Persistent Data. In this module, we'll be combining the work we've done in the previous two modules, allowing our front end to communicate with our back end. Here's what we'll be doing in this short module. First, we'll update our npm script so that we have a script that initializes the front end and the server at the same time. Next, we'll update the sagas that we created in a previous module to communicate with these endpoints on the server. In the next clip, we'll be updating our npm scripts to initialize the server and client at the same time.

Initializing the Server and Client Simultaneously

In this clip, we'll be initializing the server and the client at the same time. So we'll run a script, and that script will initialize webpack. Webpack, with its dev server, lets us work on our front-end application. The script will also initialize Express, and Express will listen up for HTTP requests coming from our application. We'll add those in this module. So let's go into our application and add the npm script. So here I am in package.json. To run both the scripts at the same time, we'll want to use an npm package called concurrently. So let's open up our Terminal, and we'll install --save-dev concurrently version 4.0 .1. Great! Now I'll close my Terminal, and I'll make a new script called start-dev. So we'll say concurrently. So the commands that come after concurrently have to

be in quotes, but since our JSON document also uses quotes, we're going to escape the quotes with a backslash. So I'll say \"\", and now in here we can put the scripts that we want to run concurrently. So we'll say npm run server, and we'll do the same thing for npm run dev. Now let's open up our Terminal, close all our other processes, and run npm start-dev. Great! So that script should get our server started, as well as launch our application. In the next clip, we'll be updating our sagas to communicate with this newly available server.

Using Client-originated HTTP Requests to Modify Persistent Data

In this clip, we'll be using client-originated HTTP requests to modify persistent data, which is a fancy way of saying the stuff we do in the browser is going to change what's on the server. So as a quick refresher, what is an HTTP request, and why is it important to us right now? So an HTTP request is a kind of web request, literally electrical signals, that can be used to communicate with a server. Of course, there's lots of opportunities for malicious programmers to use this feature to, for example, steal users' information. So there's a feature called CORS, or Cross-Origin Request Security, which prevents many HTTP requests. Basically, unless authorized, you can only make a request to the same domain. Finally, we have many front-end tools to abstract away the complexity of making HTTP requests. Axios, jQuery, and Fetch all boil these processes down to a simple GET command. So let's jump into our demo. First, we'll install Axios using npm. There are many options, but in this case, Axios is the tool we'll be using to make our requests. So we'll install --save axios@ 0.18 .0. Great! And Axios is installed. Now we're going to update our sagas to replace the sagas that we have that don't interact with the back end at all. Our task creation saga will send an HTTP request to the new task route that we've already prepared, and the task update saga will send a request to the update task route. Pretty simple. So let's open src/app/store and open our sagas.mock. We'll keep this sagas.mock file around in case, for example, we want to use it for later tasks, and we'll make a new file called sagas. So first, we'll do some imports. We'll import from redux-saga/effects, and we'll get three effects, take, put, and select. We'll also import UUID and Axios. We'll also want our mutations, so we'll import * as mutations from 'mutations'. Next, we're going to define a URL. This URL is going to be the URL that we'll use to communicate to the server. We'll update this to something more sophisticated when we publish this, but for now, we'll just say const url =, followed by http://localhost, followed by whatever the port is we're using. In the previous video, I used 8888, though you may have a different port depending on what system you're using. So I'll just put the same port as I have on my server. Now let's update task creation. I'm going to go to my mock sagas, and basically, this existing taskCreationSaga has everything we need. We just need to add a bit more. I'll copy this, paste it. So the

taskCreationSaga was interesting because before we were using it to generate some randomly originated information for reducers, but another thing saga does is communicate with our server. So I'll delete this log, and I'll say const res in square brackets, as in response, equals yield axios.post. We'll say the URL we defined above, as in localhost, followed by /task/new. And you will notice that this matches the URL that we set up in the Express example. Now we'll pass another argument, an object, and so whatever we put here will become the body property of the POST request we're handling on the server. So we'll define an object, and that object will have a property called task. Task will have an ID of taskID, which we defined above, as well as ownerID and groupID. It'll have a group of groupID, an owner of ownerID, an isComplete value of false, and a name of New task. Just in case we have to troubleshoot it later, I'll add a console.info here saying Got response, followed by the response. I'll save that, and now let's open up our application. Hmm, nothing happens yet. It seems I forgot to replace our mock sagas with our real sagas in our application. So let's go back to the app, and in store/ index.js we're just going to get rid of * as sagas from sagas.mock and replace that with * as sagas from sagas. Pretty easy to make that change. Oh, and I made another little error here in sagas.js. This should be. /mutations, just saying mutations will make it look for an npm module called mutations. Dot forward slash says hold on, look just for a JS file called mutations in the same directory. So we'll save that. Alright, when I press this Add New button, it will trigger our existing saga. And we can see here at the bottom, Got response. The response is undefined at this time, but that's okay. It does mean we received a response from the server, and we could we wanted. Let's open Robo 3T, and we can confirm that the task that we've added, complete with whatever group we might have specified, is now being persisted onto the server. We're not actually going to see this change in the application until we assemble the default state in the application server, which we'll do in the next module. But Robo 3T allows us to verify that our data is persisting here. Back in sagas I'll delete this console log. Now I'm going to make a new saga, a taskModificationSaga. And because of the way we're importing our sagas into the store using a star, this saga will run automatically, as we are doing a for loop for let saga in sagas. So we'll start this with a while (true) loop, and we'll say const task = yield take. So whatever we put in this array, we're going to pass an array of different actions, and if any of these actions are dispatched, the next line of code will run. So we'll say mutations.SET_TASK_GROUP, and we're also interested in SET_TASK_COMPLETE or SET_TASK_NAME. So the reducer is already listening for these mutations and making the appropriate change to the state. All we're going to do here is send a request to the server that essentially informs it of this user action. So we'll say axios.post, and we'll say url + /task/update, which you will note matches another route that we made on our server, and we'll pass it an object, and the object will have a task property. And now we'll just fill this in with the properties

of task. So id will be task.taskID, group will be task.groupID, name will be task.name, and isComplete will be task.isComplete. We'll save that. So here we are back in our application. You might notice that when you refresh it takes you to a task, followed by a URL, and nothing appears. That's because this task that we've created is not part of the application state we can get from the server. In the next module, we're going to add a route guard that prevents this route from being accessed without the proper permissions. So we'll go to dashboard and refresh, and this, of course, does work. Now we'll add a new task called New Task, and this time let's also modify it, so let me change its name. And I completed it. Now, if we open Robo 3T, you can see here is my new task, complete with the modifications I made after I created it. Nice! Now as far as the application goes, when we reload it, we're still getting the default state. The next chapter is going to be most exciting because when we add authorization, we'll be able to assemble the state on the server, and then when we refresh the page, our changes will persist. But that's about it as far as integration between the front end and the back end goes. All that needs to be added now is authentication, and we can start thinking about deployment. In the next clip, we'll wrap up.

