Wednesday, August 28, 2019 12:12 PM

Redux - State management library for React

Redux builds on Flux's unidirectional data flow pattern.

Why Redux?

- Redux centralizes all of the application's state in a single store.
  - Unlike flux alternatives, redux enforces keeping all state in a single, centralized object graph.
  - This makes the app easier to understand and avoids complexity of handling interactions between multiple stores.
  - It also helps to avoid storing the same data more than once.
- Redux requires less boilerplate than Flux.
  - The container components are subscribed to the Redux store automatically, so you don't have to wire up event emitters to subs cribe to the dispatcher. There is no dispatcher at all in Redux.
- Redux's architecture is friendly to server rendering the React components, commonly known as isomorphic or universal js.
- Redux uses an immutable store, which has a number of benefits, including performance.
- Redux supports Hot Reloading by which we can see our changes immediately in the browser without losing client side state.
- Redux also has Time travel debugging, which allows us to step forward and backward through state changes in the code and even replay interactions.
- Redux is small and has a small API which weighs only 2k.

Advantages of redux over flux in terms of boilerplate

Environment build

- **Webpack**
  - We'll use Webpack to bundle or compile JavaScript into a single minified file that works in the browser.
  - Webpack also includes a development web server, so we'll serve our app locally during development via Webpack.
  - Webpack is also used by create-react-app behind the scenes.

- webpack.config.dev.js

```
const webpack = require('webpack');
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

process.env.NODE_ENV = 'development';

module.exports = {
    mode: 'development',
    target: 'web',
    devTool: 'cheap-module-source-map',
    entry: './src/index',
    output: {

    }
}
```

  - We'll set the dev tool to cheap-module-source-map.
  - There are a number of dev tools to consider, but this one is generally recommended for development so that we get a source map for debugging.
  - Remember, source maps let us see our original code in the browser.
  - Because we're going to transpile our code with Babel, source maps will let us see the original code that we wrote when we view it in the browser.
  - We'll declare our apps entry point, which will be in the src directory, the index file that was created.
  - And we can omit the .js on the end. This is also the default for Webpack, so we could leave it out.
  - Next, we can declare where we want Webpack to output. But Webpack doesn't output code in development mode. It merely puts it in memory.
  - However, we do have to declare these paths so that it knows where it's serving from memory.
  - So we can say path.resolve here and use the __dirname variable that gives us our current directory name and then say build right here. So although it's not actually going to write a file to build, in memory, it will be serving from this directory.
  - And the public path we will set to a slash. This setting specifies the public URL of the output directory when it's referenced in the browser.
  - We'll set the filename for our bundle to bundle.js.
  - Again, a physical file won't be generated for development, but Webpack requires this value so that our HTML can reference the bundle that's being served from memory.
- webpack.config.dev.js

```
const webpack = require('webpack');
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

process.env.NODE_ENV = 'development';

module.exports = {
    mode: 'development',
    target: 'web',
    devTool: 'cheap-module-source-map',
    entry: './src/index',
    output: {
        path: path.resolve(__dirname, "build"),
        publicPath: '/',
        filename: 'bundle.js'
    },
```

  - Webpack devServer configuration is as follows:
    - We are going to set stats to minimal. This reduces the information that it writes to the command line so that we don't get a lot of noise when it's running.
    - We need set overlay to true, and this tells it to overlay any errors that occur in the browser.
    - We also set historyApiFallback to true, and this means that all requests will be sent to index.html. This way we can load deep links, and they'll all be handled by React Router

```
process.env.NODE_ENV = 'development';

module.exports = {
    mode: 'development',
    target: 'web',
    devTool: 'cheap-module-source-map',
    entry: './src/index',
    output: {
        path: path.resolve(__dirname, "build"),
        publicPath: '/',
        filename: 'bundle.js'
    },
    devServer: {
        stats: 'minimal',
        overlay: true,
        historyApiFallback: true,
        disableHostCheck: true,
        headers: { "Access-Control-Allow-Origin": "*"},
        https: false
    }
}
```

  - Webpack plugins

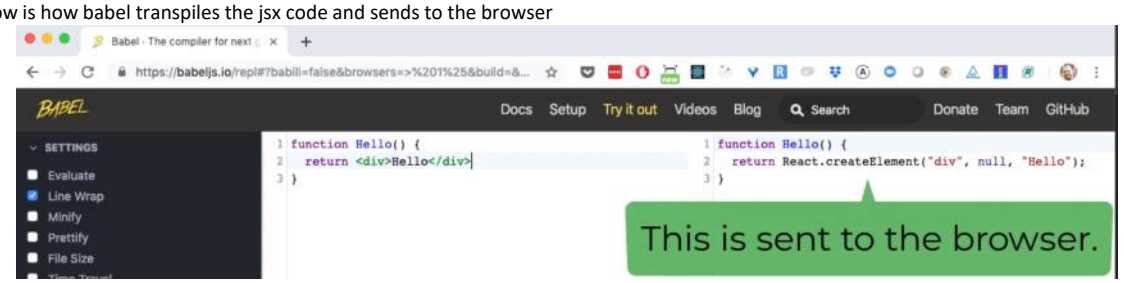
    - ```
      plugins: [
          new HtmlWebpackPlugin({
              template: "src/index.html",
              favicon: "src/favicon.ico"
          })
      ],
      ```
        - For plugins, we need to specify an array.
        - To start, we're going to configure the HTML Webpack plugin that is imported at the top of the file.
        - It accepts an object for us to configure the plugin. We're going to tell the plugin where to find our HTML template, which is in the src directory and index.html and also tell it where to find our favicon, which is in that same directory.
    - Webpack modules:
        - ```
          module: {
            rules: [
              {
                test: /\.(js|jsx)$/,
                exclude: /node_modules/,
                use: ["babel-loader"]
              },
              {
                test: /(\.css)$/,
                use: ["style-loader", "css-loader"]
              }
            ]
          }
          ```
        - We tell Webpack what files we want it to handle, and we do that by declaring an array of rules.
        - The first rule is going to be for our JavaScript. So we will tell it how to find our JavaScript files or the jsx files.
        - And we then tell webpack to ignore node_modules since we don't need it to process any files in node_modules.
        - Finally, we can use the use property to tell Webpack what to do with these JavaScript files. We want to run Babel on these files, which we will call Babel-loader which will run Babel on all of our JavaScript, and Webpack will bundle that up for us.
        - Also we mention the combination of css- loader and style-loader will allow us to import CSS just like we do JavaScript, and Webpack will bundle all of our CSS into a single file

- **Babel**
    - 

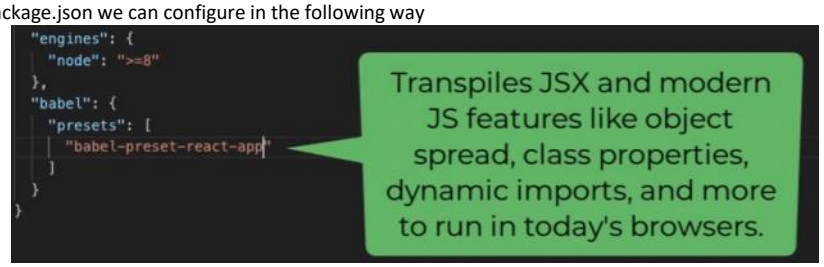
    - Babel is a popular JavaScript compiler that lets us use next-generation JavaScript today.
    - First, Babel will transpile modern JavaScript features down so that they run in all browsers. Yes, modern JavaScript support is quite good in browsers today, but some holes remain depending on the browsers that you need to support and the modern JavaScript features that you choose to use.
    - And we can't send JSX to the browser. JSX is invalid JavaScript. So Babel will transpile our React JSX down to JavaScript.
    - History of JS

	Edition	Date published	Changes from prior edition
ES1	1	June 1997	First edition
ES2	2	June 1998	Editorial changes to keep the specification fully aligned with ISO/IEC 16262 international standard
ES3	3	December 1999	Added regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and other enhancements
ES4	4	*Abandoned*	Fourth Edition was abandoned, due to political differences concerning language complexity. Many features proposed for the Fourth Edition have been completely dropped; some were incorporated into the sixth edition.
ES5	5	December 2009	Adds "strict mode," a subset intended to provide more thorough error checking and avoid error-prone constructs. Clarifies many ambiguities in the 3rd edition specification, and accommodates behaviour of real-world implementations that differed consistently from that specification. Adds some new features, such as getters and setters, library support for JSON, and more complete reflection on object properties.[9]
	5.1	June 2011	This edition 5.1 of the ECMAScript standard is fully aligned with third edition of the international standard ISO/IEC 16262:2011.
ES6 / ES2015	6	June 2015[10]	The sixth edition, initially known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015 (ES2015)[10] adds significant new syntax for writing complex applications, including classes and modules, but defines them semantically in the same terms as ECMAScript 5 strict mode. Other new features include iterators and `for-of` loops, Python-style generators and generator expressions, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies (metaprogramming for virtual objects and wrappers). As the first "ECMAScript Harmony" specification, it is also known as "ES6 Harmony."
ES7 / ES2016	7	June 2016[11]	ECMAScript 2016 (ES2016)[11], the seventh edition, intended to continue the themes of language reform, code isolation, control of effects and library/tool enabling from ES2015, includes two new features: the exponentiation operator (`**`) and Array.prototype.includes.
ES8 / ES2017	8	June 2017[12]	ECMAScript 2017 (ES2017), the eighth edition, includes features for concurrency and atomics, syntactic integration with promises (async/await).[13][12]
ES9 / ES2018	9	June 2018[8]	ECMAScript 2018 (ES2018), the ninth edition, includes features for asynchronous iteration and generators, new regular expression features and rest/spread parameters.[8]

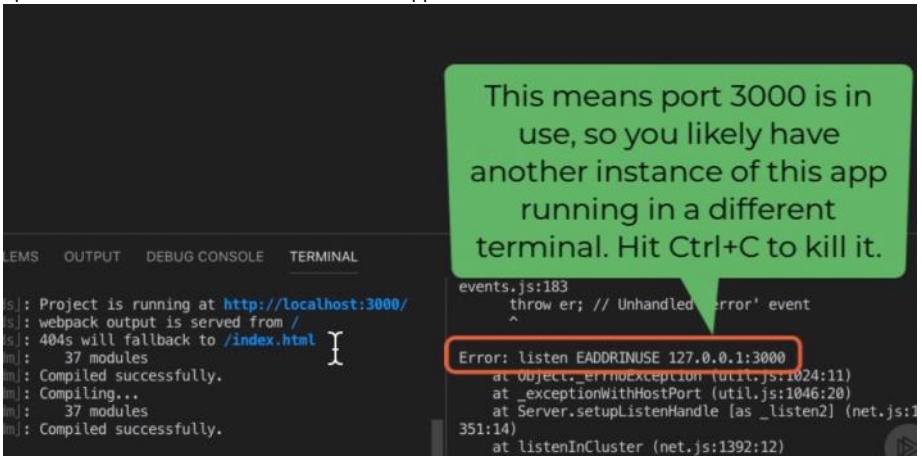
Source: en.wikipedia.org/wiki/ECMAScript

- Below is how babel transpiles the jsx code and sends to the browser
  - 

- Below are the babel related packages:
  - 

- Babel can be configured via a .babelrc file or in the package.json
- In package.json we can configure in the following way
  - 

- **Configuring npm scripts**
  - 

- npm start in the command line would run the app in the browser
  - 

- Ctrl + c to kill the processes running on Port 3000

- **Configuring ESLint**
  - We can have a separate file or else configure it in package.json itself
  - 

  - 

  - Also we need to configure eslint in webpack as below
    - 

- Steps to create react app
  - npm i react react-dom --save-dev
  - npm i webpack webpack-cli
  - npm i babel-loader @babel/core @babel/node @babel/preset-env @babel/preset-react
  - npm i html-webpack-plugin
  - npm i webpack-dev-server

  - babel.config.js
  - webpack.config.js

React Component Approaches

Ways to Create Components

- createClass
- ES class
- Function
- Arrow function

- **createClass** -
  React.createClass was the original way to create a React component when React was first launched. You can still use this style today, but most people prefer to use the more modern approaches

createClass Component

```
var HelloWorld = React.createClass({
  render: function () {
    return (
      <h1>Hello World</h1>
    );
  }
});
```

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

JS Class Component

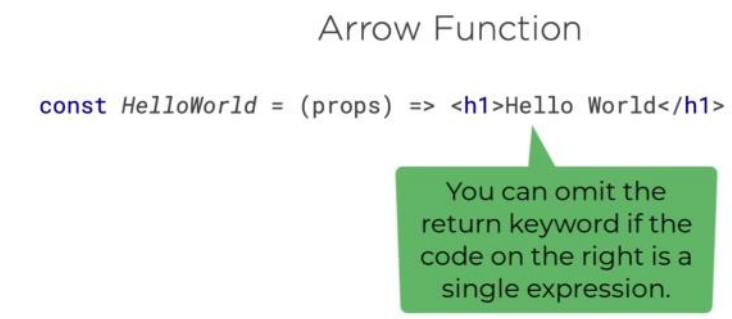
```
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

- **Function -**
  React assumes that the return statement is your render function. The only argument is the props passed in.

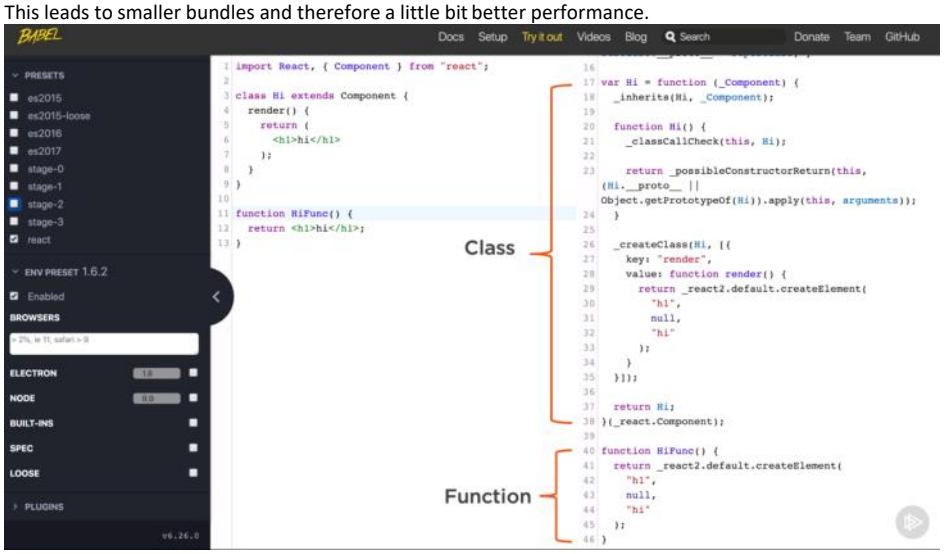
Function Component

```
function HelloWorld(props) {
  return (
    <h1>Hello World</h1>
  );
}
```

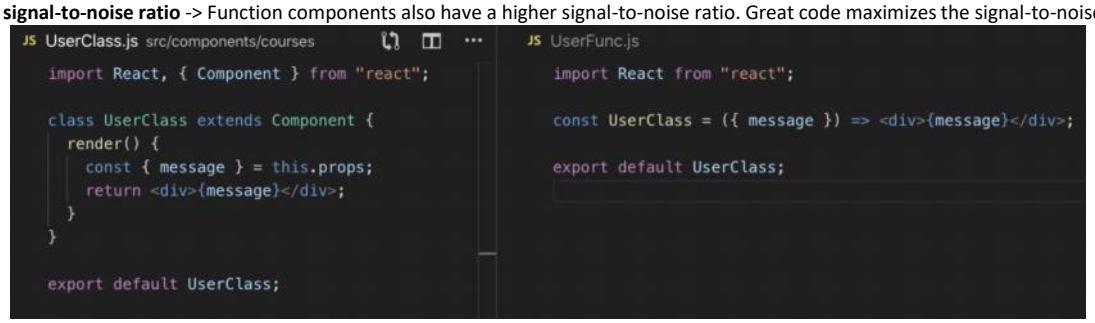
- **Arrow function -**
  With concise arrows, you can omit the return keyword if the code on the right- hand side of the arrow is a single expression. And if you have multiple lines of JSX, then you can wrap the JSX in parentheses, and that makes it a single expression.



- **Functional Component benefits**
  - It's recommended to use functional components instead of class components when possible.
  - **Easier to understand** -> They avoid class-related cruft, like extends and the constructor.
  - **Avoid 'this' keyword** -> Function components avoid the annoying and confusing quirks of JavaScript's this keyword. Doing so makes the component easier to understand.
    - Avoiding classes also eliminates the need for binding.
  - Less transpiled code -> Functional components transpile smaller than class components.
    - They produce less code when run through Babel.
    - This leads to smaller bundles and therefore a little bit better performance.



  - **High signal-to-noise ratio** -> Function components also have a higher signal-to-noise ratio. Great code maximizes the signal-to-noise ratio.



    - On simple components, with a single-line return statement, you can omit the return and parentheses. If you do this and also use destructuring on props, the result is nearly all signal.
  - **Enhanced code completion/intellisense** -> If you destructure your props, then all the data that you use is now specified as a simple function argument. This means that you get improved code completion support compared to class base components.
  - **Easy to test** -> And since functional components are just a function, assertions tend to be very straightforward. Given these props, I expect this component to return this markup
  - **Performance** -> Function component soffer improved performance as well since as of React 16, there's no instance created to wrap them.
  - **Classes may be removed in the future** ->With React Hooks, function components can handle nearly all use cases.

- **When to use Class vs. Function components**
  - Depends on React version used
  - With React versions lower than 16.8, function components lack some key features. Only class components support state, refs, and lifecycle methods, like componentWillMount, componentDidMount, and so on. But we can use function components everywhere else.

When Should I Use Each? Pre 16.8...

Class Component	Function Components
State	Everywhere else ☺
Refs	
Lifecycle methods	

- With newer versions of React, we have a new feature called hooks
  - The useState hook handles state,
  - useEffect handles side effects, so they effectively replace lifecycle methods.
  - UseRef allows you to add a ref, and useMemo allows you to avoid needless re-renders for performance.
  - Now that hooks are here, componentDidError and getSnapshotBeforeUpdate are the only two things that a function component can't do. So everywhere else, prefer function components

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page describes the APIs for the built-in Hooks in React.

If you're new to Hooks, you might want to check out the overview first. You may also find useful information in the frequently asked questions section.

- Basic Hooks
  - `useState`
  - `useEffect`
  - `useContext`
- Additional Hooks
  - `useReducer`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`
  - `useDebugValue`

- **Container vs Presentation Components**
  - **Container components**
    - Container components are concerned with behavior and marshalling data and actions. So these components have little or no markup.
    - We can think of container components as the backend for the frontend.
    - Remember, components don't have to emit DOM.
    - Container components are mainly concerned with passing data and actions down to child components. This means that they're typically stateful.
    - When working in Redux, container components are typically created using Redux's Connect function at the bottom of the file.
    - Container components pass data and actions down to presentation components.
    - Container components know about Redux. They have Redux-specific code inside for dispatching actions to the store and connecting to the store via Redux's Connect function.
    - Container components are stateful.
  - **Presentation Components**
    - These components basically rely on the markup
    - Presentation components receive functions and data that they need from a container component.
    - Presentation components typically know nothing about Redux, and this is a good thing. It makes your presentation components more reusable and easier to understand.
    - Presentation components just rely on props to display UI. They have no dependencies on the rest of the app, such as Redux actions or stores.
    - Presentation components don't specify how the data is loaded or mutated.
    - Presentation components are not stateful.

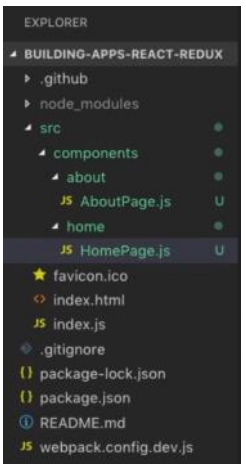
Container	Presentation
Little to no markup	Nearly all markup
Pass data and actions down	Receive data and actions via props
Knows about Redux	Doesn't know about Redux
Often stateful	Often no state

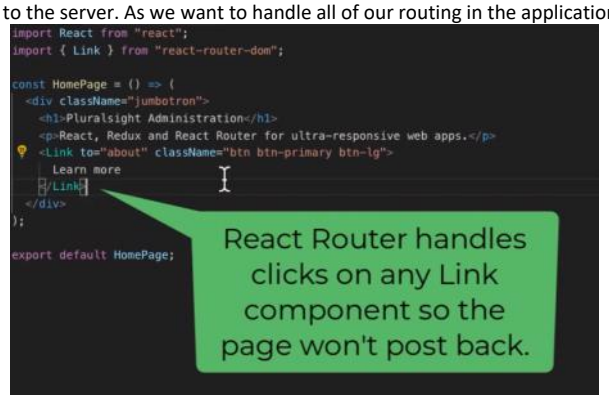
Alternative Jargon

Container	Presentational
Smart	Dumb
Stateful	Stateless
Controller View	View

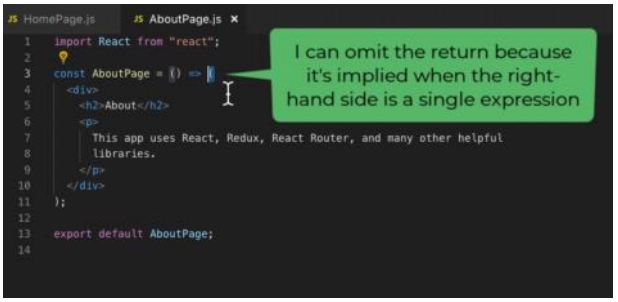
- When you notice that some components don't use props they receive, but merely forward them down, then it's a good time to introduce some container components.
- So when you find yourself having to update multiple layers of components just to pass data down to the child component that actually uses the data, that can be a good time to add some container components that connect to the Redux store.

Initial App structure

- 

- We use "**Link**" to navigate to the about page by using react-router-dom(react router) to route through the application and so that it does not post back to the server. As we want to handle all of our routing in the application on the client.
  - 

- In the about page and home page we can omit the return statement as explained in the below screenshot

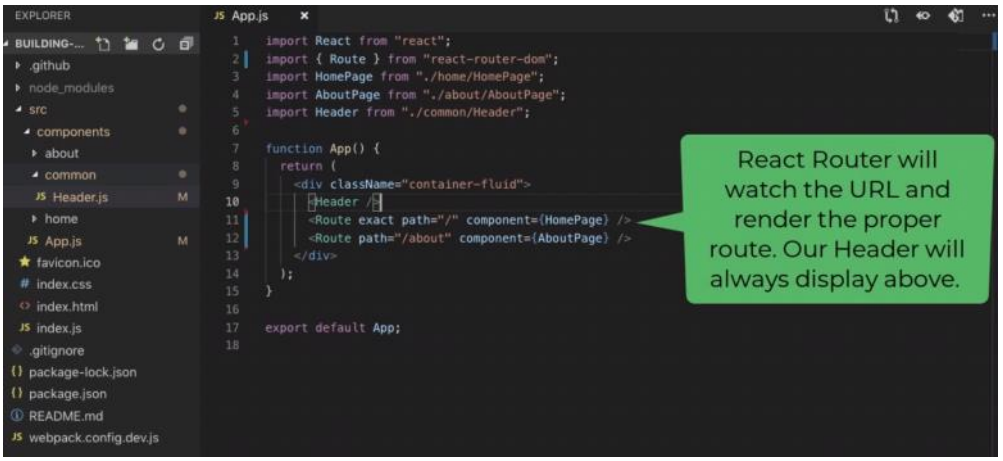
- 

- Configuring App entry point in index.js
  - We use BrowserRouter from the react router
  - ```
    import React from "react";
    import { render } from "react-dom";
    import { BrowserRouter as Router } from "react-router-dom";

    function Hi() {
      return Hi Ram ;
    }

    render(
      <Router>
        <Hi />
      </Router>,
      document.getElementById("app")
    );
    ```
  - Also we can configure bootstarp by importing the bootstrap css as below
  - ```
    import "bootstrap/dist/css/bootstrap.min.css";
    ```
  - So this will import the minified version of Bootstrap. Remember, we configured Webpack to handle CSS as well. So Webpack will bundle this up for us and inject a reference to that bundled CSS into our index.html file

- **App.js**
  - ```
    import React from "react";
    import { Route } from "react-router-dom";
    import AboutPage from "../about/AboutPage";
    import HomePage from "../home/HomePage";

    function App() {
        return (

                <Route exact path="/" component={HomePage}></Route>
                <Route path="/about" component={AboutPage}></Route>

        );
    }

    export default App;
    ```
  - ```
    <Route exact path="/" component={HomePage}></Route>
    ```
  - **"exact"** is provided that so that only the empty route matches. Otherwise, this route would also match any other route since / is in every route.
- **Header.js**
  - 

  - NavLink component like an anchor that react-router-dom manages for us. So the NavLink component will make sure that react-router-dom handles all our links.

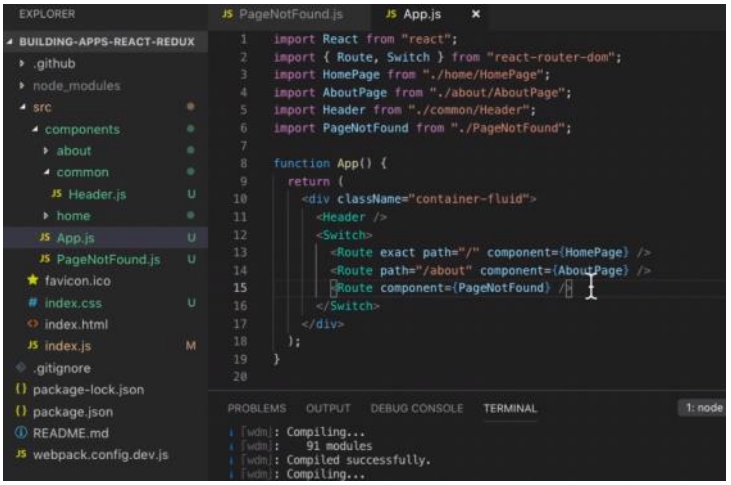
```jsx
import React from "react";
import { NavLink } from "react-router-dom";

const Header = () => {
    const activeStyle = { color: "#F15B2A" };
    return (
        <nav>
            <NavLink to="/" activeStyle={activeStyle} exact>Home</NavLink> {" | "}
            <NavLink to="/about" activeStyle={activeStyle}>About</NavLink>
        </nav>
    );
};

export default Header;
```



- PageNotFound.js

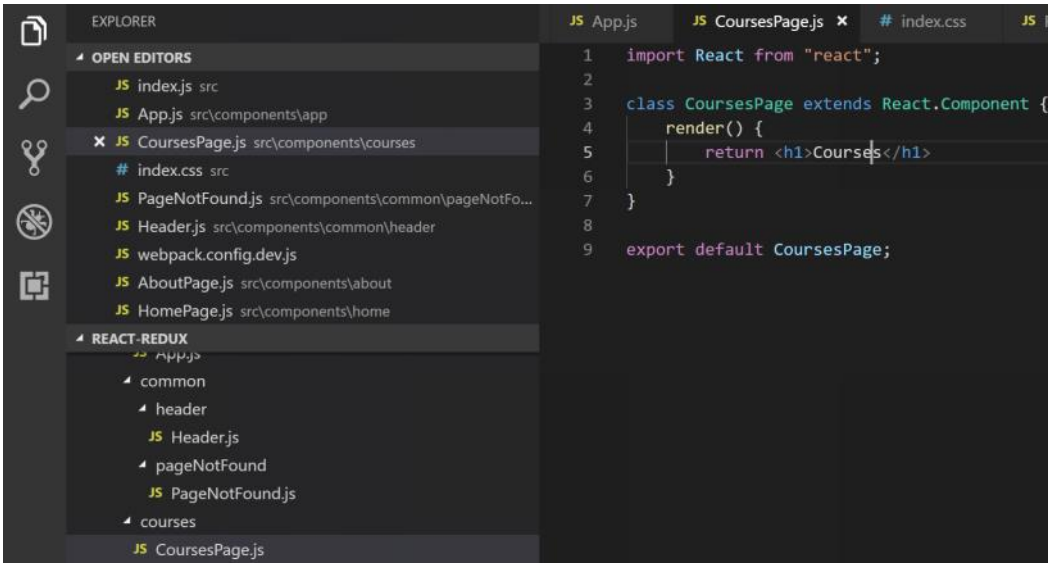
```jsx
import React from "react";

const PageNotFound = () => <h1>Oops! Page not found.</h1>;

