Building Scalable React Apps

by Hendrik Swanepoel

Start Course

Bookmark Add to Channel Download Course

Table of contents Description Transcript Exercise files Discussion Related

Course Overview

Course Overview

Hi everyone. My name is Hendrik Swanepoel, and welcome to my course, Building Scalable React Apps. I'm a freelance Full Stack Developer from Cape Town, South Africa where I work in the FinTech industry on complex cross-platform products. React is very useful, no doubt about it, but setting up a project around it, and constantly tweaking a custom stack can be detrimental to any project, especially when you use it within a team. With the stack you'll learn in this course you'll be able to focus on building complex apps instead of worrying about your foundations all the time. In this course you'll learn how to set up the very popular React-boilerplate stack on your machine. You'll learn the basics of how to work with redux-saga, and you'll use reselect to compute values on top of a redux store. By the end of this course you'll be in a good position to deliver a highly complex and large React app without even having to go near a Webpack configuration file. Before beginning this course you should be familiar with React, a bit of Redux, and JavaScript. I hope you'll join me on this journey to learn to scale React apps with the Building Scalable React Apps course at Pluralsight.

Getting Started

Introduction

Welcome to this course on Scaling React apps. What do I mean with scaling? Well firstly, it means that you can easily add new features to your app without the need to go and change the stack. It also means that you can add new team members onto your team, and they'll know what to go and do. With the proper scalable stack it will eliminate a lot of debating within your team on what technologies to go and use. You want to focus on features, and not be caught in constant debates on technology. Talking about teams, it will also be easier for new team members to join and start working on the stack rolling out features. If you're serious about your application it means that it's probably going to be more than one person working on it, so you need a stack to support that, and lastly, it should also be quick and easy to deploy your app. Let's go and have a look at everything that's involved in this stack. I would just like to extend a special thanks to my colleague and friend, Stewart Scott, who helped me in the preparation for this course. I ran a lot of ideas around him, and he gave very good feedback. He also helped to test out the material. Thanks a lot Stewart.

A Quick Look at What's in the Stack

This is what we'll be building in this course. As you can see, it's a material-like app. I couldn't include any material component libraries. I wanted to keep the course as simple and self-contained as possible. It's using BRCSS, but it's still a pretty cool look for an app. Basically, this app's functionality consists of adding new links to this link library, and each link is associated to what I call a topic, which is sort of like a category. When we click on this hamburger menu over here you can see that we get presented with a list of topics in the app. When I select a different topic over here the route updates the URL, and we see a different topic, Apps, with its associated links. You'll see when we click on this log in link that we go to a different route, and we get presented with this component. We've also got validation on here, so if we don't provide an email here and press the Log In button we get a nice validation error here. So let me type up a valid email now, and press Log In again. Cool, so we go back to the previous location, and the email is shown in the app bar. You might have noticed that in the redux dev tools on the right we've got actions firing all the time, which is quite important because everything flows through redux in this stack. We can also add new links, so when I click on this very subtle button over here we get a form that allows us to add a link. As you can see, this form is displayed inside of a dialog, and we've still got the context in the background, the list of links, and the topic. We've also got the validation on here. Let me go and add some valid information in here, so that I can show you what it looks like when it works. Click on Add again, and we get redirected to the topic, and this time the link's displaying in here, and this link is actually posted to the server, so when we do a hard reload it will load up again. So that's our sample app in a nutshell. It's very simple, but believe you me, it's going to teach you all the concepts.

Getting the Stack up and Running on Your Machine

To see what's involved in this stack head on over to this URL, github. com/hendrikswan/react-boilerplate. This is a fork that I created from the react boilerplate repo. I've made some changes to this code to go ahead and make the course a bit easier to follow along with, so it's very important that you work from this repo itself. That way you're also guaranteed that you're working on exactly the same version that I used during the recording of this course. Down here you'll find that there's a lot of links to documentation, and we'll go and have a look at some of them now. Before we do that I just quickly want to show you the original repo for react-boilerplate. You can head back here to go and have a look at the newer versions and updated documentation, but let's close that for now, and go back to my fork. So what's in react-boilerplate? What are we going to use? Down here you can see that on the JavaScript it's got a lot of libraries that it depends on. This course will mostly be focused on this part of react-boilerplate, the JavaScript part. Let's start by looking at ImmutableJS. As you can see, in react-boilerplate's documentation it gives you an explanation of immutable, and it also shows you how it's used inside of the stack. That's pretty cool. Let's click on the official documentation link. So basically what ImmutableJS does is exactly what it says in its name. That means that with every change that you make to state it will result in a whole new copy of state, so you never overwrite state. It always makes a copy when you go and make a change. Okay, back to the react-boilerplate read me. Next, we're going to have a look at reselect. Again, react-boilerplate documentation gives us a nice overview of what reselect does, and it also shows you how it's being used inside of the stack. Let's click on the official documentation again. So basically, reselect is a way to go ahead and compute and derive data from your underlying state. That means that you don't need to go and store every single calculation on state itself. You can go and compute stuff on top of it, and what's also cool is that these functions that you create will be chainable, and composable. We'll see a lot of this during this course. Okay, let's go back to the react-boilerplate read me. Next up, let's have a look at redux-saga. Again, this documentation doesn't only take you through redux-saga itself, but also how it's applied within the stack. Let's click through on the official documentation. Redux-saga is redux middleware. It's got access to all the actions being dispatched, and it can intercept it to execute logic. It also comes with a bunch of functions that you can use whenever you want a side effect to happen, which makes it more declarative and more testable. A lot of this course is going to be centered around redux-saga. Let's go back to the react-boilerplate documentation. Next up, we've got internationalization. I've made the decision not to focus a lot of this course on internationalization, but it's very easy to get going with it if you want to go do it. Basically, you just define all your messages for the different parts of your component inside of a file, and then you go and use this FormattedMessage component here that you import from React international, and point it to the correct message. As I mentioned, we're not going to use internationalization during this course, but I thought it's important to at least point it out to you. Okay, let's go back and have a look at routing. React-boilerplate routing. Here's this react-router and react-router-redux. This is cool because react-router-redux makes all your routing happen through actions that goes over the dispatcher, so it fits very nicely into our stack. We're also going to touch on routing quite a lot during this course. Although this course touches mostly on JavaScript, we're also going to do a fair amount of CSS. The most important thing in this stack to be aware of around CSS is that we're going to use CSS modules. Basically, it allows you to define your CSS using classes as usual, then you go and reference those class names inside of your component in this manner, and it will automatically be namespaced, meaning it won't clash with other classes with the same name defined on other components. The stack also relies on the post CSS preprocessor. This gives us a lot of benefits in terms of auto-prefixing, and stuff like that. We're not going to go too deep into this subject in this course, but we are going to use it here and there. So that's the stack in a nutshell. Let's move on to getting this stack on your machine.

A Quick Look at Some of the Nifty Features in the Stack

So to get the code up and running on your machine we drop into the terminal, and we do a git clone https://github. com/hendrikswan/react-boilerplate, and we can optionally specify a folder to clone it into. After it's cloned we see it enter our folder. Let me quickly do a git log here. As you can see, we've got a bunch of commits in this repo, and now let me do an npm run. I just want to show you all the commands that we've got at our disposal, so quite a bit eh? Okay, what we want to run first is npm run setup, so basically this installs all the dependencies, and sets up the app for you. It also cleans out the old repository. This installing dependencies part really does take a long time. On mine I started playing a game on my iPad while it was installing, so if it takes a long while on your machine don't worry. It's not broken. Give it a few minutes. I'm just going to fast forward this. Okay, so it's finished that setup command. Now I'm going to do another git log, and as you can see, we've just got one commit in here. So it removed the old git repo, and created a new one, took the code, the startup code, and put it inside of that git repo. Before we start it up I just want to go and open up this code in Atom. Okay, my Atom editor has opened up in the background. Now we can run npm start. This kicks off a little web server with auto and ______ loading, and ______wait back listening to changes on files, recompiling them all the way. Let's open this up in the browser. We do that by going to localhost on port 3000, and let me just show you the demo example components. So let's start in my GitHub using @hendrikswan, and it goes and talks to the GitHub API, and shows all my courses down here. If I open up the dev tools, and select the Redux dev tools you can see that it's firing actions here, and as you can see, the dev tools work out of the box with this boilerplate project, which is pretty cool.

Inspecting Different Types of Errors

Okay, let's take a quick ______ at the code, and the features that we've got in this boilerplate repo. As you can see in this app. js file, it does a lot for us out of the box. I can't tell you how many times I've had to go and do all this stuff from scratch, and it's such a relief to be able to use the React boilerplate, and it takes care of it for me. Most of our work will happen here in the components and containers folders. As you can see, there's a bunch of containers and components in here already, and we're going to get rid of them later on when we want to start plugging in our code. Let's open up this app container component, and go to its index file. Now let me go and add some code in here. I'm adding a new div, and I'm putting it in the wrong indentation. It should be indented a bit more. Now when I save it out you can see that we get a linter error here, so that's another benefit of this boilerplate project, it's got linting already configured, yet another thing that you don't need to go and set up, and it's not just superficial linting rules. Let me scroll down here and remove this propType over here. When I save that out you can see that it's complaining about this reference to this prop that doesn't have an associated propType, so the linting setup is mature enough to support proper React development. I'm going to leave that card broken, and pop back to the command line. Now in here you can see when I run npm run lint it takes a little while, and it gives us an error. Let me just scroll up a bit, so that we can see that error. There. You can see it's comparing about the children that's missing in props validation. So you can easily integrate this with your bold server. Okay, back on the code side, let's just go and undo this piece of code that I've just changed, just go and make it happy again. As you can see in this component directory, we've also got a styles. css file. I want to show you something interesting. I've mentioned earlier that React boilerplate relies on CSS modules, but I like it so much I wanted to point it out to you. Let's go back to the browser and inspect this element. See here it's got a class of logo__app-containers-App-styles, that long thing, so that's basically what it does to go and ensure that CSS is properly namespaced. If I pop back to the CSS you can see that we don't have complicated class names in here. It's quite simple. And if I pop over to the component again you can see that we just referenced these class names using className=, and then the styles that we imported, and the className, and it will go and create that unique string for you automatically, so now you can worry a bit less about clashes in your CSS. Another thing that I quickly want to point out that's pretty nifty is the internationalization. If we head on over to the index file in HomePage, and scroll down a bit, you can see that we're using these FormattedMessage component instances that we're getting form React international, and this just takes the messages that we've defined in the messages file, so for each message that we want to display you go and define a message like this with an ID, and then just put it into your component, and so you can go and support multiple languages, and it's very elegant. So that's just a few of the nifty features that we've got in this stack. Let's go and have a look at error handling.

What We'll Build in This Course

I'm inside of the index file inside of the app component, and what I'm going to do in here is input something that doesn't exist. So import bla from nowhere, and I just need to remove this character over here. Now in the browser you can see that it gives us a really nice error, so this is pretty cool. When it runs into compilation issues you actually see it in the browser. This is cool because when you make changes you don't just want to go and see a blank page with no indication of what went wrong when you get a compilation error. You don't need to drop into the terminal unnecessarily. You can just work in your code and your browser, and you'll see most of the issues. So that was a compilation error. Now I want to go and show you what happens in this stack if you get a runtime error? So again, inside of the index file in the app components folder, in this function I'm going to cause a runtime error, so I'm assigning a new constant, bla, to a variable that doesn't exist, and I'm calling toString on it, so obviously this should result in an undefined error. Now let's go to the browser and reload that. As you can see here in the console, we're getting an error here. The error message itself seems okay, but let's expand it and click on this link here, and what we expect to see is the code that caused the error, but we don't. We see this weird message here and no code around it. Luckily that's easy to fix. This is one of the defaults in the react-boilerplate project that I don't quite agree with. Let me show you how to go and override it. We're going to the internals folder, and then find webpack, and then we look for the webpack. dev. babel file. Then we scroll down, down, down, down, I think it's on line 65. Here it's got the devtool configuration for webpack. Currently, it uses this cheap-module-eval-source-map, which is great for performance, but not so great for accuracy on your stack traces, so we change that to source-map instead, and this is one of the few things that we'll be doing in this course that will require a restart of the webpack process. So we go into the command line, and stop this, and start it up again. Now in the browser we go and refresh this page, and we get the error again, but this time when I click on this link we get to see the actual code that caused the error, which is way better. Let's just go back to the code, and go and remove this line. So that last change on our webpack dev config is very important for accurate stack traces and source maps. Don't skip over this part. You'll save yourself a lot of hassle if you go and do this change now before you tackle the rest of this course.

Ramping up to Start Coding

I did some other overrides on top of the default React boilerplate stack to make it easier to follow along in this course. Now you don't need to go and apply it. It was part of the repo that you cloned from my GitHub profile, but I'd like to point it out to you nonetheless. If you go to the server folder and expand the middlewares folder you'll find an api. js file in here. This contains some Express. js server-side code that you can call into from the client for this course. I don't believe a course for a front end developer like yourself is useful without interaction with a server, so that's why I coded up this file, so that we've got something that we can call from the front end. Don't worry too much at the moment about what's in this file. We'll cover all these calls as we need throughout the course, but just as an example, let's go back to the browser and change this URL to /api/topics, and as you can see, we get a list of what I call topics back from the server. Topics is something that we'll use in our sample app as basically a list of categories that you can click on to go and filter links added to our app. Let's go and have a look at something else that I changed on top of the default stack. If you go into the app folder and expand the utils folder you'll see an asyncInjectors. js file. This file is part of React boilerplate, but I had to go and make one change in here. If you scroll down and find the injectAsyncSagas function you'll see that from this I return a function called injectSagas, and this takes a name parameter that I use to go and check whether this saga has already been injected, and if it has it just returns prematurely. The sagas get injected on routes. Let me go and show you by opening up the routes file. Here you can see that per route we go and import files necessary for that route to function, in particular, the component for that route, but also sagas and reducers that are required to make the route work. Then later on when it's imported it works with a promise, and we need to go and call the inject functions. This injectSagas one also specify an identifier when the sagas get injected. So that's the function that I changed. By default this doesn't take an identifier, and it will inject it again and again when the root is triggered, and you have to go and take care of that complexity inside of your saga logic. I decided to make it a bit easier, and keep all sagas in our course as singletons, meaning they get injected only once. Let's get this code cleaned up, so that we're ready to go and add our code onto it. We do this by going over to the command line and running another command. This time npm run clean. This one's a quick one, and now when I do a git log you can see that it added another commit in here, remove default example. If we go back to the code you can see that now the components directory is empty, and we've got very few container components left. I would advise you to go and find your process that's currently running the web server, and go and kill it, and run the command again to start it up, npm start. Go back to the browser, and this time when we load it up you can see that all the sample components are gone, and we've just got this place, all the components here. Great, so we're set up. We're ready to start building our app. That's it for this module. If you followed along you should now have React boilerplate on your machine, and it should have been cloned from my repo. That part is very important. I made some changes in there that we're going to rely on during this course. Let's get going.

An Introduction to Building Components with react-boilerplate

Introduction

In this module we're going to take a look at what's involved in building into components using react-boilerplate. React-boilerplate strings together a bunch of libraries and conventions, so what you'll learn during this module doesn't only apply to building components with the react-boilerplate. You'll be able to use all these concepts on any of your React projects. Let's do a quick overview of what you'll learn in the rest of this module. Here's an ES6 component. It's dependent on its props and on its internal state. When a component that uses this component changes the props on it, it will pass through the component, and it will rerender the component. Similarly, when this component calls setState on itself it will also go and call render with the latest state, resulting in a rerendered component. The important thing to keep in mind with ES6 components is that two things can go and cause a rerender, updated props and its internal state being updated. Now let's have a look at the stateless functional component. This is much, much simpler, and that's why we prefer using these type of components. It's only dependent on its props that it gets through its function arguments, so when the props update it results in updated JSX, as simple as that. What we'll be doing in this module, and throughout the rest of the course, is wrap stateless functional components, and sometimes ES6 classes, in container components. We want to keep our presentation components, meaning the components that receive props, and then provide updated JSX as simple as possible, and ignorant of the redux world, so we'll be wrapping them in container components. Container components have access to the redux world, meaning redux state and actions, so they will pass state to the simple presentation components through as props, and they'll also make actions available to our simple presentation components. These components can then trigger actions and render properly. The best part about that is that the simple components are totally unaware of the redux world, making them more testable and more portable to other architectures.

Using Scaffolding to Generate Components

The first thing we want to do is generate a Navigation component. We do this on the command line using scaffolding. We run npm run generate component, and we select the option for a stateless function. We want this component to be called Navigation, and yes, we want it to have styling, and no, we don't want internationalization. Now let's generate the NavigationContainer. We call generate, but this time with the option for container. We want it to be called NavigationContainer. We also want to generate the selectors, constants, and reducers for it, and we select yes for generating sagas for this component. We don't want internationalization for this component. Now that that's generated let's go and have a look at the code. This is the main file for the NavigationContainer component. As you can see, it uses react-redux to generate a container component. We've also got stuff like selectors being pulled in, and it gets its mapStateToProps from this selectors file. We'll be working with that a lot, so don't worry about that too much for now, and it does the mapDispatchToProps in here. We're going to use this container to wrap our navigation component. We've also got some default constants in this constants file, some default action creators here in the actions file, and also a reducer here in the reducer. js file. Here in sagas we've got something that might be a bit new to you. It's a way to intercept actions being fired, and to go and attach logic to that. A lot of people find it difficult to figure out where to go and put async logic in React and redux applications. This is where sagas come in. It's a nice way to hook into actions being fired, and do your logic in that way, and to compose more complex workflows. We'll be using it a lot during the rest of this course. Lastly, we've got the selectors file. This uses reselect. It's a way to go and use these functions to go and compute data on demand when their dependencies change to go and pass data onto your container components. Again, we'll be delving into this concept a lot during the rest of the course. Our Navigation component is much much simpler than the container component. It's just a normal component, a stateless function component, and it's importing a style file where we can go and define our style definitions.