Summary

Let's go ahead and summarize what we've learned. So we've learned that when you do something on the client and you want the server to know about it, you usually use HTTP and have a REST API, which is what we made in Express, to deal with it. We've learned that Axios is a tool we can use to send requests from the front end, and Express is a tool that we use to receive requests in our back end. And we've also learned that not only do side effects help our communication with the back end, but if we design them properly, we can also have a powerful set of mocks that work without the server. In the next module, we'll be adding authentication to our application.

Authentication Concepts

Introduction

Hello, and welcome. In this module, we'll be discussing authentication. Authentication is extremely important as an application which can't authenticate users can't really provide a unique experience for each one. As the bard once quoted, "Love all, trust a few. " In other words, love all your end users, but only share data with a user that they're allowed access to. So what is authentication? Authentication is a deceptively simple process. First, a user logs into an

application and demands access to certain data. Maybe, like ours, it's a daily planner application, and they want to know what their schedule is. We use the tool called authentication to determine if this individual is the user they claim to be and if they have access to the data they want. So authentication is a process which decides if it's okay for a user to access certain data. Generally what we do is we take a process called hashing where we turn the user's password into a long string of characters which can't be turned back into the original password. Then we compare it to something we have on the database. When authentication works properly, it allows many different users to have separate sets of data. In a sense, they all experience your application in their own unique way when they use it. As you may have guessed, we need authentication for many e-commerce activities since spending money is one of those things that we really don't want other people doing on our own behalf. In this course, we'll be using passwords to authenticate, but it's important to understand that there's many other ways to accomplish this. You can, of course, have your string of characters, a password, or you can use biometric data, like a fingerprint, a retinal scan, or even a microchip, believe it or not. We often use third-party authentication. For example, you may have recently logged on to an application for it to offer you to simply authenticate yourself with Facebook. In this case, the application asks Facebook, and Facebook confirms you are who you claim to be. Finally, certain applications, mainly those that deal with finances, like in the fintech sector, will need you to authenticate yourself even further by, for example, sending in a scan of your drivers license to be verified by a real person. This is all part of the process of making sure you are who you say you are. So what are we going to cover in this module? First, we're going to create a route guard that reroutes unauthenticated users to a new page. In other words, until you're logged in, you won't be able to access the application we've made so far. We will, of course, create a login page and give it a simple form that communicates with the server. We'll update our default state to include authentication information, as in passwords, to give our application an extra layer of complexity. And we'll create an auth route on the server which, if you're credentials are correct, will give you the necessary token to use the application. It's going to be really great, so stick with us. In the next clip, we'll be adding route guards.

Adding Route Guards

Just a quick reminder. You're going to want to complete at least a few courses on security available here at Pluralsight or elsewhere before attempting to secure your application in the real world. In this course, we're discussing general principles and the methods of authentication, but as data that's hidden behind authentication is often valuable, clever individuals are always trying

to break into your application and obtain that data. Thus, while you will learn great principles here, you'll need a bit more knowledge of security before you can really defend your application against all outside attackers. So let's add a root guard, or a route guard, depending on how you pronounce it. So what's a route guard? So some routes, and yes, I'll just be using root and route interchangeably in this explanation, should only be accessed by a correct user. For example, you might have a page which displays user A's schedule, and you don't want user B to see that. So route guards are a way for us to organize authentication logic. It doesn't really make sense to put that logic in React components or our Redux state, so this is a good place to put it. Route guards aren't fancy. Basically, all they do is they do a quick check, and if that check is in, if the user is not logged in, then it just redirects them either to a forbidden page or, in our case, the login page. So, let's jump right into the demo. First, let's create a route guard and add it to the routes we have. So here I am in Main.jsx. First, let's write our route guard. So I'll say contrast RouteGuard, and we'll make this a method which takes a Component as an argument. And now this is actually going to return another method, and that method will take an object as an argument. We'll destructure that object by using these curly braces, so we just want the property of the object known as match. Now we'll write our function. So the way the route guard works is you're able to run some logic inside which determines which component is returned. The simplest logic would be simply to return the component in question. So, for starters, we can say return, and we'll use JSX to say Component, and we'll say match=match. And we can also put an info log here just saying console.info, Route guard, and we'll log the match. Now you'll notice down here for a render process we already conveniently have this in the form of a function, a function that returns whatever component we want. So conveniently, we can just say RouteGuard and pass in the name of the component we're doing here. So here we'll say render, and we'll replace this with RouteGuard, passing in the ConnectedDashboard. And here we'll pass in the RouteGuard, passing in the ConnectedTaskDetail. Save that. So now if we look at our application, we can see it still looks the same, but we're getting this Route guard message. So now our route guard is somewhat in place. We have to give it some logic. So what we'll say, we have the store that's been imported here in to this context, so let's say if not store.getState .session .authenticated, and we're going to add this variable in later. So if it's not authenticated, we'll reroute. If it is, we'll say this line here, just return the Component. So to do this, we're going to need Redirect from react-router. As Redirect is part of React Router and not React Router DOM, let's go ahead and install this application. Npm install --save react-router. Great! So with React Router installed, we can import Redirect from that library. Import Redirect in curly brackets from react-router. Now here in this empty clause we'll say if the session is not authenticated, we'll return Redirect, and it will have a to property, which is the path we're going to redirect to. We'll just redirect it to forward slash