export default PageNotFound;
```

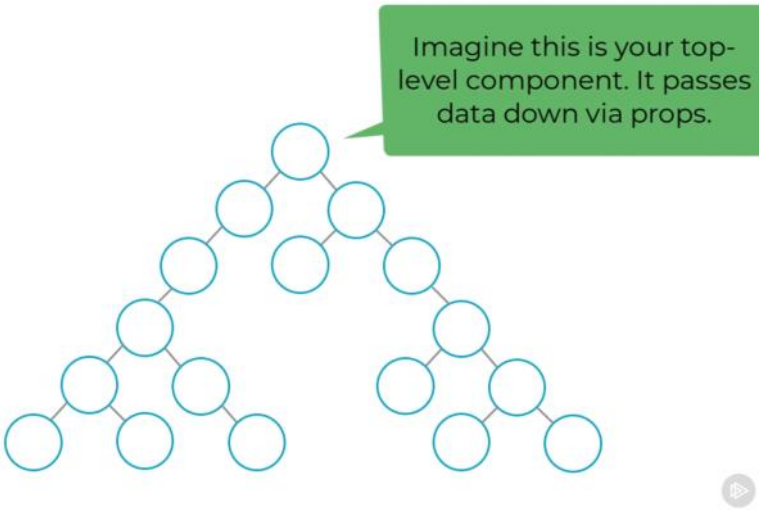
  - Switch allows us to declare that only one route should match. What we can do is wrap our route declarations in switch. And now as soon as one of these routes matches, it will stop looking for other matching routes. So switch operates a lot like a switch statement in JavaScript.

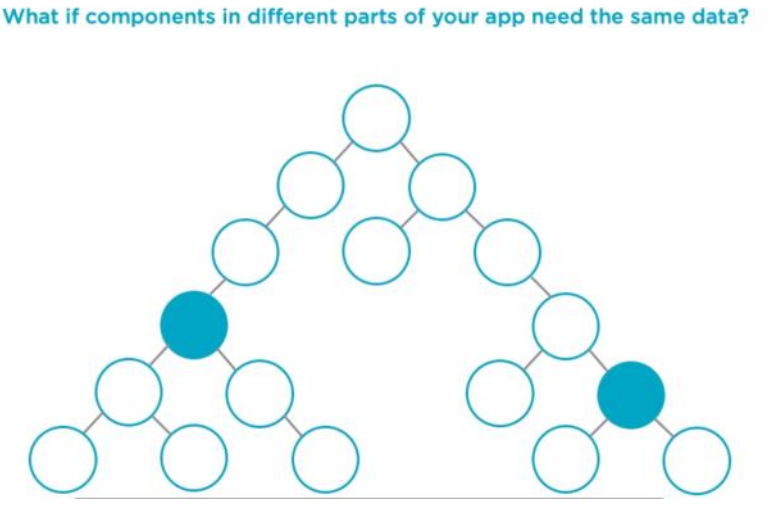


  - Notice that I don't have to declare a path here because what we're saying is if none of the routes above match, then this PageNotFound should load instead.

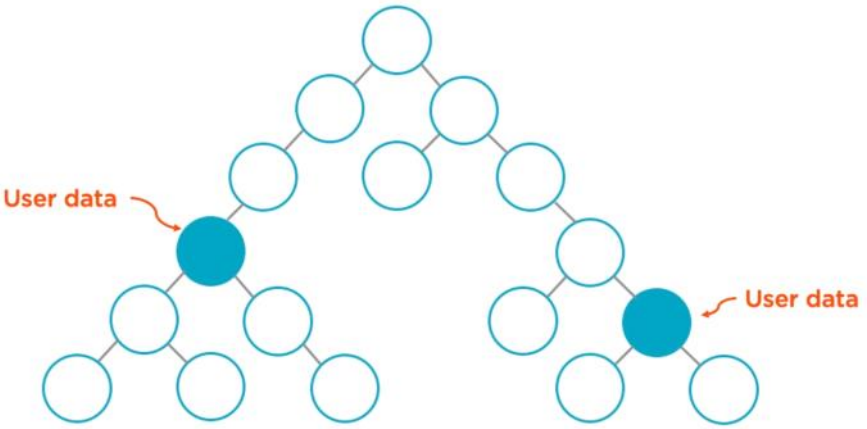
- CoursesPage.js
  - Below we have implemented CoursesPage using class by extending React.Component



- **Redux**
  - **Why Redux?**
    - Imagine all the circles below represent React components
    - The top level component passes data to the child components via props as in the below screenshot
    - 

    - 

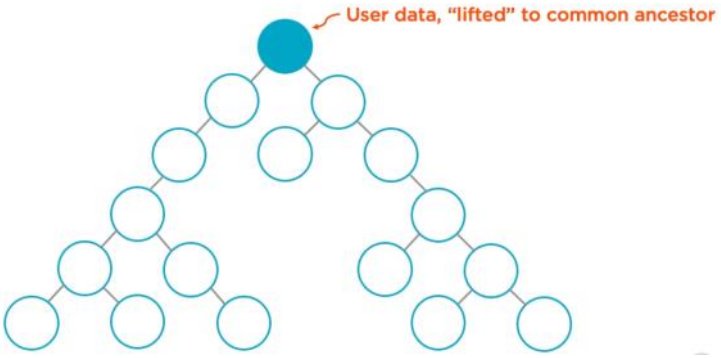
    - If both the components required to retrieve user data from the database, calling them separately would be redundant.

- 

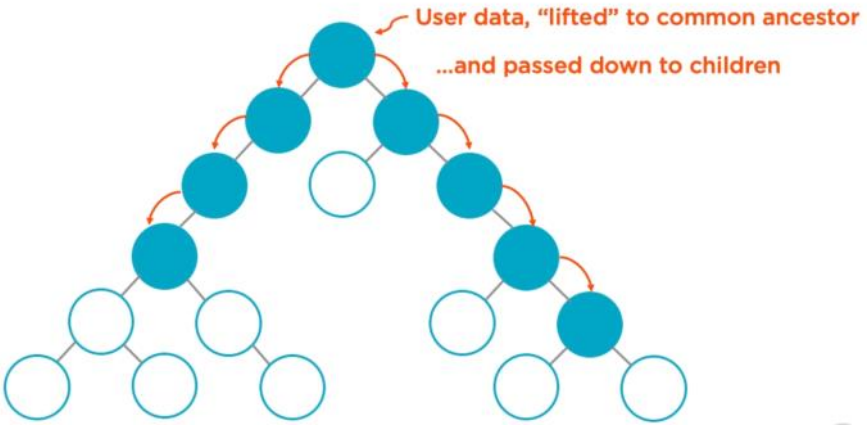
- To keep them in sync we can use any of the following options:

  Lift State -> It means to lift state to their common parent. In this case, it means moving the user data all the way up to the top-level component since that's their only common ancestor.

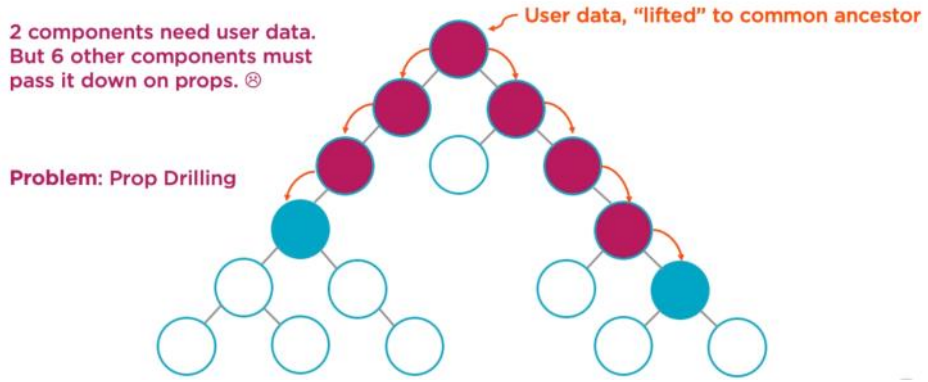
  1. Lift State



  - This is annoying though because it means that you have to pass data down on props through all these other components. These components didn't need the user's data. You're just passing the data down on props to avoid storing the same data in two spots.



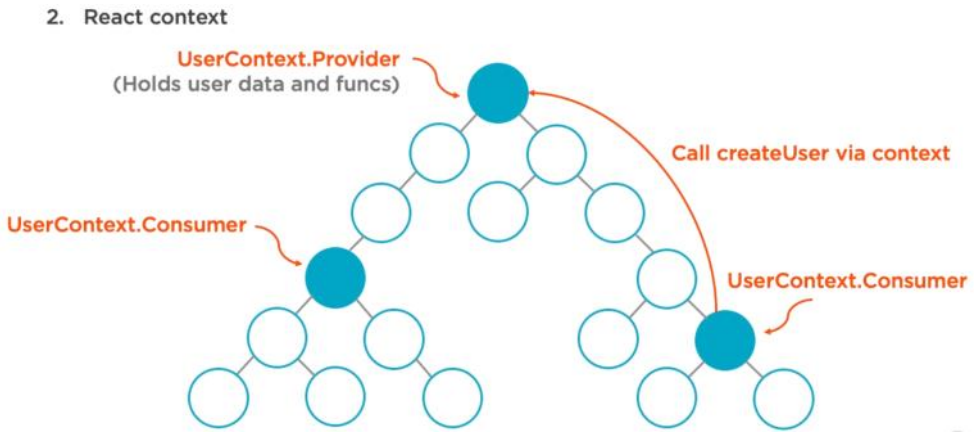
  - We added the user prop to these six components merely to pass the data down to the two components that need it. This problem is commonly called prop drilling.



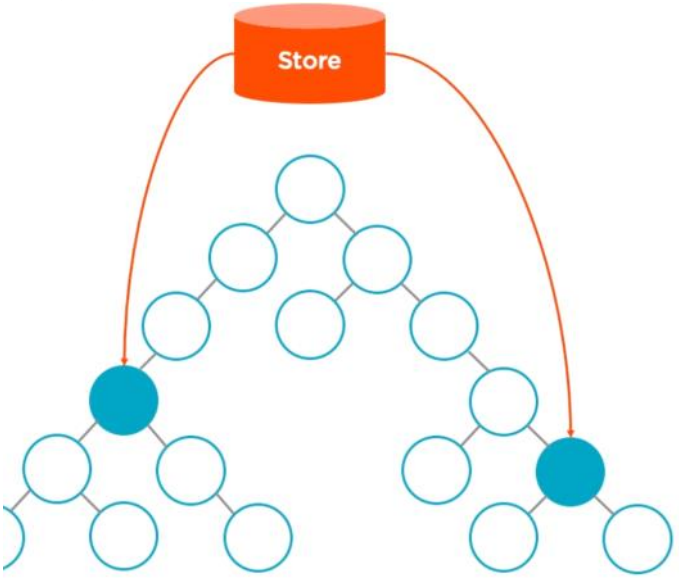
Lifting state does work, and it's a good first step for small and mid-size applications. But on larger appsthat need to display the same data in many spots, lifting state becomes tedious, and it leads to components with many props that exist merely to pass data down.

React Context -> To expose global data and functions from a given React component

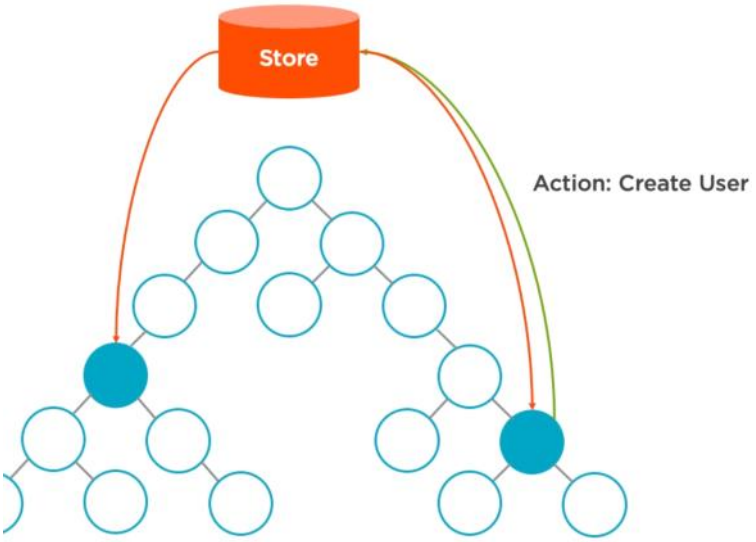
- To access this global data, you import the context and consume it in your component. So it's not truly global data. You must explicitly import the context to consume it.
- For example, the top-level component could declare a UserContext.Provider. This provider would provide user data and relevant functions to any components that want to consume it.
- So the two components that need the user data can import that UserContext and thus access the user's information via UserContext.Consumer

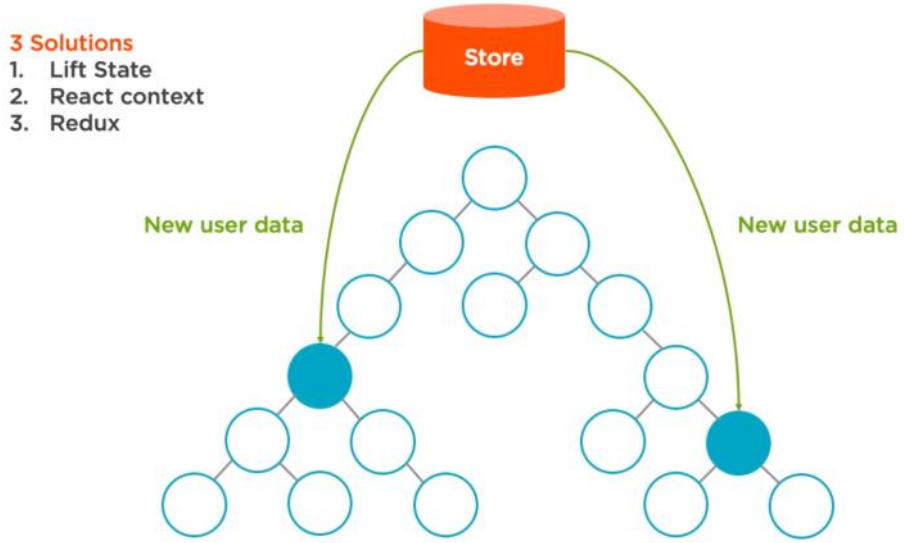


Redux -> With Redux, there's a centralized store. Think of the store like a local client-side database. This is a single spot where the app's global data is stored.

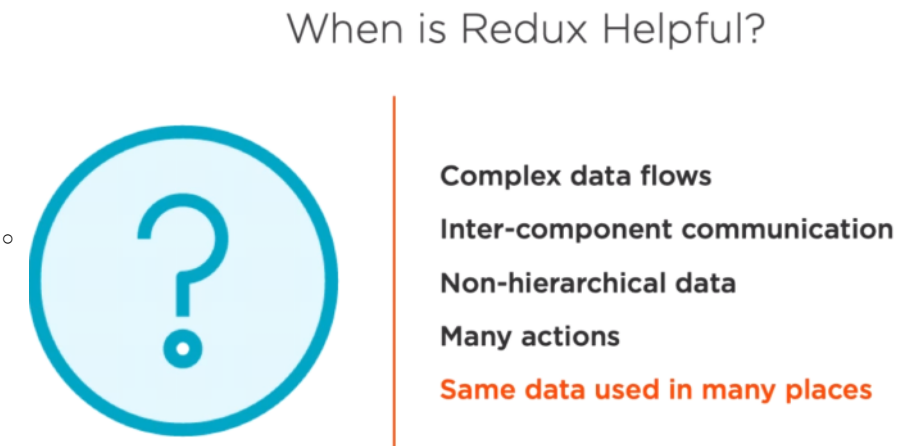


- Any component can connect to the Redux store. The Redux store can't be changed directly though. Instead, any components can dispatch an action, such as Create User.

- 

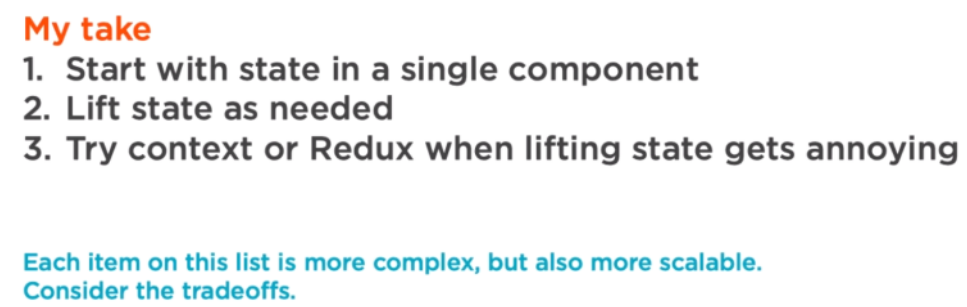
  - When the action is dispatched, the Redux store is updated to reflect the new data. And any connected components receive this new data from the Redux store and rerender.
- 

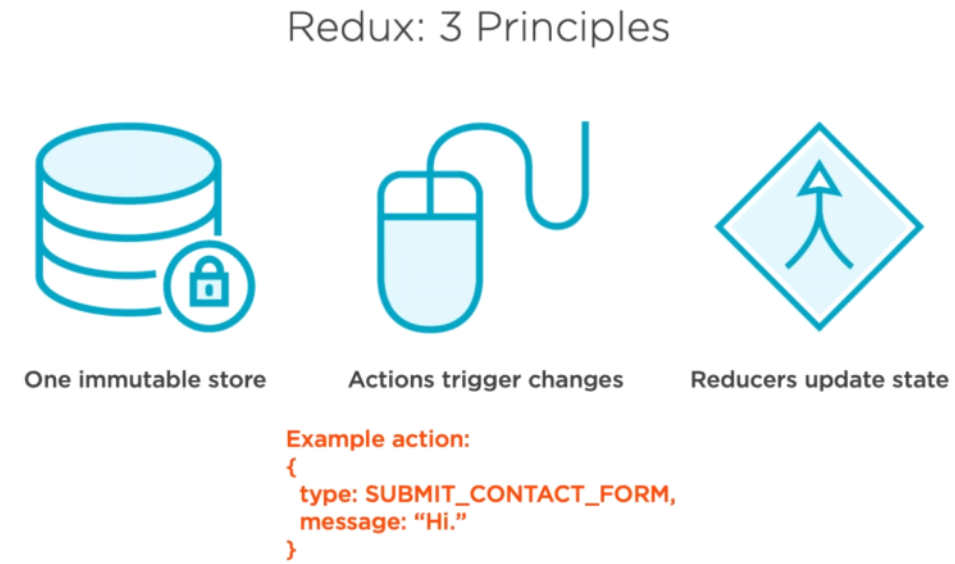
- **When is Redux helpful?**



  - **Complex data flows** -> But Redux can be useful for applications that have complex data flows. If you're writing an app that merely displays static data, then Redux isn't likely to be useful.
  - **Inter-component communication** -> If you need to handle interactions between two components that don't have a parent- child relationship, then Redux offers a clear and elegant solution.
  - **Non-hierarchical data** -> When you find two separate components are manipulating the same data, Redux can help. This scenario is often the

case when your application has non-hierarchical data.

- **Many actions** -> Also, as your application offers an increasing number of actions, such as writes, reads, and deletes for complex data structures, Redux's structure and scalability becomes increasingly helpful.
- **Same data in many places** -> The most obvious sign that you'll want something like Redux is if you're utilizing the same data in many places. If your components need to utilize the same data, and they don't have a simple parent- child relationship, Redux can help.
- 

- **Redux principles**
  - 

  - **One immutable store** -> The first is that your application's state is placed in a single, immutable store. By immutable, I mean that state can't be changed directly. Having one immutable store aids debugging, supports server rendering, and it makes things like undo and redo easily possible.
  - **Action trigger changes** -> In Redux, the only way to change state is to emit an action, which describes a user's intent. For example, a user might click the Submit Contact Form button, and that would trigger a SUBMIT_CONTACT_FORM action.
  - **Reducers update state** -> The final principle is that state changes are handled by pure functions. These functions are called reducers. In Redux, a reducer is just a function that accepts the current state in an action, and it returns a new state.

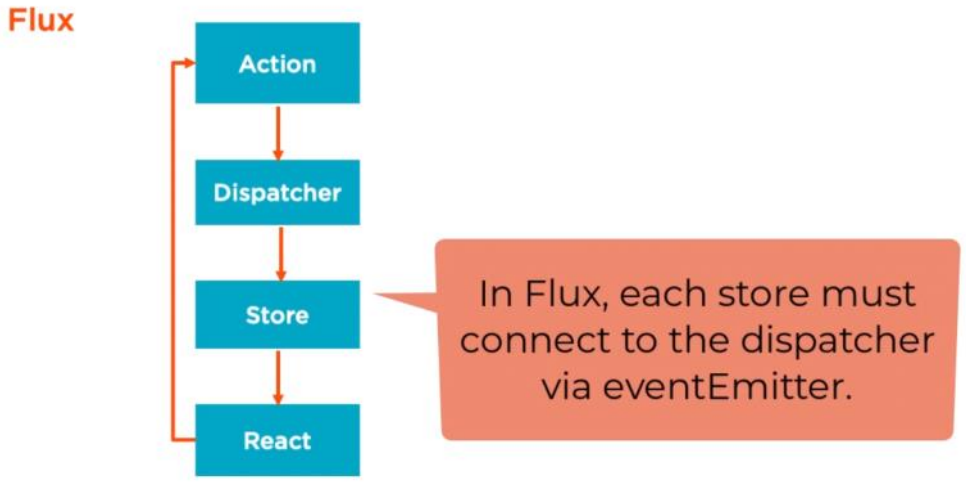
- **Redux vs Flux**
  - Flux and Redux are two different ways that you can handle state and data flows in your React applications.
  - Both Flux and Redux have the same unidirectional data flow philosophy.
  - Data flows down, and actions flow up.
  - They also both utilize a finite set of actions that define how state can be changed. You define action creators to generate those actions and use constants that are called action types in both as well
  - They both have the concept of a store that holds state, although Redux typically has a single store, while Flux typically has multiple.

- 

Redux Concepts:

- **Reducers** -> However, Redux introduces a few new concepts. Reducers are functions that take the current state in an action and then return a new state. So reducers are pure functions.
- **Containers** -> Containers are just React components, but their use is specific. Container components contain the necessary logic for marshalling data and actions, which they pass down to dumb child components via props. This clear separation helps keep most of your React components very simple, pure functions that receive data via props. This makes them easy to test and simple to reuse.
- **Immutability** -> Redux store is immutable

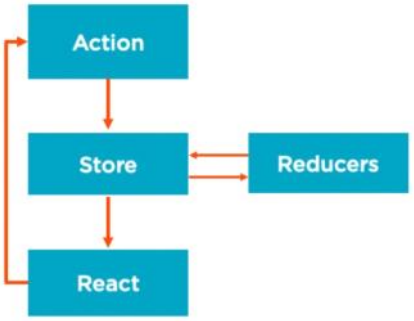
Flux Concepts

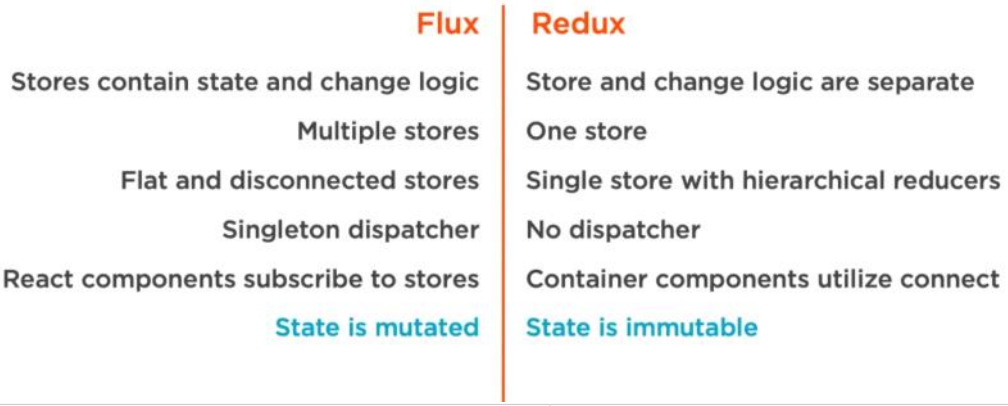
- 

- **Actions** -> When actions are triggered, stores are notified by the dispatcher.
- **Dispatcher** -> So Flux uses a singleton dispatcher to connect actions to stores.
- **Stores** -> Stores use eventEmitter to connect to the dispatcher. So in Flux, each store that wants to know about actions needs to explicitly connect itself to the dispatcher by using eventEmitter.

- In contrast, Redux doesn't have a dispatcher at all. Redux relies on pure functions called reducers, so it doesn't need a dispatcher.
- Pure functions are easy to compose, so no dispatcher is necessary.
- Each action is ultimately handled by one or more reducers, which update the single store.
- Since state is immutable in Redux, the reducer returns a new, updated copy of state, which updates the store.

- Redux

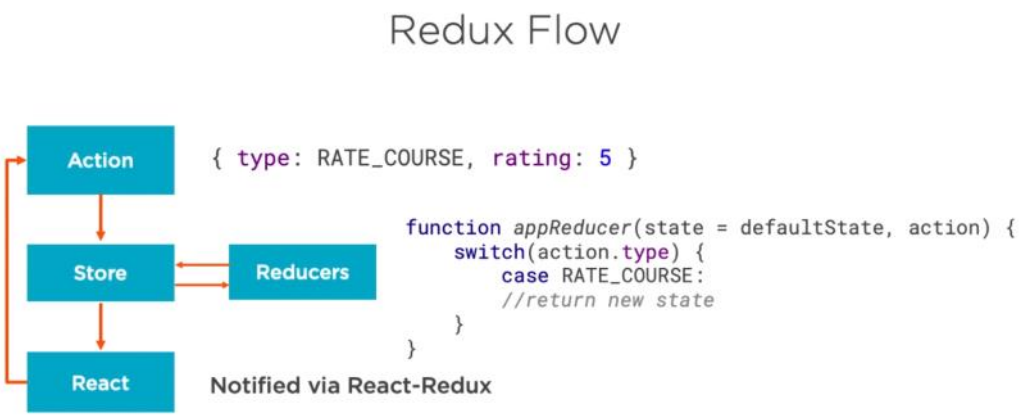




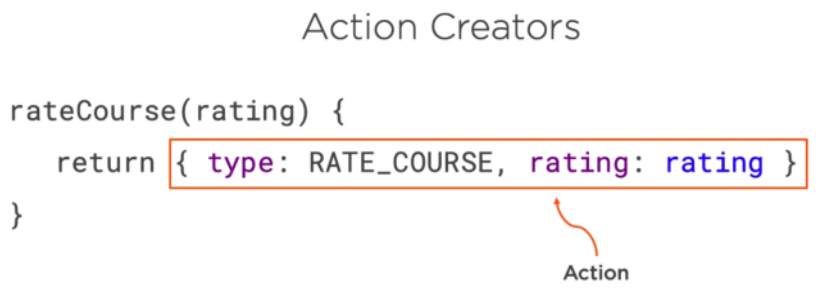
Flux	Redux
- In Flux, stores do more than one thing. - They don't just contain application state. - They also contain the logic for changing state.	- Redux honors the single responsibility principle by separating the logic for handling state. - Redux handles all state- changing logic with reducers. - A reducer specifies how state should change for a given action. So a reducer is a function that accepts the current state in an action and returns an action.
- Flux supports having multiple stores. So in a Flux app, you may have a user store and a product store. - You can have as many stores as you like.	- In Redux, you typically only have one store. Having a single source of truth helps avoid storing the same data in multiple places and avoids the complexity of handling interactions between stores.
- One common struggle in Flux is how to deal with stores that interact with one another. - In Flux, the stores are disconnected from each other, though Flux does at least provide a way for stores to interact in order via the waitFor function.	- In contrast, Redux's single store model avoids the complexity of handling interactions between multiple stores. In order to handle more complex stores with many potential actions, you can utilize multiple reducers, and you can even nest them.
- In Flux, stores are flat	- In Redux, reducers can be nested via functional composition just like React components can be nested. So Redux gives you the same power of composition and nesting in your reducers as you have in your React component model.
- In Flux, a dispatcher sits at the center of your app. The dispatcher connects your actions to your stores.	- In Redux, there is no dispatcher because Redux's single store just passes actions down to the reducers that you define. It does so by calling a root reducer that you define. Reducers are pure functions. So in Redux, there's no need for Flux's eventEmitter pattern.
- In Flux, you have to explicitly subscribe your React views to your stores using onChange handlers in eventEmitter.	- In Redux, this can be handled for you using React-Redux. React-Redux is a companion library that connects your React components to the Redux store.
- In Flux, you manipulate state directly. It's mutable.	- In Redux, state is immutable, so you need to return an updated copy of state rather than manipulating it directly

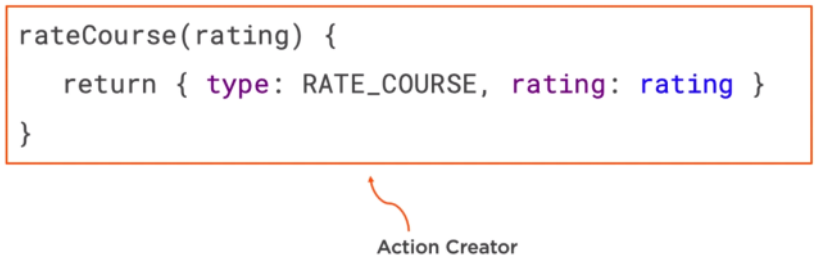
- **Redux Flow**
  - Below is an example
  - An action describes a user's intent. It's an object with a type property and some data. The data portion can be whatever shape you like. The only requirement is that an action has a type property.
  - Here's an action for rating a course. Imagine that you were rating my course on a scale of 1 to 5. Let's just say you rate it at a 5.
  - This action will ultimately be handled by a reducer. A reducer is a fancy name for a function that returns new state based on the action that you pass it.

- The reducer receives the current state in an action, and then it returns a new state.
- Reducers typically contain a switch statement that checks the type of action passed. This determines what new state should be returned.
- Once this new state is returned from a reducer, the store is updated. React re-renders any components that are utilizing the data.
- React components are connected to the store using a Redux- related library called React-Redux.



- **Actions**
  - In Redux, the events happening in the application are called actions.
  - Actions are just plain objects containing a description of an event





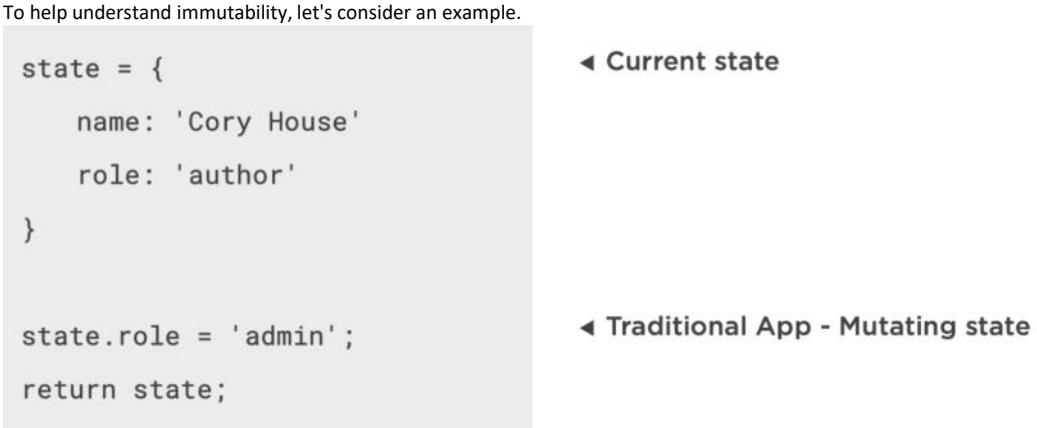
  - An action must have a type property. In the above example we are passing some data under a property called rating.
  - This could be a complex object, a simple number, a Boolean, or really any value that's serializable.
  - The only thing that you shouldn't try passing around in your actions are things that won't serialize to JSON like functions or promises.
  - Actions are typically created by convenience functions that are called action creators. Here's an **action creator** called **RATE_COURSE**.
  - Typically, the action creator has the same name as the action's type.
  - Action creators are considered convenience functions because they're not required, but I recommend following this simple convention.
  - By using these action creators to create your actions, the spot where you dispatch the action doesn't need to know about the action creator structure.
  - The app that we're creating will work with course data, so it will have actions like loadCourse, createCourse, and deleteCourse.
  - When actions are dispatched, it ultimately affects what data is in the store.

- **Stores**

- In Redux, you create a store by calling createStore in your app's entry point.
- We pass the createStore function to your reducer function.
- The Redux store honors the single responsibility principle because the store simply stores the data, while reducers, handle state changes.
- We have only one store in Redux, which is a key feature. Having a single source of truth makes the app easier to manage and understand.
- The Redux store API is quite simple. The store can dispatch an action, subscribe to a listener, return its current state, and replace a reducer.

- replaceReducer feature is useful to support hot reloading.
- The most interesting omission here is that there's no API for changing data in the store. It means that the only way that you can change the store is by dispatching an action. The store doesn't actually handle the actions that you dispatch.

- **Immutability**
  - Immutability is a fundamental concept in Redux.
  - You might be wondering how you can build an application that doesn't mutate state. I mean if I can't mutate state, doesn't that mean that no data can change? Not at all.
  - It just means that instead of changing your state object, you must return a new object that represents your application's new state.
  - It's worth noting that some types in JavaScript are **immutable** already, such as **numbers, string, Boolean, undefined, and null**.
  - In other words, every time you change the value of one of these types, a new copy is created.
  - **Mutable JavaScript** types are things like **objects, arrays, and functions**.
  - To help understand immutability, let's consider an example.



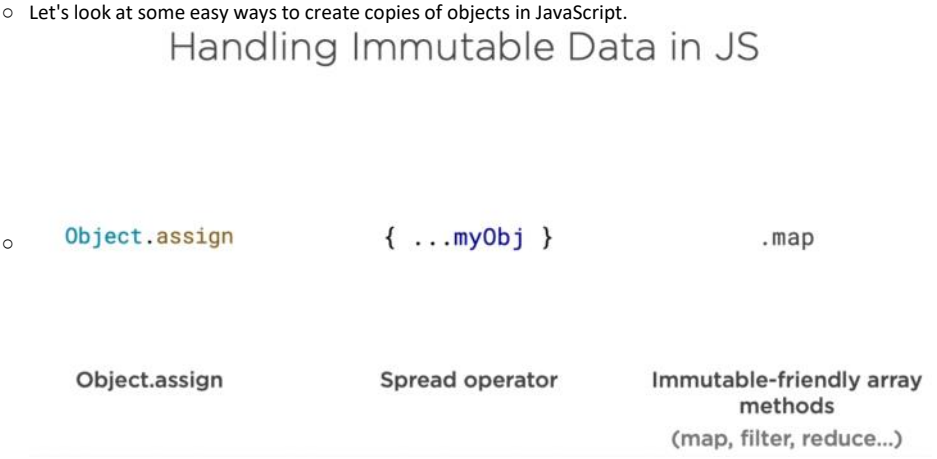
  - Imagine that our app state holds my name and my role. In a traditional app, if I wanted to change state, I'd assign a new value to the property that I want to change. So here I'm mutating state because I'm updating an existing object to have a new value for role.

- Let's contrast this approach with the immutable approach to updating state.