Getting a Container Component to Render on a Route

Let's go and get these components rendered into our app now. To do this we'll head on over to the HomePage container component. In here we remove all the code that's not being used at the moment, basically the internationalization code that's baked in by default. Then we import the Navigation component from its relative location to this container components file, and then lastly, we just return this Navigation component from render. We also need to ensure that we can see something when we test this out, and to do that we head on over to the Navigation component, and add a bit of texture inside of this function. Just say, this is the Navigation component, and now when we go and check this out in the browser you can see that our auto reloading's already kicked in, and we can see our awesome, very beautiful Navigation component displaying here, but we actually want to use the NavigationContainer don't we? So let's go and change the HomePage to go and do that. We change this import statement to import the NavigationContainer instead of the Navigation component. We just need to ensure also that we get it from the containers folder, and now we can go and use this NavigationContainer down here in the render function. As you can see, the NavigationContainer currently just returns an empty div. Let's adapt this to return the Navigation component instead. That's the whole purpose of this NavigationContainer component is to wrap our Navigation component, and abstract our Navigation component from the redux world. Okay, so we ensure that we import the Navigation component from its relative location in the components folder, and now we can go and use it down here in the render function. We add an instance of Navigation, and we go and use this syntax to go and say that we want all the props from the NavigationContainer to be passed on to this Navigation instance, so I'm going to reload this browser, and we're going to get an error. I'll explain to you why the error happens, and I'll show you how to fix them, and there you go. As you can see, it says, cannot read property toJS of undefined. That error is due to this line over here, and so when we go and call toJS on the immutable JS object that we've got on state, and this is because the selector kicks in, and it's trying to get the navigation container state from the state store, and it doesn't exist. Why is that? Well, it's because react-boilerplate uses the concept of pages, and it builds up webpacks that contains only the code necessary to go and run that page, and we still need to go ahead and ensure that all the code required by the NavigationContainer is injected for the HomePage route. Let's go ahead and do that now. This is the route for our app. As you can see, it imports the HomePage component in here, but that's the only thing that's imported at the moment. It also knows about our NavigationContainer because the HomePage itself imports that, but to make the NavigationContainer work we need to go and ensure that everything that it depends on is also imported here. The first thing we need to import is its reducer, so we use the same syntax to import the reducer from its relative location, and similarly, we also import the NavigationContainer's sagas file from its relative location. Now these imports you can see is done by using promises, by using promise. all. So this importModules. then can now expect the results of these import statements to come through here, so we add the reducer and sagas variable into this array, and now I can go and use that. We can go and call injectReducer, which gets imported up here, and we specify a name for this reducer, and we say that we want the reducer. default to be imported. This just means that the module that was imported as a result of system. import, go and use the default export to go and inject as the reducer, and we'd do the same for the sagas. We specify the name, and we specify that we want the default for this module to be imported as the sagas. Now that we go and reload it in the browser you can see that the errors have gone away, and our Navigation component is displaying again, but this time as part of the NavigationContainer component.

Getting Data from the Store on to Your Component

Let's go and make this whole NavigationContainer component useful. Its purpose is to go and provide data to our component from the store or more accurately, unidirectional data flow to our Navigation component. The first thing we want to go and do is to go and add some data into our reducer. We're using this initialState assignment here, and I'm pasting in some topics that I've got available on the clipboard. Now the topics is an array of objects that we're going to use to populate our Navigation component, and later on when you go and select one of these topics it will go and show you a list of links associated to the selected topic, so as you can see here, react-boilerplate makes use of ImmutableJS, so this is saying that we want the state to be an immutable object that contains these topics on it. Now that we've populated some default data onto the state for the NavigationContainer component we can actually expect it to be passed onto the Navigation component, so in the Navigation component let's say in here that we want to get a topics prop passed to its function, and now that we're destructuring the topics from the props on the Navigation component, let's go and add some propTypes for the topics variable. We specify that topics is of type array, and we want this array to be populated with a shape, and the shape has got a name property, and this name property is of type string, and it's also got a description property, and the description property is also of type string. Both these props are required, and the topics array is required. So now that we've wired up our topics onto our Navigation component we can go and use it here in the function itself. Let's just add a simple statement that shows us how many topics we've got inside of the nav component. Now when we reload the browser you can see that it all worked. Our statement is working here, and it's saying that we've got three topics in the nav component, so the initial data that we've added onto our reducer flowed all the way through our NavigationContainer component onto our Navigation component, and in the end we got it in the Navigation component as a simple prop making our component very portable and simple. That's pretty cool.

Loading Data from the Server with Redux-saga

Introduction

One of the core technologies in react-boilerplate is redux-saga. It's a nifty piece of technology that allows you to intersect redux actions and execute logic, and it also has a nice way to isolate your side effects, making your code very testable, and more declarative. Let's have a look at how we'll be using it in our code base. Currently, we have the NavigationContainer, and what we want to do is fetch some data or topics, to be precise, from the server instead of hardcoding it inside of our reducer as default state, and to go ahead and kick off this whole process of getting data we're going to code up some Action Creators. In particular, we're going to code up requestTopics, and requestTopicsSucceeded, so the NavigationContainer will use the requestTopics Action Creator and dispatch the resulting action. The action looks something like this. With redux when this action is dispatched it goes to the reducer, and a reducer will then make changes to the state, actually go and mutate the state to go and reflect what the action should mean on the state store, and finally, when your data updates your container receives the latest data through its prompts. Now as you've seen in the previous module, in our application we use selector functions to go and pull the relevant bits from state, and perhaps compute some data above that, and then pass it onto the container component through props. In this picture where do we go and get the data from the server? This is where we're going to start using redux-saga. Redux-saga is redux middleware, so it sits somewhere between where the action gets dispatched, and where it reaches the reducer. Now the saga can intercept these actions, and it can go and execute logic when that occurs. In our scenario we want to go and fetch some data from the server, take the resulting data, in this case, a bunch of topics, and then how does it get that data on state? It uses action creators again, so in our saga we're going to use the requestTopicsSucceeded action creator, pass the topics to that, and that will result in a new action that looks something like this. Because we used a different action creator we'll have a different action type, requestTopicsSucceeded, and this action will have the topics property on it with some data in it. The action will pass through the reducer, and the reducer will mutate the state, essentially storing the topics on state, which then gets passed through on these props to the NavigationContainer, so that's what we're going to do in this module. We're going to do some basic redux stuff, like constants, action creators, reducers, all that sort of stuff, but what we're going to do different is we're going to use redux-saga to hook into when actions get fired, so that we can go and talk to the server, and go and mutate our state as a result. What I like about redux-saga is that it sits on the side, so you don't need to go and weave it into your app and act as middleware. That's very cool. Alright, let's go and build this in code now.

Defining Action Creators

This is the URL that we can call to get a list of topics from our server-side environment. We want to use these topics instead of the hard coded ones, and bind our Navigation component to them. In our NavigationContainer component we want to look into when this component is loaded up for the first time, so that we can go and trigger an action that will result in a call being made to the server to go and get our list of topics. To do that, though, we need to go and create all the stuff around that, like constants, and actions, and so forth. Let's start. In our constants file we first want to remove this default constant over here, and now we're going to define three new constants. The first one's going to be REQUEST_TOPICS, and we're using the same convention that react-boilerplate uses by default, and this one is app/NavigationContainer/REQUESTT_TOPICS, so it's nice that our constant values are namespaced. The next one is REQUEST_TOPICS_SUCCEEDED, and we use the same convention again, and define a string that's namespaced for this constant, and now we're moving onto REQUEST_TOPICS_FAILED, and again, we define a namespace string for this one. So we've ended up with three constants. We need three constants because what we want to do is have three actions. The one is an action to kick off the logic to go and make the call to the server, REQUEST_TOPICS. The other one is for action that we want to fire when we get the data back to go and ensure that the topics get stored on state. The last one is the action that we want to fire if something goes wrong with the call, so that we can also record that on state if we want to. Now we can head over to the actions file where we can go and create an action creator for each one of these constants that we just defined. The first thing that I want to do in here is get rid of this defaultAction creator created when we generated the NavigationContainer, and then we import the newly created constants, REQUEST_TOPICS, REQUEST_TOPICS_SUCCEEDED, and REQUEST_TOPICS_FAILED. Firstly, we create a function called REQUEST_TOPICS, so this is the action creator for REQUEST_TOPICS, and from here we just return an object with a type attribute set to the constant, so this is the action that will be fired, and this is where we'll hook into for sagas, and in our reducer. Next, let's create the REQUEST_TOPICS_SUCCEEDED action creator. This one takes a topics parameter, so that we can store the topics value on the action itself, and we return an object, again, with the type set to the correct constant, and we also store the topics variable on this action. This will allow us, obviously, to go ahead and store the topics on state, and then lastly, let's define a function called REQUEST_TOPICS_FAILED. This action creator takes a message parameter. This will be the message that we want to display to the user when something goes wrong. From this function we return an action with the type of REQUEST_TOPICS_FAILED, and we also made sure to store the message attribute on this action. Now because we deleted the default action that was generated by react-boilerplate we need to head on over to our reducer, and remove all references to it from here, so we remove it from the import, and we remove it from this switch statement down here.

Dispatching an Action from a Container Component

Now we want to trigger the REQUEST_TOPICS action from the NavigationContainer when it loads. First thing we do is we import the REQUEST_TOPICS action creator we just created from the actions file, and because we're going to pass this REQUEST_TOPICS through on props to this component we create propType rules, and we specify that we want requestTopics prop, and that it's of a type of function, and it's required. Now we can go and add a componentWillMount function onto this NavigationContainer. This, as you might know, is the function that will get called when the component is mounted the first time, and in here we'll go and call the requestTopics function that we expect on props, but how does this NavigationContainer component get this on its props? Well, it won't at the moment. We need to go and wire it up. To do that we scroll down to the bottom of this file, and you can see that this mapDispatchToProps function, at the moment, just returns dispatch, which isn't a great idea. This makes dispatch available to be called for any arbitrary action. By default, if you generate a container component with react-boilerplate it adds this mapDispatchToProps, and it passes dispatch onto the container component. I think that's a bad idea because we want to pass very explicit functions onto our container component and map them in here, so let's remove this dispatch. We want a prop called requestTopics on our container component, so let's add that, and this is a function, and when this function gets called we dispatch the requestTopics action, which we get by calling the requestTopics action creator. Now why go through all this trouble of mapping it on here if we're in the same file? Well, this function, requestTopics, actually needs the dispatch available on scope, and we've got that available in the mapDispatchToProps function. We won't have that available on the component. Now when we go back to the browser and reload it looks the same as earlier, but with one difference. Let me open up the redux dev tools over here. As you can see, the REQUEST_TOPICS action was fired. At the moment, we don't do anything with this action yet. We still need to look in the logic to go and make the call to that API to go and get the topics from there, but it's a good start. We've got our action being fired from our NavigationContainer.

Intercepting Actions in a Saga to Execute Logic

Now to go and hook into when this action gets fired, and go and make the call to the server we head over to our sagas file. Now in here we import the constant, REQUEST_TOPICS, from the constants file. We're going to go and use this value to go and figure out when we want to make a call to the server. We will say import takeLatest from redux-saga. I'll show you how to use that soon. Before we do that let's go ahead and rename this saga, this generator function over here, to fetchTopicsSaga, and ensure that we export it by that name. Now basically, because we've imported the sagas file on our root it will go and execute all these generator functions. This allows us to do all sorts of funky, asynchronous logic inside of these generators. The way to go and use this action here is to go and say, yield. If you don't know generators there's a bunch of helpful information on Pluralsight on ES6 generator functions. Now because we injected the sagas file when we set up our root earlier it will go and ensure that all these functions that we export from this module will be fired up, and we can do all sorts of asynchronous stuff in here because they're generator functions. What we'll do now is we'll say, cool, wait around, and until you get the latest REQUEST_TOPICS, so the takeLatest function that we get from redux-saga is sort of a control flow function. It says, wait around until you get this REQUEST_TOPICS action being fired. Then go and execute this logic, and what we're doing now is we're just saying, logout of this log statement. So that's pretty cool. We're using a control flow function to go and hook into when actions get fired, so that we can go and execute logic. This keeps our action creators nice and clean, and gives us a nice place to go and execute behavior based on actions being fired.

Fetching Server Data from a Saga

But we obviously don't just want to log a message out to the console. We actually want to make a call to the server, and get the results to flow through the redux loop onto our state, so let's code up a new function. We call this function fetchTopicsFromServer. In here I use fetch to go and call our API URL, localized on port 3000/api/topics, and when I get a result back I just go and say, take that result, and pass it as JSON onto a normal JavaScript object. So the other thing that we get from redux-sagas, other than the ability to hook into when actions get fired, is the concept of side effects. Now you do more declarative side effects with sagas making it easier to test, so in a pure functional way you don't go and call code that has side effects. You go and call certain functions, and say this is the side effect that I want to happen. That way when you want to go and test it it's nicely decoupled. You can test that it's setting the intention for a side effect without it actually happening. To do that we want to import one of the side effects, call, from redux-saga/effects. Now we'll use a bunch of different side effects from this library during this course. For now we'll just use the call side effect. The call side effect is used to go and say, go ahead and call this function with these parameters. Now we can code up the logic that we want to execute when the REQUEST_TOPICS happens inside of an inline function right here, but I'm rather going to code up a new function to keep it a bit separate. We call this function fetchTopics, and it also needs to be a generator because we want to do async stuff inside of it. So what we do is we say we want a new constant called topics, and then we yield, and say we want to call the fetchTopicsFromServer, so we're handing off the redux saga, and it will go and execute this promise function that we defined up here, this fetchTopicsFromServer, and then it will return the result, so that's why we're using generators. It allows us to go and do async logic without having callbacks and stuff like that. In this scenario we're just going to log it out to the console afterwards, so console. log TOPICS FROM THE SERVER, and then we take the topics result from this function, and pass it in there, and now that we've got this fetchTopics generator function defined let's point this takeLatest statement to that function instead. So it will hook into when the REQUEST_TOPICS action is fired, and go and execute this generator function in a way that makes it easy for us to have asynchronous logic with side effects.

Storing Data on the Redux Store from a Saga

What are we going to do with the result of this topics? We basically want another side effect to happen. The side effect should be that this topics gets stored on state, and how do we store stuff in state with redux? We fire actions containing the data that we want to be stored. In order to go and dispatch redux actions we need to import the put side effect from redux-saga, and we also want to import the requestTopicsSucceeded and requestTopicsFailed action creators from the actions file. Now that we've got that available we can go and say yield put, and then the result of calling the requestTopicsSucceeded action creator with the topics, so in a nice declarative manner we tell it that we want this side effect to happen, but we also imported the requestTopicsFailed action creator. What are we going to do with that? Well, let's go and wrap this logic in a try catch. So we add a try, and then we move all this logic into the try block. We create a catch block, and inside of that we go and say that we want to put the result of the requestTopicsFailed action creator, and to that action creator we pass the message of the exception, so this will result in an action getting fired with a message saying what went wrong. To see if this worked let's go and have a look in the browser. When I reload this page and go and open up the redux debug tools you can see that we've got the REQUEST_TOPICS_SUCCEEDED action being fired, and that this action has a topics property, and inside of the topics we've got the topics that came from the server, so that's pretty cool. We've hooked into the action inside of our sagas, made a call to the server, and the data's now being fired on an action, and we can go and store it on state, so let's go and do that now. And in redux, how do we go and ensure that when an action gets fired we go and reflect it in state? We do that inside of our reducer function, so we go to the reducer file. In here we're currently setting default topics on state. Let's take that to an ______array again, and now let's import the REQUEST_TOPICS_SUCCEEDED constant from the constants file, so that we can hook onto that. In our reducer function we add a new case statement for this constant, so when this action gets fired what we want to happen is we want to go and sit on state, so we call state. set. Remember, we're using ImmutableJS, and we want to go and set the topics value to the topics that we get on that action. Well, when I reload the browser you can immediately see that our message is still saying that we've got three topics, and you know what, because we removed the default topics this means that we got these topics from the server. If I open up the redux debug tools you can see that it's the REQUEST_TOPICS_SUCCEEDED action that was fired, and that the navigation container value and state now is a topics value that we got from the action, so we've now managed to get data from the server, and flowed onto our redux state using sagas.

Summary

Let's do a quick overview of what we did in this module. We defined the requestTopics action, which we intersected in our saga. We then used the call side effect to kick off some logic to go and fetch topics from the server. We used the put side effect to dispatch an action containing the topics. This allowed us to mutate the data, and go and store those topics on state, making it available to our components. We're going to be following this pattern throughout the rest of this course, so don't sweat it too much if you're still a bit unclear of exactly how it works.

Handling Events with Redux-saga

Introduction

A common thing to cater for in all apps is responding to user input. In React redux apps we do this by firing actions from our components, which can then be reduced onto the store. We can also intercept these actions to go and execute logic. Let's have a look at how we'll be doing it in our app. So we're going to allow a user to go and select a topic in the Navigation component. This topic selection will bubble through the NavigationContainer component, and it will result in an action creator being called, selectTopic. This will result in the selectTopic action being fired. As you're aware, with redux it will go through the reducer, and it will have a chance to update to state. With react-boilerplate we use reselect and select the functions to go and pull values off of state and make it available to container components, but what we want to do in this module is hook into the selectTopic event to go and fetch some data. So how does that work? Well, we're going to do that with redux-saga. We're going to intersect the selectTopic action, and then go and fetch some data from the server, our links related to that topic, and then the saga will dispatch a new action using the requestLinksSucceeded action creator. There's the results and the REQUEST_LINKS_SUCCEEDED action containing the links, which then passes through the reducer, it gets updated on state, and in the end we use reselect again, with a different selector function to get the links onto the LinkListContainer, and in the end, onto the LinkList component. So that's what we're going to do. We're going to create a bunch of components, and intercept actions, and fetch data from the server based on user input.