and close that. So if we open up our application, we'll see Cannot read property 'authenticated' of undefined. We actually haven't defined a session property of our state yet, so let's go ahead and do that. We'll go here into server/defaultState, and for now we'll just add a session object, and we'll just say authenticated is false. And then in our store/index for Reducer, let's add a session, say session = defaultState.session. And we're not interested in the action just yet. We'll just say return session. Alright, so now if we go to our application, if we try to go to /dashboard, it will reroute us here to this base URL. That means the route guard is working. But let's put something here, a stub if you will, that we're going to develop later. Let's go back to the app. So I'll make a new component called Login, and for now I'll just import React from react and connect from react-redux. I'll define a LoginComponent, and we're going to be building this out later, so we'll just have it return just a div tag that says Login Here! Maybe give it an explanation mark if you're feeling high energy like I am this morning. We'll define mapStateToProps that we'll leave mostly empty for now, so it'll just return the state. And now we'll export our ConnectedLogin like before. We'll export ConnectedLogin = connect, passing it mapStateToProps, and then invoking that on our very simple LoginComponent. Now let's go back to Main.jsx. We'll import our Login from the same place that we just made it. I'll just adapt to this line of import. Oh, and I made a little mistake here. It should be export const ConnectedLogin. That is looking a lot better. ConnectedLogin. Now as we've defined it here, if our session is not authenticated, we're going to redirect to this forward slash, so all we have to do is define another route here. So we'll say Route exact path="/". And this does not need a route guard. We can just say component. In React Router, if you say render, you must provide a function, but if you say component, you just provide the component you want to display when that route is there, so we'll say ConnectedLogin and save that. And I made a little tiny error there with an extra single quote in that. Make sure that is just a forward slash. Alright, the application is looking very cool. So as you can see, here's our login page stub, Login Here!, and if we're clever and try to go say /dashboard without logging in, we're redirected back to the login page. Now we can put some logic here that'll actually let the user authenticate, which we'll do in the next clip. Very fun.

Creating a Login Page

In this clip, we'll be creating a login page and a login saga, which will allow our end user to authenticate themselves once we add the appropriate back-end logic. So for this demo, we'll update the default state to include some passwords and then reinitialize the DB to make that possible. Let's jump in and do that. So here we are in the application. You'll recall that we need something to create password hashes. We don't want to actually store the real passwords on our

database because that would allow an attacker to get a lot of information if they succeeded. So we're going to use a library called MD5. We'll open the Terminal, and we'll say npm install --save md5. Great! Now let's go to our defaultState. We'll import md5 from md5, and now we'll give the two users their own password. And we'll call it passwordHash just to be clear it isn't the raw password. But we're going to take the password, and we'll pass it to the md5 function. Let's make a password for Dev, TUPLES, and we'll also give a password to Mr. C. Eeyo. Now we're going to want to update the default state we have on the database, the users we have right now. So if I look inside my database, I can see, as I said, that the users have no password. Let's fix that up. So we're going to want to update our initializeDB method now so it's going to work also when we deploy it. So for what we'll do basically is we're going to call initializeDB every time we start the application, but we're going to check to see if there's any users. If there are, then we won't initialize, but if there aren't, then we'll just put that default state in. So I'll say let user = await db.collection ('users') .findOne, and we'll put in the ID of our default user, U1. And then we'll say if there's no user, run this initialization logic that we have here. Now in server.js, we can just import at the top initialize-db, and that'll just bring in the whole file without actually importing any methods or constants from it. So I'll save that, and now let's go into our database editing tool, and I'm just going to drop this whole database. Now let's run npm run start-dev. Oh, and I made one little error. This temporary session property we put in defaultState is going to trip this up. It's expecting all of these to actually be arrays. So let's erase this, and we're going to put it in properly in a moment. So now if I go to my database, I can see that the users are now updated with a passwordHash. You'll notice this is not the same password as original, but the hashed version of what their values are. If you're not familiar with the hash, essentially, we can always take the password TUPLES and get this out of it, but we can't take this and get TUPLES out of it. Alright, our DB is reinitialized. Now we're going to create a login page, and the page is going to have a simple form that lets users authenticate themselves. First, I'll go into store and update session. I'll say session is equal to defaultState.session or curly brackets, in other words, an empty object. That should let us develop our login page. Now let's go to Login.jsx, and let's fill this component out a little. Let's put an h2, and we'll just say Please login. And then below we'll put a form, and we'll put an input. Type will be text. Placeholder will be username. Name will be username. We'll give it a defaultValue of Dev. And we'll add another input. Type will be password, the placeholder will say password, we'll give it a name of password, and we'll give it a defaultValue of right now nothing. And then we'll put a button of type="submit", and it will say Login. Pretty cool. So here's our login form, and if we press Login, nothing happens. We now need to create a login saga that will work with this component to communicate with the server.

Creating a Login Saga

Next, let's create a saga which will work with the server to confirm the user credentials entered in the form we just made. Before we do that though, let's go ahead and actually write the mutation that the saga is going to listen for. Let's go to mutations. So we'll create a const, and we'll say export const REQUEST_AUTHENTICATE_USER. And we'll just make that equal to a string of the same name, and now we'll export a mutation, also called requestAuthenticateUser in CamelCase, and it will take a username and a password as arguments, and we'll return object with the type REQUEST_AUTHENTICATE_USER along with the username and password. Now inside our login page, we'll import * as mutations from mutations, and we'll say from store/mutations like that. Then we're going to need to dispatch this event, so we'll need to use mapDispatchToProps. We'll say const mapDispatchToProps =, and it will be a method that takes dispatch as an argument. And so we'll pass in a method called authenticateUser, which will take an event as the argument because it will occur when we submit the form. First we'll say e.preventDefault so the page does not refresh in certain browsers. Then we'll get access to our username, which will be e.target, as in the form, and then the username property of the form, and that's value. And we'll do the same for password. And with these two values, we'll just call dispatch, pass it mutations.requestAuthenticateUser, pass it in the username and password. Now we can say for this form we just wrote, onSubmit=authenticateUser and now pass authenticateUser as a destructured argument here at the top of LoginComponent. We'll save that. And we mustn't forget to pass mapDispatchToProps as the second argument to connect. Awesome! So now when we press Login, we can see in our logs that this action is being dispatched. The arguments are correct, but of course, the next state doesn't have any change. So now we have an event for the saga to listen for. Let's write our exciting new saga. Let's go to sagas, and I'll export a new function*, and we'll call that userAuthenticationSaga. So we'll put a loop, and we'll say while (true). And so that doesn't break, we'll start by taking, we'll say yield take(mutations.REQUEST_AUTHENTICATE_USER), and we'll make these values username and password. So now we'll put a try catch block. We'll say try, and then we'll try to get some data back from the server. We'll say const data = axios.post. We'll say url, and then we'll refer to a path that we haven't written yet, authenticate, passing in the username and password as arguments. And then we'll put a catch block and just say console.log ("can't authenticate"). So for now, we'll just say if there's no data, which will happen if this post fails, we'll throw a new error, which will bring us to our catch block. Now when this fails, we want the user to know somehow that they have been able to log in correctly, so we'll say yield put(mutations), and we're actually going to write a new mutation here, so we'll go to store/mutations. So we have the