```
state = {
    name: 'Cory House'
    role: 'author'
}
```
◀ Current state

```
return state = {
    name: 'Cory House'
    role: 'admin'
}
```
◀ Returning new object. Not mutating state! ☺

- Here you can see that I'm not mutating state. Instead, I'm returning an entirely new object. This is important because Redux depends on immutable state to improve performance.
- First, you might be thinking, yuck, do I have to build a new copy of state by hand every time I want to change it? Thankfully, no. That would be very impractical on an object with many properties.
- Let's look at some easy ways to create copies of objects in JavaScript.



- There are multiple ways to get this done using plainJavaScript, including Object.assign, the spread operator, and immutable-friendly array methods, like map, filter, and reduce.
  - **Object.assign**

    - Object.assign creates a new object that allows us to specify existing objects as a template.
    - The first parameter is the target, and then it accepts as many source objects as you want.

    - The first parameter is the target, so we're just creating a new empty object. And then we're mixing that new object together with our existing state and also changing the role property to admin. So the result of this statement is effectively a clone of our existing state object, but with the role property changed to admin.
    - Each new parameter that we declare on the right-hand side overrides anything on the left-hand side. And we can declare as many arguments as we want for Object.assign.
    - Note: When using Object.assign, the first parameter should be an empty object. If not used we end up mutating the state instead of creating a new object.

  - **Spread Operator**

Copy via Spread

```
const newState = { ...state, role: 'admin' };
```

Arguments on the right override arguments on the left.

- The spread operator is three dots. Whatever we place on the right is shallow copied.
- Much like Object.assign, values that you specify on the right override the values on the left. So this creates a new object that is a copy of state, but with the role property set to admin.
- We can use spread to copy an array too.
- We'll use spread in our Redux reducers to update state by returning a copy of our current state with the desired changes included.
- Both Object.assign and Object.spread only create shallow copies. So if we have a nested object, we need to clone it too.
- Notice how the user object has a nested address object.
- This means that this line doesn't clone the nested address object. It remains referenced in the userCopy. To avoid this issue, you need to manually clone any nested objects too.

Warning: Shallow Copies

```
const user = {
  name: 'Cory',
  address: {
    state: 'California'
  }
}

// Watch out, it didn't clone the nested address object!
const userCopy = { ...user };

// This clones the nested address object too
const userCopy = { ...user, address: {...user.address}};
```

Note: You only need to clone the nested object if you need to change the nested object.

- **Deep Cloning disadvantages**

Warning: Only Clone What Changes

You might be tempted to use deep merging tools like clone-deep, or lodash.merge, but avoid blindly deep cloning.

Here's why:

1. Deep cloning is expensive
2. Deep cloning is typically wasteful

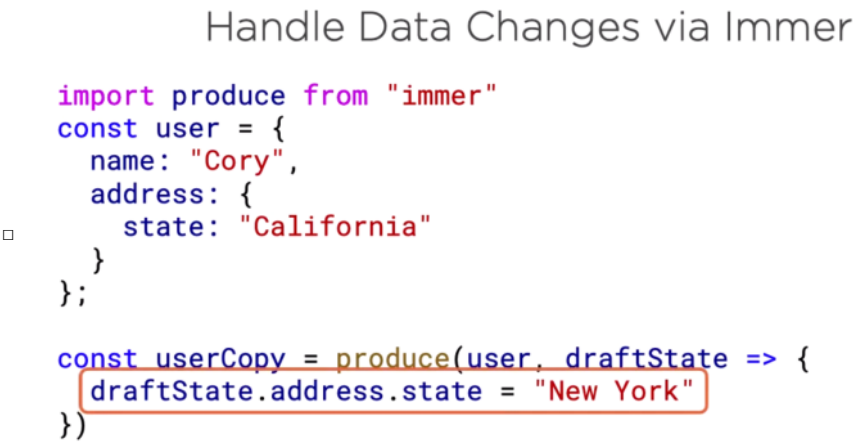
You only need to clone objects that need to change.

3. Deep cloning causes unnecessary renders

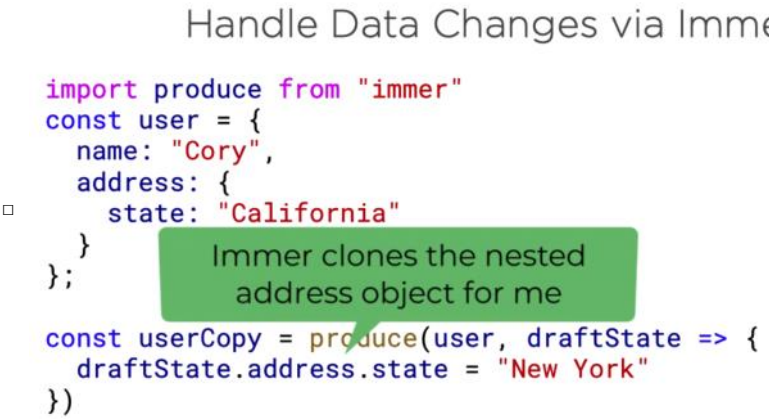
Instead, clone only the sub-object(s) that have changed.

- Deep cloning is expensive. It will needlessly slow the app down.
- Deep cloning is typically wasteful since you don't need to clone all nested objects. You only need to clone the objects that have changed.
- Deep cloning causes unnecessary renders since React thinks that everything has changed when, in fact, perhaps only a specific child object has changed.

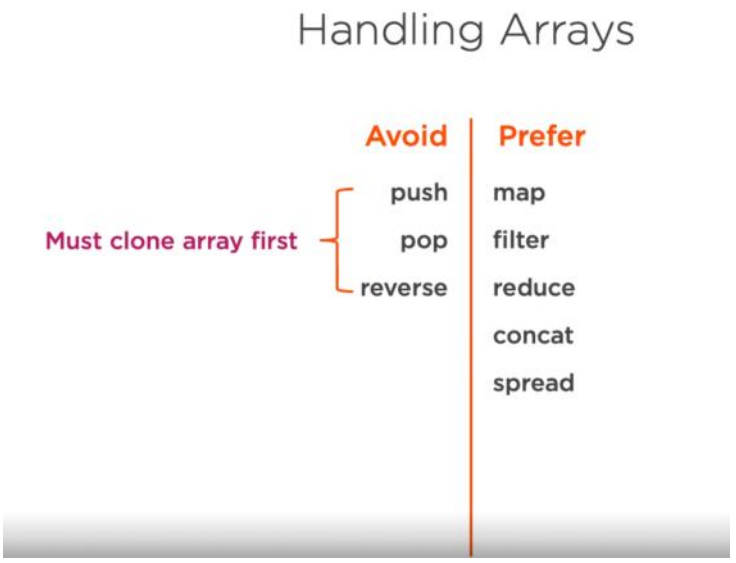
- **Immer**
  - With immer, you can write mutative code, and immer will handle the change in an immutable manner behind the scenes.



  - Note that it looks like I'm mutating the state property under address. But I'm changing a draft state here.
  - Immer sees that I want to change a value in the nested address option. So it will clone the nested address object and return the existing user object with a new address object nested inside.



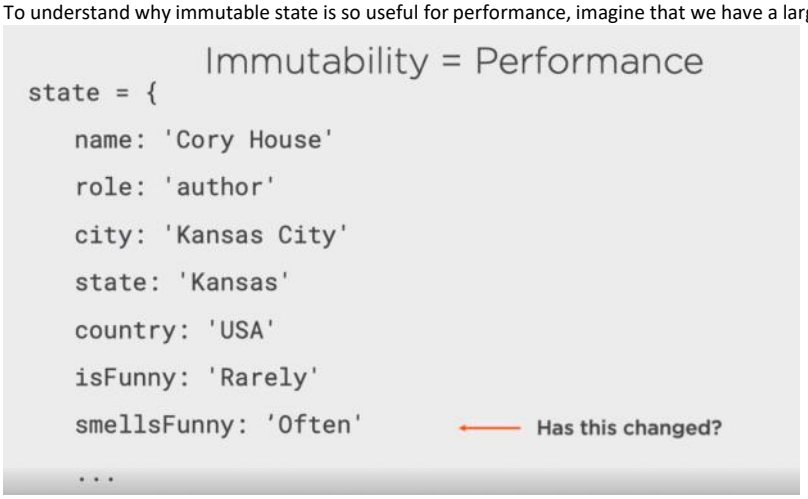
- Handling arrays in Redux



  - When working in Redux, there are some that you should avoid.
  - Array methods like push and pop mutate the array, so they should be avoided.
  - If we want to use this, you must clone the array first to avoid mutating the original array.
  - Instead, prefer using immutable-friendly array methods like map, filter, reduce, and find.
  - All of these methods return a new array, so they don't mutate the existing array.

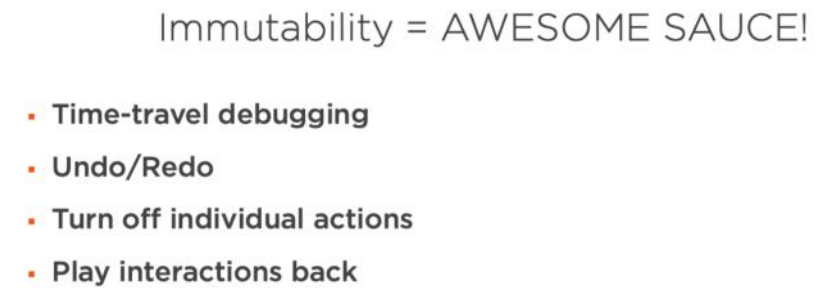
- **Why Immutability?**
  - You might be wondering why state is immutable in Redux. In Flux, you simply change state. But in Redux, it's a bit more complicated than that because state is immutable. Instead, each time you need to change your store's state, you must return an updated copy.
  - So why make state immutable? There are three core benefits to having immutable state:
    - Clarity
    - Performance, and
    - Awesome sauce.

  - **Clarity:**
    - First, immutability means clarity. When state is updated, we will know exactly where and how it happened in the reducer.
    - When state changes in a Redux app, you know where it occurred. Some code written changed in a reducer that returned a new copy of state.
    - In Redux, we don't have to worry where the state update occurred. As long as you're using Redux to handle your state, then you know that it occurred in your reducer.
  - Performance
    - The second big benefit of immutability is performance.
    - Let's consider an example to understand how immutability in Redux helps improve performance.
    - To understand why immutable state is so useful for performance, imagine that we have a large state object with many properties.

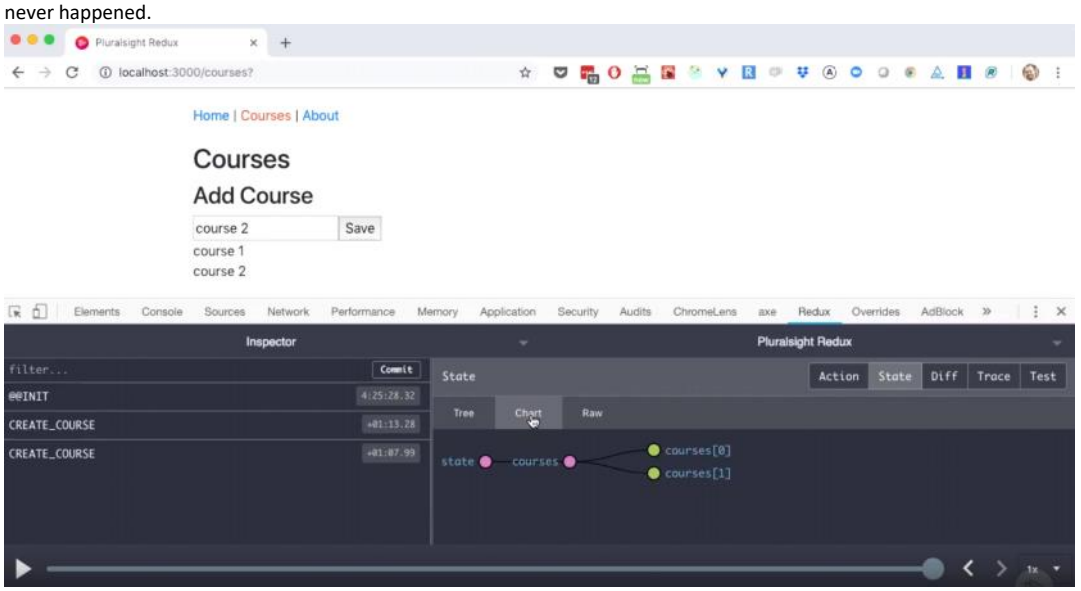


    - If state were mutable, Redux would have to do an expensive operation to determine if state is changed.
    - It would have to check every single property on your state object to determine if any state had changed. But if state is immutable, suddenly this expensive operation of checking every single property is no longer necessary.
    - Instead, Redux can simply do a reference comparison. If the old state isn't referencing the same object in memory, then we know that the state has changed. This is extremely efficient.
    - `if (prevStoreState !== storeState) ...`
    - And behind the scenes, React-Redux can use this simple reference comparison to determine when to notify React of state changes. It won't rerender the component if nothing is changed.
    - So immutability doesn't just make your app more predictable and easier to think about. It also helps improve performance.
    - React-Redux includes a variety of complex performance optimizations behind the scenes that rely on immutable state.

  - **Awesome Sauce**
    - The third big benefit of immutability is what I like to call awesome sauce.
    - Immutability helps support a truly amazing debugging experience that is unlike any other technology that I've worked in. This means that you can travel through time as you debug.



- You can go back in history and see each specific state change that occurred.
- And as you go back in time, you can undo specific state changes to see how that changes the final state.
- You can even turn off individual actions that occurred so that you can see what the state would look like if a specific action in history had never happened.



- And finally, you can play all your interactions back with a click of a button and even select the speed at which it plays back.

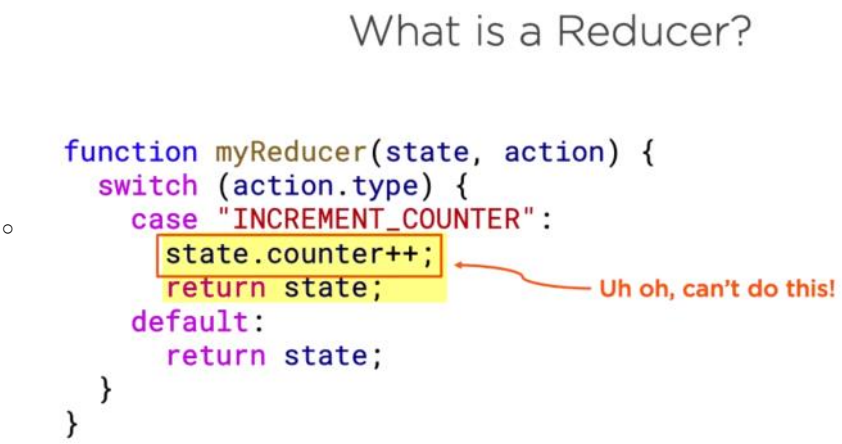
- **Handling Immutability**

- JavaScript doesn't have immutable data structures built in.
- If stores are immutable, how do we make sure that that happens?
- Well there are three approaches to consider.
  - First, the simplest way is to just educate your team and trust them.
  - If you want to put in a programmatic safety net, then we can install redux-immutable- state-invariant. This library displays an error when you try to mutate state anywhere in your app. We'll run this in our app so it will warn us if we accidentally mutate state. Be sure you only run this in development because it does a lot of object copying, and that can degrade performance in production.
  - Finally, if you want to programmatically enforce immutability, you can consider some of the libraries I mentioned earlier, such as immer or Immutable.js. Immer freezes objects so that they cannot be mutated, and Immutable.js creates immutable JavaScript data structures. Seamless-immutable is another interesting option to consider. Immer is recommended.

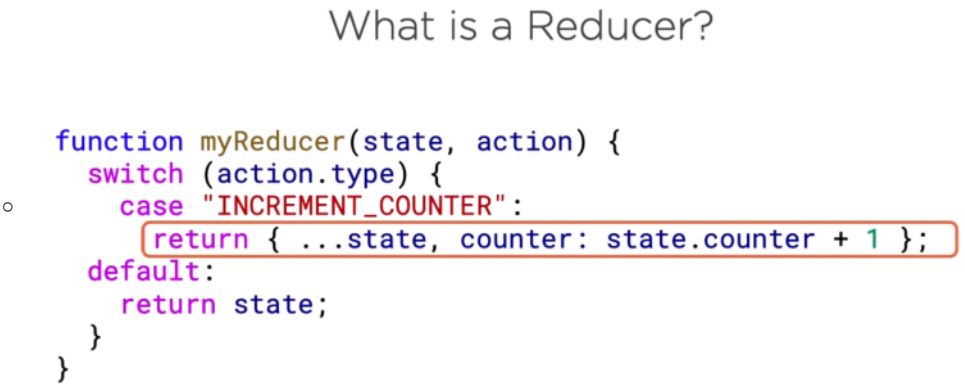
- **Reducers**

- To change the store, you dispatch an action that is ultimately handled by a reducer.
- A reducer is actually quite simple. It's a function that takes state and an action and returns new state.
- That's it. You can think of a reducer like a meat grinder. With a meat grinder, you put in some ingredients and turn the handle. The, uh, results, well, they come out the other side. In the same way, with reducers, you pass in some ingredients, in this case the current state and an action, and it returns new state.

- Here's a reducer that's handling incrementing a counter. Reducer functions just look at the action passed and return a new copy of state.



- So, for example, if the action passed was INCREMENT_COUNTER, then it would increment the counter and return the new state.
- The reducer knew what state needed to be changed by looking at the action passed, and it updated the state accordingly.
- However, we are mutating state right here. As we've discussed, in Redux, state is immutable. So in other words, it can't be changed.



- Let's update this example to return a new copy of state instead. The above example does not mutate state.
- I'm using object.spread to create a new copy of state. Create a new object by copying the existing state.

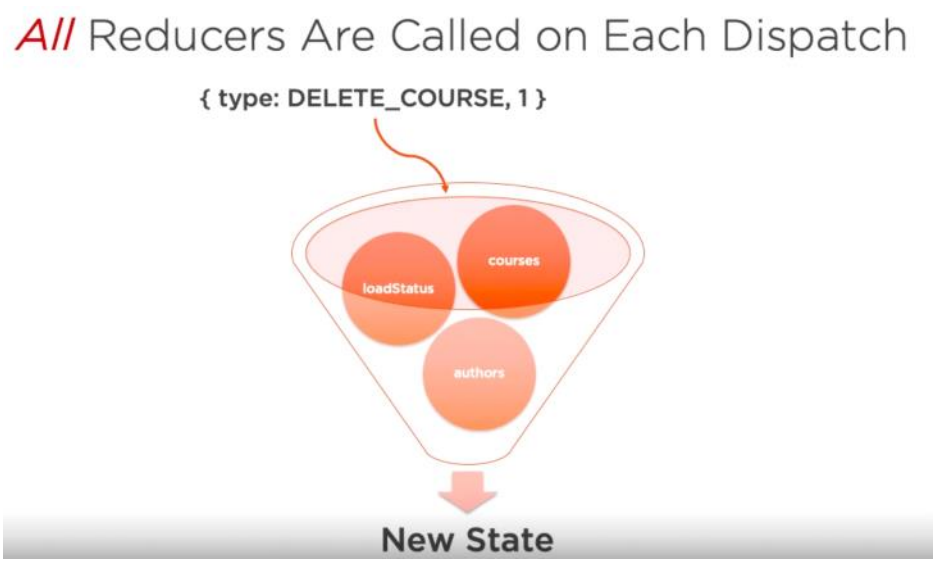
Forbidden in Reducers

- Mutate arguments
- Perform side effects
- Call non-pure functions

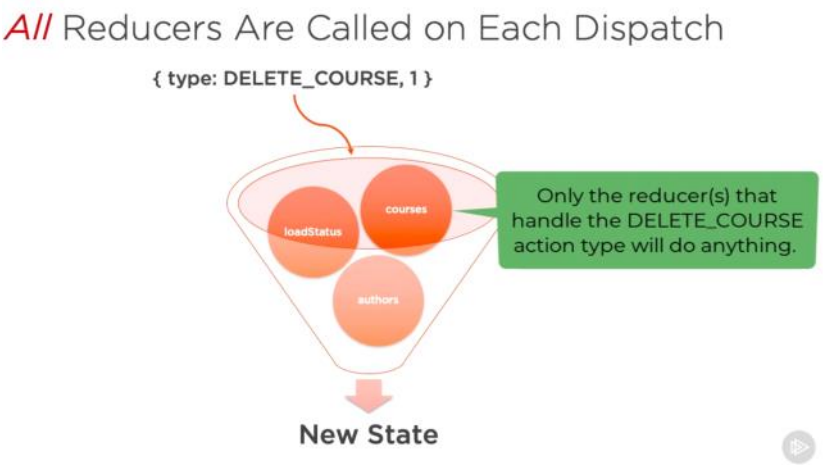
    - You should never mutate arguments.
    - You should never perform side effects, like API calls or routing transitions.
    - And you should never call non-pure functions.
        - A reducer's return value should depend solely on the values of its parameters, and it should call no other non-pure functions, such as date.now or math.random.
- Instead of mutating state, you return a copy of what was passed in, and Redux will use that to create the new store state.

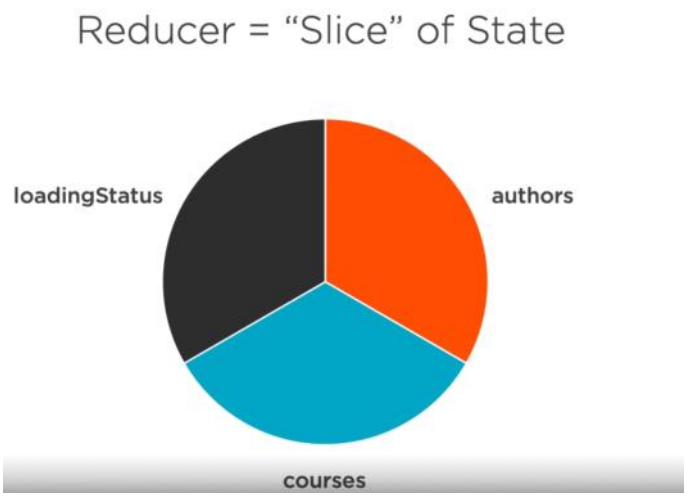
1 Store. Multiple Reducers.

- We can manage slices of state changes through multiple reducers in Redux.
- When a store is created, Redux calls the reducers and uses the return values as initial state.
- But if we have multiple reducers, which one is called when an action is dispatched? - All reducers get called when an action is dispatched. The switch statement inside each reducer looks at the action type to determine if it has anything to do. This is why all reducers should return the untouched state as the default. This way if no case matches the action type passed, the existing state is returned.



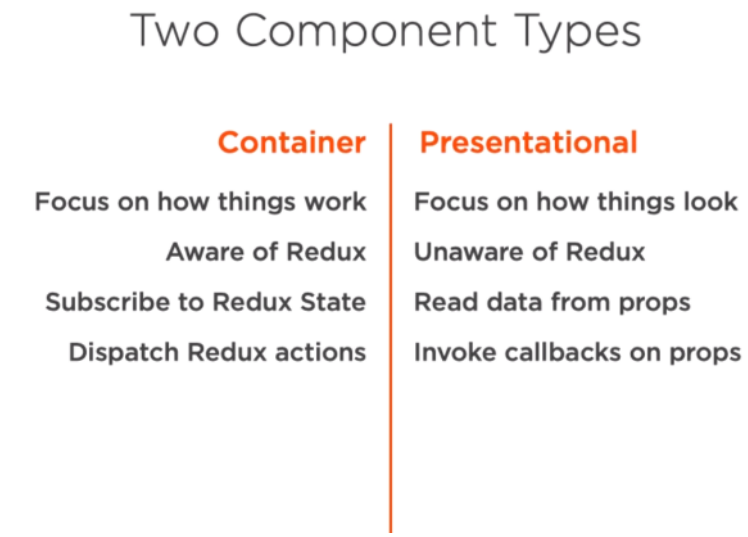
- So for example, if I dispatch the DELETE_COURSE action and my app has three reducers,
    - one for courses,
    - one for authors,
    - and one that handles loading status,
    - All three of these reducers will be called.
    - But only the reducer that actually handles the DELETE_COURSE action type will do anything.

- 

    - The others will simply return the state that was passed to them.
    - Each reducer only handles its slice of state. In fact, each reducer is only passed its slice of state, so it can only access the portion of state that it manages.
- 

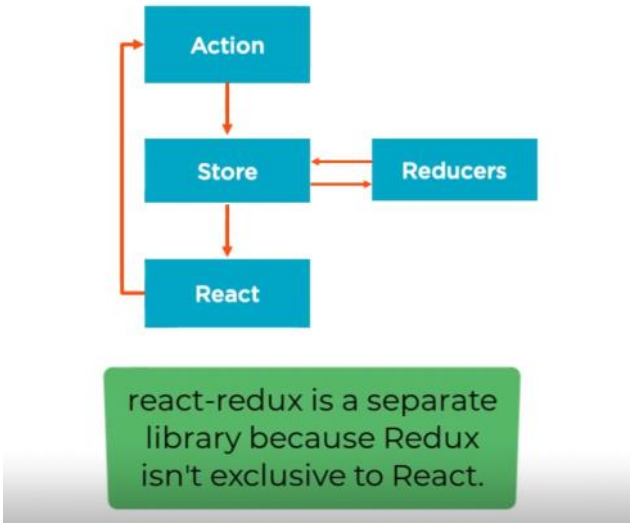
    - So while there's only a single store for Redux, creating multiple reducers allows you to handle changes to different pieces of the store in isolation. This makes state changes easy to understand, and it avoids issues with side effects.
    - There is no one- to-one mapping between reducers and actions. The Redux docs recommend using reducer composition. This means a given action can be handled by more than one reducer. Bottom line, remember, each action can be handled by one or more reducers. And each reducer can handle one or more actions.
- 

- **Container vs Presentational Components**

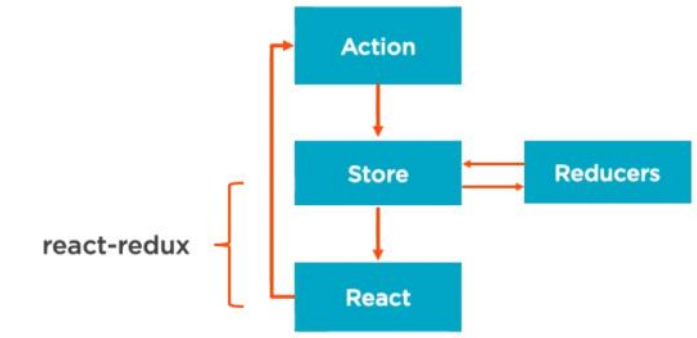


- Container components are focused on how things work. They handle data and state so that all the child components below can simply receive the data and actions they need via props. That's why they're called presentational components. They're focused on how things look.
- Container components are the only components in your system that are aware of Redux at all. This is a great thing because it means your child components are simply presentational components. They just receive data and actions via props and contain markup.
- Container components subscribe to Redux state, while presentational components read data from props. In a similar way, container components actually dispatch Redux actions, while presentational components fire off actions by invoking the callbacks that are passed down to them via props. So in this way, a presentational component isn't tied to a specific behavior. Its behavior is passed down from a container component via props.

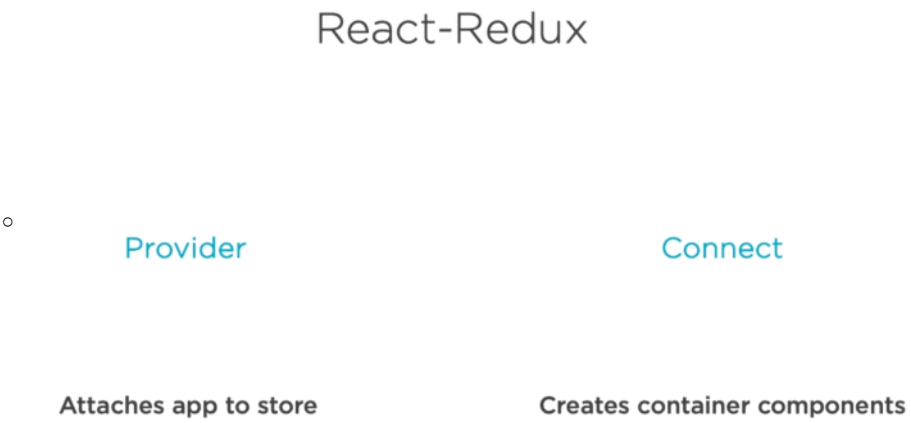
- **React-Redux Intro**
  - React-Redux is a companion library for Redux. It's a separate library because Redux isn't merely useful for React. Since Redux is simply a way tohandle state, you can use Redux with other libraries.



  - Redux can be used with vanilla js, jquery, angular, vue js etc..



- We will use the React-Redux library to connect our React components to the Redux store.
- React-Redux ties your React components together to Redux, and it consists of two core items, the **Provider component** and the **connect function**.



- **Provider component**: The Provider component is utilized at your app's route. It wraps the entire application. This is how the Provider component attaches your app to the Redux store.

  - The Provider component attaches your app to the Redux store. You declare the provider once in your app's entry point. So you use the Provider component to wrap your app's top-level component.
  - The Provider component uses React's context to pull this off. So the Provider component makes the store available to all child components without you having to pass the store down explicitly. You only need to use this once in your root component.

- **Connect function**: Connect is a function provided by React- Redux that connects our React components to the Redux store.

  - With this function, we can declare what parts of the store we'd like attached to our component's props. We declare what actions we want to expose on props as well.
  - When we use Redux with React, you use a function called **connect**. This function connects your React component to the store.
  - We pass connect two arguments, **mapStateToProps** and **mapDispatchToProps**.
  - **mapStateToProps** specifies what state you want to pass to your component on props.
  - **mapDispatchToProps** specifies what actions you want to pass to your component on props.

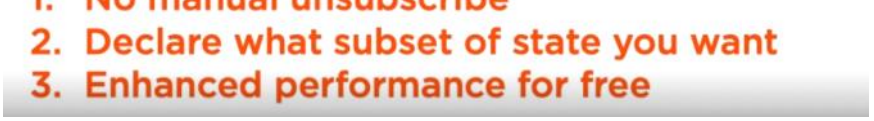
```
export default connect(mapStateToProps, mapDispatchToProps)(AuthorPage);
```

What state do you want to pass to your component?

```
export default connect(mapStateToProps, mapDispatchToProps)(AuthorPage);
```

What actions do you want to pass to your component?

- **Benefits to Redux's approach over plain Flux**



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

- **mapStateToProps**

React-Redux Connect

```
connect(mapStateToProps, mapDispatchToProps)
```

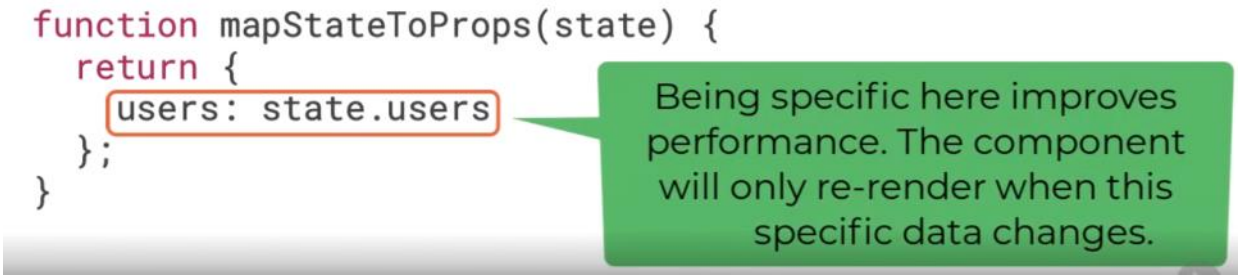
What state should I expose as props?

- The connect function accepts two parameters. Both are typically functions, and both of these parameters are optional.
- The first parameter is **mapStateToProps**. This function is useful for defining what part of the Redux store you want to expose as props on your component.
- When you define this function, the component will subscribe to the Redux store updates. And any time it updates, mapStateToProps will be called. This function returns an object.
- Each property on the object that you define will become a property on your container component. So in summary, the mapStateToProps argument determines what state is available on your container component.
- This is a logical place to filter or otherwise transform your state so that it's most conveniently shaped and sorted for your component's use.
- Let's consider a simple example.

```
function mapStateToProps(state) {
  return {
    appState: state
  };
}
```

- If you're building a simple app, you may have only one reducer and one container component.
- In this case, we just want to pass down all of your state. But as your app grows, you'll likely want to create multiple components to manage different pages or sections of your app, and you'll likely want to create different reducers to handle different slices of your store. This is an example of a simple mapStateToProps function that makes all of your state accessible to the component via props.

- With this setup, I could say this.props .appState within my component to access any state that's handled by my appState reducer.
- Now what if I only want to expose part of more store state in the component? Then I can specify the specific pieces of state that I want to expose via props here. Each object key will become a prop on my component.



- Being more specific is a win too because it helps with performance. Remember, React components re-render any time that props change. So you should only pass in props that your component needs. This way each connected component only re-renders when the props that it needs have changed.
- One important thing to note is every time the component is updated, the mapStateToProps function is called. So if we are doing something expensive in there, you'll want to use a library like Reselect for memoizing.

Reselect

- **Memoize for performance**

```
const getAllCoursesSelector = state => state.courses;

export const getCoursesSorted = createSelector(
  getAllCoursesSelector,
  courses => {
    return [...courses].sort((a, b) =>
      a.title.localeCompare(b.title, "en", { sensitivity: "base" })
    );
});
```

- Memoization is like caching for function calls. Each time a function is called, Reselect checks whether it was just called with those same parameters. And if it was, it doesn't call the function. Instead, it just returns the memoized value. This is useful for increasing performance by avoiding unnecessary operations.
- So if you're doing expensive operations in your mapping, for instance filtering, a list, or making expensive calculations, then memoization can help assure that these expensive operations occur less often.
- With Reselect, the selectors that you write in mapStateToProps are efficient because they're only computed when one of the arguments change. So if you're doing expensive work in mapStateToProps, consider using the Reselect library.

- **mapDispatchToProps**

React-Redux Connect

```
connect(mapStateToProps, mapDispatchToProps)
```

What actions do I want on props?

- The second argument that we pass to connect is mapDispatchToProps. This argument lets us specify what actions we want to expose as props.
- So this is conceptually very similar to mapStateToProps. The difference is this argument determines what actions we want to expose instead of what state.