Intercept User Events Inside of a Container Component

So let's go ahead and build out the Navigation component a bit, and go and bind to the topics that we're getting through on props. First thing we're going to do inside of the Navigation component is create a constant called topicNodes, which is the result of mapping over the topics prop. For each topic in the topics prop that we get through here we want to create a div. We need to go and set a key on this div to go ahead and uniquely identify each div that we're generating. Inside of the div we want to spit out the name attribute of the topic, and now that we've got a topicNodes array we can just go and add it in here instead of this hard-coded message, and if we go and have a look in the browser you can see that our code worked. It's mapping over the topics that it's receiving on it's props, and it's spitting out an item for each topic. While this is a Navigation component, and we actually want the user to be able to click on one of the topics, and see a list of links associated to that topic, so let's go and add a new prop onto this Navigation component called selectTopic. We want to call this function when the user selects a topic. Because we've added a new prop we need to go ahead and specify it in the propTypes, so selectTopic is of PropType func, and it's required. Now that we've validated that prop let's go up here to where we spit out the dev for each topic, and we add an onClick on here. Inside of this onClick we specify that we want the selectTopic function to be called with the topic that's being mapped over, and once I've saved that out, and go and load up the app in the browser, you can see that we're getting an error here. Because of the failed prop type we didn't get a selectTopic prop on the Navigation component, which shouldn't be a surprise because we haven't passed the selectTopic function to it yet. We need to go and do that now, and how do we do that? Well, that's the job of the container component isn't it? So we go to the NavigationContainer component, and here where we've got mapDispatchToProps we add a new one, selectTopic, and we expect this function to take a topic, and for now let's just log it onto the console. So we say, SELECTED TOPIC, and then we append the topic onto that. Now when we reload you can see that the error has gone away, so our prop validation is passing. Let's click on one of these topics. Great. You can see that our console log is working, and that it's got the correct topic being logged to the console.

Storing User Input on the Redux Store

But we don't just want to log it to the console do we? We want to go and fire an action, and go and store the selected topic on state. How do we do that? Well, we begin at constants. We want to define a constant we can use to identify our action, so we create a new constant called SELECT_TOPIC, and use this same convention to go and define a value for this constant, keeping it unique throughout our app. Now that we've got the constant defined, let's head over to the actions file. We import this constant in here, and now we can go and define a new action creator for the SELECT_TOPIC action, so export function SELECT_TOPIC. We need to ensure that we take a topic parameter on this action creator, and then we return a new action of type SELECT_TOPIC, and we ensure that we store the topic being passed in here as a property on this action. Okay, so now that we've got our constant and action defined we can head back to the NavigationContainer component, and import this selectTopic action up here, and move down to the mapDispatchToProps function, and change this logging statement to a dispatch of this action instead. To go and test this we go to the browser, and I click on the topics a few times. Okay, now I open up the redux debug tools. You can see that the SELECT_TOPIC action's being fired, and it's got a topic on it, so we're all set to go and store the selected topic on state in our reducer. In the reducer we import the SELECT_TOPIC constant, and now we go down to the reducer function itself, and add a case statement for the SELECT_TOPIC constant. When this action gets fired what we want to do is set on state a variable called selectedTopic, which is just a topic that gets fired along with an action. Let's test that in the browser again. I click on the topic here, and open up the debug tools. You can see that the correct action gets fired, and if we scroll down here and open up the state, and the navigationContainer state, cool. You can see that we've got a selectedTopic stored on state.

Adding an Additional Container Component to an Existing Route

Now that we've got the selectedTopic available on state we actually want to start showing links for that topic. To do that we need to go and generate some new components. So on the command line we use scaffolding again, and run the generate command, specifying that we want to generate the component. Again, we want it to be a Stateless Function. This time we'll call it LinkList. Yes we want styling, and no we don't want internationalization for this component. So that's the component, but we also want a container component, so we run the command to generate a container. We want to call this LinkListContainer, and we choose all the same options as we did with the NavigationContainer. We don't want any headers or styling or internationalization, but what we do want is we want sagas for our control flow, and we want reducers and constants, and all that sort of stuff. Okay, so now we've got our components in place. Let's go and have a look at them. So exactly as we did with NavigationContainer, we've got a LinkListContainer component here in the containers folder, and we've got a LinkList component here in the components folder. Let's go and pull the LinkList component into the LinkListContainer component. So the purpose of this LinkListContainer object is just to go and wrap our LinkList, and be aware of the redux world, making our LinkList component itself much simpler, so we import the LinkList component at the top here from its relative location, and where we return the JSX for the LinkListContainer we just return the LinkList itself, and this is very important. We ensure that all the props set on the LinkListContainer component gets passed onto this LinkList instance. Now you might recall previously when we wired up the

NavigationContainer that it complained with errors because we didn't wire up all of its dependencies in the routes because we're building dynamic packages per route. So we go to the routes file, and go and ensure that we also import the LinkListContainer reducer here, and then the LinkListContainer sagas. So on our route we import all these things currently, as it stands, and because we're importing more than one saga, and more than one reducer now for this route, we need to adapt this import here, this import. then statement. We need to be more specific about the parameters that we'll be getting on here. So I'm removing the existing reducer and sagas here, and rather renaming it to something a bit more specific, navigationReducer, navigationSagas, linkListReducer, and linkListSagas, and because we changed the variable names we need to go and adapt these two lines where we inject the reducer and sagas for the NavigationContainer to use the new names, and after we've done that we need to go and inject the reducer for the LinkListContainer, and we used the linkListReducer for that, and then we inject the sagas for the linkList, and we used the linkListSagas variable for that, but we're not even using the LinkListContainer anyway yet. Let's go ahead and do that instead of the HomePage container component. So import the LinkListContainer from its relative location, and now that we've got it imported down here in the render function we add it just below the NavigationContainer. When we save that out, oh, we get an error. We need to wrap it with something. We can return only one element, so I wrap it inside of a div. When we load that up in the browser we see we're not getting errors, but we don't see the LinkListContainer or the LinkList anywhere. That's because it doesn't have any content yet. Let's go and do that. Let's head over to the reducer for the LinkList. What we're going to do in here is go and add some test data. Before we do that though, let's go and remove this default constant. We're definitely not going to use that, and we need to remove it down here in this switch statement. Okay, see now it's nice and clean. Let's add some default links, which I've got on the clipboard, and it contains an array of links. Each link has got a description, a URL, a topic name, and an ID. So now we've got test data, let's flow it through onto our LinkList component and bind to it. To do that we head on over to the LinkList component itself, not the container component, but the real component, the component that takes the data and spits out some DOM elements. In here we just structure it from the props that get passed into this component, and because we're now depending on the links property here we need to go and add some property validation for it. I've already got it on my clipboard, so I can just paste it in here. Basically, we're saying that we expect links to be an array, and each object has a description, a URL, and an ID. Okay, now that we can expect links to be passed on the props to this linkList component let's map over it and go and create elements representing our links. So we're just creating a constant map over links, and for each one we called up a div. Each div should have a key to uniquely identify. You always have to do that when you map over data and create elements with JSX. Inside of this we want to spit out the link's URL, and let's also spit out the link's description. Close off that div, and close off our map function. Okay, now that we've got linkNodes let's go and plunk it in here inside of our element that we returned from render. Okay, so let's go and test it out in the browser. We're going to reload this. Great, you can see that our hard-coded link is displaying here, so it's getting its data from the redux store flowing through the LinkListContainer, onto the link, and it's being rendered, so we've moved pretty far pretty quickly.

Get Data from the Server Based on User Input

So what we want to do now is go and call the server to go and get the links, and store it on state instead of using our hard-coded links. How do we do that? Well, when we want to do asynchronous stuff we do it in sagas, so we go to the sagas file for LinkListContainer. We need to import the call side effect from redux-saga/effects because that's how we tell it to go and execute a function that will have a side effect. We import takeLatest from redux-saga. That's how we're going to subscribe to the action being fired that we want to trigger the logic to go and call the links. We import the SELECT_TOPIC constant from not the LinkListContainer's constants, but the NavigationContainer's constants. Remember, this is a constant that we defined earlier when we created an action that recalls the selected topic on state when the user clicks on it, so we're subscribing to a constant from another container component for the first time. Now we code up a fetchLinks from server function that takes a topic, and then in here we're just going to say return fetch, and we plunk in a URL, localhost port 3000/api/topics/ and because we're using a string template we say, pull in the topics name here, /links, so this is the URL to go and get the links for a particular topic, and then we say that we want to take the response and parse JSON into an object. Now that we've got a function that we can use to go and call the server and return the results we should go and code up a generator function. Now we use a generator function because we can go and handle async logic quite tell inside of it. This generator function we call fetchLinks, and it also expects the action that was fired that triggers this logic. We cut it up in a try block from the start, and we're going to say here that we want a constant called links, and this should be the result of yielding on the call side effect, parsing that the fetchLinksFromServer function, and the topic that we've got on the action parameter, so this basically says, go and call that, and wait for the results to come back. That's the benefit of these functions being generator functions. Then let's just add a comment here. Later on we want to dispatch an action to store these links. For now we just console. log a little message for ourselves here, LINKS FROM SERVER, and pin the links onto that. For our catch for now we'll just go and say let's dispatch an action to this store to go and store the error, so we've got our fetchLinks generator function with our side effects in there, so in our saga, which is also a generator function because it can kick off a bunch of async stuff, we use yield to say that we want to wait until takeLatest has picked up that the SELECT_TOPIC action has been fired, and when it's fired we want the fetchLinks logic to execute. Let's go and test that out in the browser. I'm quite nervous because that was quite a lot of code, and we haven't tested in the meanwhile. Okay, so when I select on a topic, great, you can see that our message is being logged, and when I expand that, cool. That's the link that came back from the server. Let's select another topic and expand. Okay, great. So it's definitely talking to the server and getting the links back, and spitting them out here in the console.

Storing Data from the Server on the Redux Store

Okay, so logging it out to the console is pretty cool, but we obviously want to go install those links on state instead of using the hard coded links, so we head over to the constants file of the LinkListContainer, and we add some new actions in here, which we can go and dispatch when we get our data or when we get an error. So we export a new constant called REQUEST_LINKS_SUCCEEDED, and we give it a namespaced value, REQUEST_LINKS_SUCCEEDED, and we create one called REQUEST_LINKS_FAILED, and we create a similar value, but for REQUEST_LINKS_FAILED, and let's also delete this default generated constant from here. Now we can head over to actions to go and create some action creator functions for these constants. Let's remove this default constant from here, and import the newly created constants instead, REQUEST_LINKS_SUCCEEDED, and REQUEST_LINKS_FAILED. Okay, and now that we've got that imported let's go ahead and flesh out our REQUEST_LINKS_SUCCEEDED, and REQUEST_LINKS_FAILED action creators. We start with requestLinksSucceeded. This action creator takes a links parameter and we return an action object of type REQUEST_LINKS_SUCCEEDED, and we also store the links value on there. Let's move onto the requestLinksFailed action creator. This one takes a message parameter, and we return an action for that, and it's got the type of the REQUEST_LINKS_FAILED constant, and we ensure that we store the message on there. We also need to delete this generated default action from here. When I save that out you can see that all the linting issues go away. Love it when that happens. Okay, and let's go and make use of these action creators now. We head back to the sagas file. First thing we want to do is make sure that we import the put side effect from redux-saga/effects because we want to dispatch actions. We also import our newly created action creators from the actions file, and now here when we wrote some comments earlier we replace that by calling yield put requestLinksSucceeded, and passing the links to that, and we remove the console. log here, and for our error handling in the catch block we yield put requestLinksFailed, and the message on the exception. So again, we're using the side effect functions from redux-saga to go and trigger our side effects. Basically wrapping it in a nice way that we can go and switch it out with something else if we want to, and when I head back to the browser, and open up the debug tools, and click on a topic, okay so, you can see that REQUEST_LINKS_SUCCEEDED has been fired, and it's got the links on it, so we're in a good position to now go and store it on state, and where do we do that? Inside of the reducer, and in here we import the REQUEST_LINKS_SUCCEEDED constant, and we also set the links to just an empty array. We don't want any default hard-coded data in here, and now we can go and add a case statement for the REQUEST_LINKS_SUCCEEDED constant, and when this action fires we want to go and set on state on a links variable the links being passed through on the action. When we're testing the browser you can clearly see when I click on the topics here that it loads up the correct links for that topic. I don't even need to show you the debug tools.

Summary

So we're handling user input, and we did this by defining a new action, selectTopic. We dispatched this action from the navigation component, intercepted it in our saga, and kicked off a call to go and fetch some data from the server, which we got into our store. Hopefully you're starting to see how you can combine redux, redux-saga, and reselect to do cross component logic while still keeping your components decoupled.

Styling Your Components with CSS

Introduction

We've done some pretty advanced stuff so far in this course, but our app looks a bit ugly, but don't worry. We're going to sort that out now by adding some CSS to the mix, and we're also going to bolster our components to cater for a bit more interaction for the user. At the end of this module we'll have something that looks like this. As you can see, it looks much better than what we've got now. You can also click on the icon in the app bar and it slides a drawer containing the topics in with a nice transition. When you select a topic it goes away, again, with a nice transition.

Styling the AppBar Component

In order to add icons to our application we use npm to install react-fontawesome, just a React wrapper around fontawesome. It's very nice to get icons into your application this way, as you don't need to do all sorts of stuff like spriting and manage a bunch of images. To get fontawesome to work you also need to include its CSS file into your application, and we're doing this using @import from the apps styles. css file. We're also importing the Google font called Roboto with all the weights that we might use during the applications development. There are also some other defaults that we need to go and set at our application level. I want the container to have a bit of a margin around it, and let's go and specify some rules for our body. We want a font-family of Roboto and a background color of a grayish sort of color, just so that we can have cards that are over layered, and have a bit of contrast. Now let's go and add a rule for all elements, and go and say that we want box-sizing to be border-box. I don't know about you, but I prefer working with border-box. I just find it much easier to reason about my element sizes that way. Let's go and generate our AppBar component. We use scaffolding again, and run npm run generate component. I want this component to be a stateless function. Let's call it AppBar. We definitely want styling on it, and we don't want internationalization. We're going to use the AppBar mainly for navigation, so let's go ahead and pull the AppBar component into the Navigation component. We import AppBar from its relative location, and we go and add an instance to the AppBar component just above where we pull in topic nodes. Okay, let's go and flesh out our AppBar component a bit. We add some text, Coder daily, so this will be the title of our application ones we'll see at the top. We go into the styles for this, and we go and flesh out this appBar rule over here. We want the AppBar component to take up as much width as the container allows it. We want it to have about 20 pixels of padding, and we give it a background color of a very bright blue. We want the font color to be white, and we want it to have a rather large font size. We also said that we want it to be a flex box. You'll see why soon, and we justify the content with space between. We want a lot of space between the elements of this flex box. It'll make sense in just a bit more time when we start adding more elements to the AppBar component. Let's go have a look in the browser and see what it looks like. Great. You can see it's starting to take some shape, and this little bit of style is making our application look like something, so let's add to this AppBar component. I wanted and icon to show on the left hand side of this AppBar, so let's import fontawesome from react-fontawesome, and now that we've got that imported we can go and add an icon. For more control of the icon styling I'm going to add a wrapper div around it, and we give that a class name of styles. iconButton. Inside of this div we add a FontAwesome component, and we set a className of styles. icon on that. With fontawesome you use the name property to specify what icon to display? We want the bars to display. We close that off, and sort out a bit of indentation here, and let's also wrap this Coder daily title in a div, so that we can style it. We give this div a class name of styles. heading. We'll go and style everything afterwards. Let's close that off. We also want to show a link that allows the user to login. Let's create a div, and give this a class name of styles. linkContainer. Inside of this div, for now let's just add some text, log in. If we have a look in the browser you can see it doesn't actually look too bad. We've got our bars displaying here on the left, and we've got a title, and we've got a log in link. We still want to do quite a bit on this styling, and we're going to do the rest purely in the styles. css file of the AppBar component. Okay, so we go and create a class, iconButton, and we give this a cursor style of pointer. Remember, we made the button be a div element, so we need to control the pointer, so that it looks clickable. We give it a margin-right of 15 pixels. Then we go on to style the actual icon. We give it a font size of 20 pixels. Because it's an icon font you control the size of the icon through font-size, and we give it a color of white. We add a class for the icon:hover. Here we want the color to be this color. We'll see what it looks like soon enough. Then we specify a rule for our linkContainer. Here we want the font-size to be 16 pixels. We don't want it to be the same font size as the heading. Now we go and define a class for heading. I want this text to be aligned on the left, so that's why we add a margin right on the icon button, and I also wanted to display flex, and we set a flex-grow rule on it of two. Okay, so let's go and have a look in the browser. That actually doesn't look too bad. The title looks good, and I like the spacing between the bars and the title.