REQUEST_AUTHENTICATE_USER, but we're also going to have an action called PROCESS_AUTHENTICATE_USER. Let's create a few more consts. I'll just call it PROCESSING_AUTHENTICATE_USER and give it the same name. And now we'll give a couple of consts. We'll make a const for AUTHENTICATING, AUTHENTICATED, AND NOT_AUTHENTICATED. Great! Now at the bottom we'll export const processAuthenticateUser, and this will just sort of contain a status and a session property. The status will be equal to AUTHENTICATING, the variable we just made, and the session will be equal to null, and it will just return an object with type PROCESSING_AUTHENTICATE_USER, the session value if it existed, and an authenticated value that's equal to the status that's returned. So now here back at our saga, we can yield put(mutations.PROCESSING_AUTHENTICATED_USER and pass it in the status of mutations.NOT_AUTHENTICATED. I made a little mistake here. This should be the lowercase .processAuthenticateUser. There we go. Now let's make a reducer for the session that we're working on. Go to store/index, and here in this session reducer we'll add some more logic. So the second argument will be action, and we'll destructure action, so we'll say let type, authenticated, and session equal the action. Now we'll add a switch on type. When the type is REQUEST_AUTHENTICATE_USER, we'll return the session, but we'll have an authenticated value of mutations.AUTHENTICATING. In other words, this is telling the application we're in the process of doing it. Then we'll also listen for mutations.PROCESSING_AUTHENTICATE_USER. And once again, we'll just return the session and authenticated. We'll also give it a default value, and we'll just copy and paste this up here to return the session. So let's call this version up here, the default one, we'll just call it userSession, and we'll just change these here so that userSession is returned as the default value. And now if we run the application, when we click Login, we can see that we're getting our NOT_AUTHENTICATED message returned from redux-saga. Now let's update our LoginComponent. So here's our LoginComponent, and now we have a property of the state that we're interested in, authenticated. So we'll update mapStateToProps, so we'll destructure the original state into just the session property, and we'll return authenticated is session.authenticated. Now in our LoginComponent we have an authenticated property. Now here under the second input we can put in JSX brackets. We'll say authenticated equals. Now this isn't going to be true or false, but rather one of several different constants. So we'll say if authenticated equals mutations.NOT_AUTHENTICATED, we'll put in a paragraph and say Login incorrect, or in the case that the property authenticated is not equal to that, we'll just have nothing, as in no problems here. Cool! So now if we press Login, we're going to see Login incorrect. This is actually a whole interaction going on between our component, our store, our sagas, and back to our component. Now until we update the server, it's always going to say Login incorrect, so in the next clip, we're going to go ahead and do that.

Adding Authentication to the Server

In the previous clip, we added some authentication functionality to the front end, but without corresponding functionality in the back end, we simply won't be able to authenticate users, so in this clip, we'll be adding authentication to the server. So in this clip, we're going to do two things. We'll add an authentication route to the server which returns the appropriate response based on whether the user's authentication credentials are valid, and we'll also take this opportunity to make a utility that allows the server to make a unique application state for whatever user has decided to log in. So let's jump into the application. So here I am at the application. We'll go ahead and make a new file called src/server/ authenticate.js. We'll start with some imports. We'll import UUID and MD5, and we'll also need the connectDB utility we made from the file of the same name. We'll define a constant called authenticationTokens, which will just be an array. Now we'll make our actual route. We'll export a method that takes app as an argument. The app that's passed in is the same app we've been referring to as the server, so we can say something like app.post so we can define the functionality associated with a route. Let's say /authenticate is the route, and we'll define the handler as an async function taking req and res as arguments. So from a previous clip, we know we'll be expecting an object with a username and password property. Next, we'll connect to the database so it can verify this information relative to the information that we have. So we'll await connectDB. Now we're going to need access to the users collection, so we'll define collection, and we'll make that equal to db.collection ('users'). In this case, we're interested in the user's password and username matching what we have in the database. So we'll say let user = await, and then we'll say collection.fineOne and pass in the username as the name. So here we'll put a little bit of logic. If it can't find the user, it will return res.status (500), which means internal server error, and we'll just send a message, User not found. In this case, we're not sending any state or any information that any user couldn't have accessed. Now that we know we have a user, let's convert the hash that was provided into an MD5. So we'll say let hash = md5 and pass it the password that was provided as an argument up here. Now comes the simple part. We'll define a variable called passwordCorrect, which will be a Boolean, and that Boolean will be true if hash is equal to the user's passwordHash. Now we'll have a bit more logic. We'll say if the password is not correct, we'll send a different error message, return res.status (500) .send ("Password incorrect"). Now if we've gotten this far in the file, that means that the password must match. So we'll define a special token to be that user's authorization token, and we'll push the token into our array of tokens. AuthenticationTokens.push, and it will just be an object with a token value and the userID, which is the user's ID. We will not actually be using these tokens in this course, but you'll need them if you want to add some more security to the application later.

So for now, we'll just imagine the state as just an empty object, and we'll fill that in later, and then we'll say res.send, and we'll send the token and the state. Now we have this route, so let's go to server.js, and we'll import from /authenticate the authenticationRoute. Now here in the application we can just call authenticationRoute(app), and our app will now have the necessary functionality. Now, as the last step for our authenticationRoute, let's add a utility which assembles the appropriate user state upon authentication. So here in our authenticate file, if we've gotten pass this password not correct clause, we know we're dealing with a valid user. So here at the top I'll define a function called assembleUserState, and it will take a user as an argument. So at the top of this method, we'll await for the database to be connected. First, we'll assemble all the tasks which are owned by the user in question, so we'll say let tasks = await db.collection ('tasks') .find, and we'll make sure that the owner matches the ID of the user provided. And we'll say toArray, which tells MongoDB to take this pointer, which is called, and convert it into an actual JavaScript array full of data. And we'll also get the necessary groups, so we'll say let groups = await db.collection ('groups') .find. And we'll say the same thing, the owner must match the user.id, and we'll say to Array. Now we have all the tasks and groups that this user owns, so we can now return an object, and we'll say the object includes the tasks object and the groups object and also a session property. And the session will have an authenticated property of AUTHENTICATED and an id of user.id. Now back down in our authentication route, we can say state = assembleUserState(user). And since that's an async function, we'll actually just say await assembleUserState. So now our server has authentication logic. In the next clip, we're going to link up our front-end application to this logic to finally add authentication to the whole thing.