```
function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(actions, dispatch)
  };
}
```

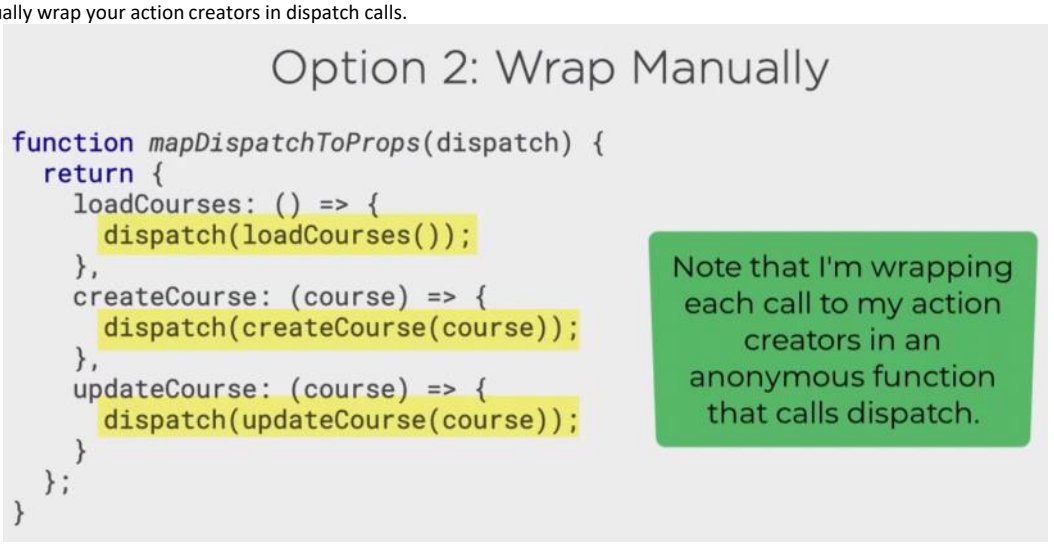
- MapDispatchToProps receives dispatch as its loan parameter. It returns the callback props that you want to pass down.
- The bindActionCreators function you see here is also part of Redux.
- To clarify it's use, we need to consider the different ways of passing actions to components using Redux.
- Redux is lightweight and rather unopinionated. So there are many ways to handle mapping your actions to props.
- Remember, mapDispatchToProps is how you expose your actions to your components. There are four options.

4 Ways to Handle mapDispatchToProps

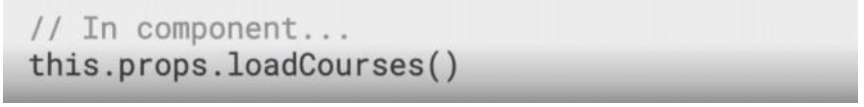
1. Ignore it
2. Wrap manually
3. bindActionCreators
4. Return object

1. To ignore the mapDispatchToProps function altogether.

   - It's an optional argument when you call connect. When you omit it, then the dispatch function will be attached to your container component. This means you can call dispatch manually and pass it an action creator.
   - However, there are a couple of downsides to this approach.
   - First, it requires more boilerplate each time you want to fire off an action because you have to explicitly call dispatch and pass it the action that you'd like to fire.
   - Second, this means that your child components need to reference Redux-specific concepts, like the dispatch function, as well as your action creators. If you want to keep your child components as simple as possible and avoid tying them to Redux, then this approach isn't ideal.
2. Manually wrap your action creators in dispatch calls.

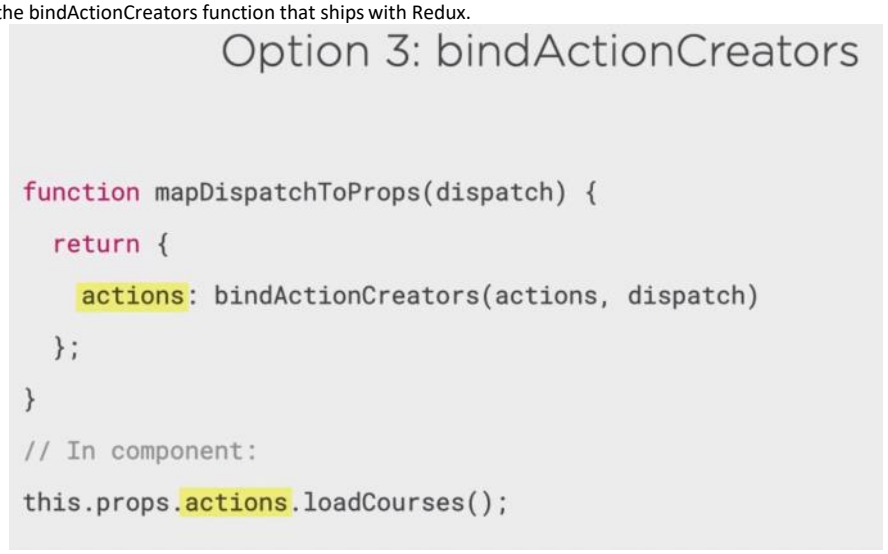


   - Here I'm specifying the actions that I want to expose to my component explicitly.
   - One by one, I wrap each action creator in a dispatch call.
   - When you're getting started, this is a nice option because manually wrapping action creators makes it clear what you're doing. But as you can see, it's rather redundant.



   - Each of the actions that I've specified above in mapStateToProps could be accessed under this.props in the component.

3. Use the bindActionCreators function that ships with Redux.

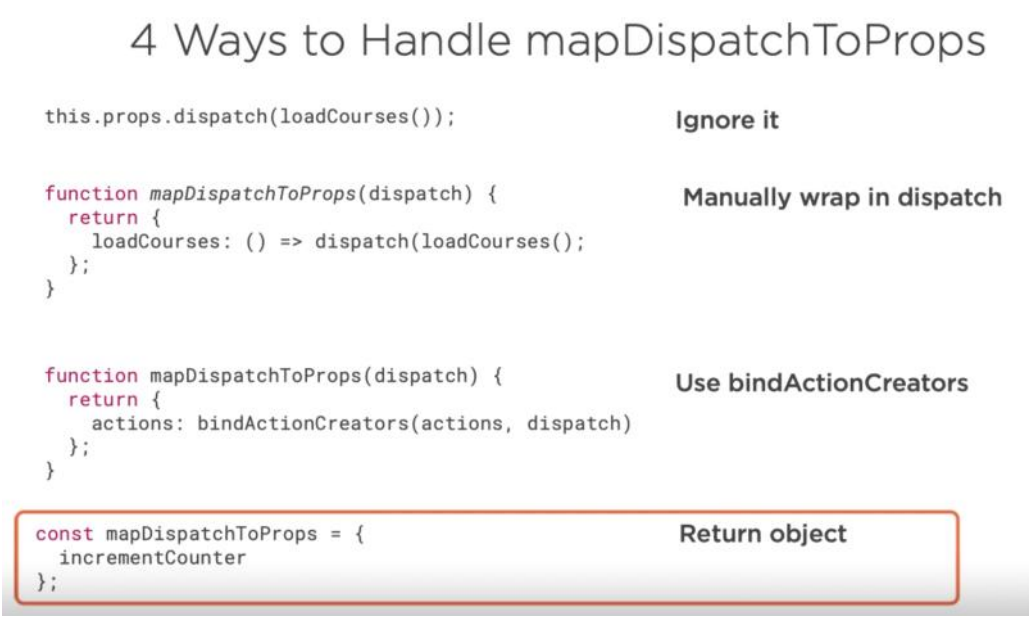


   - With this approach, you call bindActionCreators, and it will wrap the actions passed to it in a dispatch call for you.
   - Now the props created by these two examples will be slightly different. Notice that the prop that will be exposed to component will be called actions in this case.
   - So accessing the bound actions with this approach would require saying this.props .actions .actionName.
   - In summary, bindActionCreators wraps your action creators in a dispatch call for you to save you a little bit of typing.

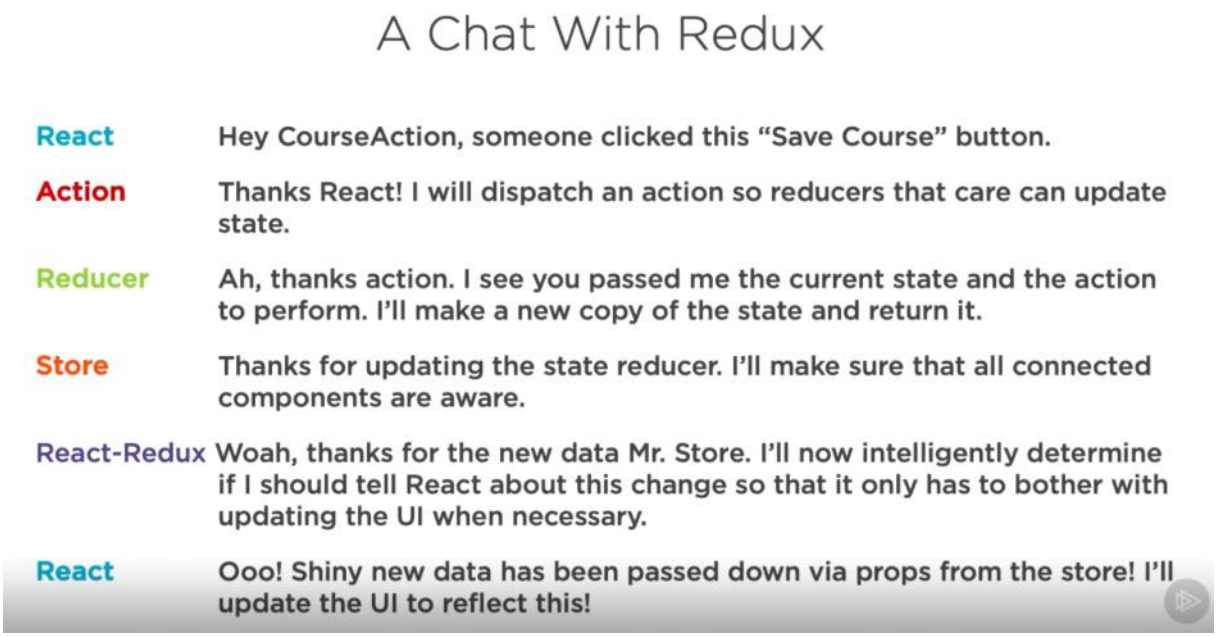
4. To declare mapDispatchToProps as an object instead of as a function.
   - In this example, assume that loadCourses is an action creator.

   - When you declare mapDispatchToProps as an object, Redux's connect will automatically wrap each action creator and dispatch for you.

- This approach is good because it's quite concise.

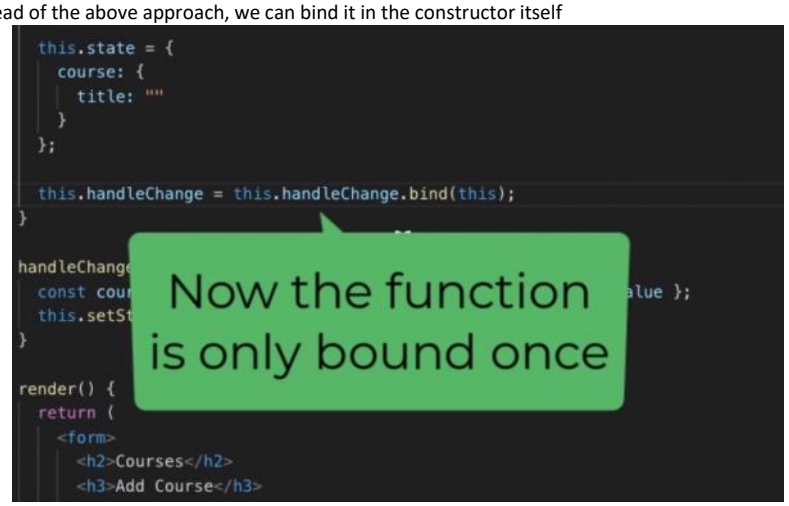


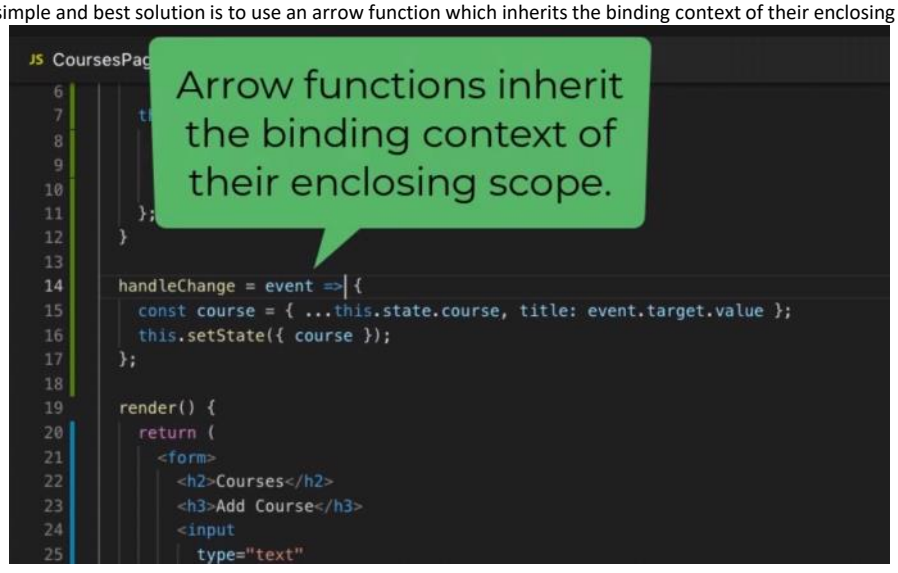
- **Data flow in Redux in form of chat**



- **Binding in classes**
  - Handling onChange events in input text using bind to bind the context of this will create a function every time the component is rendered.

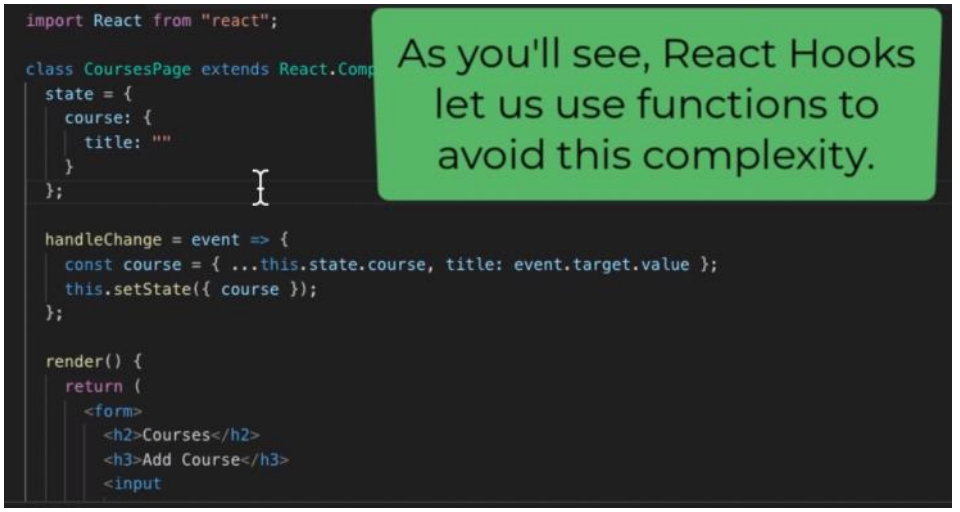
- 

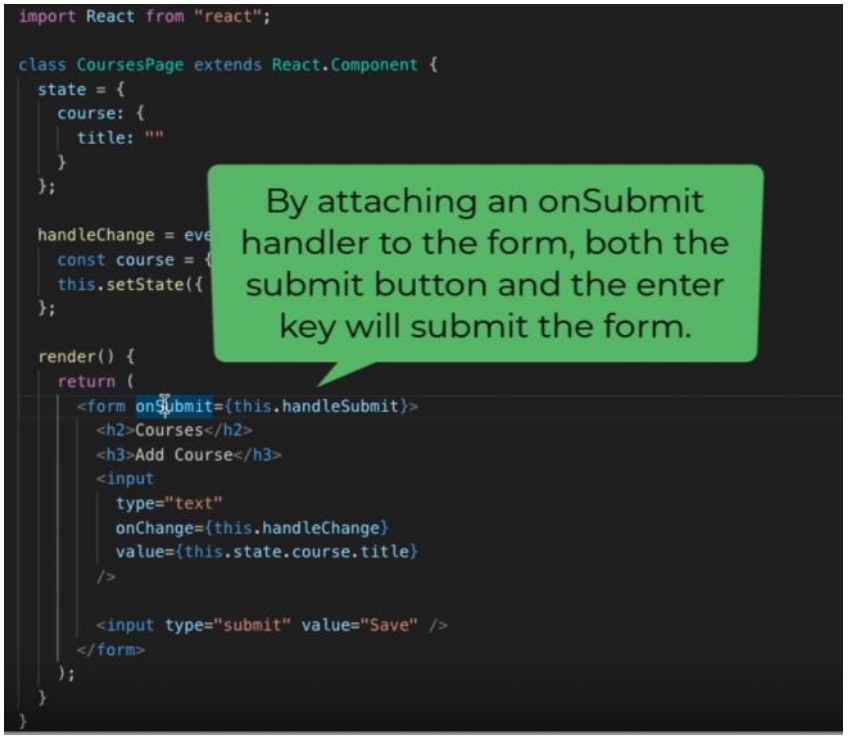
- Instead of the above approach, we can bind it in the constructor itself
  - 

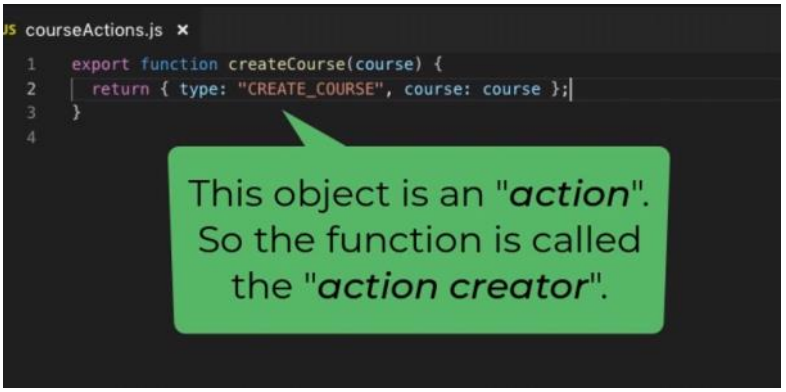
- The simple and best solution is to use an arrow function which inherits the binding context of their enclosing scope.
  - 

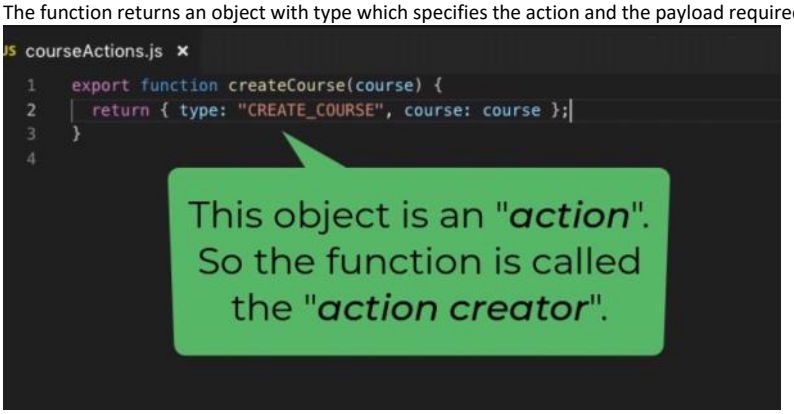
- We can also remove the constructor and modify the code as below

- 

- 

- **Handle Submit**
  - 

- 

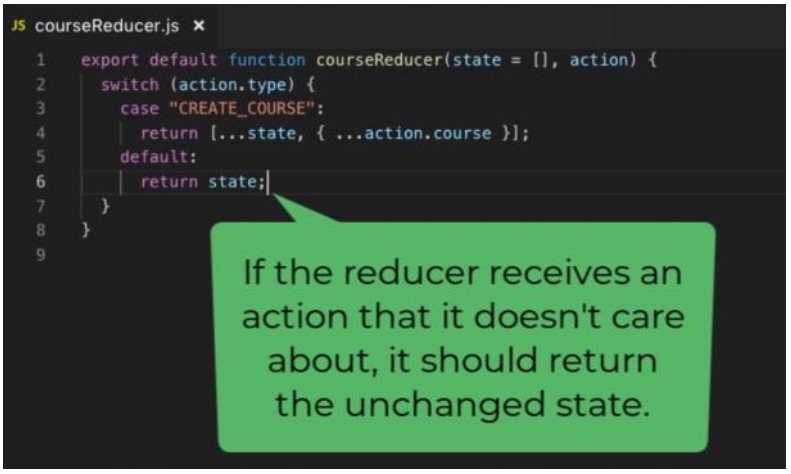
- **Creating Course Action**
  - We create a function and export it so that wherever courseAction is required it can be imported.
  - The function returns an object with type which specifies the action and the payload required.
  - 

  - 

- **Create Course Reducer and Root Reducer**
  - All reducer functions will have state and action as arguments.
  - 

    - In the above code, the state mutates which should not be done as state is immutable.
    - Below is the code to clone it and return new state.

- 

  - Here we make a clone of the state using spread operator and also clone the course passed.
  - This will create a new array with the list of all the existing courses and the additional course passed in the action.
- 

- Course is an simple array of objects here.
- But for larger data-sets we can consider storing by id instead as in the below screenshot.

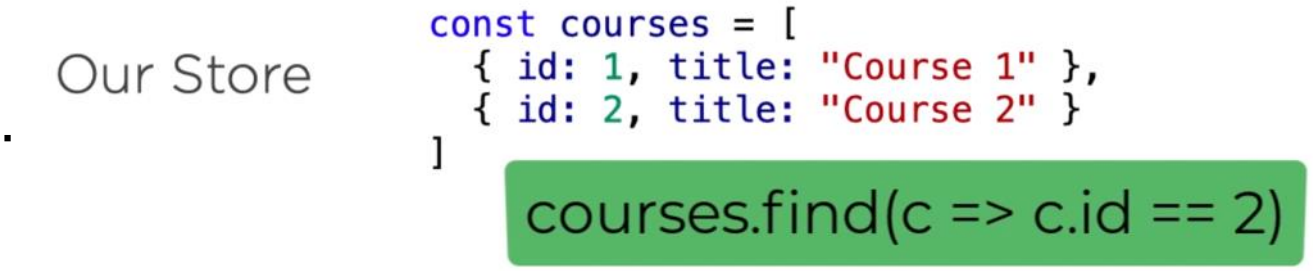
Our Store

```
const courses = [
  { id: 1, title: "Course 1" },
  { id: 2, title: "Course 2" }
]
```

By ID

```
const courses = {
  1: { id: 1, title: "Course 1" },
  2: { id: 2, title: "Course 2" }
}
```

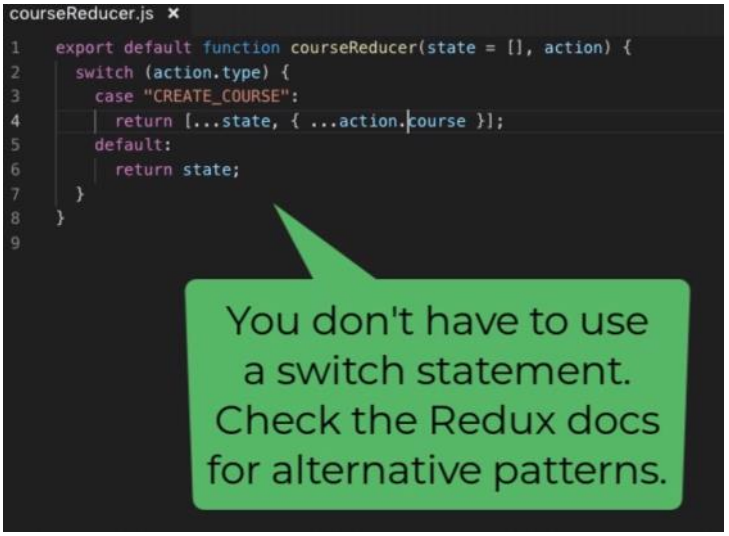
- Using array of objects, when we try to find an object we have to as below

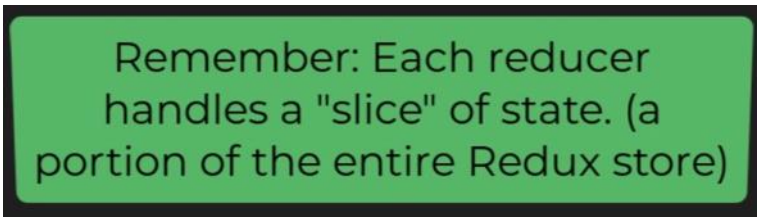
- 

  - But when we use storing by id, it will be simpler than using object as below

- 

  - For more information we can refer to "Normalizing state shape" in Redux docs.

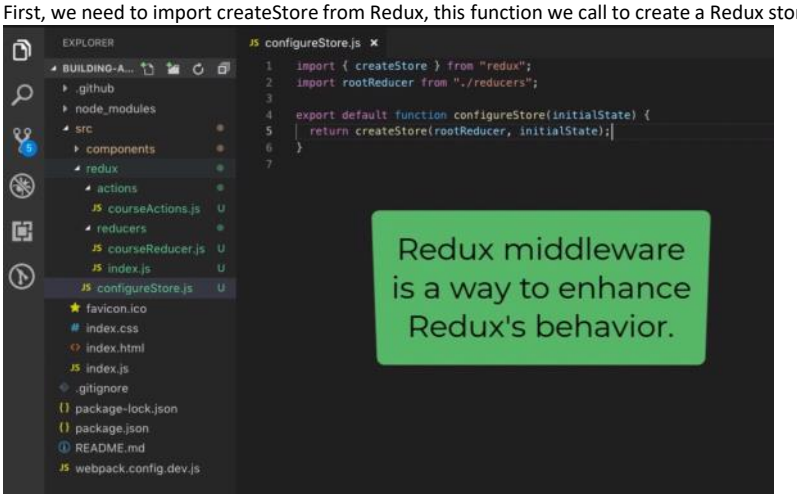
- 

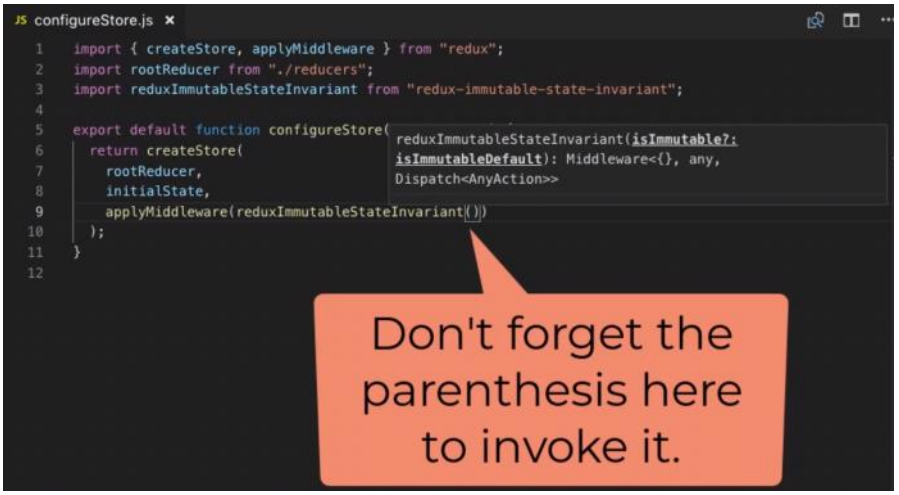
- 

  - With default export we can name the courseReducer to whatever we want when importing in another file.
  - Like in the below example, we imported from courseReducer as courses

- **Creating Root reducers using combineReducers**

-
```
index.js
1  import { combineReducers } from "redux";
2  import courses from "./courseReducer";
3
4  const rootReducer = combineReducers({
5    courses
6  });
7
8  export default rootReducer;
9
```

- **Creating store in react app**
  - There's no need to put this in a folder since we'll only have one of this type of file.
  - First, we need to import createStore from Redux, this function we call to create a Redux store.
  - 

  - Remember, Redux middleware is a way to enhance Redux with extra functionality.
  - To work with middleware, let's add another import up here called applyMiddleware.
  - The third parameter for createStore accepts the applyMiddleware function. And the middleware that we would like to apply is called reduxImmutableStateInvariant.
  - 

-

```
configureStore.js
1  import { createStore, applyMiddleware } from "redux";
2  import rootReducer from "./reducers";
3  import reduxImmutableStateInvariant from "redux-immutable-state-invariant";
4
5  export default function configureStore(initialState) {
6    return createStore(
7      rootReducer,
8      initialState,
9      applyMiddleware(reduxImmutableStateInvariant())
10   );
11 }
12
```

This will warn us if we accidentally mutate Redux state

-

```
configureStore.js
1  import { createStore, applyMiddleware, compose } from "redux";
2  import rootReducer from "./reducers";
3  import reduxImmutableStateInvariant from "redux-immutable-state-invariant";
4
5  export default function configureStore(initialState) {
6    const composeEnhancers =
7      window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose; // add support for Redux dev tools
8
9    return createStore(
10     rootReducer,
11     initialState,
12     composeEnhancers(applyMiddleware(reduxImmutableStateInvariant()))
13   );
14 }
15
```

  - **Redux DevTools**: They're handy for interacting with our Redux store.
    - And with this, we can now come down to our call to applymiddleware and prefix it with a composeEnhancers call because this composeEnhancers variable that we declared on line 6 gives us a function that we can call.
    - So composeEnhancers calls apply Middleware, and reduxImmutableStateInvariant is a piece of middleware that we're using. And now we'll be able to interact with our Redux store using the Redux DevTools in the browser.

- **Instantiating store and provider in index.js**
  -

```
index.js
1  import React from "react";
2  import { render } from "react-dom";
3  import { BrowserRouter as Router } from "react-router-dom";
4  import "bootstrap/dist/css/bootstrap.min.css";
5  import App from "./components/App";
6  import "./index.css";
7  import configureStore from './redux/configureStore';
8
9  const store = configureStore();
10
11 render(
12   <Router>
13     <App />
14   </Router>,
15   document.getElementById("ap
16 );
17
```

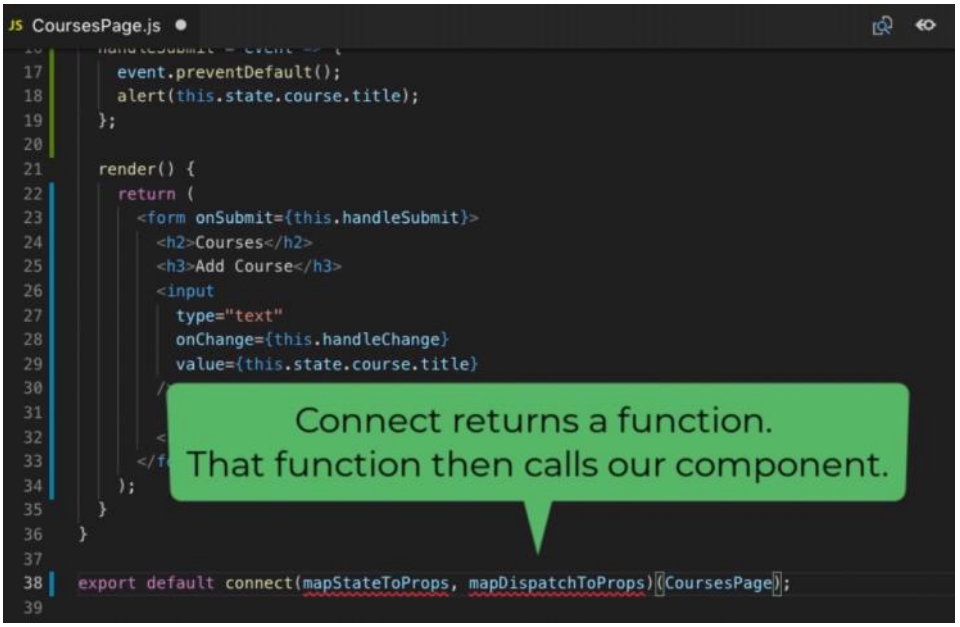
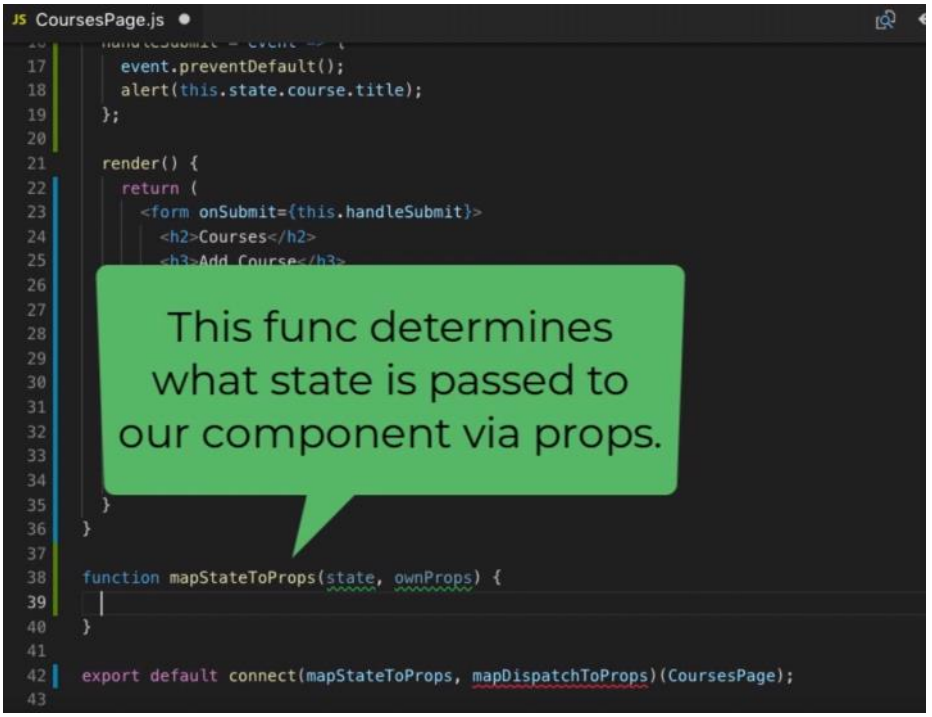
It can be useful to pass initial state into the store here if you're server rendering or initializing your Redux store with data from localStorage.

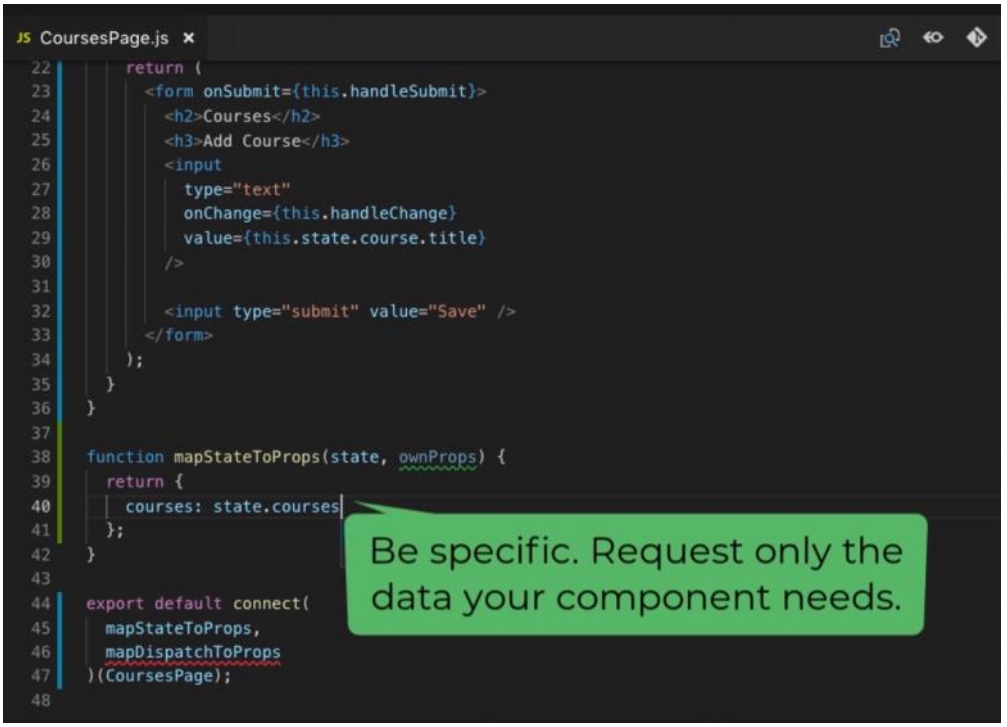
```
index.js
1  import React from "react";
2  import { render } from "react-dom";
3  import { BrowserRouter as Router } from "react-router-dom";
4  import "bootstrap/dist/css/bootstrap.min.css";
5  import App from "./components/App";
6  import "./index.css";
7  import configureStore from './redux/configureStore';
8  import { Provider as ReduxProvider} from 'react-redux';
```

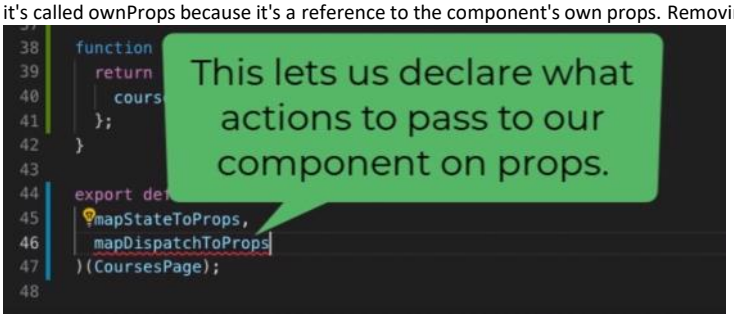
-
```
import React from "react";
import { render } from "react-dom";
import { BrowserRouter as Router } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import App from "./components/App";
import "./index.css";
import configureStore from './redux/configureStore';
import { Provider as ReduxProvider} from 'react-redux';