Creating an Action to Toggle the Drawer Component's Visibility

Now we're going to work on the functionality that when you click on those bars that a drawer slides in from the left allowing you to select a topic from there. To do that we need to add some functionality. We need to trigger an action, and it needs to go and store a value in state, and this value will decide whether the drawer is open or not. To do that we start in constants in the NavigationContainer, and we'll add a new constant called TOGGLE_DRAWER. Now that we've got the constant defined let's go and create an action creator around it. In actions we go and import the new constant, and then down here in the bottom of the file let's go and create a new function called toggleDrawer. This function returns the action with the type of TOGGLE_DRAWER. Now let's go and make this action available to the Navigation component, so that it can pass it onto the AppBar component. We do this by starting in the NavigationContainer component, the component that's aware of the redux world. We ensure that we import the toggleDrawer action creator from the actions file, and down in mapDispatchToProps we go and create toggleDrawer, and we specify that this is a function that when it's called will go and dispatch the result of calling the toggleDrawer action creator. Now that it's being bossed through on the Navigation component let's go into the Navigation component and use it. First thing we need to do is to structure it here on the function from the props parameter. Then we go and add a prop on the AppBar instance called toggleDrawer, and we set that to the parameter. Now that we've got this dependency on the toggleDrawer prop we need to go and add the propType for that, so just toggleDrawer, and it's of type func, and it's required. Okay, so we go into the AppBar component now. That was the whole purpose for this toggleDrawer action that we've been passing down. First thing we do in here is we destructure it from the props being passed through to the function, and now we add an onClick prop on this div, and we set that to this toggleDrawer function that gets passed in here, and because we're depending on a new prop now we need to go and specify some property validation rules. Set up a new propTypes on AppBar, and go in and create a rule for toggleDrawer. Again, it's a propType of function, and it's required. Let's go and have a look in the browser to see what that's done. I click here on the bars in the AppBar, and let's open up the redux dev tools, and I'm hoping to see. Great. We've got the toggleDrawer action being fired, so we've wired it up correctly.

Creating the Drawer Component

So we've got our action firing, but what's it going to do? We still need to go and build our Drawer component. Let's go and generate it now. So again, we use scaffolding here. We also want this one to be a Stateless Function. We want to call it Drawer, yes we want styling, and no we don't want internationalization, so in some places in this course I go and build the most specific component first, and then refactor away to more generic components. We're going to do a lot of that later on, but with a drawer we're going to start by building the generic component, meaning it can be reused in interesting ways. It's not going to be specific to navigation. So I'm going to add on all the props that we need to do this right off the bat. The first prop we need is items. This contains all the items that we want to bind to in the list in the drawer. Then we want selectItem. This is a function that we'll call when the user selects an item in the drawer. We want itemLabelAttr. This is the property that we'll use on each item to pull up the label to go and display in the list in the drawer. An itemKeyAttr. This should be unique across all items. Because we're going to iterate over items we need to specify a key in the JSX, and isDrawerOpen. This is an indication of whether the drawer should be in open or closed state. Okay, so let's go and use these props now shall we? I'm assigning a new constant called itemNodes, and we are setting this to the result of mapping over the items being passed in here. Now we've got a loop where we can go and generate new elements. What we want to generate's a div, and we're going to set some properties on it now. We want a className of styles. item. We'll flesh that out soon enough. We want a key attribute on it. Now we want the key to be calculated by going and pulling off the itemKeyAttr that was specified from each item. Then we specify that we want an onClick attribute on this div. We want this to be a function, and when the user clicks on this we want to call the selectItem function that's being passed in our props, and pass the current item to that. Let's close off this div, and now we can add its contents, and we get this label by using this itemLabelAttr prop that was passed in, so for each item it will go and get a field that's passed in to use as the label. Okay, so we've generated a bunch of itemNodes. Let's go and use it by pulling it into this element here. We also need to go and add some property validation now by going and specifying a propTypes object onto the Drawer component. The first prop we need to validate is items, and this is of type array, and it's required. Then we want to add a rule for selectItem. This is of prop type func, and it's also required. Next one, itemLabelAttr. This is of type string, and it's required. Then itemKeyAttr. This is of type string, and it's also required. Then we've got isDrawerOpen. This one's of type Boolean, and it's also required. We're not passing it in yet, but we've at least done our validation, and we've made it available to this component.

Adding Conditional Styling Based on State

We're going to add a class name onto the drawer based on whether the drawer is open or not. When you find yourself doing conditional stuff on classnames inside of JSX do yourself a favor, and go ahead and install the classnames npm package. I'm installing it now on the command line, and I'm specifying a very specific version. I don't want it too bright for you when you follow along, so let's go and use class names in our Drawer component now. We import classNames from the classNames component. How do we use this classNames function that we've pulled in? Well, let me go and show you. On this className over here we're going flesh out this className attribute on this container div, so let's make a bit of space for that. Now inside of this we go and say, call classNames, and we pass the existing styles to the drawer in there. Now we can go and add another style onto this, so you can add a multitude of styles as parameters to classNames, and it will combine it for you onto the element, but you can do conditional stuff on it too, and we're going to do that now. To do that we pass an object in here, so we wrap this in curly braces, and then you add a colon after someotherstyle, and pull in the isDrawerOpen variable. So what's this doing? Well, we're using the syntax to go and say only apply some other style if the variable isDrawerOpen is truthy. That allows us to go and say that, if for example, this is false, some other style won't be applied, but if it's true then some other style will be applied or based on the prop being passed up here. I'm sure you can see how that could be useful. In our case, we want a style from our style sheet called drawerOpen to be applied when the isDrawerOpen prop is set to true, so let's just type that in quite ______, but when I save it out you can see we get an error, and that's because it's not a string, and it doesn't conform to a valid key in JavaScript. Luckily, in modern JavaScript we've got the concept of computed attributes. If we just wrap this in square braces you can see the linter is happy. It uses this dynamic call then, and computes it, and gets that value and uses it as the key. That's pretty cool.

Maintaining the Drawer's Open State on the Redux Store

So let's go and use the Drawer component now, and we're going to use it in navigation, so we duplicate this import line, and adapt it to import the Drawer component. Down here in the function let's go and add the Drawer component just below the AppBar. Now the Drawer component takes a bunch of properties, so to make it easier on ourselves let's go into the Drawer component, and go and take these propTypes over here, and copy that. Go back here to the Navigation component, and go and paste it onto the Drawer here. We'll now adapt this to go and pass all the correct values, so for items we want the topics, which we've already got available. For selectItem we go and use selectTopic. This is also already being passed in onto the Navigation component. For the itemLabelAttr we specify name, so this string name will be used to go and pull a field value from each object that's being passed into the drawer, so it's essentially doing the same as we're doing up here where we map over the topics. For itemKeyAttr we also specify name, so this is basically, in a dynamic way, doing what we're doing up here where we specify the key when we map over topics, and then we specify isDrawerOpen, and we assign that to isDrawerOpen, which we still need to make available to the navigation component, so let's add isDrawerOpen up here on the props where we destructure them, and we can remove all the other logic where we map over the topics to go create and elements. The Drawer will do that for us. Now that we've depended on a new prop, isDrawerOpen, we need to go and validate that quickly, and this is of type Boolean, and it's required. As you can see, you're in the browser. It's complaining about the isDrawerOpen prop that's not being passed through to navigation, and the drawer component, so let's go and fix that quickly. Let's go and make it available, and we need to do that inside of the NavigationContainer's reducer file. This is where we maintain all the state for the navigation code, so just below topics we add isDrawerOpen, and we set that to false. Now when we reload the app in the browser you can see that we're not getting any property validation errors anymore, so the NavigationContainer is passing the isDrawerOpen false value through to navigation, which in turn is passing it through to the Drawer component. Now remember, when I click on this icon over here, and open up the debug tools, you can see that we're already firing the TOGGLE_DRAWER action when the user clicks on this icon, so all we've got to go and do now is handle this action and set the variable on state accordingly. This, again, is done in the reducer, and in here we import the TOGGLE_DRAWER constant, which we defined earlier, and now we can go and add a case statement for when this action gets fired, so case TOGGLE_DRAWER. What we want to happen is we want to go and set the isDrawerOpen value to the opposite of what it is at the moment. So we go and say, set isDrawerOpen to the opposite of state. get isDrawerOpen, so when it's true, and this action gets reduced, it will turn it to false, and when it's false it will change it to true. So let me show you what I mean. Let me just position the redux dev tools here at the bottom of the page, and now when I go and click on this icon you can see that the NavigationContainer state now has an isDrawerOpen value of true, and when I click on it again, you can see that it changes back to false.

Show or Hide the Drawer Component Based on State

So let's go and style this drawer now to make it meaningful. We go into the styles. css of the drawer component, and I've got some style code already on my clipboard, which I'm pasting here. Basically, the important part here is that its position is fixed, and that its lift is -300 pixels, the same as its width, so you actually can't see it on the screen, so by default you actually can't see it on the screen, and now we add a class called drawerOpen where we set the left position to 0 pixels. And if you go back to the Drawer components code again, see this drawerOpen is the class that we assign to this element conditionally based on the isDrawerOpen prop. When the isDrawer prop is set to true it means that this drawerOpen class will be assigned to this element, meaning that it's left position will be at 0 pixels, making the drawer visible to the user. Okay, let me go and show you that in the browser. So when we reload this, and okay, let me just get these debug tools out of the way, you can see that the drawer actually comes in from the left with a nice transition. It doesn't look very nice at the moment, but at least we've hooked it up to state, and we're maintaining it's open state through redux. Let's go and flesh out the rest of this drawer component style. Again, I've got some style on my clipboard, which I'm just pasting in here. Basically, we've got two classes, just item, which maintains the style for the items that we bind to, nothing weird in here, and we've got a rule for item:hover where we change the background color a bit. Let's go have a look in the browser. This is starting to take shape. It looks much better now. We've still got a bit of work left to do on this component though. Let me show you. When I select these topics you can see that it's still got its effect that we want. It changes the links in the background, but we want this drawer to go away when the user selects a topic. Let's go and do that. In the NavigationContainer's reducer you're in the SELECT_TOPIC action handler. We go and say, after setting the selectedTopic we want to go and set the isDrawerOpen value to false because the user's selected a topic. We don't want to keep on showing the drawer. Let's go see what that looks like in the browser. When we open up the drawer go and select the topic. Great, it transitions away. I actually like seeing that so much I'm going to do it again. Excellent. That's working now.

Creating the Link Component

Now we want to go and make the links look a bit better, and we're going to start by creating a component called link. So we use the scaffolding again to generate a new component. We want this to be a stateless function. We want it to be called link. Yes, we want styling, and no we don't want internationalization. Let's go and flesh out the Link component a bit. First priority is to get the right data on the Link component. We're saying here that we expect an object called link to be passed through on props, and what we do with this is we spit out this objects. url property as the text for the link component, and because we've added a new prop on here we need to go and do validation for it. Luckily, we've done validation for the link data type before. We did this in the LinkList, so let's head over to the LinkList, and from here we copy the rules for a link. Back in the Link component we add propTypes onto link, and in here we specify that we want the link to be defined by these rules, and we paste in the rules that we've copied. Now let's go and use this Link component, and we do this inside of the LinkList component. First thing we do is we import Link from its relative location. Now here in this code where we map over the links we want to go and create a Link component instance instead, so we remove this, and replace it with Link. Let's just close that tag before we start adding props. We still need to go and set a key on this Link instance, and that's still going to be the links ID, and then we set the link prop on it, and we pass in the value that we're getting from this map function. Let's go ahead and test that out in the browser just to go and verify that we haven't made a mistake. Okay, so when I select a topic, okay it shows our link there, but clearly I made a mistake in the property validation. Let's go and fix that. Well it seems for the Link rule I just passed an object in there hoping that it would work, but we should actually say that it's a PropType of shape, and pass this object to that, and now when we reload in the browser you can see that our property validation error has gone away, so let's go and make this link look respectable now. We can head on over to the styles. css file for the Link component, and again, I've copied and pasted some style in here. Nothing strange in here. We've just got background colors, I made it a flex again, we've added some shadow around it, stuff like that. Nothing weird going on here. Let's have a look in the browser though. That looks pretty decent. You can see that we've got this material looking card for the link now. Let's go and focus on fleshing out the Link component to show all the information that we want to see.

Styling the Link Component

So first off, I'm going to create all the components that I want as children to this Link component first before worrying about their content. First, we want to go and create a div with a className of styles. votingContainer, and we want to close off that div. This will contain our voting count a bit later on, and the rest of the link will go into this div. This div's got a class name of detailsContainer, so we're going to add some more complex markup in here. In fact, let's go and flesh out this detailsContainer div now. It's going to have two child divs. We create two child divs quickly, and inside of the first child div we want to add an anchor tag. We set the href of this anchor tag to be the link. url prop, and then we add a className, and we set that to styles. linkAnchor. We'll go and worry about the styles soon enough. For the text of this anchor we also want to use the link's URL property. Let's go and sort out the indentation on this anchor to be more consistent with what we've been doing the whole time. Okay, the second div over here is for the description, so let's add a className attribute on here, and set that to styles. description, and for its contents we want to use, wait for it, the link's description attribute. Each link coming back from the server has a voteCount attribute, which we haven't used before. Let's go and add some validation in the propTypes for voteCount, and this is a number, and it's required. Now we can go and use it in our mockup. So inside of our votingContainer we add yet another div. This one's got a className of styles. votingCount. We want to close off this div, and inside of it go and add the styles. voteCount as the text. So we've added some complex markup. Let's go and define some styles that will make this markup worthwhile. Again, I've got some style code on the clipboard, and I'm pasting it in here. We've got a rule that targets the votingContainer, and creates some padding around the element, and for votingCount we go and set a larger font, we align the text, and we go and set some padding. Okay, so more style. DetailsContainer, it's got a display value of flex because we want fine grain control of its layout. We set its flex-direction value to column, and we say that we want to justify the content center. That's actually the reason for making it flex. Then we want to say we want some space on the left. That's what the margin-left value is for, and we give it a medium sized font. We've also got a class for description where we go and change the color to make it a bit gray, and for the votingIcon we go and set a font-size of 40px. This class we are actually not going to use immediately, only later on. Last, but not least, we add a class called linkAnchor. We say that we want text-decoration none. Because this is an anchor in our HTML it will underline it by default, so we're just overriding that and saying, don't do that for us.

Then we've got this peculiar syntax over here where we say &:hover where we go and say text-decoration: underline. I want it so that if the user hovers their cursor over the anchor it shows a line underneath, so this is a pseudo rule, but this is with a special syntax, and we can do this syntax, which might look familiar to you if you've used SAS in the past because react-boilerplate gives us post CSS out of the box, so that makes our style code a bit easier. Let's go and have a look in the browser to go and see what it looks like now. When I select a topic you can see that the card is starting to look decent. Select another one. Great. I think I made a mistake somewhere with the votingContainer though because I'm not seeing the voteCount come up here on the left where I expect to see it. Oh dear. That was a bit silly of me. I tried to get the voteCount data off the styles object. Now when we go and have a look in the browser, ahh that looks better. You can see that we've got the voteCount displaying here on the left on the card. Great, so our app is starting to look like something. I especially like this drawer that opens up from the left with a nice transition, and closes when we select the topic.

Summary

That's it for this module on styling. I hope you agree that we got very far with very little effort. What I like about styling in react-boilerplate is the way that we automatically get namespace CSS, so we don't get any conflicts. The one thing that stood out for me while working on this module is the way that we applied conditional style onto a component. We had to go and use the class names in ______ module, and we had to use computed properties JavaScript syntax. Maybe I'm just excited by this because this is the first time ever that I've found a real world use for computed properties.

Adding Routes to your Application

Introduction

In this module we're going to introduce routing to our app. I'm sure I don't need to sell you on routing. All developers love it, and users too. There's nothing as irritating as pressing the back button and be thrown out of the app, and we're going to be doing routing the redux way, meaning that all our route changes will go through the dispatcher. This has got some benefits in testing because you can time travel. You can go and undo actions and go back to the previous route. It's pretty cool. You'll see. Let's have a quick overview of what we'll be doing in this module. What we're going to do in our app is trigger an action, SELECT_TOPIC, from the NavigationContainer. We're already doing this at the moment, but what we'll do differently is we'll intercept this action in the saga, and trigger another action in state using the push function that we get from React router, which will result in this @@router/LOCATION_CHANGE action. We're going to set up a route that will go and render the LinkListContainer. The LinkListContainer will go and trigger an action, REQUEST_LINKS, to go and get the links for the topic currently on route. The sagas will intercept this action, and go and fetch the links associated to the topic from the server. It will then dispatch another action, REQUEST_LINKS_SUCCEEDED, which then gets reduced, and the state gets updated. In the end our LinkListContainer gets its links that it should display. Great. Let's go and do this in code now.

Intercepting an Action to Change the Route

Because we're now going to fetch the links when the URL changes instead of hooking into the SELECT_TOPIC action being fired, let's remove it here in the LinkListContainer. So we remove the import, and we remove the takeLatest statement down here. Now in order to go and change the URL when the SELECT_TOPIC action gets fired we need to go over to the NavigationContainer's sagas file. In here we import the SELECT_TOPIC constant, and I'm going to create a function here called pushTopic that takes the action that was fired. For now we'll just log out the action's topic. Now let's go and create a saga function for this one. We call it, selectTopicSaga, and in here we say yield on takeLatest when the SELECT_TOPIC action is fired. Go and execute the logic in pushTopic. So what we want to happen is when the SELECT_TOPIC action gets fired we want to go and execute the pushTopic function, and in there we'll handle the logic to go and change the URL for the relevant topic. We also need to export this new selectTopicSaga function, and now that we've done that let's head over to the browser, and go and see if this worked. You just open up the debug tools here, and when I select a different topic, great, you can see that it's being logged to the console. So we've intercepted that action in our sagas, and we're now logging it out. We're now in a good position to go and trigger a URL change. Now to go and trigger the URL change we need to import the push function from react-router-redux. Now that we've imported that let's go and change the pushTopic generator function to go and cause the URL change. Now when we say that we want to put that means that we want an action to be fired, and we get that action by calling push with the URL. In our case, we're using a string template to build up this URL, so that we can fold in our actions topic. name. Great, so we're using a redux-saga side effect to go and put an action, and we're using push to go and get that action. Let's go ahead and test that in the browser. Okay, so now when I select a topic, great, you can see that the URL changes. We haven't set up our routes or anything, but it's good that our action's being fired, and it's triggering a URL change.