Finalizing Integration between Client and Server

In the previous clip, we've created mechanisms for authentication on a client and the server. Now in this clip, we'll integrate them creating the first holistic full-stack application experience that we'll see. So in this clip, we'll create a new mutation that's capable of taking whatever state is returned from the server and applying it to the Redux store. We'll then be able to verify that our full-stack application is functional. Let's jump into it. First things first. In my application, I'm going to go ahead and restart the server. This is so that the new route that we added can take effect. Our application should nearly be ready to work. If we have a look at the userAuthenticationSaga, we can see that it's already written to communicate with this route. I made a small mistake here though. This should be yield axios.post since this is an async method, and we'll add at the bottom of the userAuthenticationSaga console.log ("Authenticated! ", followed by data. And let's actually just put this console.log here at the bottom of this try statement instead. Let's save that. Great! So

when we press the button, we can see Authenticated! with the token and the user states. Very nice. We can see these are the correct tasks and the correct groups. Let's try with some incorrect information, and we see Login incorrect. Wow! So now the server is only sending the state to the user to which it applies. Very cool. Now all we have to do is take this state and apply it to the application. So let's make a new mutation to set the state of the application. We'll go to mutations, and we'll say export const SET_STATE = 'SET_STATE'. And then we'll make one at the bottom, export const setState =, and state will be an empty object by default, and it will return an object just like all the other mutations with a type of SET_STATE in capitals and the state object. Now back in sagas, we will now yield put, and we'll invoke our mutations.setState mutation. So we'll pass in setState with data.state. Now let's go to our index which contains our reducers and write some clauses for the setState. So this is a different sort of reducer. It doesn't act on any one property. In fact, it acts on many properties. So let's add a case for the session. Do it at the top actually for consistency. And we'll say case mutations.SET_STATE, and we'll return the whole userSession, so ...userSession, and then we'll also update the id to be action.state .session .id. And it should be action. Now let's do the same thing for tasks. We'll say case mutations.SET_STATE, and we'll return action.state .tasks. So we're literally taking the very same tasks that were provided and making this the property of our state. Finally, we'll do the same thing for groups. And you may notice this comments. You'll be able to add this functionality to the comments yourself in the challenge task section. We'll add a second argument to the groups handler called action, and we'll switch on action.type. We'll say if the case's mutations.SET_STATE. And you probably are in the rhythm by now, and just say return action.state .groups. And we'll save that. Now that we have this functionality put in, we no longer need these default values, so we'll keep the user session the same uniquely, but we'll change the default value of tasks to an empty array and the default value of comments and groups to an empty array, even users to an empty array. Now let's take a look at the application. Alright! So when we log in with the correct credentials, we're seeing our SET_STATE action occur. In our previous state, we had no tasks, no groups, nothing, but the user session status of AUTHENTICATING. In the next status, we have the correct groups and the correct tasks belonging to the right user. That means all we really have to do now is just redirect to another page, and it'll also fix this session AUTHENTICATING to make it finalized. So back to WebStorm. We now know our saga works, so we can finish the job here in sagas. So we'll yield put(mutations.processAuthenticateUser, passing it a mutations.AUTHENTICATED. Finally, we're going to redirect to the dashboard, thus we will import history from our history file we've written here, and we'll say history.push ('/dashboard'). Save that, and let's see our application. Alright, so here we are at the application ready for testing. If I erase this password and type the wrong password, let's type some random keystrokes here, we

see Login incorrect. No login for us. But if I type the correct password, which I believe in capitals TUPLES, the application loads as expected. And this is Dev's unique experience. There could be any number of users, and they could all see a list of different groups and to-do items. Very, very cool. So now we're seeing a real live full-stack application. As we can see, the data is now persistent, so I can take this Update component snapshots, move it to Done, and rename it. We'll just call it our Rewrite unit tests and press Done. You can see it's moved. And if I refresh and log in again, there it is still in Done, so it's persisted. We've created a full-stack application from scratch. It has been very challenging, but I'm sure you've definitely enjoyed it. In the next clip, we'll wrap up this very exciting module.

Summary

I don't know about you, but I just love it when a good plan comes together, and in this module, everything we did in all the previous modules came together in what I hope was as illuminating for you as it was for me. Let's recap what we've learned here. So we've learned that the fundamental principle is that if you can't restrict access to certain data or routes, then your full-stack application can't do its primary purpose of giving users a unique experience. We learned that we can use a simple Express route to validate any user-provided information, and we've also learned that as a tool for organizing authentication logic on the client route guards make a lot of sense. Here's what's coming up in the next module. In the next module, we'll be deploying. So first we'll be configuring our deployment tool Heroku, and then we'll update our URL for deployments. Once we're done, we'll actually be deploying the application to production, which should be very, very exciting and a great way to finish this course off. It's going to be excellent, so I hope you join us for it.

Deployment Concepts

Introduction to Deployment

In this module, the final applied module of this course, we will tackle deployment, which is the process of putting our application on the World Wide Web. Now it can be hard to think of your application out there in the big world without you to watch over it. But as Charles Dickens said, "The pain of parting is nothing to the joy of meeting again. " In other words, there's no feeling quite like seeing your application up on the web. But let's get on the same page. What is

deployment? So deployment as I've implied is the process of taking an application and putting it on the internet. On the internet, an application can literally be accessed by anyone almost anywhere, and if yours is an e-commerce application, the people who visit it could sign up or even purchase one of your products. There are a number of deployment services. A certain category of deployment services, which we'll call cloud-based services, such as Heroku or AWS, which are very similar, are a good choice for our Node application. As we've discussed a bit, some deployment environments will not support a Node application, the old-fashioned kind that only support, for example, PHP apps. And, of course, when you're ready to take deployment to the next level, it's easy to add features like automatic deployment every time you update your application or even integration with your tests to make sure that an application with failing tests doesn't get pushed to production. So here's the roadmap for what we'll be doing. First, we'll sign up for Heroku and configure a repository. Then we'll update our application to replace certain values that we have with constants that we'll need to use for production. And, finally, as a little bonus, we'll add Bootstrap to make our application presentable, as well as functional. In the next clip, we'll be setting up Heroku.