const store = configureStore();

render(
  <Router>
    <App />
  </Router>,
  document.getElementById("app")
);
```
This will provide Redux store data to our React components.
  - Provider is a higher-order component that provides your Redux store data to child components.

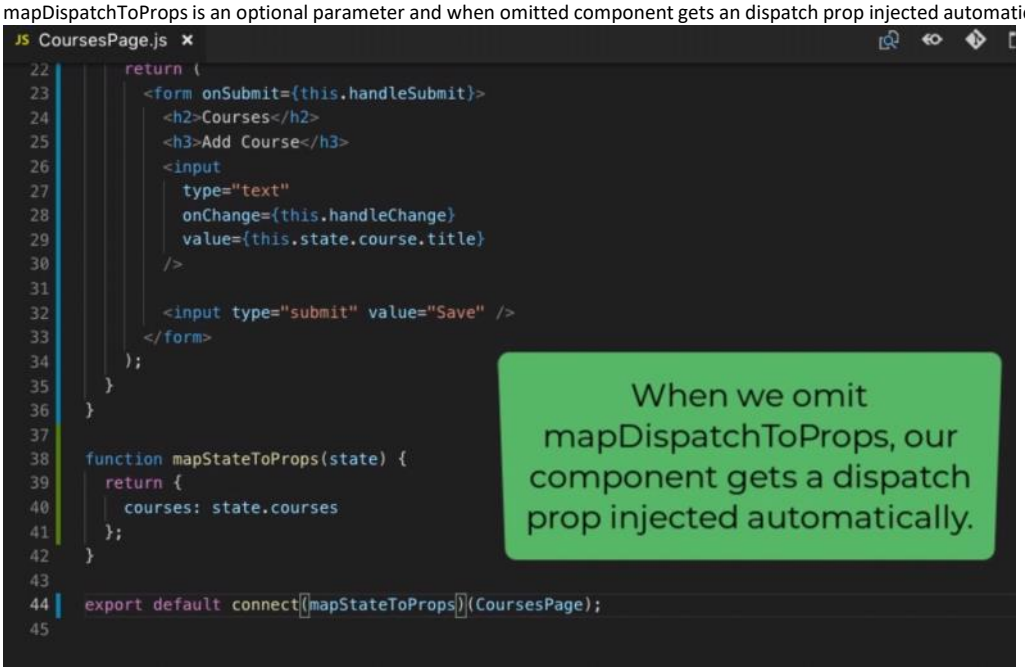
```
import React from "react";
import { render } from "react-dom";
import { BrowserRouter as Router } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import App from "./components/App";
import "./index.css";
import configureStore from "./redux/configureStore";
import { Provider as ReduxProvider } from "react-redux";

const store = configureStore();

render(
  <ReduxProvider store={store}>
    <Router>
      <App />
    </Router>
  </ReduxProvider>,
  document.getElementById("app")
);
```

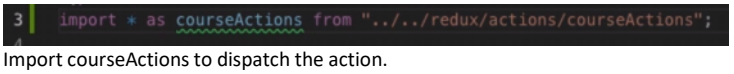
- **Connect Container Component**
  - Import connect from react-redux library
  -
```
import React from "react";
import {connect } from 'react-redux'
```
  - The connect function connects our components to Redux. And we typically call these components container components.
  - The connect function takes two parameters. The first is mapStateToProps, and the second is mapDispatchToProps. Then we take the results of this and call CoursesPage.

- 
CoursesPage.js
event.preventDefault();
alert(this.state.course.title);
};
render() {
return (
<form onSubmit={this.handleSubmit}>
<h2>Courses</h2>
<h3>Add Course</h3>
<input
type="text"
onChange={this.handleChange}
value={this.state.course.title}
Connect returns a function.
That function then calls our component.
export default connect(mapStateToProps, mapDispatchToProps)(CoursesPage);
- 
CoursesPage.js
event.preventDefault();
alert(this.state.course.title);
};
render() {
return (
<form onSubmit={this.handleSubmit}>
<h2>Courses</h2>
This func determines
what state is passed to
our component via props.
function mapStateToProps(state, ownProps) {
export default connect(mapStateToProps, mapDispatchToProps)(CoursesPage);

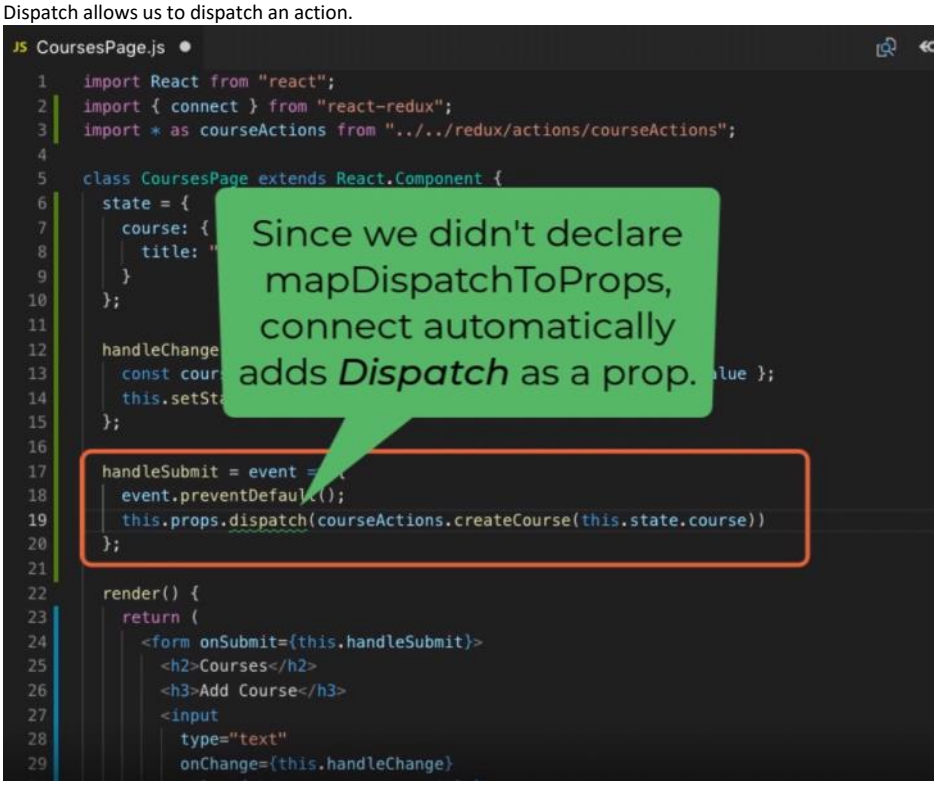
- 

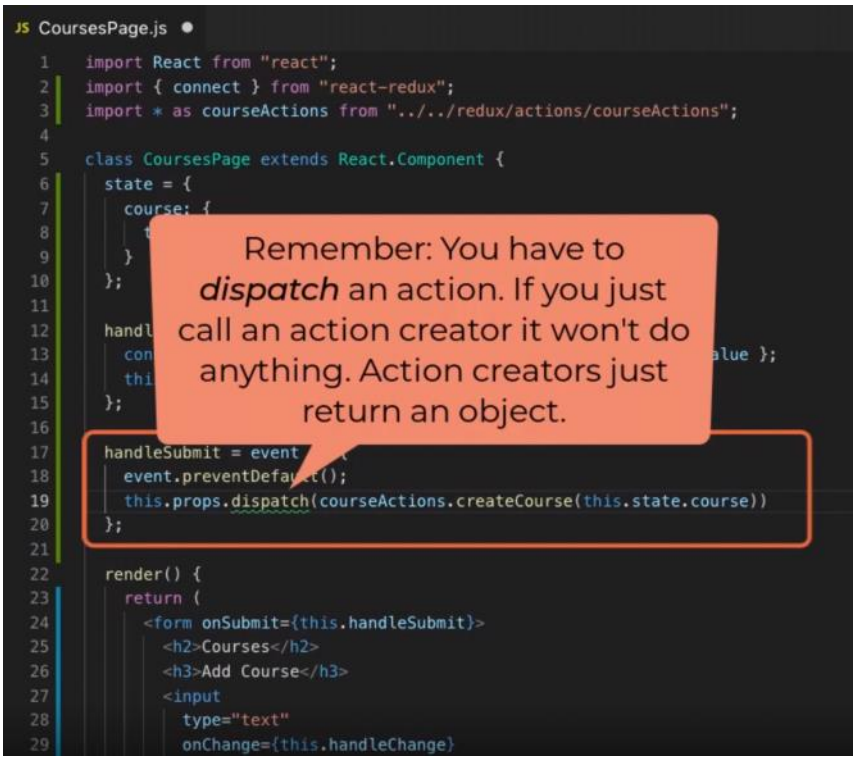
  - When declaring mapStateToProps, be as specific as possible about the data that you expose to the component.
  - For example, if you expose the entire Redux store, then the component will re-render when any data changes in the Redux store, which is not good. Request exactly the data that your component needs and no more.
  - We're not using the second parameter, ownProps. This parameter lets us access props that are being attached to this component. That's why it's called ownProps because it's a reference to the component's own props. Removing it as it is not required now.
- 

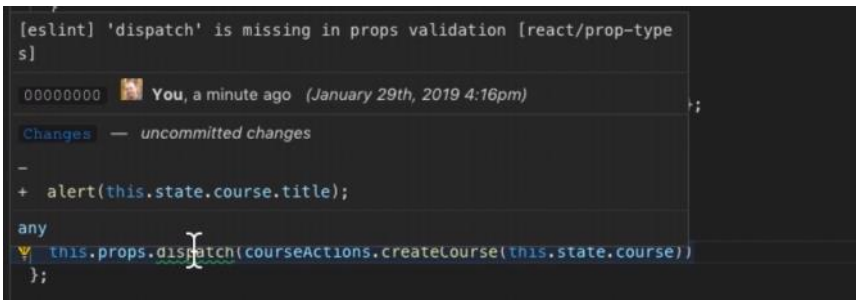
  - mapDispatchToProps is an optional parameter and when omitted component gets an dispatch prop injected automatically.
- 

- 

  - Import courseActions to dispatch the action.

- Dispatch allows us to dispatch an action.
- 

- 

- 

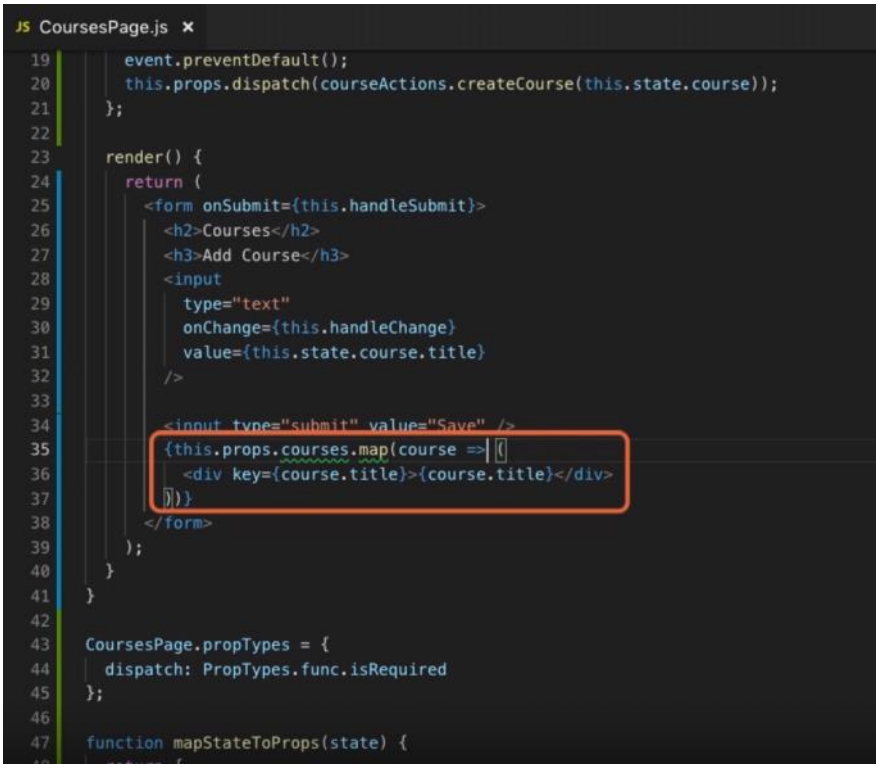
- We are getting an ESLint warning because dispatch isn't declared as a prop type for our component.

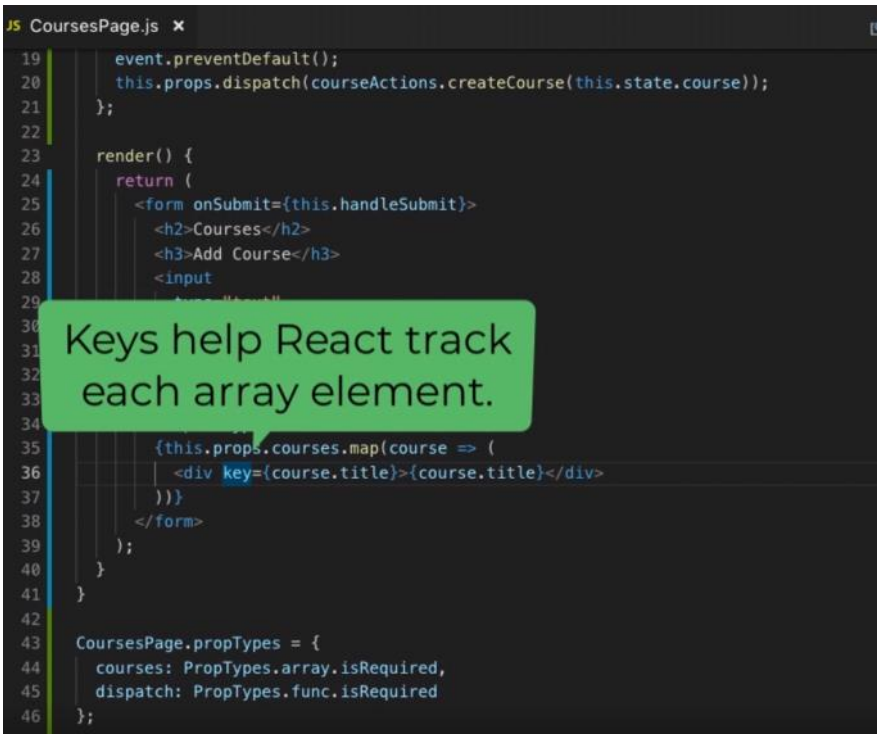
- Prop types help us specify the props that our component accepts, and this helps us catch errors when we're not getting the data that we expect, and it also documents each component's interface.
- ```
  import PropTypes from 'prop-types';
  ```
- ```
  CoursesPage.propTypes = {
    dispatch: PropTypes.func.isRequired
  };
  ```
- CoursesPage.js

  ```
  return (
    <form onSubmit={this.handleSubmit}>
      <h2>Courses</h2>
      <h3>Add Course</h3>
      <input
        type="text"
        onChange={this.handleChange}
        value={this.state.course.title}
      />

      <input type="submit" value="Save" />
    </form>
  );
  }
  }

  CoursesPage.propTypes = {
    dispatch: PropTypes.func.isRequired
  };

  function mapStateToProps(sta
    return {
      courses: state.courses
    };
  }