Getting a Component to Display on a Child Route

So let's start the work to go and show the links when the URL changes. To go and do that we need to head on over to the home page Container component, and in here where we currently import the LinkListContainer we remove that line. We also remove the LinkListContainer from the render function, and what we do instead is we go and say, go and show this. props. children there. So this is going to be provided by the React router, and because we're now depending on children we need to add some property validation rules in here. Okay, so let's go ahead and do the work, so that React router will pass the component onto this component, so that it shows here in render. What we need to go and do is go to our terminal, and generate a new route. So I'm using scaffolding here, npm run generate route. It's asking us which component we want the route to show. Well, we wanted to show the LinkListContainer. Next, we need to specify a path for this route, which will be /topics/ and then the syntax, :topicName. This just means that topicName is a parameter that's going to be passed from the route. Now we need to go to our routes file to go and do some further configuration on our newly generated route. Right off the bat we can see something's wrong here. Our home route currently still imports all the stuff necessary to render the LinkListContainer, and because this is a variable route now it's not always going to show on the home route. We need to go and remove that, so we remove the import statements for LinkListContainer's, Reducer and Sagas, and down here we resolved the promise. We also removed the statements to go and inject the reducer and saga for the LinkListContainer. Now if you scroll down a bit you can see that our nearly generated route is right here, topics/topicName, but what we want is for this route to be a child route of our main route. This means that when this route gets hit it will be passed on a prop called children to the HomePage component, and be rolled into there dynamically. So let's go and create a childRoutes prop on this route, and now this needs to be in the array, because you can have multiple childRoutes, and we move our newly generated route in here. Now the scaffolding script that we use to generate this route did a pretty good job. We just need to go and change one tiny thing, and that's because I made the decision to inject our sagas as singletons, so here where we inject the sagas we just need to pass a string to go and identify the sagas that we inject here, so we just call it linkListContainer.

Using Reselect to Access Route Parameters

Now we've got a route that, when it gets hit, goes and says that the linkListContainer should be slotted into our app, but how will our linkListContainer know what topic is selected, so that it can go and fetch the correct links? We're going to do this inside of the selectors file in the LinkListContainer component. In here we go and create a new selector function called selectRouteTopic, and when this function gets executed we can expect a state and props parameter to be passed to it. From the props parameter we pull out the params. topicName. React router passes to parameters from the URL onto the components props, so that's why we're coding up this selected function like this. We need to access the props for this component in here, so that we can pull out the parameter from the URL. Now that we've coded up the selector that goes and gets the props for this component to go ahead and pull out the topicName that came from the route, we can go and combine this selector onto the selectLinkListContainer. So we go and do that over here. We go and say, cool, go and combine the selectLinkListContainerDomain selector, and the selectRouteTopic selector, and we can expect this function, this next selector in the chain, to have the routeTopicName now, so we add that as a parameter. Now we basically want to combine these, and we're using Object. assign to go and say, take this subject. toJS result, and on that go and add the routeTopicName. Now everything that we return from this selector will make it onto the LinkListContainer, and the LinkListContainer passes everything on to the linkList, so we can head on over to the linkList, and go and use the routeTopicName over there. So we add the routeTopicName on the function, and let's also go and add it here inside of the div, so we can see what the routeTopicName is, and because we've now got a dependency to a new prop let's go and do some validation for that. It's just a string, and it's required. Let's go and test our new route in the browser. When I reload this page, great, I'm a bit relieved. It's working, and you can see that it's got the topic that we've passed through on the URL displaying here in the app.

Loading Data for a Component When It's Loaded by a Route

Now we want to go and trigger a new action when the LinkListContainer loads for the first time with the topic that it gets from the route. To do this we go and define a new constant in the LinkListContainer's constants file. We duplicate this line and adapt it for REQUEST_LINKS. Great. Now that we've defined that new constant let's head on over to the actions file where we import the REQUEST_LINKS constant at the top, and after doing that we can now go and create a new action creator. So we export a new function called requestLinks, and this function has one parameter, topicName. We're going to get this parameter from the route, and we return a new action from this of type REQUEST_LINKS, and we also want to store the topicName value on this action. Great, so we've set up a constant, we created an action creator around that constant. What do you think is next in triggering the logic to go and request the links from the server? Wade on over to the index file of the LinkListContainer. We want to import the newly created action creator inside of this file, so import requestLinks from the actoins file, and we want to make dispatching this action available down here in the mapDispatchToProps function, so we create a new prop, requestLinks, and we say that this is a function that takes a topicName, and when this function is called it will dispatch the result of calling the requestLinks action creator. Great, so we've rolled in the action creator, and it's now available for the LinkListContainer component. Let's go and create a componentWillMount function on the LinkListContainer. So this function will be called when the component is mount for the first time, meaning that we want to go and request the links, so we call this. props. requestLinks, and we pass from props the routeTopicName in there. This routeTopicName prop is passed to the LinkListContainer because we've adapted the selector for the LinkListContainer to go and pull it off the route. Now that we're depending on these two new props we need to go and set up some propTypes for it, so we set up the propTypes, and we add a rule for the routeTopicName, which is of type string, and it's required. RequestLinks is of type func, and it's also required, and it's also fixed indentation on that. So we've created a constant, an action, and we're now triggering that action at an appropriate time. Let's head on over to the sagas file, so that we can hook in to when that action gets fired, and kick off the logic to go and fetch the links from the server. First thing that I need to do in here is import the REQUEST_LINKS constant from the constants file. Now we removed some code from this default saga a bit earlier. Let's go and adapt it to go and say, when the REQUEST_LINKS action gets fired we want to go and trigger the logic in the fetchLinks function. We used to have the full topic on this action, but now we'll only have the topic name, so we adapt this call over here to only pass the topic name from action onto the fetchLinksFromServer function. The fetchLinksFromServer function also needs to be adapted. We rename this topic to topicName, and here in this string templating we rename this topic. name to topicName. So let's go and test that out in the browser. When I reload this page, great, I know it's working because it's displaying links, and it's only our new code executing now. Let me open up the redux dev tools to go and show you. When I scroll up here to go and see which actions fired, see, here we've got the REQUEST_LINKS action fired with the topic name of apps, and that resulted in the links being retrieved from the server, and being stored on state, but there's a catch. There's something in there that we still need to sort out. We'll now go and select a different topic. It doesn't update the app. That's because we've hooked into componentWillMount, and that's where we trigger the action to go and request the links. When the props change on that component, that already mounted component, it doesn't hook into that, and it doesn't go and fetch the updated links. So let's go and fix that now. So let's go into LinkListContainer where we trigger the REQUEST_LINKS action, and go and see what we need to do there. As you can see, we've hooked up the componentWillMount and requesting links from there. How will we be able to go and trigger this when the selected topic changes, when the route changes and we've got a new topic on there? Well we need to go and add componentWillReceiveProps onto this component, and this takes a parameter called newProps. This is before the props are passed to the component, and before render gets called, so in here we can go and say that if the newProps. routeTopicName is different from the current props. routeTopicName, then we want to go and trigger requestLinks, and we want to do it using the routeTopicName on the new props. Okay, let's go and test that out in the browser. Then I reload here, select the different topic. Great. You can see that the links are updating when I select a different topic, so we've done the correct thing with componentWillReceiveProps.

Preventing Code Injection Through Route Parameters

We can also use the back and forth button to change the URL, and it loads the corresponding topic and links, but there's a little bit of a catch in the way that we've coded this up, and you might have picked up on it if you were paying attention. The problem is, if we go and change this URL up here it will pass through all our code, and then display in the app itself, which isn't a great idea. Let me show you. See what I mean? I can add any arbitrary string onto the URL, and it will take it through all the different layers, and add the text onto the web badge. This is really bad because somebody can send you a link that injects a bunch of content in there doing all sorts of malicious stuff, so we need to go and use this value from the URL, but then just use it as a lookup to the topics to go and pull out the actual topic. If it doesn't find the topic it will just display nothing. If it does find the topic it works as expected, but to get the selected topic based on the parameter passed through on the route is a bit more challenging than you might think because the LinkListContainer component is the one instantiated by React router, and it's the component that will get the variable from the route, and the data containing the topics is all managed by the Navigation container, so what we're going to do now is we're going to import the selectNavigationContainer inside of the selector of the LinkListContainer, so import selectNavigationContainer from its relative location to this file, and now we're going to use this one, and create a new selector function called selectTopic, so we go and say createSelector, and we say that it should be a combination of selectNavigationContainer, and this selectRouteTopic function that we created a bit earlier. Now when these two are combined we can expect a function to be called with the two variables, navigationState, and routeTopicName. Now that we've got access to the navigationState we can go and create a new variable, selectedTopic, and we get that by calling. find on the navigationState's topics variable, and for the filter criteria we say that if the topics name matches the value that we got through the selectRouteTopic, then that's the topic that we want. Then we return that selectedTopic or we just return an empty object with a name set to an empty string, and we do this to ensure that we've always got a default. It'll make a bit more sense pretty soon, and now that we've introduced this new selector function we need to go and adapt this selectLinkListContainer selector to go and use this one instead. So if we change this function to selectTopic, we change this parameter to the callback function, and now we go and set the routeTopicName to the result of calling the. name property on the topic itself. Now when we go and reload this in the browser with a bad topic variable in the URL it just does nothing. We could actually hook into this and provide a more useful message, but you get the idea. At least now somebody can't inject malicious content into our app, and when we go and select a valid topic it still works as expected, so we didn't break anything with that new selector code. Back here in the selector for the LinkListContainer we're currently setting a variable called routeTopicName, and making that available to the LinkListContainer. I actually don't like this variable name. I don't want my component to know that this variable is coming from the route. There's no reason for it to know that, so we change it to topicName. By changing this variable that we're passing onto our components we need to go and adapt our components to go and use the new name and state. We start by going into the LinkList component and changing this variable to topicName over here, and we also need to go and change it in the propTypes, but we're also using this variable name in the LinkList container, so let's head on over there, and select all instances of this incorrect variable name, and change that to topicName. So let's go and test that out in the browser, just to go ahead and make sure we didn't mess this up. So when we reload, great. It still sets the topic name here, and it loads the correct links. Let's go and change the topic. Cool. It's still works.

Redirecting to a Route Based on Server Data When the App Loads

In terms of topics and loading the links for it, there's something that we still need to go and do. Let me show you by adapting this URL, and removing the topic URL part. When we load that up you can see that it doesn't load a topic, and it doesn't display any links. This isn't a great experience, and it's a good opportunity for us to do a bit more complicated stuff with loading default data. So what we're going to do now is make it so that if this route is hit it will go and select defaultTopic and redirect to that. To do that we need to head on over to our Navigation container's sagas. First thing we do in here is we ensure that we import the REQUEST_TOPICS_SUCCEEDED constant. Now that we've got access to that let's go and create a new saga function. So we export a function called selectDefaultTopicSaga, and in here we yield on a takeLatest, and we target the REQUEST_TOPICS_SUCCEEDED action, and we go and say go and execute the logic in the selectDefaultTopic function, which we'll code up now. So let's code up the selectDefaultTopic function. Function selectDefaultTopic, and in here let me write some code. This isn't the actual code, but I want to show you what I want to do. If the state doesn't have a selected topic, then what we want to do is say, yield a put, so go and trigger an action, and that action is the result of calling push, the React router push, and to push we pass a URL that we want to redirect to, /topics, and then we insert the first topic that we can find on states, name. So we expect a topics around state, we get the first one out of the array, and use the name for that to pin onto the URL. Now we don't have access to state here do we? We need to go and find a way to get the selected topic of off state. So far we've been getting state into our components by using selectors, and we're also going to do it here in our saga. We're going to use our existing selectors to go and get some values from state, and luckily redux saga has a way to go and pull data from state, and it's by using the select side effect, so we need to go and import it up here, and we also need to go and import the selector for the Navigation container, so selectNavigationContainer from selectors. Now that we've got a selector function for that we can go and adapt our selectDefaultTopic function, and we go and say that const state = yield on select, and we pass the selector into that, and that will give us the stack that we want. See how well these concepts are starting to play together? We also need to go and ensure that we export the selectDefaultTopicSaga at the end of this module. Let's test that out in the browser now. So I'm on the route URL, and I'm reloading that. Great. It redirects us to a default topic, and it's showing us links and everything. That's pretty cool. I'm excited. We've done all that hard work to show it, let's go and make it look a bit better than what it does now. I'm talking about the name of the topic here in the linkList, so we just wrap this topicName in an h1, and we go back to the browser. That looks better.

Summary

So that's it. We've added routing to our app, and we can now press the back and forward button, and it works. To do this we intercepted an action in our saga, and we dispatched another action using the put side effect to go and instruct our router bindings to go and change the route. That way everything goes through redux, and you can go and use the redux div tools to go and check everything out. What was also interesting for me in this module is that we used the selector functions, together with reselect, to go and pull our route parameters off the route, and make it available to the rest of our components. In the next module we're going to change our app to take some input from the user in order to build a login component. We're even going to add some validation onto the input to go and ensure that the user inputs a valid email.

Building Forms to Gather User Input

Introduction

We've gone through a lot of work already, but we're not taking any input from the user yet. We're going to change that in this module, and build a login component that takes an email from the user. We're not going to wire this up to the server, we're just going to build out the functionality, so that you can see what's involved in taking input from the user, and throwing that into the rest of your app. Although there are dedicated libraries like redux form to help you to work with inputs in the redux world I decided to keep it simple in this module, and build it out ourselves, so that you can see what's involved. What this means in our app is that we're going to have stateful components because we need to keep track of the state that the user is inputting before we can flow it into the redux world. If you use redux form you can still keep on using PS stateless functions, and it will remain working, which is pretty cool. Let's have a look at what we'll do in the rest of this module. We're going to build this functionality here. See when I click on this login link in the app bar it changes to a different route, /login, where we allow the user to logon with the user's email. If I press Cancel here we go back to the previous location. If I provide an invalid email we'll see some validation happening here. We're going to do this by doing a stateful component, and handling some internal state. When I change this to a valid email and click on the Login button you can see that it goes back to the previous location, but now we've got the email that was provided showing here in the app bar, so we'll be doing some selector code to go and get the email that was provided on a different location, and on a different part of state, into the app bar. Cool. Let's go and do that in code.

Getting a New Component to Show on Its Own Route

To build out our login functionality we start by generating some components. First, we're going to generate the login component using npm run generate component. This time we're going to choose an ES6 class because we want to contain some state within the component. It should be called Login. Yes we want styling, and no we don't want internationalization messages. Next, we generate our Container component. We call this LoginContainer, and we don't want headers or styling for this container component. We do, however, want all the redux assets to be generated, and we want to use sagas. Next, we generate the route. We want the route to show the LoginContainer, and we want the route to be /login. Great. That's it for generation. Let's go and make the login functionality work now. First thing we need to do is go to our routes file. We generated a route, and it's mostly okay, but we want this route to be a child route of our main route, our /route that we did earlier, very similar to what we did with the LinkListContainer component. So we select this newly generated route, and we cut that, and we go and paste it into the child routes of our main route. The one other thing that we need to do in here is to give a name to this injectSagas function. This is so that it can keep track of the sagas that it's been injecting to ensure that we don't inject it more than once. So we've set up the route. Let's go and adapt our components a bit, just to go and ensure that it's been slotted in correctly. We head down over to the LoginContainer component. In here we import the Login component from its relative location, and we just go and adapt the render function to go and use this Login component, and we use this syntax to ensure that all the props passed to the LoginContainer component also get passed onto the Login component itself. Now we head on over to the Login component, and we adapt this render function to show a simple message. This is the Login component, so this is what we expect to see in the browser. Now let's go and check that out in the browser. I'm adapting this URL to be /login. When I press Enter, oh no. It redirects us to the libraries topic. Why is this? In the previous module we added some code to redirect to a default topic inside of the NavigationContainer. The problem is that it does it on every route, and we don't want it to do it on the login route, so we're going to add some code to go and target a specific URL, and we'll only redirect to a default topic on that URL, and the way we're going to target that route is by heading over to the navigationContainerReducer file. In here we add another case statement. This time we target @@router/LOCATION_CHANGE. This is the value of the type of the action fired by react-router's redux bindings, so we can use it to look into that action being fired inside of our reducer. When this action gets fired we go and set a value on state called routerLocation, and we set that equal to the action's payload, and on that the pathname. By doing this, this whole NavigationContainer world is aware of the current router location through its store. Let's go and use this value now to go and conditionally redirect to the default topic. Where do we do the redirect? We do it over here in the sagas file of the

NavigationContainer. Here's the selectDefaultTopic function. What we want to do is adapt this if statement by saying that, yes, if it doesn't have a selectedTopic, and on the state we've got a routerLocationValue equal to /, meaning we only want to redirect to the default topic if we're on the route of the application. Let's go and test that out in the browser again, just doing a hard reload just to go and make sure that we've got all the latest stuff, and now I'm changing the URL to /login. Great, and our Login component loads and it displays correctly, so we fixed that bug. Next, let's go and make this login link that we see in the app bar functional, and we do this by heading over to the AppBar component. In here we import Link from react-router, so don't confuse this link with the concept that we've got for links inside of our application. This link is from the react-router, and it allows you to go and create hyperlinks that triggers the routing to happen correctly without a hard reload occurring. Now that we've imported that we go down here to the bottom of the render function, and change this simple label here to an instance of Link, and we add a prop called to. We set that to /login, we make the label login again, and close off the Link element. Let's see what this looks like in the browser. So I'm just going to select a topic here in the app bar just to go ahead and make sure that when we do click on the login link that it does the correct stuff. Okay, now when we click the login, great. It loads up the Login component without doing a hard reload, so that's exactly what we wanted.