Configuring Heroku

In this clip, we'll be configuring Heroku. So what is Heroku or Heroku as you might hear it pronounced? Either is fine. So Heroku is a tool for hosting web applications. It's distinctive since it's a bit more sophisticated than other tools and can host various kinds. In this case, we're interested in its ability to host Node applications. The end result is that once your application, your Node application, is up on Heroku, people can access it via the World Wide Web. You might call Heroku a Platform as a Service or a PaaS. There are many such tools, and many are appropriate for deploying Node applications. Heroku is the one I've chosen to use since it's simple and popular. Let's jump into the demo. First, let's sign up for a Heroku account. So here I am at heroku.com, and I'm just going to click the Sign up button. I'll fill out this form, and once the form is complete, my account should be all set up. So once you're signed up or just logged in if you already had a Heroku account, you should see a page like this. This page lists all your repositories. I have quite a few repositories since I've been using Heroku for a long time. But for you, it might be blank. In order to update our application at our command line, we'll need a tool called Heroku CLI, so let's install that. So I've gone here to this URL at the top, devcenter.heroku .com /articles/Heroku-CLI. This might have changed a little bit when you're watching this application, so if you have any trouble at all, just search the web for Heroku CLI. I'm going to click Download and install and then click the appropriate installer for my needs. I'll let that download, and then I'll

run it once it's finished running. So once the Heroku CLI is installed, I can access it in various ways. The way that my Windows PC lets me access it is by, say, opening up git bash with my regular git bash open. So here I can type heroku -v, and you should get a message here telling you the Heroku version. If you did, you've installed Heroku CLI correctly. Next, we're going to have to create a staging area or application on Heroku. So let's go back to the Heroku website. So here at the top of my Heroku page, I'll press New, and I'll create a new app. Now whatever name you give your app is how you'll access it via the World Wide Web. You have to give it a unique name, so it can't match exactly the name I'm going to give my app. I'll just call mine react-express-application, and I'll just give it a number. So if you give yours a number, you can choose such a number that the name isn't taken. Choose the region appropriate for you, and press Create app. Once you do that, you'll be taken to the app's page. Note at the top how Heroku is using similar routing as the kind we developed for our application. Pretty cool. Finally, we need to have an instance of MongoDB for production just like we did for development. So we have to go ahead and provision a MongoDB instance. As most things involving Heroku, it's just point and click. So I'll click Resources, and I will go to this add-on tab that lets me quickly find add-ons. And I'll type in MongoDB. It gives you a number of options. I'm sure with minimal configuration, any would work. But we're going to go ahead and use mLab MongoDB. So click that. It gives us a bunch of options, and we probably want Sandbox - Free. Now I'd like to point out that the free tier of Heroku is very good, but it does tend to load a bit slower. If money isn't much of an object to you, then by all means update your Heroku application and MongoDB instance to the least expensive paid kind, and you'll notice that your application has a lot more room to grow. So let's provision this sandbox, and we can see we have an mLab and MongoDB instance provisioned. And that's it. Everything set up on the Heroku side. In the next clip, we'll be updating our application.

Updating the Application / Deployment

In this clip, we'll be updating the application for production. So what does this entail exactly? So the URL for HTTP request is going to be different if the application's on the World Wide Web. We're not going to use localhost, so we need to update that to something, which makes sense when it's on Heroku. We won't have the luxury of choosing our own port in production. We'll need to use a special variable provided by our deployment tool, which tells us what port we can use. We'll need to write an npm start script. This is going to be used by our Platform as a Service, in other words Heroku. And this will give Heroku instructions on how exactly to kick our application off. Finally, we need to bundle the application into the ES5 since running it directly from ES6 will be quite slow and resource consuming for production. So we're not going to do that. Alright, let's

jump into it. First, we'll add an npm script to bundle all the code into a single ES5 file. Before we proceed, we're going to want to make some changes to package.json. First, let's add a build script, and we'll say webpack --p. The -p is standing for production. We're actually going to build the application locally since this makes it easier to deploy it down the line. Let's give our build script a shot. So I'll say npm run build. And now we should see our finished bundle in this dist directory. This is our application in a form that even a regular browser could read. Of course, it's rather hard for a human to read. Now we want to go to our .gitignore file, and we want to make sure that dist actually isn't there. Let's just only ignore node_modules. Finally, let's go back to server.js and just add one more import. We'll import path from path as we're using the path library here. Finally, we want to change our start script. We no longer want to run Webpack when we start, rather we just want to say npm run server. This will run our server script, which will start server.js, which will serve our newly created dist file. By deploying this way, we've really removed as much complexity from the deployment as possible, but you still may run into an unexpected difficulty due to new package versions, or perhaps discrepancies between your operating system and mine. So do be prepared to do a little bit of personal troubleshooting as the final step of deployment. Next, we're going to update Express to serve our bundled file. So here we have dist/bundle.js. This is the file we want to serve. So let's go to server/server. So, first, here at the top, we've specified port 8888. However, what we'll say here is we're going to use process.env .PORT or 8888. Basically this means that if it's on Heroku, process.env .PORT will be defined to a unique port for our application. If it's not, we'll just default to our basic 8888. Now let's add an if clause here. We'll say if (process.env .NODE_ENV == 'production'), we'll do the following block. We'll say app.use (express.static). Now whatever we put in this URL, this folder, everything inside this folder will be served just like it was the base directory of our application like in, for example, a standard Apache folder. So we'll point it in the direction of our dist file. We'll say path.resolve followed by the __dirname, which is a special variable that resolves to the directory of the file, and we'll pass in the string.. /.. /dist. So that's our dist directory. So we'll also say app.get ('/*'), so any path, and we'll get the usual req, res. And what we'll do is we'll serve the index.html file that we've created manually. So I'll just say res.sendFile (path.resolve (' index.html ')). This will allow us to not use Webpack dev server in production. Now let's close that there. Alright, that's looking good. Now let's update our database configuration. So let's go to connect-db, and we have our convenient URL here at the top. Similar to what we did before, if we put this on Heroku, there'll be a special variable called process.env .MONGODB_URI, which will be the correct URL for production. And what we'll do is we'll say that or the development URL we have, /myorganizer. So that should be all we need for the database. Now we'll need to update our sagas as well. So here're our sagas. And you can see at the top, here's the URL we're using. It's almost like there's a

pattern to updating things for production. So here for the URL, we'll say process.env .NODE_ENV == 'production'. If it is, then we're actually not even going to have a URL. It's just going to be an empty string. So if this application were running at mysite.com, all the HTTP requests would go to mysite.com /tasks for example. So only one domain. And if it's not production, then we can just use the existing localhost with the port following. Finally, we need to make sure the database will initialize in the production environment. As you can see, initialize-db is already set up to do this. It's going to check for a user, and if it doesn't find one, it's going to initialize it with the user. Finally, it's time to deploy the application. So I'll go to my Heroku page and click the Deploy tab. Here if I scroll down, I'll see Deploy using Heroku Git. First, we'll log in. So on the right here is my command. So I can say heroku login, and it will conveniently open up the browser and just let us log in with a single click. That's pretty cool. Now that we're logged in, it's telling us to create a new Git repository, but we already have one. We will need to add the URL. It actually gives you the exact thing you need to type for your application, including its unique name. So I'll just copy this and paste it into my command line, and it should tell me it set my remote as expected. Now let's commit all our files. Just check that. It's all committed for me, but I'll just commit that anyone, git add -A., git commit -m "Final commit before production! ". Now all we need to do is say git push heroku master. Now it should give us a whole bunch of output regarding what we put on production. This can be a tricky step. We're going to work through any errors that we get here, but if you get unique errors on your system, you can troubleshoot them using the instructions on heroku.com or post to the discussion. Now once this finishes, I'll start up the application. It will take quite some time to load as it's still going to build our project with Webpack. And if you pick the free repository, it is sometimes a little bit sluggish with provisioning those resources. And if that all went smoothly, the application should now be visible on heroku.com at the URL of your app. Don't worry if yours doesn't look styled yet. We're going to be adding that in the next clip. So, voilà, our application has been deployed to production. If you ran into a hiccup or two during the production deployment, don't worry. Just whatever error you've received in the Heroku logs, just Google it, and I'm sure you'll find the advice of many people who've faced the unique problem before. Alright, in the next clip, we'll be adding style.