  export default connect(mapStateToProps)(CoursesPage);
  ```

  connect automatically passes dispatch in if we omit mapDispatchToProps here.
- Now we can expect dispatch to be passed in to the CoursesPage component, and it will be passed in because connect automatically passes dispatch in if we omit that second argument, which was mapDispatchToProps.

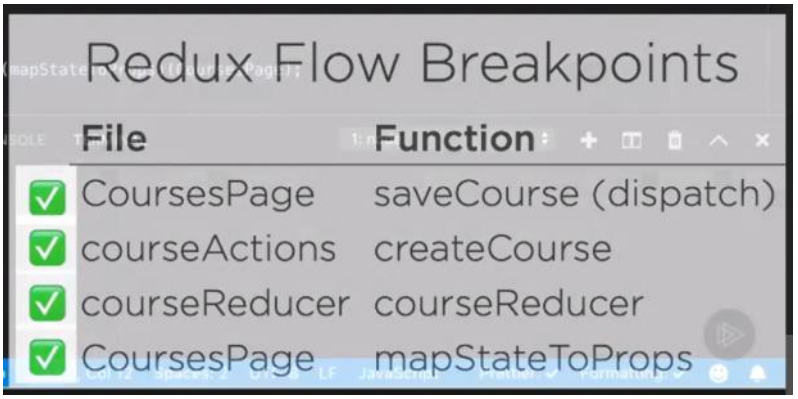
- **Step through Redux Flow and Try Redux DevTools**
  - First let us change the render method to list all the courses

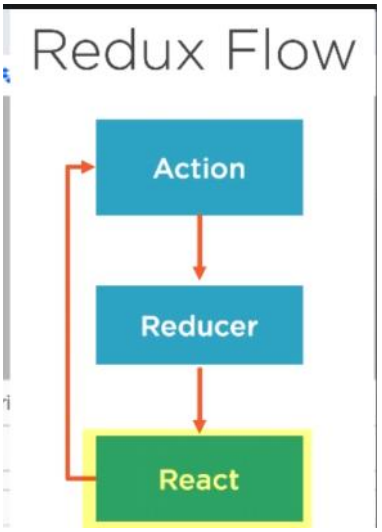
- 

- 

- We get a warning over here if we use courses on props, as we didn't declare that down on our PropTypes. So we should add courses in propTypes.
- Any time we're iterating over an array, React expects us to provide a key. Keys help React keep track of each element in an array, and they're useful for performance reasons. A key should beunique to the array. So ideally, we'd set the key to an ID.

- Below is the Redux flow for coursesPage

- 

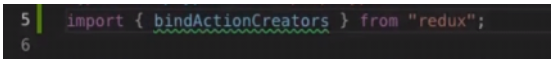
- 

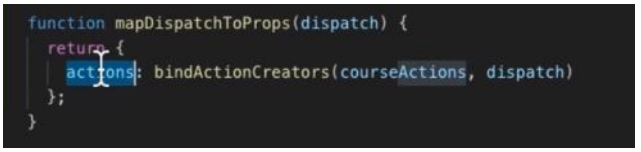
- **mapDispatchToProps: Manual Mapping**
  - 

  - Dispatch is the function that notifies redux about an action.
  - When we manually write the mapDispatchToProps below is the code to handle it to dispatch an action

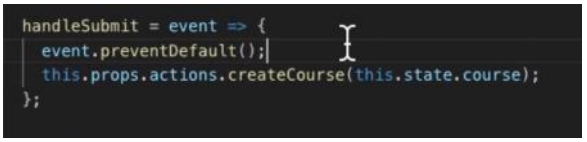
- 

- 

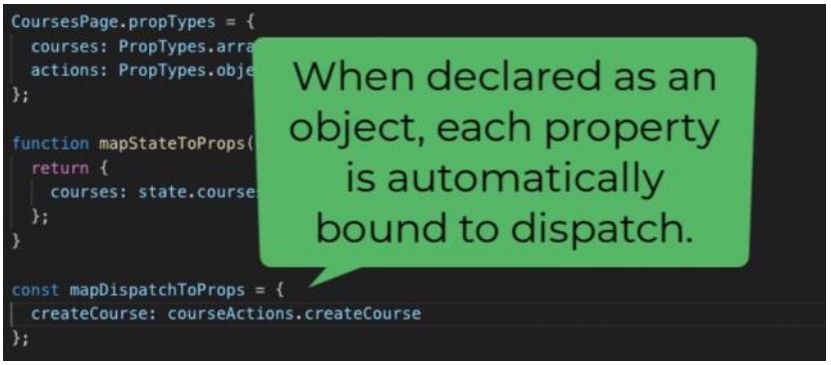
- **mapDispatchToProps: bindActionCreators**
  - Using mapDispatchToProps helps simplify dispatching our action within our component. But our call down here inmapDispatchToProps is still quite verbose.
  - Now Redux comes with a helper function to save us from having manually wrap our action creators in a dispatch call, and this function is called bindActionCreators.
  - 

  - 

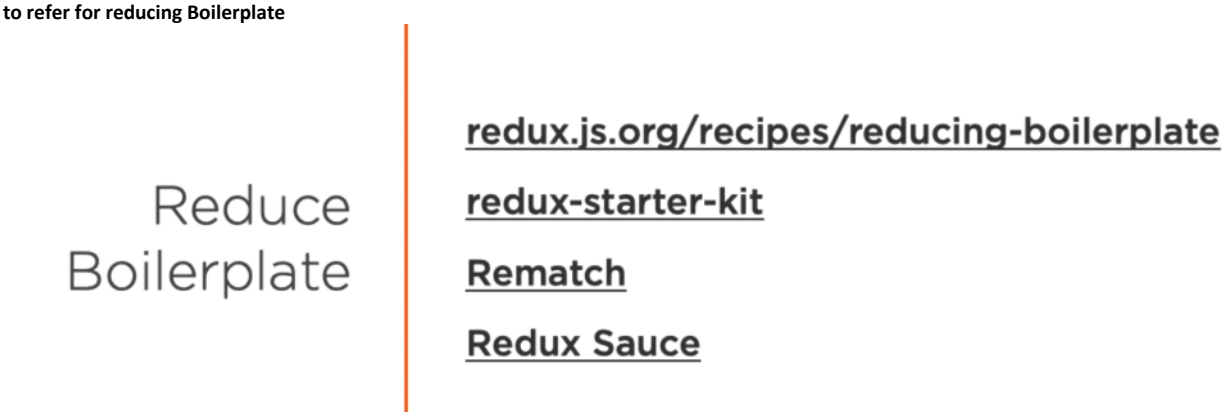
  - Here in the above code we are passing all actions available in courseActions but we can specify even a single one action. Actions will be an object.
  - 

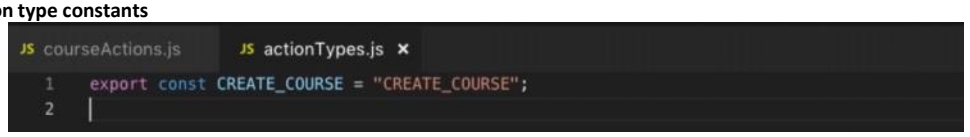
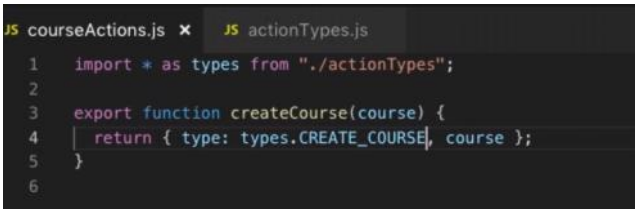
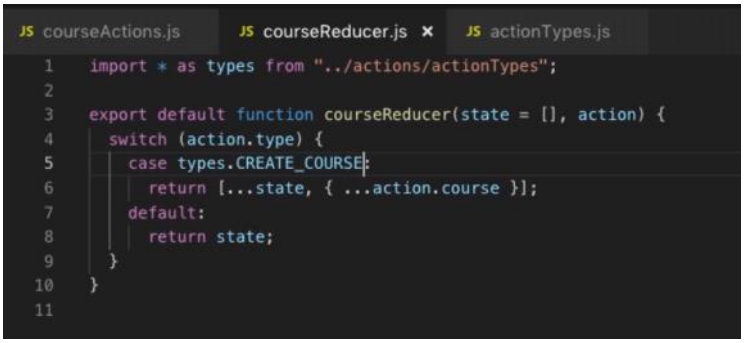
  - 

- **mapDispatchToProps: Object Form**

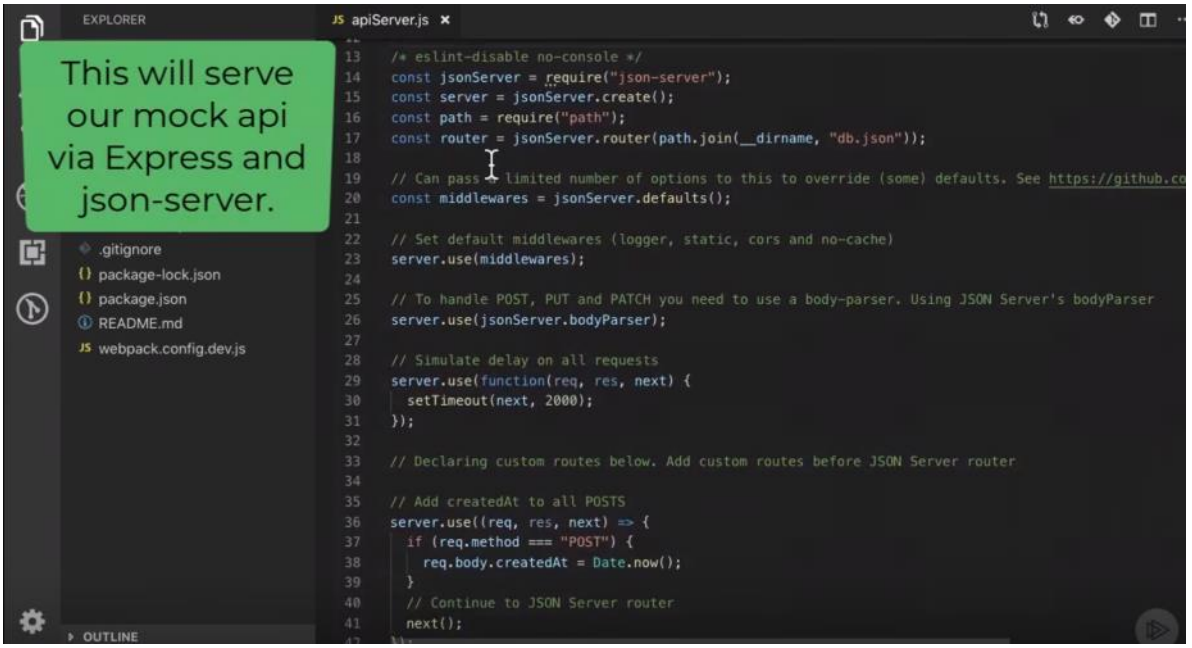
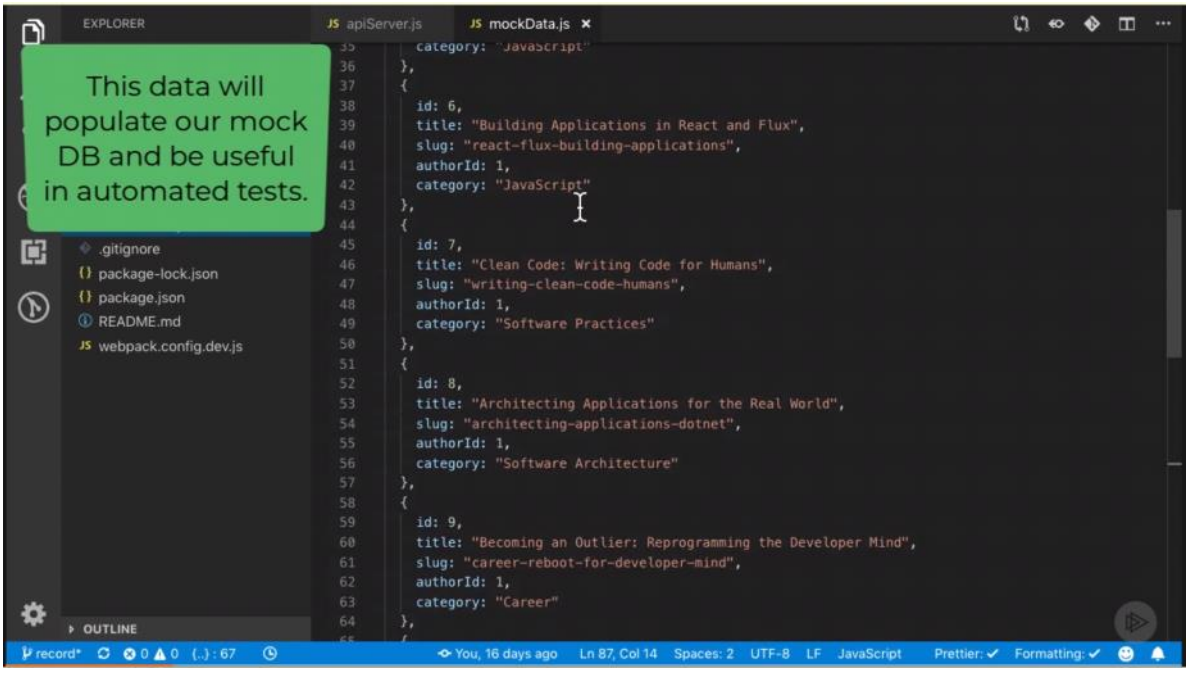
- 

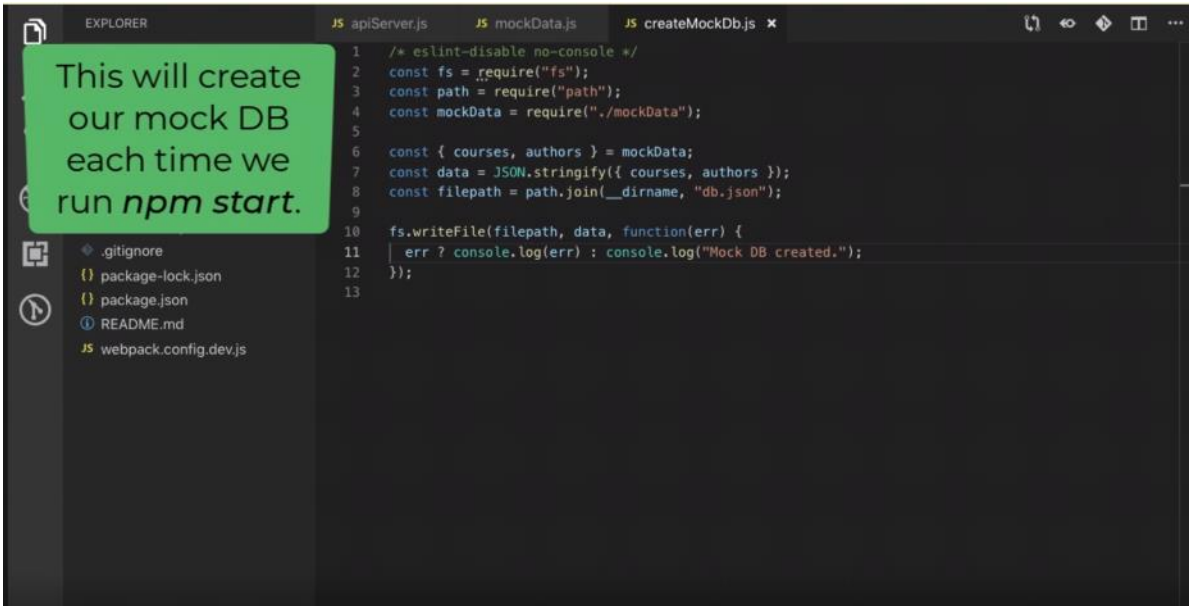
  - When we declare mapDispatchToProps as an object, then each property is expected to be an actionCreator function, and that's exactly what courseActions.createCourse is.
  - What happens is connect will automatically go through and bind each of these functions in a call to dispatch for you.

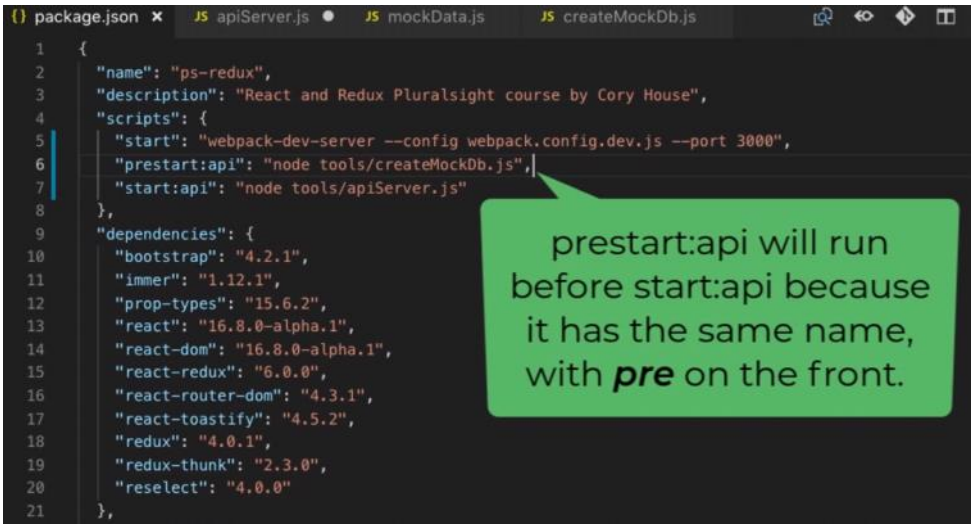
- **Docs to refer for reducing Boilerplate**
  - 

- **Action type constants**
  - 
```
courseActions.js   actionTypes.js ×
1  export const CREATE_COURSE = "CREATE_COURSE";
2
```
  - 
```
courseActions.js ×   actionTypes.js
1  import * as types from "./actionTypes";
2
3  export function createCourse(course) {
4    return { type: types.CREATE_COURSE, course };
5  }
6
```
  - 
```
courseActions.js   courseReducer.js ×   actionTypes.js
1  import * as types from "../actions/actionTypes";
2
3  export default function courseReducer(state = [], action) {
4    switch (action.type) {
5      case types.CREATE_COURSE:
6        return [...state, { ...action.course }];
7      default:
8        return state;
9    }
10 }
11
```

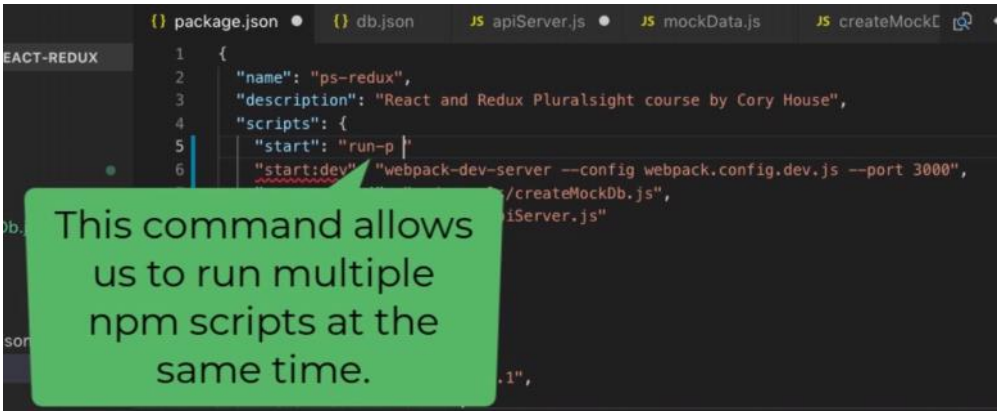
- Mock API Setup

- 
This will serve our mock api via Express and json-server.
EXPLORER
apiServer.js
.gitignore
package-lock.json
package.json
README.md
webpack.config.dev.js
OUTLINE
/* eslint-disable no-console */
const jsonServer = require("json-server");
const server = jsonServer.create();
const path = require("path");
const router = jsonServer.router(path.join(__dirname, "db.json"));
// Can pass a limited number of options to this to override (some) defaults. See https://github.co
const middlewares = jsonServer.defaults();
// Set default middlewares (logger, static, cors and no-cache)
server.use(middlewares);
// To handle POST, PUT and PATCH you need to use a body-parser. Using JSON Server's bodyParser
server.use(jsonServer.bodyParser);
// Simulate delay on all requests
server.use(function(req, res, next) {
setTimeout(next, 2000);
});
// Declaring custom routes below. Add custom routes before JSON Server router
// Add createdAt to all POSTS
server.use((req, res, next) => {
if (req.method === "POST") {
req.body.createdAt = Date.now();
}
// Continue to JSON Server router
next();
- 
This data will populate our mock DB and be useful in automated tests.
EXPLORER
apiServer.js
mockData.js
.gitignore
package-lock.json
package.json
README.md
webpack.config.dev.js
OUTLINE
category: "JavaScript"
},
{
id: 6,
title: "Building Applications in React and Flux",
slug: "react-flux-building-applications",
authorId: 1,
category: "JavaScript"
},
{
id: 7,
title: "Clean Code: Writing Code for Humans",
slug: "writing-clean-code-humans",
authorId: 1,
category: "Software Practices"
},
{
id: 8,
title: "Architecting Applications for the Real World",
slug: "architecting-applications-dotnet",
authorId: 1,
category: "Software Architecture"
},
{
id: 9,
title: "Becoming an Outlier: Reprogramming the Developer Mind",
slug: "career-reboot-for-developer-mind",
authorId: 1,
category: "Career"
},
record*
You, 16 days ago
Ln 87, Col 14
Spaces: 2
UTF-8
LF
JavaScript
Prettier:
Formatting:

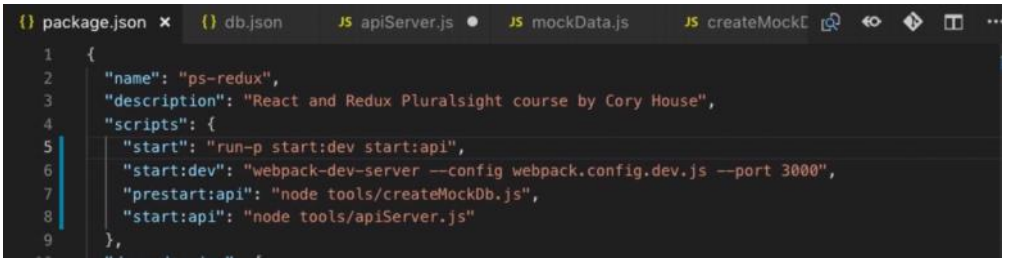
- 

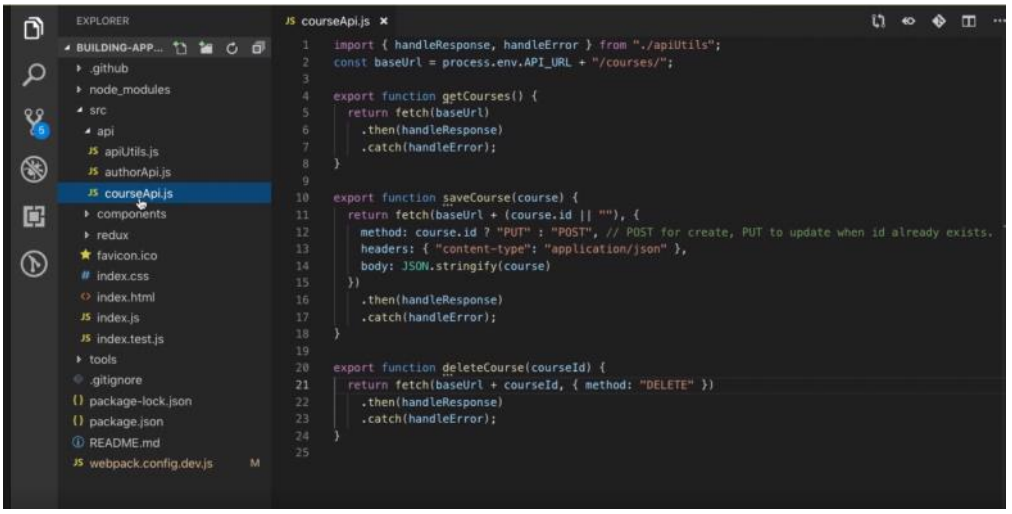
- 

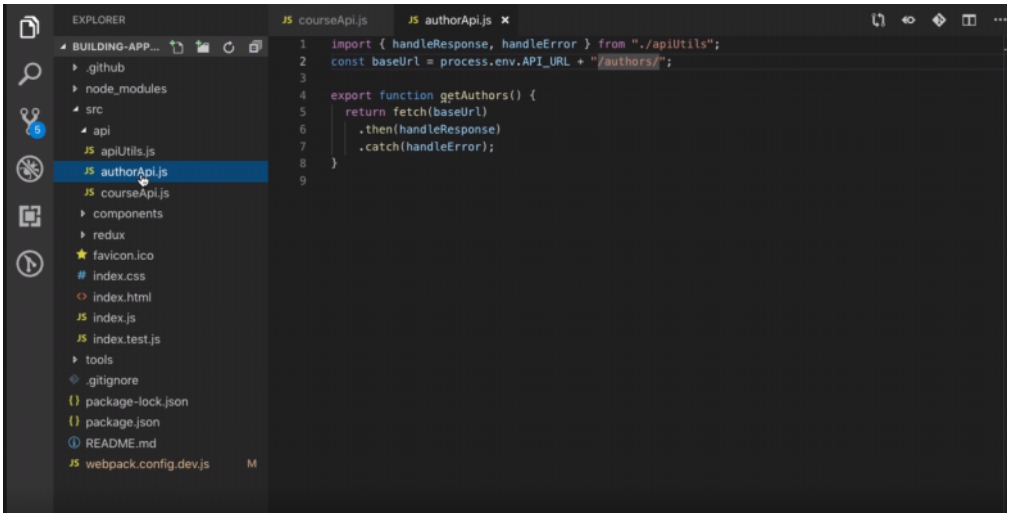
- `npm run start:api`
- 

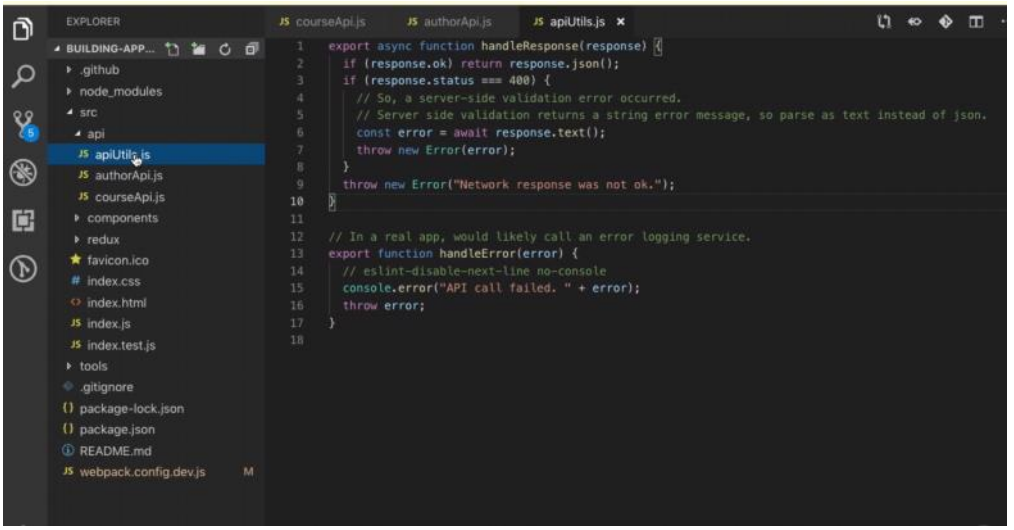
- 

- **run-p** : This is a commandthat comes with npm run all, which is already installed since it's one of the packages listed in package.json. The run-p command allows us to provide a list of npm scripts that we want to run in parallel. That's what the p stands for.







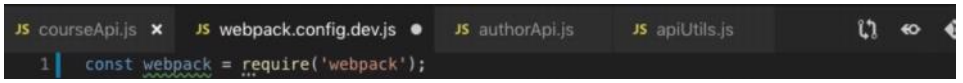
- 

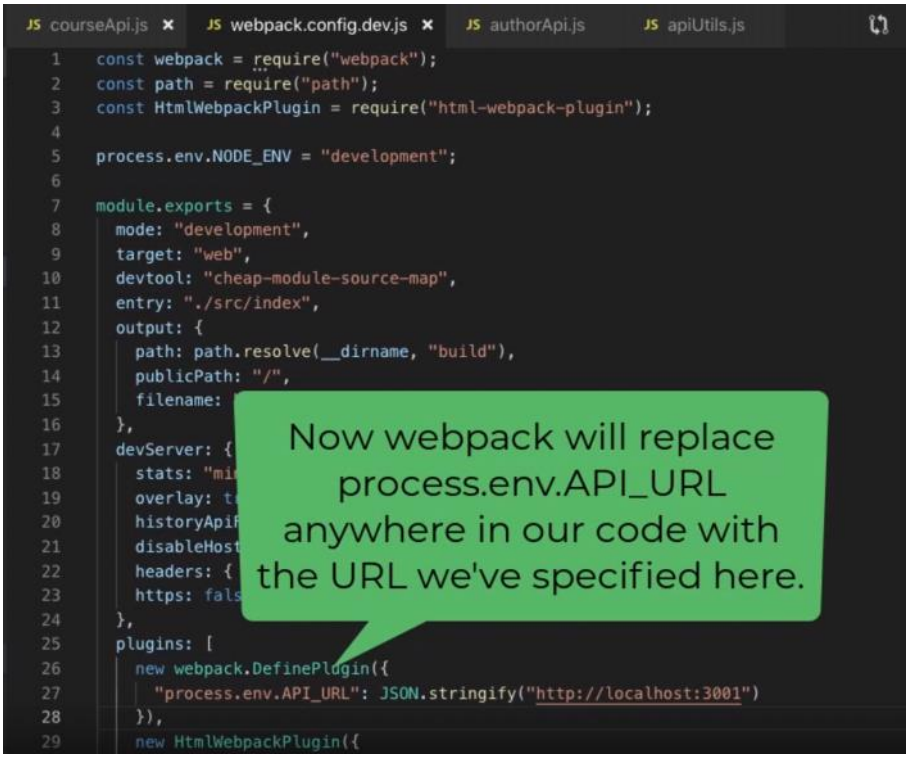
- 

- 

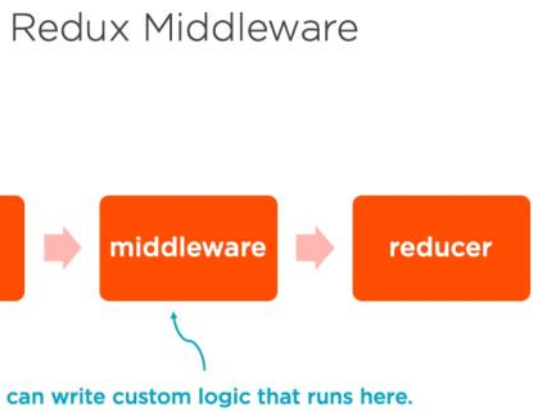
- 

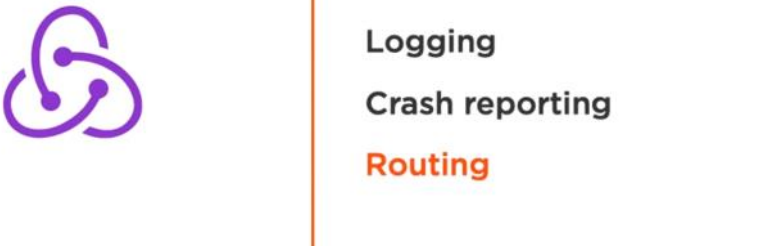
- Fetch to make API calls. Fetch is built in to modern browsers, so we can make API calls without installing an extra library. Fetch is a promise-based API. With promises, the then function is called when the asynchronous call is complete. And if an error occurs, the catch function is called.
- Fetch defaults to GET
- Http verbs: GET, PUT, POST, DELETE
- `const baseUrl = process.env.API_URL + "/courses/";`
- We can configure the above in webpack
- Import webpack

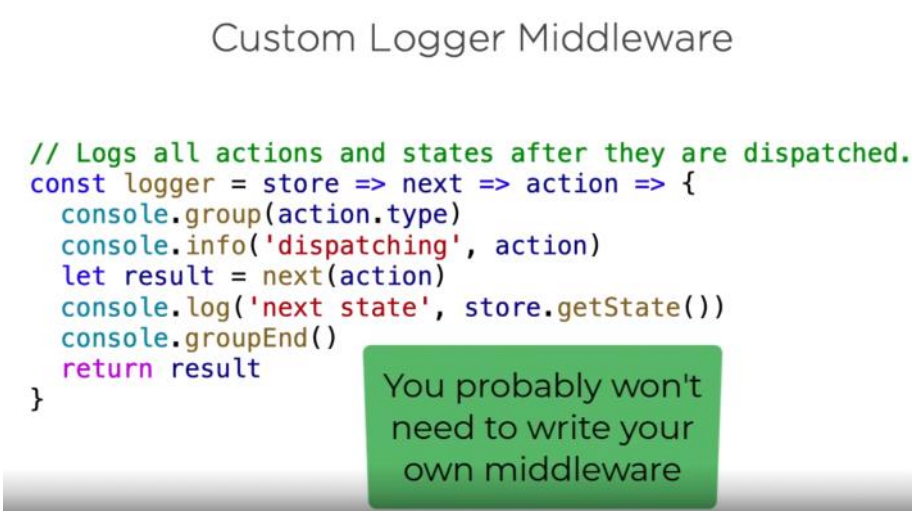
- 

- 

- **Redux Middleware**
  - Redux middleware runs in between dispatching an action and the moment that it reaches the reducer. If you've ever used popular libraries like Express, then you're already familiar with the idea of middleware. Middleware is a handy way to enhance Redux's behavior.
  - Redux middleware can handle many cross- cutting concerns, including handling asynchronous API calls, logging, reporting crashes, and routing.







- Above is an example of middleware that logs all actions and states after they are displatched

- **Async in Redux**

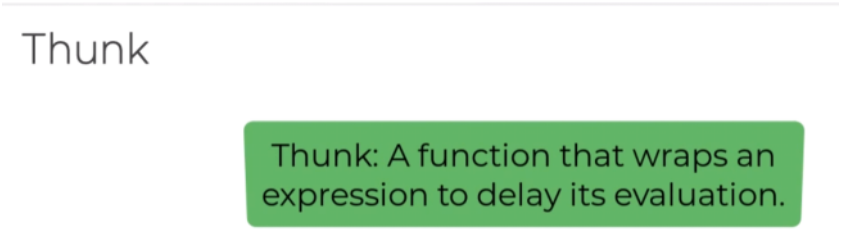
- **Redux-thunk** is quite popular and was written by Dan Abramov who's also the creator of Redux. It allows you to return functions from your action creators instead of objects.
- **Redux-promise** allows you to use promises for async, and it uses Flux standard actions to bring some clear conventions to async calls.
- **Redux-observable** allows you to dispatch observables. So if you're already familiar with RxJS, then you may be interested in this approach.
- **Redux-saga** takes very different approach. It uses ES6 generators and offers an impressive amount of power with what's basically a rich domain-specific language for dealing with asynchrony.

With redux-thunk, you actions can return functions instead of objects. A thunk wraps an asynchronous operation in a function.	With sagas, you handle async operations via generators instead. Generators are functions that can be paused and resumed later. A generator can contain multiple yield statements. At each yield, the generator will pause. They're a powerful tool.
Thunks are a bit clunky to test because you have to mock API calls, and you have no easy hooks for observing and testing individual steps	Sagas are easier to test because you can assert on effects, which simply return data. You don't have to mock anything, and your tests

in the async flow.	generally read more clearly.
The benefit of thunks is they're conceptually simple. And much like Redux, the API surface area is very small. This makes learning thunks quite easy.	Sagas, on the other hand, are hard to learn because you have to understand generators and a large API. Once you do, there are many ways to introduce subtle bugs in your code if you don't fully understand the implications and the interactions of the effects that you're composing. That said, once you know sagas and see the elegance that generators afford, you may prefer them over thunks.

- **Thunks**
  - Normally, we can only return actions from our action creators.
  - With redux-thunk though, we can return a function instead.
  - Here's an example of a thunk for deleting an author.

```
export function deleteAuthor(authorId) {
  return (dispatch, getState) => {
    return AuthorApi.deleteAuthor(authorId)
    .then(() => {
      dispatch(deletedAuthor(authorId));
    })
    .catch(handleError);
  };
}
```



  - A thunk is a function that returns a function.
  - Thunk is a computer science term. A thunk is a function that wraps an expression in order to delay its evaluation.
  - So in the above case, the deleteAuthor function is wrapping our dispatch function so that dispatch can run later.
  - `return AuthorApi.deleteAuthor(authorId)`
  - In the above line we are calling a regular action creator called deleteAuthor.
  - But note that you don't have to call a separate action creator function. We could also inline the action within the thunk if preferred.
  - Thunks enable us to avoid directly causing side effects in our actions, action creators, or components.
  - Instead, anything impure is wrapped in a thunk. Later, that thunk will be invoked by middleware to actually cause the effect.
  - By transferring our side effects to running at a single point in the Redux loop, in this case at the middleware level, the rest of our app stays relatively pure.
  - Redux-thunk has access to the store, so it can pass in the store's dispatch and get state when invoking the thunk. The middleware itself is responsible for injecting those dependencies into the thunk. Thunks are passed dispatch automatically.

```
export function deleteAuthor(authorId) {
  return (dispatch, getState) => {
    return AuthorApi.deleteAuthor(authorId)
    .then(() => {
      dispatch(deletedAuthor(authorId));
    })
    .catch(handleError);
  };
}
```

  - They also receive getState as a second argument. The getState argument on thunks is useful for checking cache data before you make a request or for checking whether you're authenticated, in other words doing a conditional dispatch.

- reduxjs / redux-thunk

Watch 164 | Star 10,919 | Fork 558

Code | Issues 1 | Pull requests 2 | Insights

Branch: master ▾ | redux-thunk / src / index.js | Find file | Copy path

gaearon Add withExtraArgument() 41aefcc on May 10, 2016

2 contributors

15 lines (11 sloc) 352 Bytes | Raw | Blame | History

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

- Middleware isn't *required* to support async in Redux. But you probably want it.
-
```
export function deleteAuthor(dispatch, authorId) {
  return AuthorApi.deleteAuthor(authorId)
    .then(() => {
      dispatch(deletedAuthor(authorId))
    })
    .catch(handleError);
}
```

Thunks pass dispatch in for me so I don't have to.

Thunk

Thunk

The big win with thunks: Your components can call sync an async actions the same way.

- Why Use Async Middleware?

Consistency

Purity

Easier testing

- Import thunk using redux-thunk library in configureStore.js and use it in applyMiddleware

- configureStore.js

```
import { createStore, applyMiddleware, compose } from "redux";
import rootReducer from "./reducers";
import reduxImmutableStateInvariant from "redux-immutable-state-invariant";
import thunk from "redux-thunk";

export default function configureStore(initialState) {
  const composeEnhancers =
    window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose; // add support for Redux dev tools

  return createStore(
    rootReducer,
    initialState,
    composeEnhancers(applyMiddleware(thunk, reduxImmutableStateInvariant()))
  );
}
```

```
applyMiddleware(middleware1: Middleware<any, any, any>,
middleware2: Middleware<{}, any, any>): StoreEnhancer<{
dispatch: any; }, {}>
```

3/7

- For our first thunk, let's load courses when the app initially loads.

- courseActions.js

```
import * as types from "./actionTypes";
import * as courseApi from "../../api/courseApi";

export function createCourse(course) {
  return { type: types.CREATE_COURSE, course };
}

export function loadCourses() {
  return function (dispatch) {

  }
}
```

Redux thunk injects dispatch so we don't have to.

- courseActions.js | actionTypes.js

```
import * as types from "./actionTypes";
import * as courseApi from "../../api/courseApi";

export function createCourse(course) {
  return { type: types.CREATE_COURSE, course };
}

export function loadCourseSuccess(courses) {
  return { type: types.LOAD_COURSES_SUCCESS, courses };
}

export function loadCourses() {
  return function(dispatch) {
    return courseApi
      .getCourses()
      .then(courses => {
        dispatch(loadCourseSuccess(courses));
      })
      .catch(error => {
        throw error;
      });
  };
}
```

- Options for when to load courses:
  1. When app loads
  2. When course page is loaded.

- Options for when to load courses:
  1. When app loads
  2. When course page is loaded.
-

```
JS CoursesPage.js ×
1   import React from "react";
2   import { connect } from "react-redux";
3   import * as courseActions from "../../redux/actions/courseActions";
4   import PropTypes from "prop-types";
5   import { bindActionCreators } from "redux";
6
7   class CoursesPage extends React.Component {
8     componentDidMount() {
9       this.props.actions.loadCourses().catch(error => {
10        alert("Loading courses failed" + error);
11      });
12    }
13
14    render() {
15      return (
16        <>
17          <h2>Courses</h2>
```

-

```
JS ManageCoursePage.js ×   JS CoursesPage.js
1   import React from "react";
2   import { connect } from "react-redux";
3   import * as courseActions from "../../redux/actions/courseActions";
4   import * as authorActions from "../../redux/actions/authorActions";
5   import PropTypes from "prop-types";
6   import { bindActionCreators } from "redux";
7
```

Section 1: Imports

-

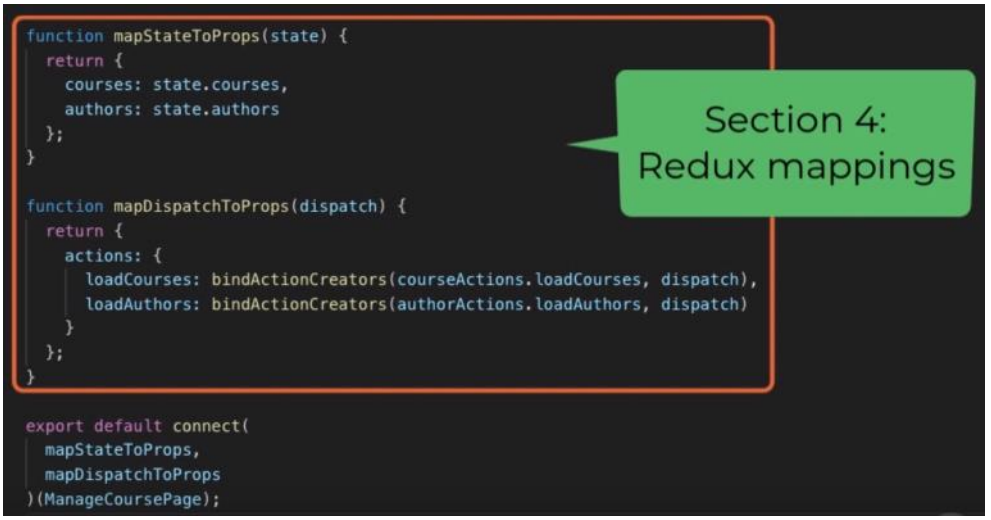
```
class ManageCoursePage extends React.Component {
  componentDidMount() {
    const { courses, authors, actions } = this.props;

    if (courses.length === 0) {
      actions.loadCourses().catch(error => {
        alert("Loading courses failed" + error);
      });
    }

    if (authors.length === 0) {
      actions.loadAuthors().catch(error => {
        alert("Loading authors failed" + error);
      });
    }
  }

  render() {
    return (
      <>
        <h2>Manage Course</h2>
      </>
```

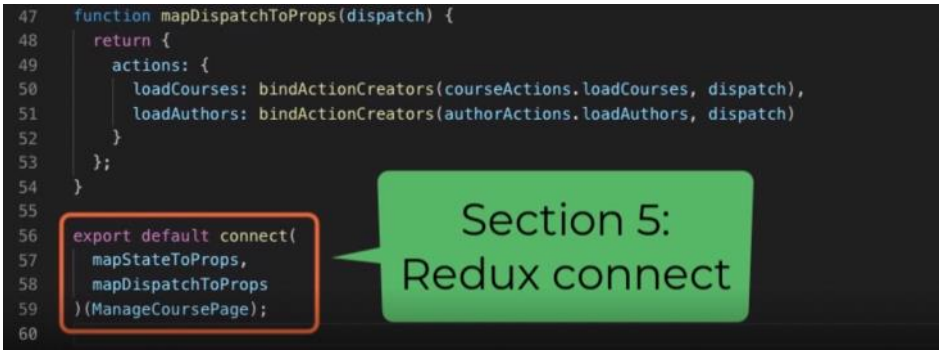
Section 2: Component

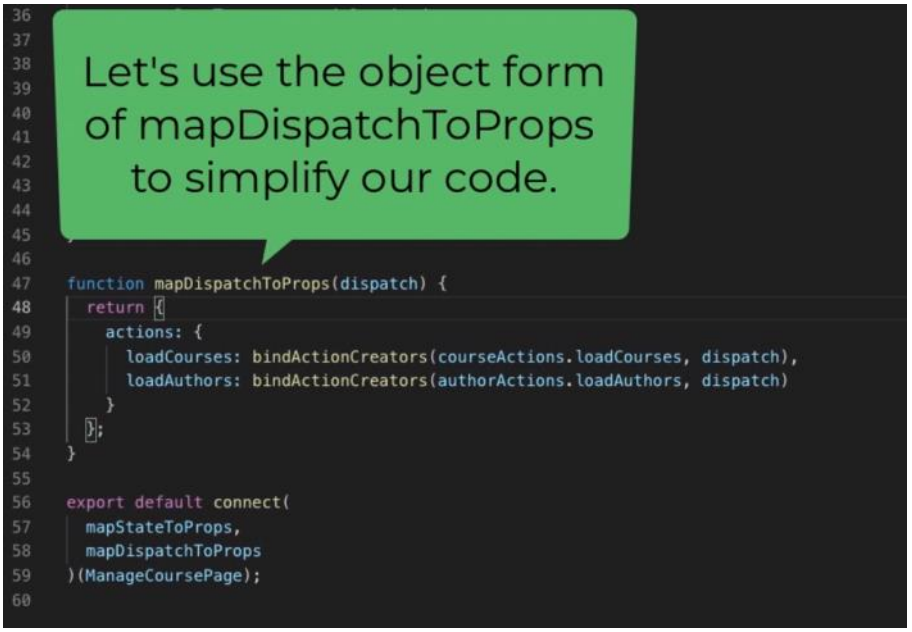
-