Laying out a Form

Let's go and style out our Login component a bit. We want it to look like a sort of material card, so let's go and bold that out. I've copied in some style that I had on the clipboard, just to speed it up a bit. You can see there's nothing funny happening here. All this style is just to go and make it look like a white card with some shadow, and padding, and stuff like that. Let's head on over back to the Login component, and now I want to go and create a div element that will contain a heading for this card-like Login component, and on this div I add a className prop, and set that to styles. heading, and we give it a label, Login with your email, and we close off that tag. And we head on over to the styles file, and I'm pasting in some style to target that heading element that we just created. When we head over to the browser you can see that our Login component is starting to take shape. We've got the white card look that we've wanted, and we've got our heading with its larger text displaying there. Right, now in the Login component again, we want to flesh out this render function, and add the ability to input your email, so we add an input, and we give it a className of styles. input. We give it a placeholder, and we set that to Your email, so this is just a prompt. We add a ref prop, and this ref takes a function with a parameter, and what we do with this parameter is we go and store it on this. emailField by assigning that to the parameter. This is so that we can keep track of this element created through JSX, so that we can pull the value off of this instance later on. We also add a type of text onto this input, and let's close that off. When we go and have a look at this in the browser you can see we've got our input here, but it's in dire need of some styling. We can definitely do better than this. Okay, now I'm just going to paste in some style here. Again, there's nothing weird going on in this style, it's just stark, standard sort of stuff. Let's have a look at this in the browser. Now that looks much better. You see the border at the bottom? That's to give us a sort of material look. Now let's go and create some buttons on this component. We want two buttons, a Cancel and Login button. First, we start by creating a container element for our buttons, a div, and we'll give it a className of styles. actionContainer, then under that we create another div, give that a className of styles. button, and the label for this button we want to be cancel. Then we add another div next to that one with a className of styles. button, and we want that one to have a label of log in. We close off that button, and then we close off the action container. It's going to find some style for these elements that we've just coded up. We paste in some style for the action container. You might be interested to see that this has got a display type of flex, and some space at the top. Then we paste in some style for the button. This is also pretty standard, but you might be interested to see that we're transforming the text with CSS to be uppercase, and we're setting a cursor explicitly to pointer. We're also using this syntax over here to go and say that when it's in a hover state we want the background color to be different. We're going to give it a light gray background, and we want it to have a bold font-weight, and we can use this syntax because react-boilerplate uses post CSS allowing us to add pseudo rules like this. When we have a look at this in the browser, great. That looks pretty cool. That's starting to look like a real component isn't it? If you just hover here over these buttons you see what we did with the background color, and the bolder font-weight on hover state.

Wiring a Form with Validation

Now let's go and make this Login component functional. We start by adding an onClick prop for this div that represents our Login button, and we say that this should execute this. login. Now we go and define a login function on this component, and in here we just logout to the console a simple message, login button clicked. On the browser I just do a hard reload just to go and make sure that we've got the right code. Load it up already, and I press the Login button. Great. As you can see, it's logging on to the console. Login button clicked. Let's go and try and adapt it to go and show us what the value was in the input, so let's naively go and adapt this console log statement, and append this. emailField. value. I say naively because this won't work, but I first want to show you how it breaks. Let's just fill in a value here. It doesn't matter what, it won't actually be able to get it, and then we press the Login button. As you can see, we get an error here in the console, Uncaught TypeError: Cannot read property 'emailField' of null. Our function doesn't have a reference to the emailField. But we did create a ref right? That's because there's a login function, even though it's defined on the class, because we used it on onClick on the button it doesn't execute bound to the instance of this component, and there's a little hack that we can use for this. We can use. bind inside of the prop on the button, but let's use this way. We go and assign this login to an arrow function, and this just binds it to this instance of the class, this weird syntax. I'm still not sure whether I prefer doing it this way or by using a. bind where I go and reference the function, but nonetheless, let's go and test it out in the browser. This time around I type test again, and I press on this Login button. Great. You can see that we had access to the value of the inputs where we logged out the message. Let's go and add some validation for this input field. Just go ahead and make sure that we only execute our logic if the user provided a valid email. We do this by going over to the terminal, and installing using npm, a package called email-validator. Okay, now that that's installed let's go and use it inside of our Login component. We import validator from email-validator, and now in our login functionality we get rid of this existing log statement, and we assign a new constant of email to this. emailField. value, so we're just pulling the value off our ref, and now we go and add an if statement to go and say that if the validator. validate returns false for this email value we want to go and logout a little bit of an error message to the console, so in the browser we go and do a reload, we type in a bad email, and click on the Login button. It's definitely working because it's showing us our error message, but let's test with a correct email. Just typing in my email over here, hendrik. swanepoel@gmail. com, and I'm going to press the Login button. Great. You can see it's not adding any further error messages. So it's not giving us a false error there. That's good, but obviously our user won't be looking at the debug console. We want this component to go and show validation properly. There was a reason that we went with an ES6 class for this Login component. That's because it's going to be a state full component, and we're going to store the error message on this components internal state.

Presenting a Validation Message on a Form

We assign a variable called state to an empty object. I'm using this modern syntax of saying state = object just in the middle of the class, and it might look a bit weird, but all it does is exactly the same as going and saying this. state equals that inside of a function like the constructor. Now that we've done that we go and change this code here to go and say that if the validation failed, and we want to go and set a property on state called errorText, and we set that equal to Please provide a valid email, and if validation failed we don't want to do anything after setting this on state. We just want to return there, and we also want to do the inverse of that. If the validation did pass we want to go and call this. setState, and go and set the errorText to null. Just to go ahead and show that if there was an error, and it was subsequently fixed that the error goes away. Now let's go and use this state inside of render. We create a new constant called fieldError, and we set this to a value conditionally based on this. state. errorText. If we do have error text we want it to be a div with the className of styles. errorMessage. Inside of this div we want to use this. state. errorText, class of. div. If we don't have errorText on state we would just want this to be null. Now we can just go scroll down here to the input field, and add the fieldError in here. Okay, let's go and test that out in the browser. In here we add a wrong email, an invalid email first. Press the login button. Great. You can see that our message is displaying. We're going to style it to look more like an error message soon, but that's something already. Okay, let's go and test the inverse, and go and type in a real email in here, hendrik. swanepoel@gmail. com, press Login, and the error message goes away, so that's exactly what we wanted. As promised, let's go and add some style for this error message. We go to the styles. css file, and login, and I paste in some style here. It's basically just to give some padding, and to make the error red, and this should have already updated with hardware loading, so let me change this email to fake. Press the login button. Great, that error message looks much better, but it would actually be nice that when this login component is in an error state that the text input itself looks a bit more like an error, and to do this is surprisingly complex. We're going to have to do some funky stuff with our class names. Let me show you what I mean. In an earlier module we also used this functionality. We're importing classNames from the classNames npm package that we installed earlier. We're going to use this to go and add some conditional styling to the input element when it's in an error state. So on the input we go and add a class name of, and now we call the classNames function that we just imported, and we go and say, always going to use this styles. input, but then conditionally go and add the styles. inputError based on whether there's a value on state for errorText. Remember it uses this computed property syntax, these square braces over here, as I've explained in a previous module, so what we want is for this class to be added only if the state has an error on it. Let's go and see if that worked, but before we can go and test this we just need to go and add some style for it in the styles. css file. I'm pasting in some rules for input error, and I'm basically just saying that I want a border-bottom-color of red. Let's go see what this looks like in the browser now. It seems our hardware loading already kicked in. As you can see, I've already got the components in an error state, and we've got the red bottom border, so that conditional style worked perfectly.

Getting Form Values onto State

Now we can go and use this newly defined constant to go and create an action creator around it. We changed this import statement to import the LOGIN constant in state, and now we go and define a new action creator function called LOGIN, and this one should take an email as a parameter, and then we return the action. It's of type LOGIN, the constant, and we want a property on it called email with this value passed into the action creator, and then we close that off. Now we head on over to the reducer file, and here again, we change this input to go and use the LOGIN constant instead, and we adapt this case statement to use this login constant, and if this action fires we want to go and set a value on state. We want to set email to the action. email attribute. Okay, great. So we've defined a constant. We've created an action creator function around it, and we've added some code to the reducer to go and handle that action, and store the email on state, but we still need to go and make all this available to the login component, and we do this inside of the LoginContainer component's index file. You'll import the login action from actions, and we want to go ahead and make this available to the LoginContainer component itself, but in the context where it's aware of dispatch down here in mapDispathToProps. We add a login attribute, and we set that to a function that takes email on its parameter, and when this function executes we want to dispatch the login action, which we get by calling the LOGIN action creator with the email that was passed through, and because we passed through all the props that we get on the LoginContainer to the Login component itself we can expect this login function to be passed to the Login component, so let's go and use it there. So inside of the login component the first thing that we want to do in here is add propType rules, so that we can go and validate that we're getting the login function, and I'm doing this by using the static keyword for propTypes, and setting that to an object, and adding a rule for login, and specifying that this is a propType of function, and it's required. So this is just a different sort of syntax that you can use to go and specify propType rules on an ES6 class component. Okay, now that we've validated that we're getting the login function through on props let's go and use it, and we do that down here in the login function. We go and call this. props. login, and we pass it the email that we've pulled out up here onto a constant to that function. Okay, let's go and test that out in the browser. Let me type my real email in here. It's very long, right? Okay, and press on the Login button. Okay, we don't see anything happening, but that's okay because we're just doing some redux stuff, aren't we? So we open up the redux dev tools, scroll up here. Okay, great. We can definitely see that the correct action is being fired, and that it's got the email on there, and if we scroll down here and expand the LoginContainer on state you can see that we're storing the correct email on the LoginContainer part of state, which is exactly what we wanted to do. Okay, so that's all good and well. We're triggering some actions, and restoring the email in state. In the real world we would have probably spoken to a server with this email, and it would have validated the email for us. I'm not going to do server-side communication for the login functionality in this course, but we've still got our work cut out for us on this component. I want to show you enough of the real world scenarios that you would deal with with react-boilerplate, but I don't want to go and add some code just for the sake of it, so what we'll do further on in this module is go and hook into these actions being fired with sagas. We won't be talking to the server, but what we'll do is we'll go and redirect to the previous location when the user presses cancel, and when the user does a successful login, and the email has been stored on state. What we're also going to do is we're going to pull out the email from state, and show it up here in the app bar. So let's go and do that now.

Navigating Back to the Previous Route

Okay, so let's head on over to the LoginContainer, and in here we import the LOGIN constant from constants. We import the put side effect from redux-saga's effects. We're going to use this to dispatch more actions to the redux world. We import takeLatest from redux-saga. Remember, we used this to intercept actions being fired, so that we can hook some logic onto it. Then we import goBack from the react-router-redux bindings. This is so that we can go back to the previous location when the user clicks on the cancel button or when the user manages to logon successfully with a valid email. Now we get rid of this generated code that we've got in here, and start defining the saga logic that we want. We start by creating a generator function called doLoginSaga, and here we want to say that we want to wait around until we get the LOGIN action being fired, and then we want to execute a function called handleLogin, so let's go and define that now. The handleLogin function is also a generator function, and here we say that we want to yield on doing a put, so this means dispatch a new action, and we call the goBack function from react-router-redux to go get the action that we can dispatch, so we're actually triggering a route change by dispatching a redux action. And the last thing that we need to do in here before we can go and test that out is ensure that we export the doLoginSaga, which we just defined. Okay, and let's go and test that out in the browser. I just do a hardware reload just to go and make sure we've got all the latest code. Type in a valid email here and press Login. Great. It redirects us to the linkList, which is what we wanted. So that complicated piece of code that we've done in sagas has paid off. Now when we've got the Login component showing we've also got this cancel button. So far we haven't added any functionality to it. What we're going to do now is hook it up through redux and all that sort of stuff that when you click on it it navigates to the previous location that we've got for this app. Inside of the constants file of the LoginContainer I define a new constant called CANCEL_LOGIN, and we set that to a CANCEL_LOGIN action type. We head on over to the actions file, and import this newly created constant in here, and now I can go and create a new action creator function called cancelLogin, and from this we return an object, and we give that a type of CANCEL_LOGIN, and that's it. That action is just an indication that it was canceled. It doesn't have any other data. Now we need to go and make this action available to the login component, so that when the user clicks on the Cancel button it can trigger this action. We do this by going over to the index file, and LoginContainer, and importing cancelLogin up here. Now in mapDispatchToProps we add a prop to this component to cancelLogin, and this is just the function. When this function executes we dispatch an action that we get by calling the cancelLogin action creator function, so at this point the Login component itself should receive a cancelLogin prop containing this function. So inside of the login component let's go and add some property validation for the cancelLogin function that we expect, and we want this to be required. We scroll down to where we defined the cancel button inside of our JSX, and we add an onClick prop on this div, and we set that to this. props. cancelLogin. We didn't have to define a local function on this component for cancel because we don't need to go and interact with internal state. We just want to trigger the action directly. Let's go and check in the browser that we've wired it up correctly. We're not doing anything with this action yet, but when I click over here, and go and open up the redux dev tools you can see that the cancel login action is being triggered when we click on this cancel button, so that's cool. So we head on over to the sagas file, so that we can go and intercept this action, this CANCEL_LOGIN action, and execute some logic to navigate back to the previous location. Now because we want to do exactly the same thing when we did a login, and when we do a cancel, we just rename this function to handleDone, and rename it to where we call it from the doLoginSaga. Now we define a new saga function, so we create a generator function called cancelSaga, and here we wait around for the latest CANCEL_LOGIN action being fired, and then we execute the handleDone function. Cool, and we just need to ensure that we export this cancelSaga. Okay, let's go and test this out in the browser. When I reload this page the login shows, and we click on the Cancel button. Great. It redirected to where it was prior to hitting the login route, and that's exactly what we wanted it to do.

Selecting Data from Another Container Component's Part of State

Now let me show you what we want to do next. When I click on login here, and I provide a valid email, and click on Login, it redirects to the previous location, but it still shows us that login link there in the app bar. That's not good. We want it to show us the logged in email, the email that the user provided. So let's go and do that now. The challenge is that this is the NavigationContainer's world, isn't it, this whole Navigation, and AppBar, and all that sort of stuff, and the email is stored inside of the LoginContainer world, meaning inside of the LoginContainer's part of redux state. How do we get access to that value on the LoginContainer's part of the state inside of the NavigationContainer world? Well, we go into the selectors file inside of NavigationContainer, and in here we import the selectLoginContainer from its relative location, so we're going to pull the email from that part of the redux state, and return it from the selector functions, so that it's available to the NavigationContainer object, and so also to the Navigation component. And now that we've got a selector function available for that let's go and combine this selectNavigationContainer with the selectLoginContainer. So this createSelector can take another parameter below selectNavigationContainer domain. We go and call the selectLoginContainer, and we adapt this function, this callback function now to contain another parameter, loginSubstate. Now we can combine substate and loginSubstate, so we do an Object. assign on this substate. toJS result, and we combine it with the loginSubstate, so all the properties that we've got on loginSubstate will be added on to the state for the selectNavigationContainer domain. So basically, just combining the two. Let's go and have a look to go and see what's happened there. Okay, so when we reload this in the browser you can see that we're getting an error. I actually knew that this was going to happen, but I wanted to show it to you in any case, so that we can go and fix it. Let's go and fix it now. The problem is here in the selectLoginContainer function. Because we pulled this in from the NavigationContainer logic, but this code executed without the LoginContainer's reducer logic executing, so there's no state for this LoginContainer yet, so what we're going to do now is just adapt this to handle that scenario of where there is no LoginContainer state yet. And in here we just say that if we do have a substate value go and call toJS on that. That's the part where it was getting an error. It was calling toJS on undefined, and if it doesn't just return an empty object. So if it does have the value on state it will go and work correctly with it. It if doesn't it will just return an empty object, which suits our purposes. Rightio. Now that we're going to test that out in the browser again, you can see that at least our app is working again. Let's go and test that it's combining the state properly, and making the email available on the NavigationContainer state, and we're going to do that by changing the AppBar component, and using that value from state that it's getting from the selector function, and showing it here instead of the login link. Because the AppBar gets its prompts from the Navigation component we'll head on over to the Navigation component first, and we'll add a dependency to an email prop being passed to this, and now that we've done that we need to go and do property validation for this new prop. Now we can add an email prop on the app bar, and pass it the value that we're getting on the props of the Navigation component itself. Now we head on over to the AppBar component where we do the same. We just structure the email prop on the function, and we add some validation for this email prop. Now we head on over to the top of this function, and define a new constant called loginLink. We go and say that this value should be the email, so if there's a value coming in on the props for email, if it doesn't have a value we set it to some JSX, which will go and render the link that will allow the user to go and trigger the login logic. Now we can scroll down here to the bottom, and go and replace this to loginLink. So it will go and display the email if it's got it or it will go and display a loginLink. Let's also head on over to the styles for the AppBar. I just want to go and fix a bug that's been bothering me with the positioning of the link the whole time. So I'm setting a margin-top of 6 pixels in the linkContainer. Let's go and check that out in the browser. Okay, so we're not logged in at the moment, so let's go and click this login link to see what happens. Great. We're getting our login component. Let's type in a valid email and go and login. Click on the Login link. Great. You can see that the email is being displayed inside of the app bar, which is exactly what I was hoping for. I'm glad all that worked.