Adding Styling and Presentational Elements

Well, it certainly has been an exciting course. In this clip, we'll be finishing off by adding some style and presentation to our application. It might just be because I have a background in graphic design, but I just can't leave a project finished without making it look sharp first. So let's talk a bit about styling and presentational elements. End users have high expectations of how your

application looks. They've seen thousands of tight applications before, and they also expect yours to look clean and professional. Moreover, good presentation increases the confidence your end user might have in your application. If your application's going to ask them to hand over, for example, their credit card information, it's really good that your application looks professional. Finally, there is a tool which you probably have heard of called Bootstrap that allows us to add styles in a way that is standard. In other words, if another developer were to come along and look at the work we've done, they could easily understand what we were doing and how to make changes if necessary. So, first, we'll install Bootstrap via the content delivery network. So here I can go to bootstrapcdn.com, and I will get a link here. So what I want to do is click this arrow, and I'll get the HTML link and copy that. Now I'll go back to my application and go to index.html, and I'll paste this right at the bottom of the head tag. You'll notice it has this long integrity SHA. Don't worry about that. That's just for security, and the browser will make sure that is or isn't valid. Now if we go to our application on localhost, we can see it already looks a lot different. Just adding Bootstrap makes it look really sharp. It needs a little work though. Now we'll go into our React components and add some Bootstrap styles. I'm just going to be breezing through this, so if you get lost, feel free to refer to the finished files. First, in index.html, we'll add a class to the body called container. And you'll notice that moves our whole application a bit to the middle there. Now let's go through these components one by one and add some styles. Starting with the dashboard, we'll give the main div a className of row, and we'll give each task in the connected TaskList a className of col, as in each one is a column in one big row. This is how it looks so far, not quite right. Let's add some more styles. Let's jump to the TaskList, and we'll give the topmost div a className of card p-2, as in padding 2, m-2, as in margin 2. And we'll give this button a className of btn btn-primary btn-block mt-2, or margin top 2. Excellent. That's looking good. Alright, I'm going to Dashboard and actually just get rid of this Dashboard h2. I'm not liking the way it's looking. Save that. And now let's jump to the login component and add some styles to that. We'll give the top-level div a className of card p-3, padding 3, col-6 as in a width of 6. We'll give this first input a className of form-control, and the second input a className of form-control mt-2, margin top 2. And we'll give this button a className of form-control mt-2 btn btn-primary. Save that. Very nice, so now our application's looking sharp. Here's the login page. And here are the tasks. Let's work on this TaskDetail. That's looking a little rough around the edges. So let's go to the TaskDetail, plenty of styles here to add. I'll add a style to the topmost div of card p-3 col-6 just like before. We'll add a style to this input, className of form-control form-control-lg, or form-control large. We'll add a className of mt to this next div, mt-3 so there's a little space above it. We'll take this TaskCompletion button and add a className to it right here after button. And we'll give it a className of btn btn-primary mt-2. It looks good. We'll give this select a

className of form-control. And, finally, we'll add some styles to this button, className btn btn-primary mt-2. So that's looking pretty great now. Let's just update these task items. They look a little more like buttons or something. Let's go back to the application. So here in tasks, in the div that contains a task's name, we'll just give that a className of card p-2 mt-2. Bingo, and our application is stylish. Look at it go. Wow! I sure am excited about this. Are you excited? It sure is great to see an application come together like this, and it really completes our application. All you need to do now is update the styles in your production and start making the application your own. There are many features that I wanted to add that I just didn't have time to in this course. So in the next module, we're going to discuss what these challenge tasks are and get you started on the rest of your journey to being a full stack expert.

Summary

Well, that certainly was a great module, folks. I sure do love to see a project come together. Let's review what we've learned. So we've learned that web applications must be deployed before an end user can access them. And the environment that we deploy them to is often referred to as production. We've learned that we can use Heroku or a number of different Platforms as a Service to deploy Node applications. We learned that generally speaking, we use an npm start script to begin listening to the port with Express that our Platform as a Service has specified. And we've also learned that URLs are different for production than from development, but we can use conditional statements to have an application that works in either setting. Coming up in the next module, we'll go through a high-level summary of all the ideas and topics that we've learned. We'll review a few challenge tasks which you can complete to help increase your knowledge.

Conclusion

Course Summary

Hello, and welcome to the summary module of this course. This has been a truly excellent and action-packed course, and in this module, we're going to slow it down a bit. There's no coding. We're just going to review what we learned and look at some next steps. So here's what you can expect from this module. First, we'll review the high-level concepts that underscore everything we need to know about full stack applications. Then we'll look at some challenge tasks, which are some fun and innovative features you can add to the application to increase your understanding

and ability to independently add features to full stack application. And, finally, we'll look at the next steps that you can take to continue learning after completing this course and the challenge tasks if you choose to do them. In the next clip, we'll look at the high concept review.

High-level Concepts Review

Let's review some ideas we've learned during this course. It is, of course, very difficult to summarize everything we covered in a full-length course in just a few minutes, but hopefully these ideas will help crystallize what we've learned. So we learned that a full stack application is an application that's composed of two parts, a client application, which is what the end user sees, and the server, the server being the component that handles secret operations and data persistence. We learned that the goals of a full stack application as opposed to just a regular single page application include security, protecting a user's data from other users, and data persistence, making sure the user's data is still there when they come back. We learned that React and Redux can be used to create a fast and easy-to-maintain client component. This is true not just of our app but almost any style of web application. We also learned and applied some techniques with Express and Node observing that they're pretty useful for creating a tool which can respond to REST requests. We learned that we can combine React, Redux, Node, and Express to create authentication, which allows each user to have their own personalized and private experience. Finally, we learned that it's easy to deploy full stack Node applications by using a Platform as a Service.