```
JS ManageCoursePage.js ×   JS CoursesPage.js
33
34  ManageCoursePage.propTypes = {
35    authors: PropTypes.array.isRequired,
36    courses: PropTypes.array.isRequired,
37    actions: PropTypes.object.isRequired
38  };
39
```

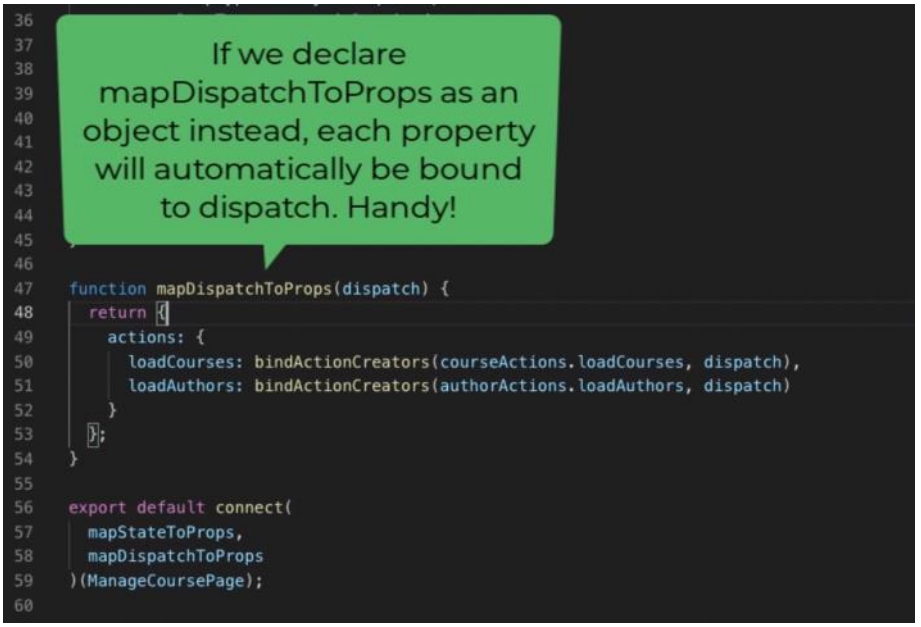
Section 3: PropTypes

- 

- 

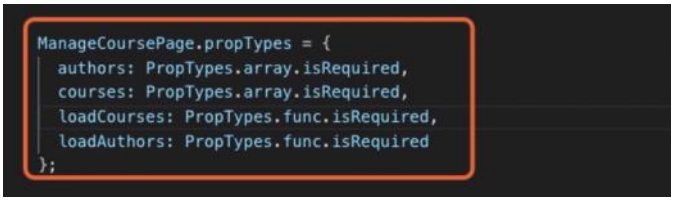
- 

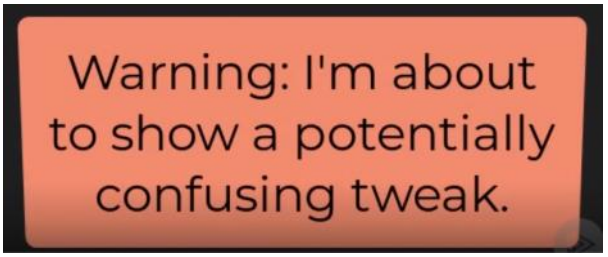
- We also have an alternate way to handle mapDispatchToProps, that is via an object form

- 

- 

- We need to tweak a bit to load courses and load authors as below
- We can also remove bindActionCreators from the imports
- 

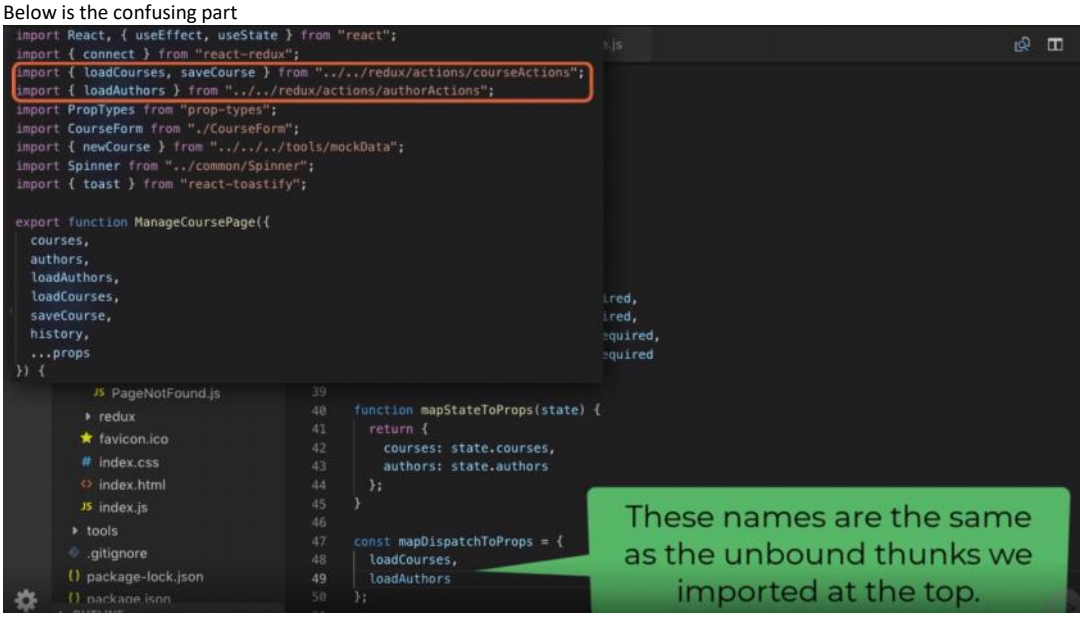
- 

- 

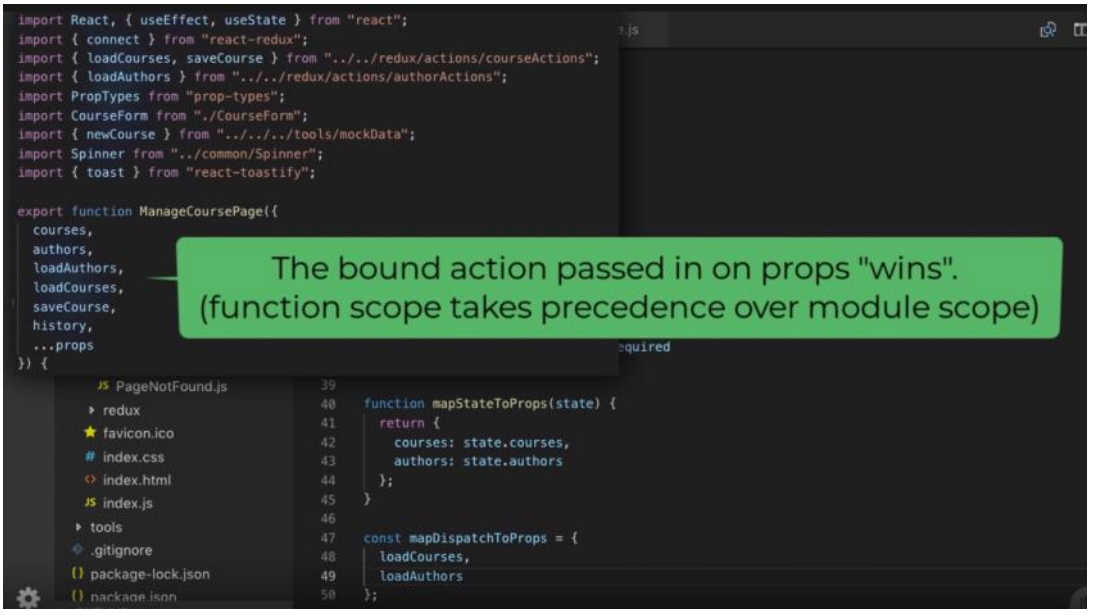
- We can make mapDispatchToProps more shorter

- ManageCoursePage.js | CoursesPage.js

```
1  import React from "react";
2  import { connect } from "react-redux";
3  import { loadCourses } from "../../redux/actions/courseActions";
4  import { loadAuthors } from "../../redux/actions/authorActions";
5  import PropTypes from "prop-types";
6
```

- Below is the confusing part
- 

- 

- 


```
import React, { useEffect, useState } from "react";
import { connect } from "react-redux";
import { loadCourses, saveCourse } from "../../redux/actions/courseActions";
import { loadAuthors } from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import CourseForm from "./CourseForm";
import { newCourse } from "../../../tools/mockData";
import Spinner from "../common/Spinner";
import { toast } from "react-toastify";

export function ManageCoursePage({
  courses,
  authors,
  loadAuthors,
  loadCourses,
  saveCourse,
  history,
  ...props
}) {
```

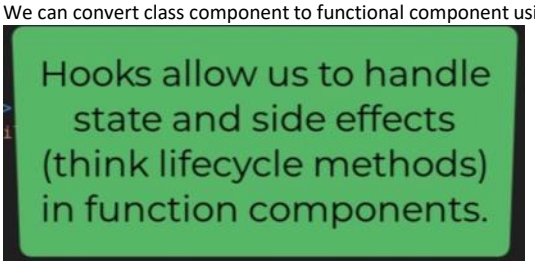
```
function mapStateToProps(state) {
  return {
    courses: state.courses,
    authors: state.authors
  };
}

const mapDispatchToProps = {
  loadCourses,
  loadAuthors
};
```

- App.js

```
1  import React from "react";
2  import { Route, Switch } from "react-router-dom";
3  import HomePage from "./home/HomePage";
4  import AboutPage from "./about/AboutPage";
5  import Header from "./common/Header";
6  import PageNotFound from "
```

- 

  - Remember, our routes are wrapped in switch, so only one route inside switch can match. The moment the React Router finds a matching route, it stops searching. So if it finds a slug in the URL, it will stop here. This is why we need to make sure that the slug route is declared first. Otherwise, the shorter course URL would match, so our slug route would never load.

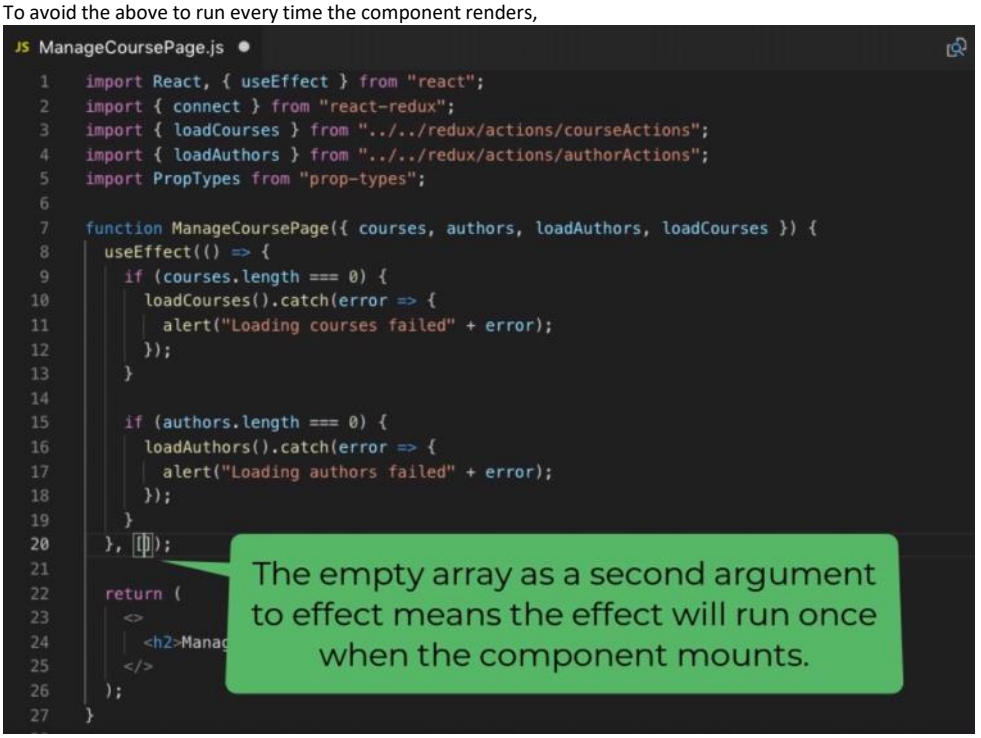
- **Converting class component to functional component**
  - We can convert class component to functional component using **Hooks**
  - 

  - Hooks only work with function components
  - We're going to use **useEffect** hook to replace our usage of componentDidMount because useEffect lets us handle side effects.
  - UseEffect accepts a function that it will call.
  - We can remove render function also as it is implicit that function call will return

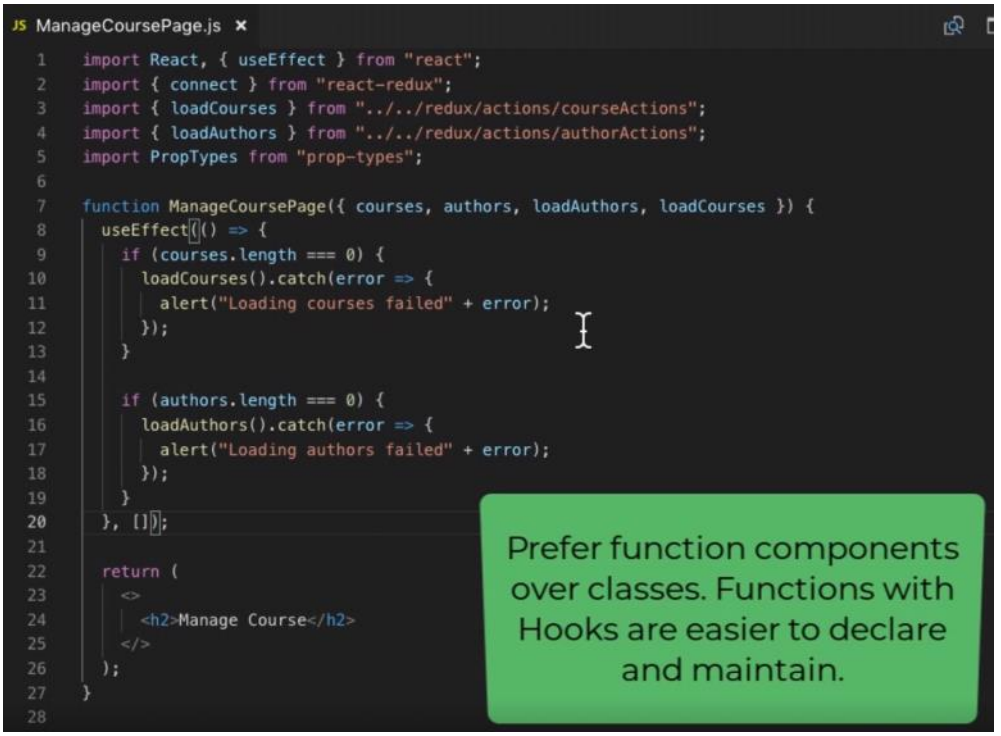
- JS ManageCoursePage.js

```
import React, { useEffect } from "react";
import { connect } from "react-redux";
import { loadCourses } from "../../redux/actions/courseActions";
import { loadAuthors } from "../../redux/actions/authorActions";
import PropTypes from "prop-types";

function ManageCoursePage({ courses, authors, loadAuthors, loadCourses }) {
  useEffect(() => {
    if (courses.length === 0) {
      loadCourses().catch(error => {
        alert("Loading courses failed" + error);
      });
    }

    if (authors.length === 0) {
      loadAuthors().catch(error => {
        alert("Loading authors failed" + error);
      });
    }
  });

  return (
    <>
      <h2>Manage Course</h2>
    </>
  );
}
```

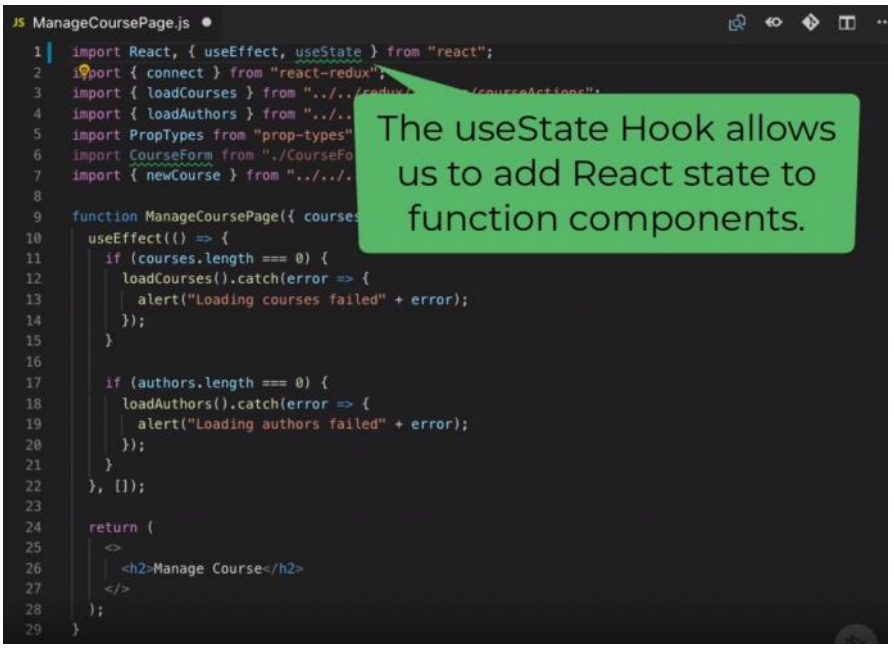
- 

- To avoid the above to run every time the component renders,
- 

- We want this to only run once when the component first mounts. So to do that, we can declare a second argument to useEffect.
- We declare that argument as an array of items for it to watch. And if anything in that array changes, it will rerun this effect. Since we only want this to run once, we can declare an empty array here, and this is effectively the same as componentDidMount.

- 

- We can useState hook for state management
- UseState is a hook that lets us add React state to function components.

- 


- ```
  function ManageCoursePage({ courses, authors, loadAuthors, loadCourses, ...props }) {
    const [ course, setCourse] = useState({...props.course});
  ```
  - We are going to declare course as our state and the setter as setCourse. We will call useState. And for the default value, we'd like to set it to the *course* that's getting passed in on props.
  - UseState returns a pair of values. We use array destructuring syntax to assign each value a name.
    - The first value is the state variable, and
    - the second value is the setter function for that variable.
  - UseState accepts a default argument. We're specifying that it should initialize our core state variable to a copy of the course passed in on props.

  - Rest Operator
  ```

- ManageCoursePage.js

```
import React, { useEffect, useState } from "
import { connect } from "react-redux";
import { loadCourses } from "../../redux/act
import { loadAuthors } from "../../redux/act
import PropTypes from "prop-types";
import CourseForm from "./CourseForm";
import { newCourse } from "../../../tools/mockData";

function ManageCoursePage({ courses, authors, loadAuthors, loadCourses, ...props }) {
  const [ course, setCourse] = useState({...props.course});
```

So this says, "Assign any props I haven't destructured on the left to a variable called props."

- ManageCoursePage.js

```
import React, { useEffect, useState } from "react";
import { connect } from "react-redux";
import { loadCourses } from "../../redux/actions/courseActions";
import { loadAuthors } from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import CourseForm from "./CourseForm";
import { newCourse } from "../../../tools/mo

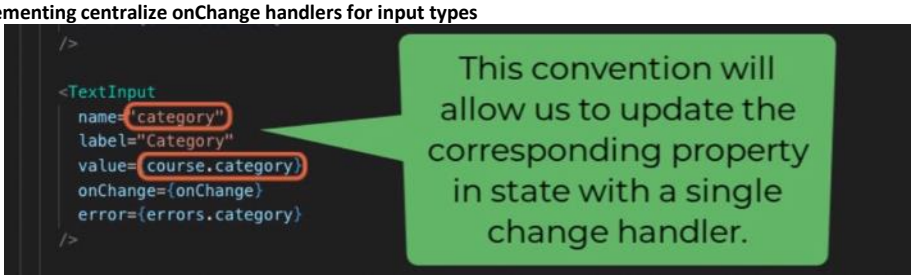
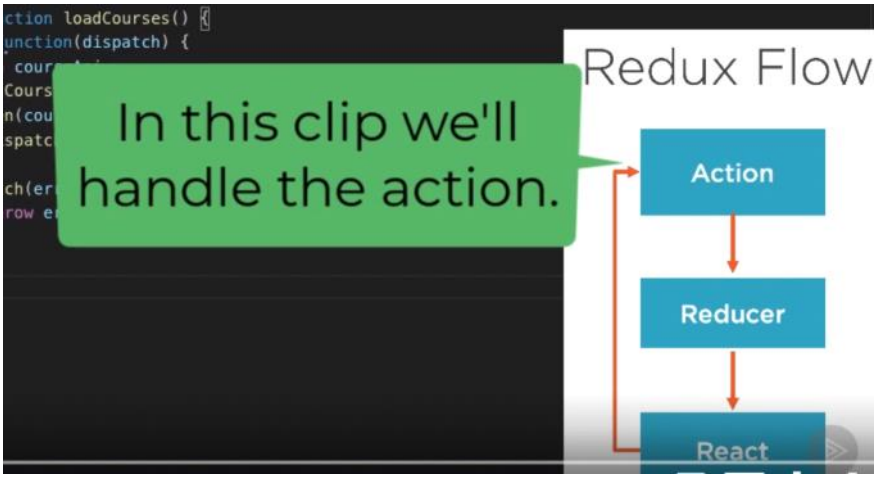
function ManageCoursePage({
  courses,
  authors,
  loadAuthors,
  loadCourses,
  ...props
}) {
  const [course, setCourse] = useState({ ...props.course });

  useEffect(() => {
    if (courses.length === 0) {
      loadCourses().catch(error => {
        alert("Loading courses failed" + error);
      });
    }

    if (authors.length === 0) {
      loadAuthors().catch(error => {
        alert("Loading authors failed" + error);
      });
    }
```

Avoid using Redux for all state. Use plain React state for data only one few components use. (such as form state)

- ManageCoursePage.js

```
import React, { useEffect, useState } from "react";
import { connect } from "react-redux";
import { loadCourses } from "../../redux/actions/courseActions";
import { loadAuthors } from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import CourseForm from "./CourseForm";
import { newCourse } from "../../../tools/mo

function ManageCoursePage({
  courses,
  authors,
  loadAuthors,
  loadCourses,
  ...props
}) {
  const [course, setCourse] = useState({ ...props.course });

  useEffect(() => {
    if (courses.length === 0) {
      loadCourses().catch(error => {
        alert("Loading courses failed" + error);
      });
    }

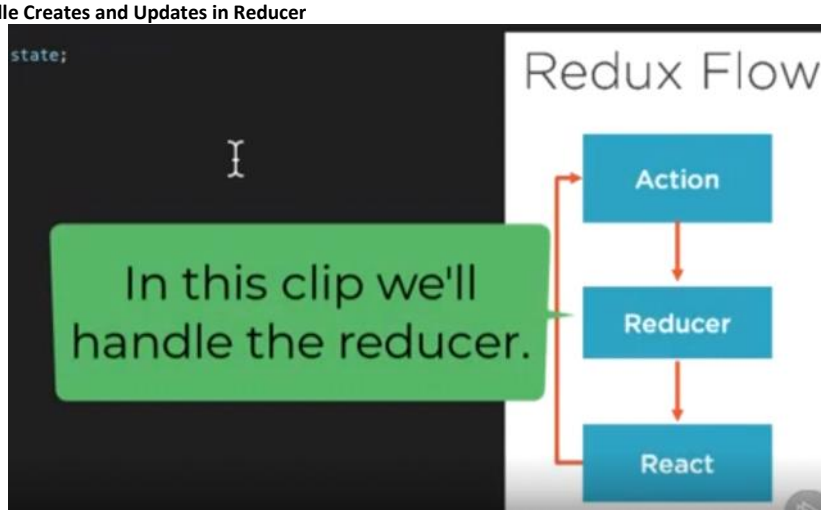
    if (authors.length === 0) {
      loadAuthors().catch(error => {
        alert("Loading authors failed" + error);
      });
    }
```

To choose Redux vs local state, ask: "Who cares about this data?" If only a few closely related components use the data, prefer plain React state.

- **Implementing centralize onChange handlers for input types**
  - 

  - 

  - We can use object form or functional form setState. In the above example we are using the functional form of setState.
  - 

  - 

- Add Save Course Thunk

- 


- ```
  export function saveCourse(course) {
    //eslint-disable-next-line no-unused-vars
    return function(dispatch, getState) {
      return courseApi
        .saveCourse(course)
        .then(savedCourse => {
          course.id
            ? dispatch(updateCourseSuccess(savedCourse))
            : dispatch(createCourseSuccess(savedCourse));
        })
        .catch(error => {
          throw error;
        });
    };
  }
  ```

- ```
        .then(courses => {
          dispatch(loadCourseSuccess(courses));
        })
        .catch(error => {
          throw error;
        });
    };
  }

  export function saveCourse(course) {
    //eslint-disable-next-line no-unused-vars
    return function(dispatch, getState) {
      return courseApi
        .saveCourse(course)
        .then(savedCourse => {
          course.id
            ? dispatch(updateCourseSuccess(savedCourse))
            : dispatch(createCourseSuccess(savedCourse));
        })
        .catch(error => {
          throw error;
        });
    };
  }
  ```

getState lets us access the Redux store data.

- ```
  export function createCourseSuccess(course) {
    return { type: types.CREATE_COURSE_SUCCESS, course };
  }

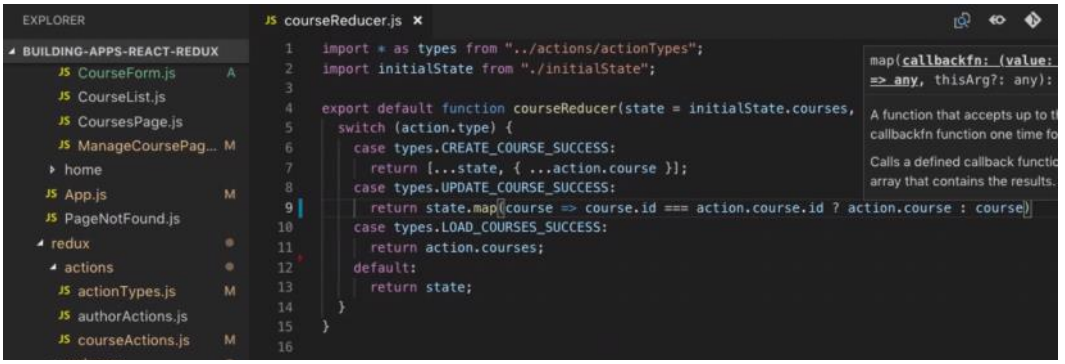
  export function updateCourseSuccess(course) {
    return { type: types.UPDATE_COURSE_SUCCESS, course };
  }
  ```

- courseActions.js | actionTypes.js

```
1  export const CREATE_COURSE = "CREATE_COURSE";
2  export const LOAD_COURSES_SUCCESS = "LOAD_COURSES_SUCCESS";
3  export const LOAD_AUTHORS_SUCCESS = "LOAD_AUTHORS_SUCCESS";
```
```

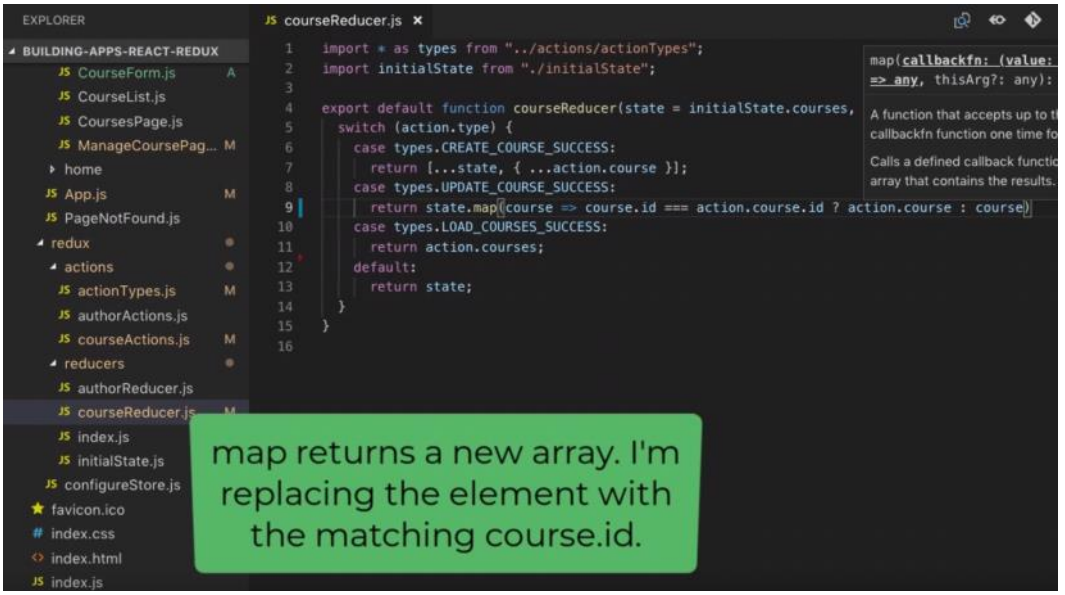
-

```
courseActions.js   actionTypes.js ×
1  export const CREATE_COURSE = "CREATE_COURSE";
2  export const LOAD_COURSES_SUCCESS = "LOAD_COURSES_SUCCESS";
3  export const LOAD_AUTHORS_SUCCESS = "LOAD_AUTHORS_SUCCESS";
4  export const CREATE_COURSE_SUCCESS = "CREATE_COURSE_SUCCESS";
5  export const UPDATE_COURSE_SUCCESS = "UPDATE_COURSE_SUCCESS";
6
```

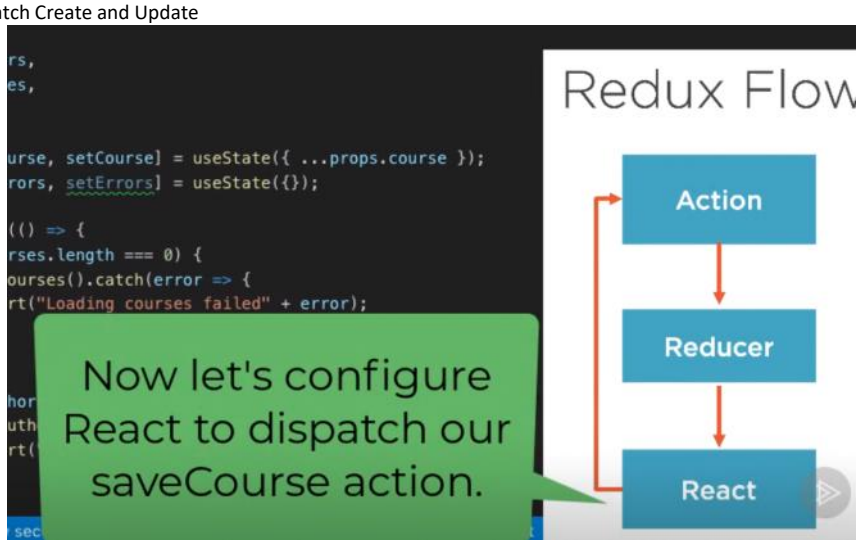
- **Handle Creates and Updates in Reducer**
- 

- 

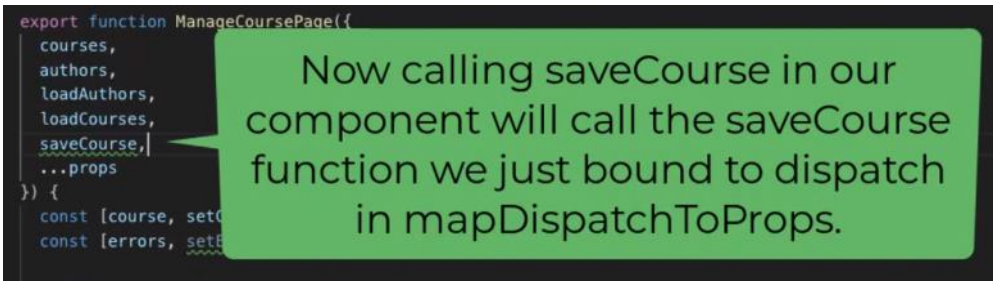
- 

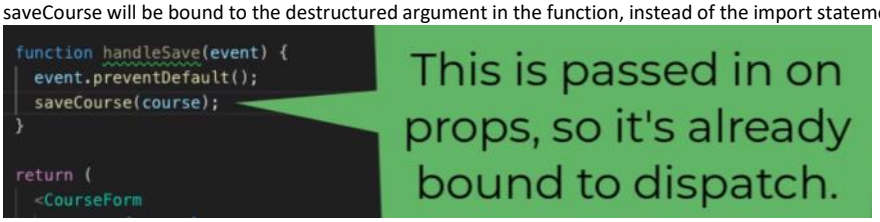
- 

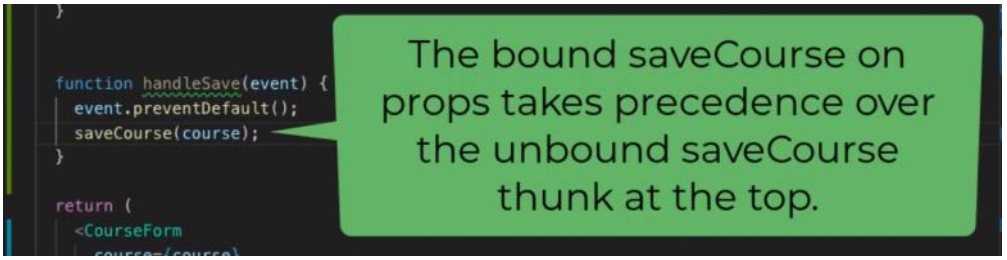
- 

- Dispatch Create and Update
  - 

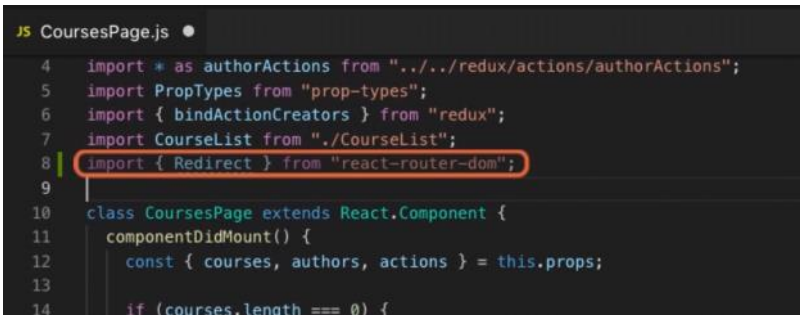
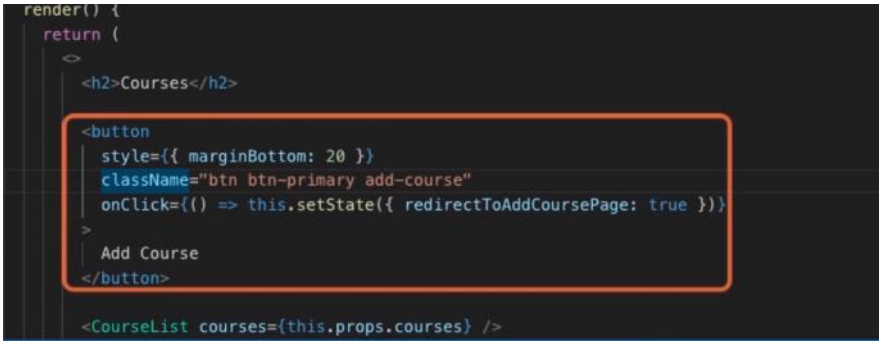
  - 