Summary

So that's it. We're taking input from the user, and flowing it through the rest of our app. We took the input from the Login component with its own Container component, and then we made it available inside of the NavigationContainer component, so that we could show the email in the AppBar. So that was also an example of getting data from the one Container component's ______, the LoginContainer component, and to flow it onto another Container component, so that it could display it in its componentry, in this case, on the app bar. In this module we opted to use ES6 class components. We did this so that we could keep track of the values of our input components with references. If you go with redux form you won't need to do this. You can use stateless functional components, and everything will still work. So keep that in mind. In the next module we're going to refactor some of our components, so that we can make our text inputs more generic, and make it overridable in terms of style, and so forth.

Achieving Component Reuse

Introduction

In this module we're going to take even more input from the user, and we're going to do this so that the user can add a new link to the app. We'll achieve this by refactoring the code from the previous module and making it more generic. Let's have a look at what we'll work towards in the rest of this module. We're going to add a new route to our app. When you add /add onto the URL you'll be presented with this form. The interesting part is that you can still see the links in the background, so we're showing this new form component in an overlay while we've still got a bit of context in the background. We achieve this by using nested routes. When I press on the Add button you can see that our validation kicks in. We're going to create a new component called TextInput and we're going to reuse it inside of this Add form, and we're going to make it take props for validation messages and style overrides. We're also going to refactor the existing Login component to use this new TextInput component. Okay, let's go and do it in code.

Getting the Link Form Component to Render on a Route

Let's start by generating a component for our LinkForm. We'll choose ES6 class for this component because it's going to contain some state. Call it LinkForm. Yes we want styling, and no for internationalization. Now we generate the LinkFormContainer component. We call it LinkFormContainer. We don't want headers or styling. We want all the redux stuff. We want sagas, and we don't want the internationalization. Now let's generate the route that's going to show our LinkForm. We specify that we want the LinkFormContainer to show on this route, and we want the path of /add, and we want this component to display inside of our template with our NavigationContainer, all that sort of stuff, so we go to the routes file, and we go and grab the route that was just generated, put it on the clipboard, and move it into childRoutes here next to the one we've got for links, and because I've adapted this injectSagas function, so that they behave as singletons, we need to go and specify a name to go and identify what we're injecting here. Now if we scroll up a bit you can see that we've got another childRoute, /topics/:topicName. This is what we currently use to show the links for a specific topic. If you think about it, what we want to do is add a link to a specific topic, so we're going to copy this route, and adapt it for our add route. Then we scroll down, paste this route in here, and append /add on the end. Let's go and check this out in the browser. Just go ahead and make sure that we didn't break anything. If I add add onto the URL over here, great, it's still loading. Obviously we haven't fleshed out our LinkForm component, so it's not doing anything at the moment. Let's go and flesh that out. So the first thing we want to do is the SAN that we did for all our previous container components. We're going to the LinkForm container, and we import the LinkForm component from its relative location. Then we take this LinkForm component and return this as the component from the render function, and we ensure that we pass on all the props that we get on the container. Now we head on over to the LinkForm component, and in here just pop in a simple message, just so that we can verify that we've got the right component showing. Head on over to the browser. Great. It's definitely working. It's showing our components in here.

Fleshing out the Link Form Component

Now if you think about it, we've already got a form that takes input, and it's got buttons and everything inside of the Login component, so let's head down over there, and put all the stuff on our clipboard, and then go back to the LinkForm component, and paste it in here. Let's just change the class then to styles. linkForm before I forget. Immediately I can see that we're using a classNames reference here, so we need to import that at the top of the file, so we import the classNames function from the classNames module, and because this is a stateful component let's just go and define the state variable and initialize it up here, and let's also go and steal the styles from the Login component, so we go over to its styles file, copy all this, and paste that into the linkForm styles file. Before this will work we just need to rename this to LinkForm up here. Okay, so let's go and adapt it in from component now, so that it looks like a form for adding links. We can change this heading text to add a link, and this input that we've got over here, let's change the placeholder to URL, and on the reference we also store it on urlField. We can get rid of this error reference that we've got here. Doesn't exist yet. Okay, now let's go and duplicate this input, and adapt it for description, so we change the placeholder, and we go and store it on descriptionField, the reference. Okay, and let's go and have a look in the browser. That doesn't look too bad, so we've copied and pasted and started adapting it. We still need to go and extract some reusable fields, but all in all good progress so far. The one thing that's a bit weird on here is that we're going to trigger an add when you've got the links showing for a topic, and then you'll see this thing pop-up, so you'll lose all the context of where you're adding it to. I think what we should go and do is go and change our route, and make it a child route of the actual list one.

Displaying a Component in a Dialogue

So we go to the routes file, we grab our route over here, and scroll up a bit. Let's add a childRoutes attribute on here, make that an array, and now we can paste our route in here. Let me just scroll up a bit. Okay, so you can see that we've got our route, topics/:topicName, which will display the links, and as a child to that we've got this topics/:topicName/add, so this will allow us to go and select this form into the app while the links are still displaying, and then we're going to overlay it over the links a bit later on. Now before it will display we need to head on over to the LinkList component, and in here we add a new prop onto this component, children, and then we need to go and output the children here in the JSX, and because we've added a new prop we need to go and do validation for some children, and I'm making this an element. Okay, so what does this look like in the browser? As you can see, it's got the LinkForm displaying below our links at the moment. Obviously, this is not the best user interface. We're going to fix it, so that it's overlaid over the links, and you've still got the context of the links. Now in the LinkForm component this can create a div around this linkForm, and we give this div a className of styles. overlay, which we'll go and flesh out soon, and we move down to the end of where we return the JSX, and close off this div, and we fix the indentation. Now let's go and adapt this style for this linkForm class. First we remove the box-shadow. We're going to create an overlay, which will be like a shadow behind this whole class, so there's no sense in having it. Let's also remove the margin that we use to center align this item, and now we make it a fixed position item, and we set its top to 25%, left to 50%, and we set it's margin-left to half of its width to center align it. Because it's going to be sort of like a pop-up we're starting to use fixed positioning now instead of relative positioning, so that we can position it over the links. Now let's flesh out our overlay class. We add a height of 100%, width of 100%, give it a fixed position. Ensure that it's above the other elements with this z-index. We add a left positioning of 0, top of 0. We give it a black background color, but a bit transparent, so that we can still see the links through it, and we specify that we don't want it to go and create scroll bars. Okay, and let's go and have a look in the browser. That looks pretty cool. You can see we've got our links in the back, so the user doesn't lose context, and the form is displaying on top of that. Now we've taken some liberties with the copying and pasting to get this component to look like this. Let's go and refactor our code a bit, and go and create some reusable components. We're going to start with the TextInput.

Creating a New Generic Component for Text Input

So for the text input that we're going to create and factor out we generate a component. This one needs to do be an ES6 class because we're going to have internal state to keep track of errors and stuff like that. We call it TextInput, we definitely want styling, and we don't want internationalization messages. We go and find our TextInput that we just generated in our editor, and we're going to do it a bit different this time around. We're going to start with the propTypes, so TextInput. propTypes. We've got an errorText, so if the validation failed, and we want to display an error for this component, and this is type string. Placeholder, what is the placeholder text that we want in our input element? And class name, so this is if the component that uses the TextInput wants to override some style. We're going to use the classNames function again, so let's import that from the classnames module. Now for our error state we go and define a constant, errorText, that we go and pull from prompts. Now we go and define a new constant called fieldError, and then we go and do a conditional statement. We go and say, if we've got errorText being passed through on the props we want this to be a div with the className of styles. errorMessage, and we want the errorText to display in here. We close off the div. If errorText wasn't passed in we'd just set it to null, and let's remove this className from this container div. We're not going to need that, and now I'm pasting in an input. We've done this input inside of the Login component and LinkForm components, so you should be familiar with it, but let's go over it quickly. For the class name we use the classNames function. We use styles. input as default, and we go and add the className that we're getting from props over there. Then we go and add some conditional logic to go and say that if we've got errorText coming through on props we should add the inputError class as well. We set the placeholder equal to the placeholder prop. We add a reference to keep track of this field, so that we can get the value later on, and it's of type text. We also want to display the fieldError that we calculated a bit earlier. Now this is a reusable component, and we've added this reference to keep track of the input field, so how will the value of this input be made available from this TextInput component? Easy. We go and create a function called value, and from this we return the reference, value, so any component that uses this TextInput component will just have to keep a ref of its own, and then it can go and call value on the ref to go and get the value off this input contained inside of this component. Now we have defined some styles already for inputs and errors and all that sort of stuff. Let's head on over to the LinkForm's styles, and copy this part over here, the errorMessage and inputError, paste it inside of the styles of the TextInput component. Head on back again, to the LinkForm styles. We want this input class to move over as well, so we go and paste it inside of the TextInput styles, and while I'm here I'm going to add a rule to this className to go and specify some CSS for its focus state. We want the outline to be none, a border-bottom-color of gray, and a nice thick border-bottom-width of 2 pixels.

Using the New Generic Text Input Component

Let's go and use this TextInput component now inside of the LinkForm component. First, we need to import it. Now that we've got it imported we get rid of this old input code, and we start creating a TextInput. This one's for the URL, so we specify a place on the ref URL. We duplicate this, and this one's for description, so a placeholder of Description. So in the browser when we refresh it looks the same, so that's good. When I click on the URL field you can see that it's got our new style with a border bottom there that we defined inside of the TextInput component, and you know what, I'm also going to open up the debug tools, and open up the React debug tools. I'll try and select this URL field, inspect it. If we go back to the React debug tools here you can see that it's definitely using the TextInput component, so now that we've refactored out of the TextInput component let's go and get rid of some of the styles inside of the LinkForm that we're not using anymore. So you have already cut and paste the errorMessage and that stuff, so it's basically just the input className. If we go back to the browser here you can see it doesn't look too great at the moment, these multiple fields below each other. We definitely need to go and add some spacing between them. The interesting part about adding this spacing is that we're going to do it inside of here, the LinkForm's style. We're recreating an input className, but this time with a different purpose. This is to style our inputs in the context of the LinkForm, nowhere else, so we specify a margin-top of 10, and a margin-bottom of 10 pixels, so to be clear, I'm not defining these styles inside of the TextInput styles, I'm defining it inside of the LinkForm, and we're going to use it now to go and add to the styles of the TextInput in this context. And now that our styling is a bit simpler inside of the LinkForm we can actually remove this classNames import up here, and now we can go and define a className prop on these text inputs, and this we set to styles. input, which we just defined for the margin top and bottom. We copy that, and paste it onto the TextInput below. So why am I setting a class name prop on this TextInput? You might have forgotten. Let's head on over to the TextInput quickly, where I'll show you what this is setting. See here where we're defining the input we're using the classNames function, and we're saying, use styles. input, which is defined on its own styles. css, and then use the className prop that we can expect on this component to go and add onto that class, so we've got our default styles defined inside of this TextInput, styles. css, and then we allow any users of this component to go and add onto this style, and even override it using a prop called className. Now when we go and have a look in the browser you can see that we've got some nice space below the TextInputs.

Changing the Login Component to Use the Text Input Component

Our Login component is still using a low level input, and all this logic here to go and define its input, and we would rather want to use the TextInput that we just defined, so let's start doing that. We scroll to the top of the file, and import the TextInput. Now in the render function we can actually get rid of this fieldError logic. All that sort of stuff is contained inside of our TextInput component. We can get rid of this fieldError down here, and now we can start replacing this input with a textInput instance. Actually, I'm going to adapt it to be a textInput in state because we can reuse some of its props, so I rename it to TextInput, get rid of this className stuff, we leave the placeholder, and we leave the reference, and we want an errorText prop to be set to this. state. errorText, and now because we removed all that className stuff we can scroll up to the top of this file, and remove the classNames input, even the styles of the Login component can be simplified. We try and find input. There it is. We remove that, and we can go down to the bottom here, and remove these classes as well. Let's go and have a look in the browser. When I click on the login link you can see that our component shows, and when I select this item, great, we've got this border at the bottom. Now let's try and login with an invalid email. Great. As you can see, we've got a validation message. Now here's going to be the catch. I'm going to try and login with a valid email, and when I press the Login button you can see that it still thinks I've got an invalid email. Let me go show you how to go and fix this issue. Here in the Login component where we used the ref to the TextInput that we've used for email we're just saying. value, so this is actually returning the function that we've added on the TextInput component, so we need to go and add some parentheses here. Now let's go back to the browser. I refresh, and type in my valid email, press the Login button, great. So we've wired up our TextInput correctly inside of the Login component.

Summary

Great. So we've got a generic TextInput component. We can override the styles through props, and we can set an errorMessage on props, and it just displays inside of the component. A key takeaway for this module is that if you want to do a dialog sort of overlay you need to use nested routes, otherwise, you'll lose the context in the background, like the LinkList in our example. In the next module we'll tackle the rest of the work that we need to do in order to make the Add LinkForm functional.

Tackling a Realistically Complex Feature with Your New Skills

Introduction

Now we're going to tackle the rest of the work to allow the user to add a new link to our app. First, we're going to code up another generic component, an IconButton component. We'll use this to allow the user to start the process to add a new link. We'll also use this new component inside of our AppBar component to show the drawer icon. In the end, we'll link our form up to redux, and ensure that the links make their way all the way to the server, and immediately show inside of our app. See this button over here? When I click this we'll get presented with an overlay form, meaning you can still see the links in the background, and you've now got a form that allows you to add in your link. What's interesting is that you can see that this is a nested URL, so that's how we're getting the links to stay in the background with this form showing over it with a different route. If I click the Add button without providing input you can see that we've got validation similar to the previous module, but we're actually going to make these textInputs reusable, making them work with validation, and taking some override style. When I provide some valid input into these fields we immediately see the newly added link inside of the LinkList, and what's interesting is when I hard reload the page you can see that the new link is still being displayed. That means we actually spoke to the server to go and save this new link. Let's go and do this in code now.

Creating an Icon Button Component

Now we're going to create a generic component for our icon buttons that we want to use in our app. We generate one on the command line. We want the Stateless Function. We want it to be called IconButton. Yes it should have styling, and no we don't want internationalization messages. Inside of our icon button component we import FontAwesome from react-fontawesome, so that's what we use for icons, and we've actually defined some good markup for our icon button already inside of the AppBar component, so inside of the AppBar component we grab this spot, we go back to the IconButton component, and paste it in here. Now we can start making this a bit more generic and reusable. We're going to get the click handler through on props, and we call that onClick, and then we wire up this onClick to that prop. We do the same for the icon that we want to display on the button, so we add a prop for icon, and use it here on the name prop where we use FontAwesome, and now we can go and specify some propTypes for these props that we've just defined. We create one for icon, and this is of type string, and it's required, and onClick. This is a func, and it's also required. When we save that out our linter is happy. Let's go and define some styles for the IconButton. We want the text to be aligned in the center, and we want it to have a cursor of pointer, so it needs to look clickable. Now let's head on over to the LinkList where we want to use this new icon button to allow the user to add a new link. We import

IconButton from its location, and now I can go and use it down here in the JSX part. Between linkNodes and children I add IconButton, and I specify an icon prop on it, and I want plus icon to display. Let's close that off. So here in the browser we've got our add button displaying, but it looks horrible. It must be one of the most user unfriendly buttons ever. So what we want to do is make the generic IconButton able to take classNames through its prop, like we did with the TextInput, to add to its styling, so that it makes sense in its context.

Extending the New Icon Button Component with Style

To do this we start by importing the classNames function from the classNames module. This is what we always use when we want to combine classnames for a component. Now we add two props onto this IconButton component, iconClass and buttonClass. We want this component to take these two props for overrides because you might want to override how the icon itself looks, and you might want to override how the button around it looks, and because we're depending on new props now, guess what we need to go and do. Yes, you've guessed it. Property validation. So I specify a rule for iconClass, which is type string, and buttonClass, which is also a type string. None of these are required, and now we use the classNames function over here to go and say that we want to combine our own internal iconButton class with the buttonClass that might have been passed through on the props, and on the FontAwesome instance we specify that the className should be a combination of classNames and iconClass. You know what, we just want to actually say that we want to use for the className on the FontAwesome instance, this iconClass prop that's being passed through. Now we can go and use these props inside of the LinkList component, so we go and set buttonClass to styles. button, and iconClass to styles. icon. Now we can go and define that inside of the CSS. I'm sure you didn't want me to go and type it up bit by bit. We've got some shadow run on the button. We've got border-radius to make it a round button. We've got some display rules to go and say display it as an inline-block. Position fixed, and then we specify some fixed positioning, so 30 pixels from the right, 110 from the top. We give it a pink background color, give it some nice padding, and then we override the padding for the left and right. Then we specify a hover state for this, and we give it a different background-color. For our icon we just specify that we want a white color for it. Let's go and have a look in the browser. Great. So our button is displaying here on the LinkList, and it's sort of like a floating action button that you get with material designs. When we hover over it you can see that it's got this nice effect where it changes background color. Cool. So our icon button is working. Let's also go and use it inside of the AppBar now where we've got this hamburger menu. Let's remove this FontAwesome reference in here, and replace this with an import for the IconButton instead, and now let's go and replace the FontAwesome code down here with IconButton code, so we change this to an IconButton, we specify an icon of bars, buttonClass, we use styles. iconButton, iconClass, styles. icon, and onClick we use toggleDrawer. We also want to remove this wrapping div around it. We don't need it anymore, just the icon button contains all this stuff for us. Okay, let's go and see if this worked in the browser. I reload, and the icon button is still displaying there, so that's a good start, and when I click on it it brings up the app bar, so it's definitely hooked in correctly.