Challenge Tasks

In this clip, I'll set out the challenge tasks that I provided for you to increase your learning if you so desire. So what are these challenge tasks? They're basically assignments, kind of like homework, but more fun. They're optional. You don't have to do them, but they're also entertaining. Personally as I completed all these challenge tasks, I had a blast. The challenge tasks reinforce ideas that you learned in this course, for example, about REST APIs, React components, and managing state with Redux. You can complete them at your own pace. If you're a very skilled user, you get all the challenge tasks done in an afternoon. If you're more of a beginner, the more challenging tasks could take several days to complete. But as long as you have the spare time to do so, you'll find that completing these tasks really does increase your knowledge a lot. And the solutions are all available on the GitHub at the URL indicated here shortly after the publication of this course. So let's get into it. Challenge task #1, easy difficulty, a connected username

component. This is really simple. You are to create a React component which takes a single prop, ID. The prop is to match the ID to a user and display that user's name. So in our course, dev had the user ID U1. So providing this as the prop to the component would make the component say dev. Now there're two ways that you can do this. You can choose to rewrite the user state assembly so that any usernames that might be needed can be loaded initially, or you can have the component be more dynamic, and upon being created to make a REST request to grab the information for the relevant user at that time. The second option is more difficult. They both get the job done, and I'll leave the rest to you. Challenge task #2, add comments, difficulty medium. You saw this feature at the beginning when we looked at the finished application. So comments are a separate collection, a collection of messages that users can attach to tasks. Here're the specifications. Each comment when it's stored in your database must remember who created it, the text that's in it, and which task it applies to. Here's what you'll have to do. You'll have to create a new saga that handles common creation. You'll have to create a new reducer that reduces changes to comments. And you'll have to add a comment creation route to your server, which will allow your server to create new comments in the database. And also, of course, you'll need an attractive form that allows the user to interact with this feature. Challenge task #3, sign-up page, difficulty medium. So the sign-up page is an extra component and route that you'll be able to access through a link on the login page. Just like a normal account, the link will say, if you don't have an account, sign up here. This sign-up page will contain a form which creates a new user on the server. You'll need to add a user creation saga and a user creation route, for example, user/new. You'll be able to adapt much of the functionality from the login page. But after creating this component yourself, the knowledge will definitely persist in your mind. Finally, task #4, secure the application, difficulty hard. So throughout the whole application, you are to add logic to prevent a user from getting data without the right credentials. Most importantly, you'll need to update the Express routes and see if there's a gatekeeper of the information. A valid auth token will be required to access any data. We created auth tokens in the main application. You just have to put them to use. You'll need to create some permission tables to decide which tokens are valid for accessing what information. The ultimately updated routes will return a 403 or similar response on invalid credentials instead of handing over user information that they probably shouldn't. This is hard since there are many holes to fill, and you'll have to be thorough, as well as possess a very real understanding of security. So those are the tasks, and be sure to check out the GitHub page for the conclusions and explanations at your own pace. In the next clip, we'll look into next steps.

Next Steps

So you've reached the end of this course. What now? Here are some next steps that I recommend, which you can take before or after doing the challenge tasks if you choose to tackle any of them. First, you can increase your understanding of web security. Not only is this a fun and interesting task, but web security experts are among the best compensated professionals in the internet field. So here I am at Pluralsight.com. And the course I'd like to recommend for security is Pluralsight LIVE 2018: Get Your Geek On, Security Edition. This is a really cool course because this is a bunch of different talks by different experts who are all complete pros. I would need to study for decades, literally decades to have the expertise on security of even one of these individuals. The URL for this course is here at the top. And, of course, on the topic of security, I can't recommend the courses by Troy Hunt enough as he is considered not just one of Pluralsight's but one of the world's leading security experts, and he has many great courses right here on Pluralsight. So if you do want to take security a step farther and as I've mentioned, it is an extremely lucrative calling, do check out the courses by Troy Hunt. Next, you'll want to improve your understanding of React components. The methodology which I use React components in this course, which in addition to being my preferred method and the method I do used to make real applications, is also the simplest and most intuitive way to create components. But there's actually with no exaggeration a half dozen different ways to initialize React components. And the course I'll recommend for React components will be Creating Reusable React Components by Cory House. Cory House is truly one of the leading experts on JavaScript and React. Though I consider myself an expert, this is a person whose input into any JavaScript discussion I take very seriously. This course is a very long, detailed, and thorough exploration of React components. Weighing in at a bit longer than this course and talking only about components, you know that this is going to give you a really thorough understanding of those tools. Basically what you learned in this course will allow you to create React components to get the job done. But if you complete Cory House's course on creating reusable React components, you could, for example, go into any workplace, be asked to, for example, fix a React component that's problematic, and have no trouble identifying what the component is and where the problem is. This is great since skilled React developers are extremely employable. Finally, I recommend that you master Redux-based state management. Redux is a very popular, simple, and useful tool that's used by many large applications. Redux in my opinion is so effective at managing your state that once you master using it, you're really free to focus on the components and business logic of your application. The course I'll recommend here is one of my own courses, Mastering Flux and Redux. In this course, we start by learning about Flux, which is the precursor to Redux. Then we jump

right in and learn about all of Redux's excellent features. It's an easy-to-understand and fun course that will arm you with tools for building almost any kind of application that relies on having a state. In the next clip, we'll wrap the course up.

Thank you!

This is just a quick clip to say thank you. You know, very few people who start a course actually make it all the way to the end. If you're looking at this slide, you did it, or you just skipped to the end, in which case go back and watch the course. However, if you really did just complete this course, I'd like to congratulate you. You are really taking your future into your own hands and learning skills that are valuable not just for a great career, but for keeping a mind strong and able for an entire lifetime. Finally, I'd just like to say that for me the greatest reward is truly knowing that I've helped someone on their journey towards being a skilled web developer and computer programmer. Please, if you enjoyed the course, do tweet at me. I read them all. And don't forget to give this course---and don't forget to provide me with feedback. If you really enjoyed the course, please do give it a five-star rating. And if you had any issues at all, do share them with me in the discussion so that I can improve this course and future ones. This has been Daniel Stern, the Code Whisperer, wishing you a great day.

Course author

Daniel Stern

Daniel Stern is a coder, web developer and programming enthusiast from Toronto, Ontario, where he works as a freelance developer and designer. Daniel has been working with web technologies since...

Course info

Level	Advanced
Rating	★★★★★ (81)
My rating	★★★★★
Duration	3h 13m

Released 7 Feb 2019

Share course