Triggering an Action to Show the Add Route

Now let's go and make the LinkListContainer button functional. We want to do this through redux, so we head on over to the constants file inside of the LinkListContainer. In here we define a new constant called START_ADD, and give it a unique string. Now we can go and define an action creator using this constant. We do this by going to the actions file, importing the newly created constant, and now we start creating a function called startAdd that takes a topicName, and this will return an object with the type of the START_ADD constant, and we'll also have an attribute containing the topicName. So in the LinkListContainer we can now go and import the startAdd action container, and now we should go and make it available to the LinkList component. We do this by scrolling down here to the mapDispatchToProps function, and we add startAdd as a prop, and we say that this is a function that takes a topicName, and when it's executed it should dispatch the result of calling the startAdd action creator with the topicName. Now we can go and use it inside of the LinkList component. First, we need to go and make it available on props, and now that we've got it on props we can go down to the IconButton code, and add an onClick, and we say that when onClick executes we want to go and call the startAdd function with the topicName that we got through props, and now that we've got a new prop that we depend on we need to go and add it to propTypes. So I startAdd, it's a PropType of func, and it's required. So let's just test this out in the browser to go and ensure that we've wired up everything together. We're opening the redux dev tools, and I'm pressing Add. Okay, it hit the breakpoint there. I'm just going to continue there. Back in the redux dev tools let me try and scroll down here. Okay, so here you can see that we've got the START_ADD action being fired. We don't do anything with it yet, but let's go and do so now. We're going to hook into it, and go and execute some logic, and this logic we're going to execute in the sagas file inside of the LinkList container, so we head on over there. We want to ensure that we import the START_ADD constant, which we defined, and we also want to import push from react-router-redux because we want to trigger a URL change when this add button is clicked. Now we start creating a new generator function called startAddSaga, and here we yield on a takeLatest, and we say that when START_ADD is fired we want to execute a function called startAdd. Now let's go and build out the startAdd function. This one takes an action, and we yield on a put, so put is what we use to go and dispatch actions, and the action we get is by calling the push function that we got from react-router-redux, and we pass a URL to that using the topicName attribute on the action to go and build out the URL, so we're saying intercept the START_ADD constant, go and yield a put to go and dispatch an action, which we get by calling push. So in the end we want it to redirect to a different route. We also need to remember to go and export this saga that we've just created, so that it can go and intercept on actions. In the browser we do a reload, and I click on that Add button, and it works. It dispatched the action and everything, and that resulted in a URL change, showing us our Add form.

Hooking up Validation for the Link Form

Now at last we're in the position again, where we can go and make our form that allows us to add links functional. The first thing that we want to do in here is go and expand from this default state. I'm adding default state for urlError, an empty string, and descriptionError, also an empty string, so we'll be using these state variables when we do validation in our form. We'll go and set their values, and pass them on to the TextInput instances, so that they can go and show their error messages. Let's go and set up prop errorText on this first TextInput, and we set that to this. state. urlError, and we do the same for the description TextInput, errorText, and we set that to this. state. descriptionError. We also want to keep track of these TextInputs; otherwise, we won't be able to get their values, and what's the use of TextInputs without getting values? So we add a ref on this first one, and we say that this is a function, and we go and store the field being passed in here as this. url. We go and do the same for the second input. Again, we pass a function, and we say, go and set this. description to the field being passed in. Now let's go and wire up the behavior where we get the values from these TextInputs. We start by defining an onAdd function on this component, and let's just log out a simple message in here, and then we scroll down to where we've got our buttons. On this button that's currently labeled login we go and say, onClick should execute onAdd, and let's also change its label to add. Let's go and test that out in the browser. I reload this, and press the Add button. Great. You can see in the console our log message is working, so let's go and change this to go and get access to the fields. So let me get rid of this console. log, and now we define a new constant called url, and we set this to the value that we get from the url ref. We do the same for description, setting a constant, and getting the value from the description reference. Now let's work with errors. We create a variable, urlError, and we set that to null, and another variable, descriptionError, and also set that to null. Now for the validation of the URL I'm just pasting in some code in here, and I'm saying if the string that we go from url doesn't match this whole regex, then we set the urlError to, please provide a valid URL. I'm sure you didn't want to watch me type in that whole regex, so that's why I pasted it in here. You know what, let's do the same for description. Pasting in some code here, and it says that if we don't have a description set the descriptionError to, please provide a valid description. Now to go and use these variables where we keep track of validation errors we go and say that if we've got a urlError or a descriptionError go and setState, and we go and set the urlError, and the descriptionError. We're using the shorthand syntax because we want the state variables to be the same as these variable names, and if we had an error after setting state just return. Don't go on with this logic. Now when we go and test that out in the browser I'm clicking Add without providing any input, and as you can see, both our error messages get in, and when I change the description to something and press the button the error message goes away, so the inverse also works. When I type in an obvious non-URL and press Add the error message remains. Let's test with a valid URL now. This time around we've got a bit of a weird behavior. This is obviously a valid URL, but it's still displaying us an error message. Well, I've got a good idea of why that's happening. Let me go and show you. We should actually move this setState outside of this if statement. We want to return if we had an error and not execute any further logic, but we always want to go and set the state with this because if we don't have an error, then it should overwrite it if it previously had one. Let's go and test this out. So I press Add again, and get the error messages. Now I provide a valid URL, and that error message goes away. Now for the big test. If I go and add all valid fields do we end up with no error messages? Yes we do, so that change resolved that issue. Let's go and make this form functional now.

Getting an Action to Fire Containing a New Link

To make this LinkForm functional we need to get some props through to it that contains the actions that we can trigger to send through the data, so I'm adding propTypes onto this LinkForm component, and we need to do it here before state. We specify a propAdd link, and this is of type function, and it's required. Let me just fix the indentation there. Now to go and make this function available to this component we need to go and do some redux stuff, so we can head on over to the LinkFormContainer's constants file, and adapt this default constant to be ADD_LINK, and because we removed this default constant we need to head on over to reducer, and go and remove the input, and the case statement for default action, and now to go and define the actions for it. In the actions file we import the ADD_LINK constants instead, and we change this function to be addLink, and it takes a link parameter. The type will be ADD_LINK, and we want to store the link on the action itself. Now to go and make this addLink action creator available to the LinkForm itself we need to go into the LinkFormContainer component, and import the addLink action from the actions file. Once we've done that we can go down here to the mapDispatchToProps function, and go and specify that we want a prop called addLink to be a function that takes a link. When this function gets called we want to dispatch an addLink action, and we pass the link onto the addLink action creator. Okay, so now we've made this action available to the linkForm, so let's head on over there, and go and use it. So we've already added prop validation for it. Let's just go and call it. After this if statement down here where we check our urlError and descriptionError we add some code to go and say, call the addLink that we get on props, and we pass an object to that with the url and description on it, so that's our link that we're passing to this function, so let's go and test that in the browser. Okay, so we reload, and I type in a URL in here, and a description, and click the Add. That validation is just too good. Let's change this URL to abc. Okay, when I press Add now I'll show you here in the redux dev tools, there you go. There you can see that the ADD_LINK action is being fired, and that it's sending a link through on the action itself. Let me just expand that link. There you can see that the links got all the right fields too. So let's go and intercept this, and send this through to the server.

Saving the New Link to the Server

Now to create a new link on the server I thought it high time that we create a dedicated place in our app to go and store logic that talks to the server. I'm creating a new file in api/index, and in here we're going to define a function that talks to the server to go and create a new link. So we export it, and we call it createLink, and it takes an object, and this object represents a link. It should have a topic name, a URL, and a description. We then return a promise by calling fetch with this URL, localhost3000/api/topics, and then we specify the topicName that we got through on this object, /links. We specify that this fetch should be made with the method of POST. On the fetch headers we specify that we accept JSON, and that the content that we're sending through is also JSON. For the body we specify JSON. stringify, and we serialize the link object, so the url, description, and topicName. We chain a then onto this fetch promise, and we say that we want to take the response, and we want to pass the JSON into JavaScript. Okay, so we've got a function that we can easily import and call to go and create a link on the server. Now to go and use this API function we head on over to the sagas file in LinkFormContainer. We import the ADD_LINK constant from constants. We import the takeLatest function from redux-saga, so that we can intercept actions. We import the call side effect from redux-saga, so that we can trigger the fetch logic, and we import the createLink function that we just set up inside of the API folder. Now let's get rid of this defaultSaga, and create a new saga called addLinkSaga. In here we yield on takeLatest, and we intercept the ADD_LINK actions, and when that happens we execute a function called addLink. Now we create a generator function, addLink, which takes an action. We do a try. We say, yield on the call, the call side effect, and we pass it the createLink function, and as parameters we pass the link that we got on the action, and then we type up our catch block, and close it off. Then we just ensure that we're exporting this addLinkSaga function, so that it can hook into the actions. As you might have noticed, in the createLink function in our API folder we expected a link to pass through a topicName, and currently we're not sending through a topicName from the LinkForm, so we need to remedy that, so we're going to say that we expect to get the topicName on the LinkForm's props, so that we can send it along. So we just start with the propType, and now that we can expect a topic named prop we can use it down here where we call addLink. We add topicName onto this object that we passed to addLink function, and we set its value to the value of the prop being passed through, but how does this component get this topicName? We still need to wire it up, so that it comes through on this component's props, and that we're going to do in the LinkFormContainer's selectors function, so I'm defining a new selector function called selectRouteTopic, and this expects to get the state and props from this component, and for the results of this selector function we're just saying that we want to return the topicName that came through on the route params on the props of the LinkFormContainer. Remember react-router passes through property parameters to the component that the route has been configured with. That's why we're accessing props here. Let me just fix the indentation there. Okay, so now we can go and use the selectRouteTopic selector function, and go and combine it into the selectLinkFormContainer selector. So we just add a new line in here, and we call the selectRouteTopic function to get that selector function, and now our callback can expect a topicName to come through here, so we add a topicName parameter, and we want to combine this substate and topic name, so we do an Object. assign. For the first parameter we pass the results that we get from calling toJS on substate, and then we say, on that go and add the topicName. So in the end we get a combination of what we always used to return from this selector, just a state for this container component, and the topicName that came through on the route for this component. Let's go and test this in the browser. I do a reload, just to go and get the latest code, click on Add, and type in a valid URL, and just give it a description, and click on the Add button. Now in here in the Network tab you can see that it's actually doing a post to links, so that actually worked, and when I do a reload, oh, I need to go and remove this last part from the URL, then great, you can see that our new link is displaying here. So we've done a lot of work to get a link to be created on the server. Just a reminder that a lot of that work was to go and factor out generic components where we started duplicating functionality. Now let's go and do the rest of the story of adding links.

Showing a Newly Created Object Inside the App

As you can see here in the addLink function in my sagas file, we're currently doing the call to go and create the link on the server, but then we're not doing anything after that. We actually want to go and trigger actions from here, so that our app can either go install the new link if it was successful or go and optionally provide an error message to the user if it failed. So let's go and create these actions now, and we do this by going to the constants file. I'm duplicating the ADD_LINK constant, and adapting it first for success by pinning SUCCESS on here, and then for failed by and pinning FAILED on here, and now we can go over to the actions file where we need to import the constants that we just defined, ADD_LINK_FAILED, and ADD_LINK_SUCCESS. So let's go and create some action creators for this quickly. We create an addLinkSuccess action creator that takes a link. This returns the action with the correct constant on its type, and the link as well. Then we create an action creator for addLinkFailed. This one takes a link as well, and a message. Now we return an action of type ADD_LINK_FAILED, we store the link on there, and we store the message on there. Now in the sagas file the first thing we do is we also import the put side effect because we're going to dispatch actions, and I'll import the action creators for the actions that we want to dispatch, addLinkSuccess, and addLinkFailed, and we also import a function called goBack from react-router-redux, so this is also an action creator that you get with react-router-redux, which you can call to get an action that will instruct it to go back to the previous URL. Because we want to dispatch the link that was created on the server, we go and assign the result of this yield call to a constant called serverLink. Now we can go and dispatch and action that we get by calling addLinkSuccess, and passing the serverLink to that. Do you remember, addLinkSuccess takes a link as a parameter, and after this we dispatch another action. This time the action that we get by calling the goBack action creator, so what we want to do is tell react-router to go back to the previous location after we've recorded the new link. In our catch statement we go and dispatch an action that we get by calling addLinkFailed, and we pass the link, and the exception message to that, so that should be it for the sagas file. Now we want to go and do something with the actions that we dispatch on success and failure of adding links. This we want to do in the linkListContainersReducer. I know we're in the LinkFormContainer now, but the linkList is interested in when links are added, so in the LinkListContainer's reducer we import ADD_LINK_SUCCESS from the LinkFormContainer's constants, and because the logic that we want to do is a bit more complex than usual, when we reduce actions I'm creating a function to do it, and I'm calling this function addLink, and it's got two parameters, state, and link. State is the existing state, and link is the link that was added that came back from the server. In this function I create a new constant called Links. I set this to the value that we get when we call get on state, saying that we want the current links. To these links we push the new link, and then we say what we want to return is a mutation of state where we override the current links with this new link array, and now that we've got a function we go and add a case statement that calls this function, so when the action that's being fired is ADD_LINK_SUCCESS we go and say that we want to return the result of calling addLink, passing in the current state, and the link that we get from action. Let's just fix the indentation here. Let's go and test this out in the browser. So I'm reloading. We've got the latest code, and now I'm going to add a new URL, and a description, and press Add. Great. You can see that worked. Our new link is displaying here at the bottom, and it's gone back to the previous location, so that's pretty cool.

Dismissing the Add Link Dialogue

When we show the ADD_LINK by clicking this button we can't cancel it at the moment. We still need to go and fix that, and we're going to do that by creating an action and making it available to this component so that it can trigger that action. In the LinkFormContainer's constants file let's define a new constant called ADD_LINK_CANCELLED, and this should be app/LinkFromContainer/ADD_LINK_CANCELLED, and now we can head over to the actions file where we import the ADD_LINK_CANCELLED constants. With that constant imported we can now create an action creator around it called addLinkCancelled, and we return an action form this of type ADD_LINK_CANCELLED. Now in the index for LinkFormContainer we import the addLinkCancelled action creator from actions, and now you're in mapDispatchToProps. We make this action creator available to the link form by saying, addLinkCancelled equals a function, and when you call this function we'll dispatch the result of calling the addLinkCancelled action creator. So we've made this action creator available to the LinkForm component. Let's head on over to the LinkForm component, and go and add the PropType rule for addLinkCancelled. It's a function, and it's required. Now when we scroll down here to the bottom where we define our buttons, and on this cancel one we just say, go and execute addLinkCancelled instead of this one. Just to show you that we're firing this action correctly in the browser, I open up the add form, and I've got the redux dev tools open now. Now I go and click this Cancel button, and just scroll down here. You can see that the ADD_LINK_CANCELLED action is being fired correctly. Now we can go and intercept it, and hide this form. The first thing that we need to do here in the sagas of the LinkFormContainer is to import the ADD_LINK_CANCELLED constant, and I can go and create a new saga function called addLinkCancelSaga, and we yield on takeLatest. We're interested in the ADD_LINK_CANCELLED action, and we want to execute a function. We go and say, go and put, meaning dispatch, the result of calling the goBack action creator, so we're just dispatching our action using the side effect functions, and this will hopefully take us back to the previous location. We also need to export this saga. Now when I reload this and press Cancel, great, it closes the dialog, and it goes through the previous route.

Summary

So that's it for this module. We've now got the ability to add new links to the app, and we did this in a nice, elegant way by defining some reusable components that we used throughout our app. The TextInput component that we defined also has validation baked in, and everything works well through props. We also started moving some of the logic that talks to the server into a dedicated file that we called API. It's probably a good idea in your real world app to move all the code that talks to the server into a file like this, so that you can see all of it in one place, and maybe build it out from there, and modularize it. So what's next for you? Well, I've got good news for you. If you want you can go and build out the voting functionality in the app. The code base that you've got on your machine that you cloned from my repo already has code on the server-side that will work for voting, so you can just go and call the API. Let's have a look at this call, so that you can go and use it if you want. Have a look here in the api. js file under server, middlewares. Here you can see that we've got some code that allows the user to vote on a link. Basically, it takes the links ID on the URL, and it expects an email and an increment to be passed through on the body. I know it's not super secure, but there's a reason I did this. I wanted you to be able to play around with getting errors from the server, and handling it properly, so that's why I coded up some logic to prevent double voting based on email. So to go and test out this call we first need to get a valid ID for a link in our app. Luckily, this is easy to get using the redux dev tools. We expand our state, and go and have a look under the LinkListContainer state. Expand links, and expand this first link. We get this ID and put it on our clipboard. Now that we've got a valid ID we just go and plunk it here into our URL, so in the end we've got /api/links, the links ID, /vote, and we sent this through as the body. An increment of 1, and an email. I just need to adapt this email here because I've already voted on this link using this email. Now when I click Send you can see that we get a valid response back with the updated link. When I press Send again, we get the Forbidden HTTP code coming back, so that's what you can code against to test out your error logic. And just to show you, when I go and reload the app in the browser this vote count is now 2, so it's definitely updated properly. So that's it. Go and use this call. Go and test your new found knowledge of this stack, and please don't forget to give this course a good rating if you did find it useful. I'd really appreciate it. Thanks, and goodbye.

Course author

Hendrik Swanepoel

Hendrik kicked off his career in 2000 installing MS Outlook on PCs in coal mines throughout South Africa. Seriously. He had to wear a hard hat and everything. Luckily he got his foot into the door...

Course info

Level	Intermediate
Rating	★★★★☆ (200)
My rating	★★★★★
Duration	3h 48m
Released	27 Oct 2016

Share course