-
```
const mapDispatchToProps = {
  loadCourses,
  loadAuthors,
  saveCourse
};
```
-
```
ManageCoursePage.propTypes = {
  course: PropTypes.object.isRequired,
  authors: PropTypes.array.isRequired,
  courses: PropTypes.array.isRequired,
  loadCourses: PropTypes.func.isRequired,
  loadAuthors: PropTypes.func.isRequired,
  saveCourse: PropTypes.func.isRequired
};
```
- 

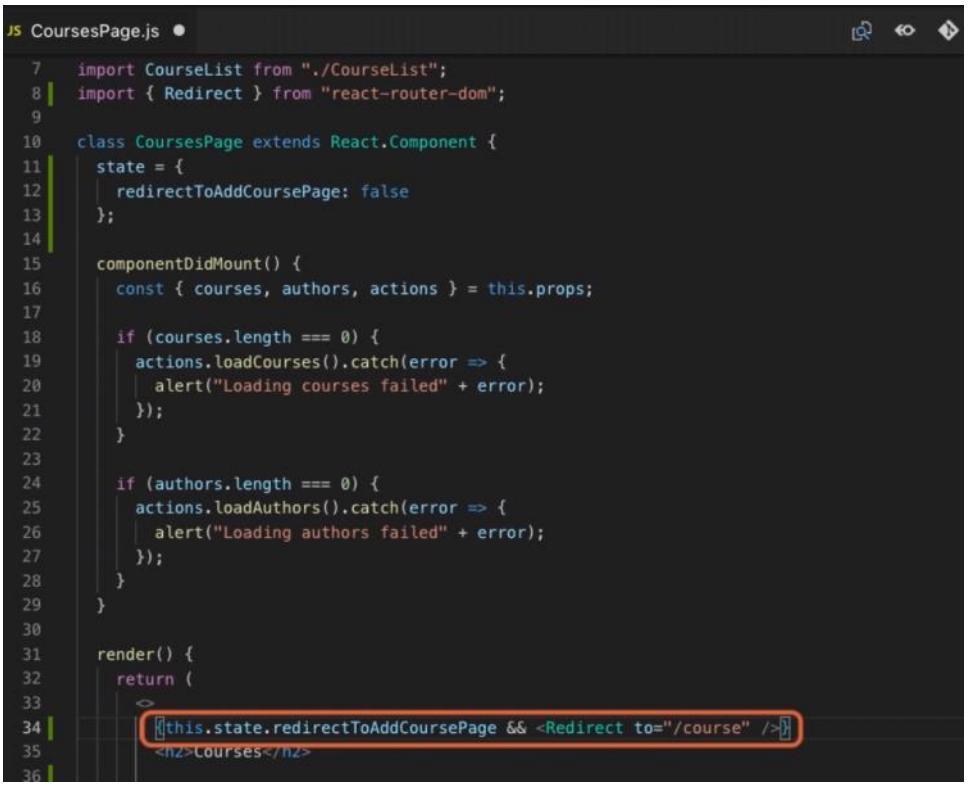
- saveCourse will be bound to the destructured argument in the function, instead of the import statement saveCourse.
- 

- 

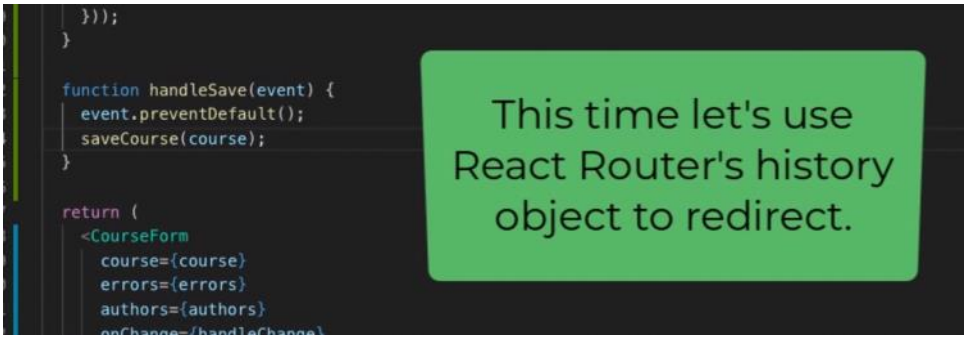
- **Redirect via React Router's Redirect Component**

- 
CoursesPage.js
import * as authorActions from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import { bindActionCreators } from "redux";
import CourseList from "./CourseList";
class CoursesPage extends React.Component {
componentDidMount() {
const { courses, authors, actions } = this.props;
if (courses.length === 0) {
actions.loadCourses().catch(error => {
alert("Loading courses failed" + error);
});
}
if (authors.length === 0) {
actions.loadAuthors().catch(error => {
alert("Loading authors failed" + error);
});
}
}
render() {
return (
<>
<h2>Courses</h2>
<CourseList courses={this.props.courses} />
</>
I'll show two different ways to handle redirects.
- 
CoursesPage.js
import * as authorActions from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import { bindActionCreators } from "redux";
import CourseList from "./CourseList";
import { Redirect } from "react-router-dom";
class CoursesPage extends React.Component {
componentDidMount() {
const { courses, authors, actions } = this.props;
- 
render() {
return (
<>
<h2>Courses</h2>
<button
style={{ marginBottom: 20 }}
className="btn btn-primary add-course"
onClick={() => this.setState({ redirectToAddCoursePage: true })}
>
Add Course
</button>
<CourseList courses={this.props.courses} />

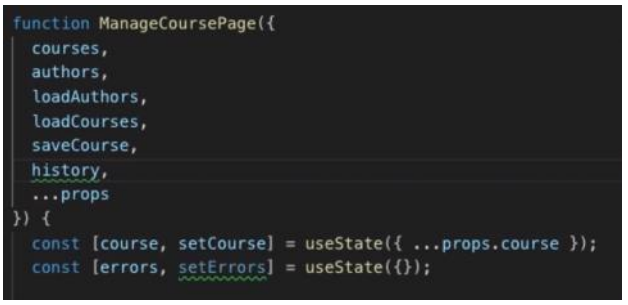
- 

- 

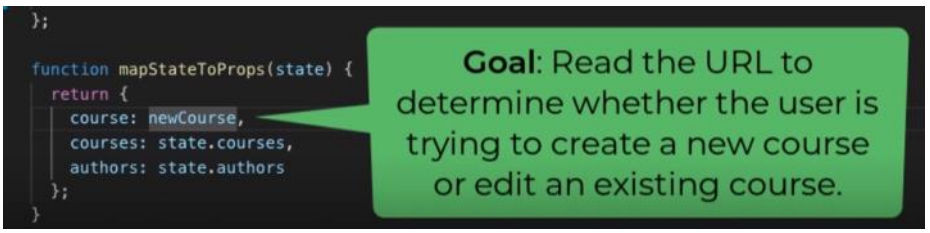
- Redirect via React Router's History

- 

- 

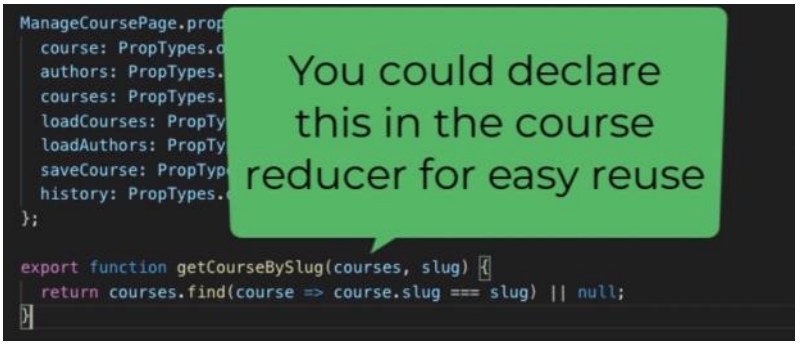
- 
- 

- Populate Form via mapStateToProps
  - To populate a form with existing course data when editing it
  - 

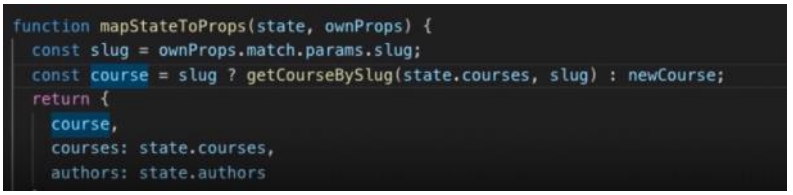
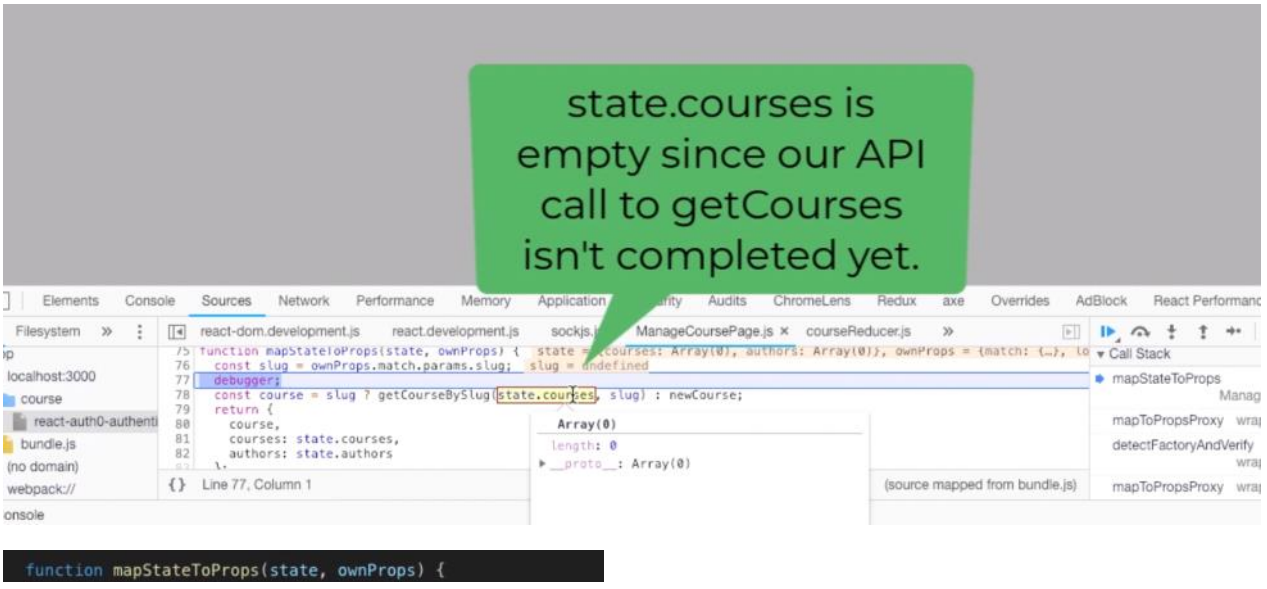
  - 

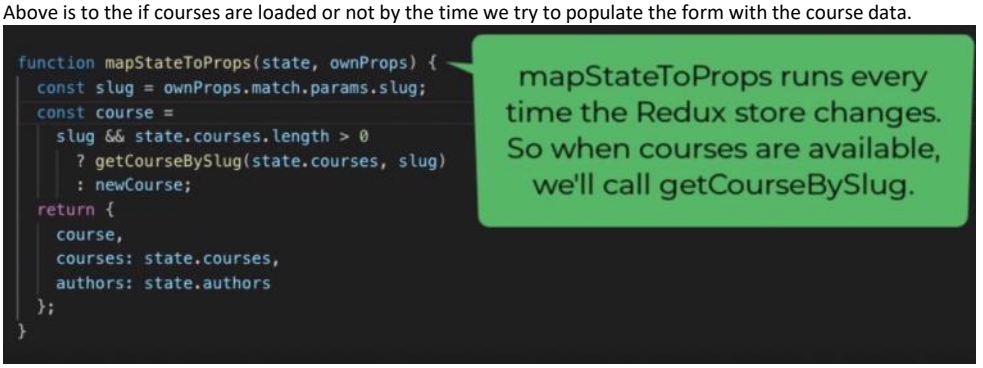
-
```
ManageCoursePage.propTypes = {
  course: PropTypes.object.isRequired,
  authors: PropTypes.array.isRequired,
  courses: PropTypes.array.isRequired,
  loadCourses: PropTypes.func.isRequired,
  loadAuthors: PropTypes.func.isRequired,
  saveCourse: PropTypes.func.isRequired,
  history: PropTypes.object.isRequired
};

function mapStateToProps(state, ownProps) {
  return {
    course: newCourse,
    courses: state.courses,
    authors: state.authors
  };
```
This lets us access the component's props. We can use this to read the URL data injected on props by React Router.
  - ownProps is automatically populated by Redux.
  - As the route is configuresd with slug as param as shown below
    -
```
import React from "react";
import { Route, Switch } from "react-router-dom";
import HomePage from "./home/HomePage";
import AboutPage from "./about/AboutPage";
import Header from "./common/Header";
import PageNotFound from "./PageNotFound";
import CoursesPage from "./courses/CoursesPage";
import ManageCoursePage from "./courses/ManageCoursePage";

function App() {
  return (
    <div className="container-fluid">
      <Header />
      <Switch>
        <Route exact path="/" component={HomePage} />
        <Route path="/about" component={AboutPage} />
        <Route path="/courses" component={CoursesPage} />
        <Route path="/course" component={ManageCoursePage} />
        <Route path="/course/:slug" component={ManageCoursePage} />
        <Route component={PageNotFound} />
```
Access in mapStateToProps via ownProps.params.match.slug.
  - Using the above we can use ownProps to the slug value
    -
```
function mapStateToProps(state, ownProps) {
  const slug = ownProps.match.params.slug;
  return {
```
  -
```
function mapStateToProps(state, own
  const slug = ownProps.match.paran
  return {
    course: newCourse,
    courses: state.courses,
    authors: state.authors
  };
```
Goal: Populate the course object based on the URL.
  -
```
function mapStateToProps(state, ownProps) {
  const slug = ownProps.match.params.slug;
  const course = slug ? getCourseBySlug(state.courses, slug) : newCourse;
```

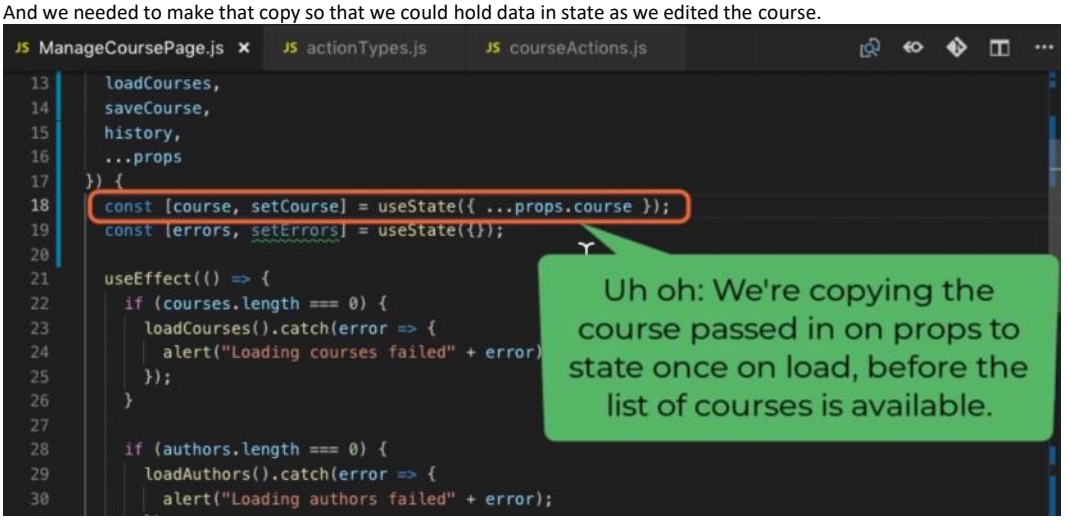
- 

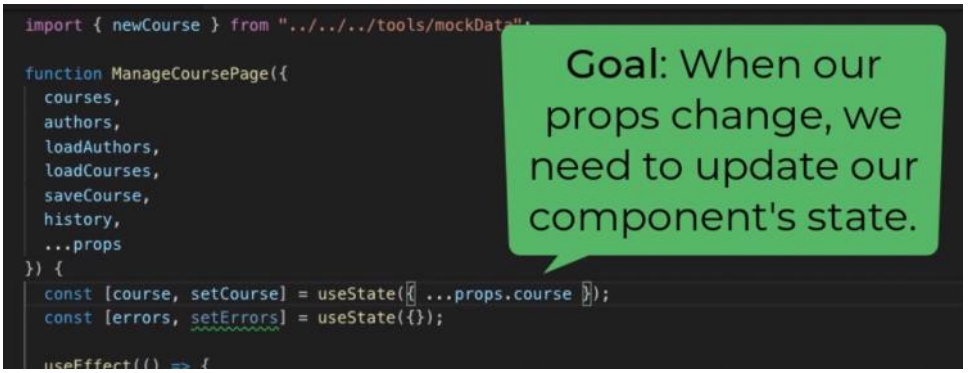
- 

- 

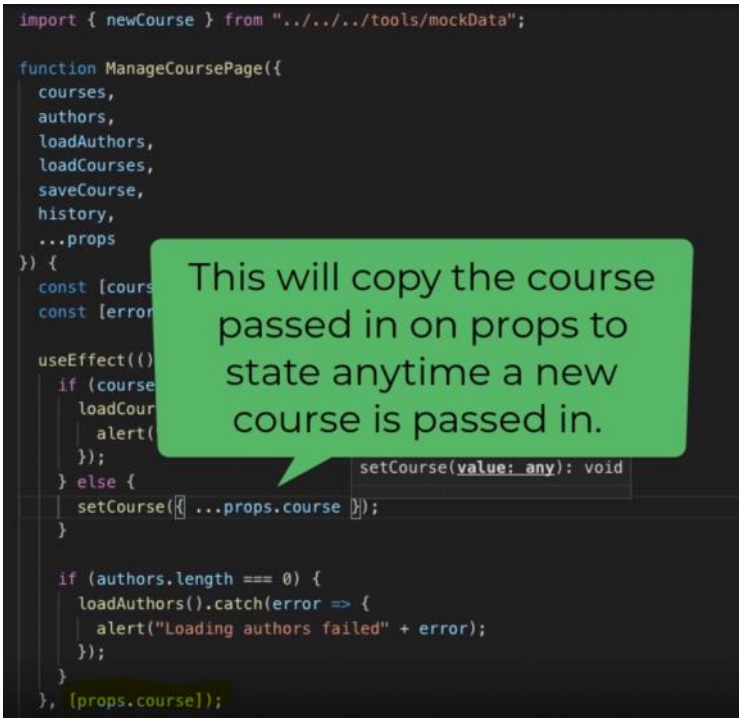
- For more information we need to check out "**Selectors in Redux**"
- 
- 

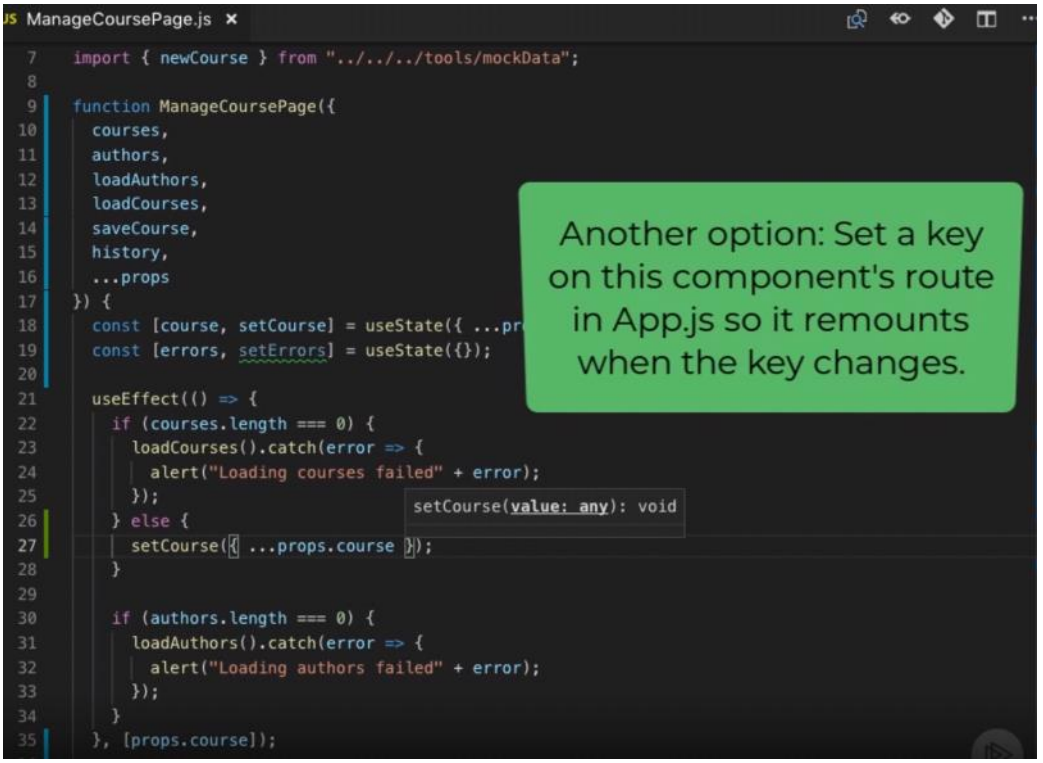
-
```
function mapStateToProps(state, ownProps) {
  const slug = ownProps.match.params.slug;
  debugger;
  const course =
    slug && state.courses.length > 0
      ? getCourseBySlug(state.courses, slug)
      : newCourse;
  return {
    course,
    courses: state.courses,
    authors: state.authors
  };
}
```
- Above is to the if courses are loaded or not by the time we try to populate the form with the course data.
- 

- But when we directly load an existing course it does not load the form with the data
- In ManageCoursePage, this course that we're passing down to CourseForm isdeclared in state up here at the top. So it is a copy of props.course. And we needed to make that copy so that we could hold data in state as we edited the course.
- 

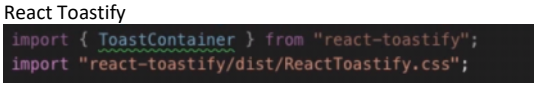
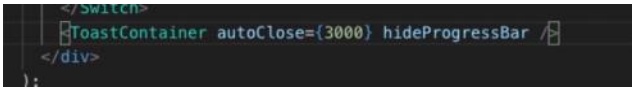
- 

- This can be easily solved by tweaking our useEffect hook. Right now, the useEffect hook only runs once when the component mounts.
- Instead, we want it to run any time that a new course is passed in on props.

- 

- 

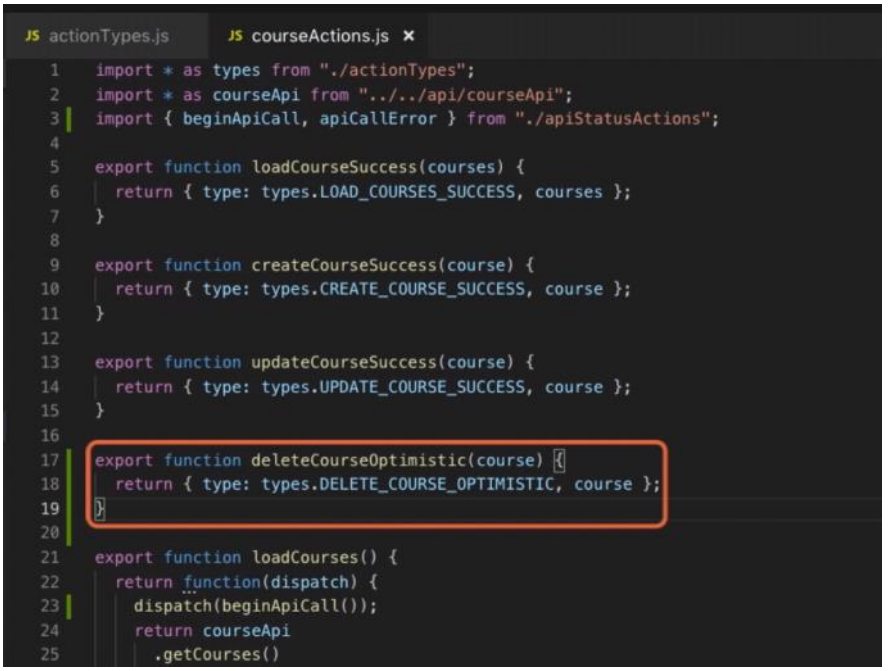
  - But the above alternative needs a lot of code change, so let us stick to the 1st approach.

- Using React Toastify
  - 
```
import { ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";
```
  - 
```
    </Switch>
    <ToastContainer autoClose={3000} hideProgressBar />
  </div>
);
```

-
```
App.js   JS ManageCoursePage.js ●
1  import React, { useEffect, useState } from "react";
2  import { connect } from "react-redux";
3  import { loadCourses, saveCourse } from "../../redux/actions/cour
4  import { loadAuthors } from "../../redux/actions/authorActions";
5  import PropTypes from "prop-types";
6  import CourseForm from "./CourseForm";
7  import { newCourse } from "../../../tools/mockData";
8  import Spinner from "../common/Spinner";
9  import { toast } from "react-toastify";
0
```
-
```
function handleSave(event) {
  event.preventDefault();
  setSaving(true);
  saveCourse(course).then(() => {
    toast.success("Course saved.");
    history.push("/courses");
  });
}
```

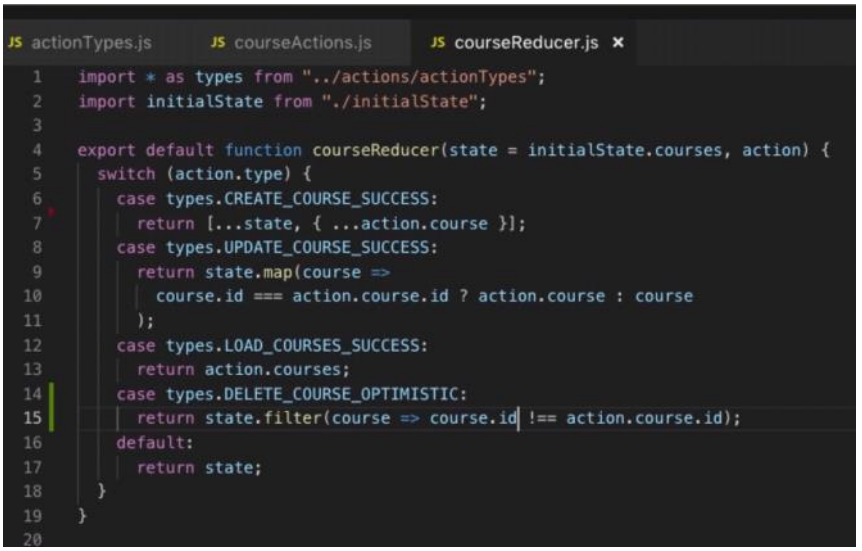
- **Deleting a course optimistically**
-
```
actionTypes.js ×
1   export const CREATE_COURSE = "CREATE_COURSE";
2   export const LOAD_COURSES_SUCCESS = "LOAD_COURSES_SUCCESS";
3   export const LOAD_AUTHORS_SUCCESS = "LOAD_AUTHORS_SUCCESS";
4   export const CREATE_COURSE_SUCCESS = "CREATE_COURSE_SUCCESS";
5   export const UPDATE_COURSE_SUCCESS = "UPDATE_COURSE_SUCCESS";
6   export const BEGIN_API_CALL = "BEGIN_API_CALL";
7   export const API_CALL_ERROR = "API_CALL_ERROR";
8
9   // By convention, actions that end in "_SUCCESS" are assumed to have been the result of a completed
10  // API call. But since we're doing an optimistic delete, we're hiding loading state.
11  // So this action name deliberately omits the "_SUCCESS" suffix.
12  // If it had one, our apiCallsInProgress counter would be decremented below zero
13  // because we're not incrementing the number of apiCallInProgress when the delete request begins.
14  export const DELETE_COURSE_OPTIMISTIC = "DELETE_COURSE_OPTIMISTIC";
15
```
-
```
        ? dispatch(updateCourseSuccess(savedCourse))
        : dispatch(
    })
    .catch(error =>
      dispatch(apiC
      throw error;
    });
};
}

export function deleteCourse(course) {
  return function (dispatch) {
    // Doing optimistic delete, so not dispatching begin/end api call
    // actions, or apiCallError action since we're not showing the loading status for this.
    dispatch(deleteCourseOptimistic(course));
    return courseApi.deleteCourse(course.id);
  };
}
```
Differences:
1. Immediately dispatching deleteCourse
2. Not dispatching beginApiCall

- actionTypes.js | courseActions.js

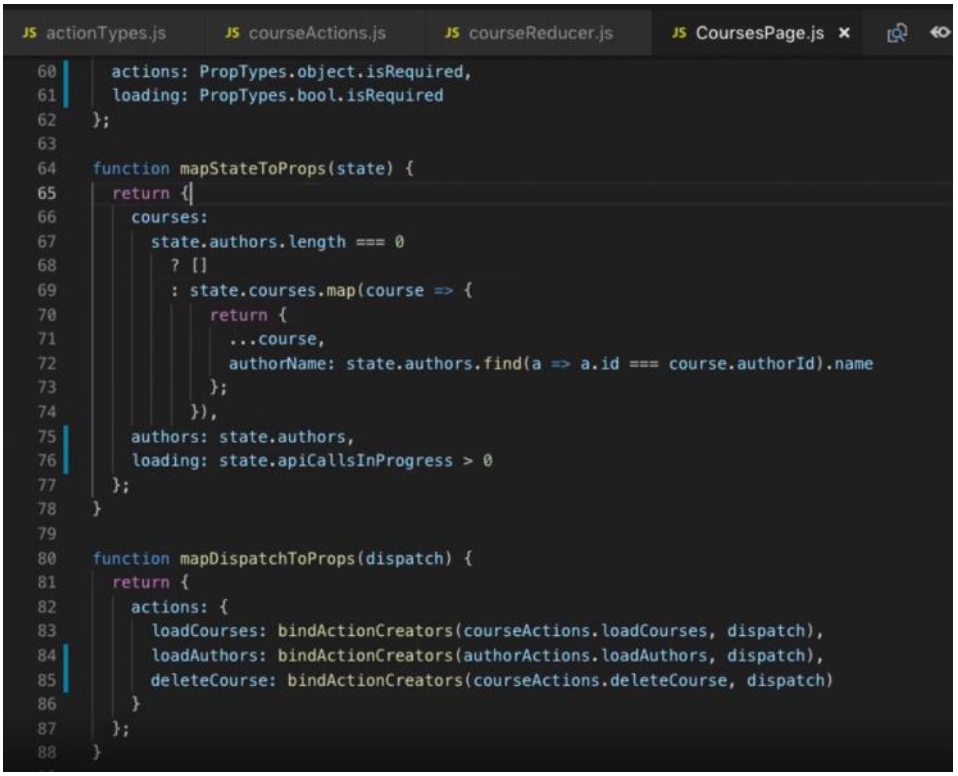
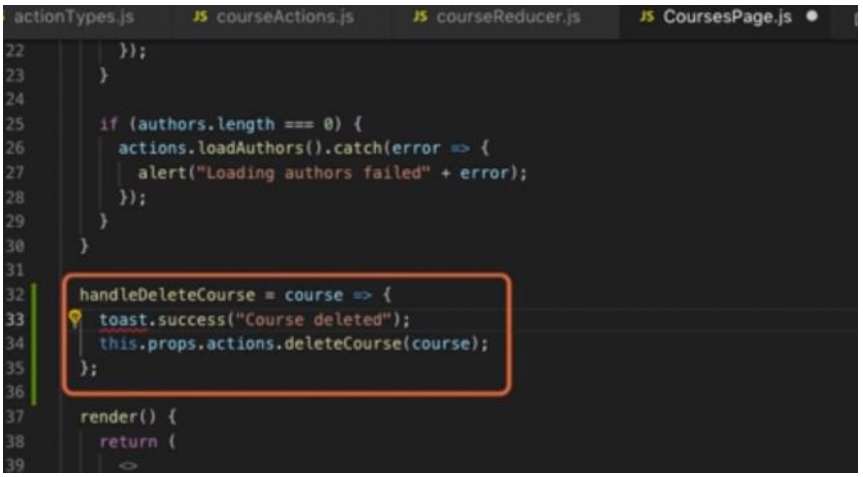
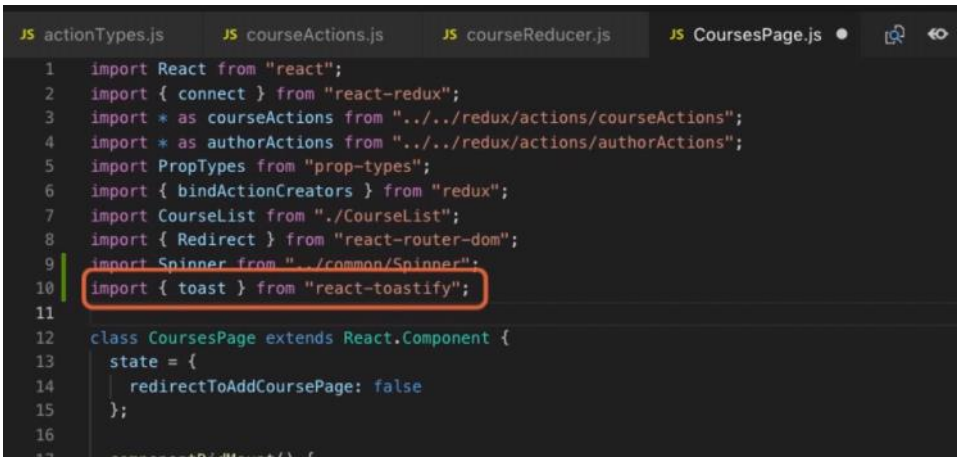


```
import * as types from "./actionTypes";
import * as courseApi from "../../api/courseApi";
import { beginApiCall, apiCallError } from "./apiStatusActions";

export function loadCourseSuccess(courses) {
  return { type: types.LOAD_COURSES_SUCCESS, courses };
}

export function createCourseSuccess(course) {
  return { type: types.CREATE_COURSE_SUCCESS, course };
}

export function updateCourseSuccess(course) {
  return { type: types.UPDATE_COURSE_SUCCESS, course };
}

export function deleteCourseOptimistic(course) {
  return { type: types.DELETE_COURSE_OPTIMISTIC, course };
}

export function loadCourses() {
  return function(dispatch) {
    dispatch(beginApiCall());
    return courseApi
      .getCourses()
```

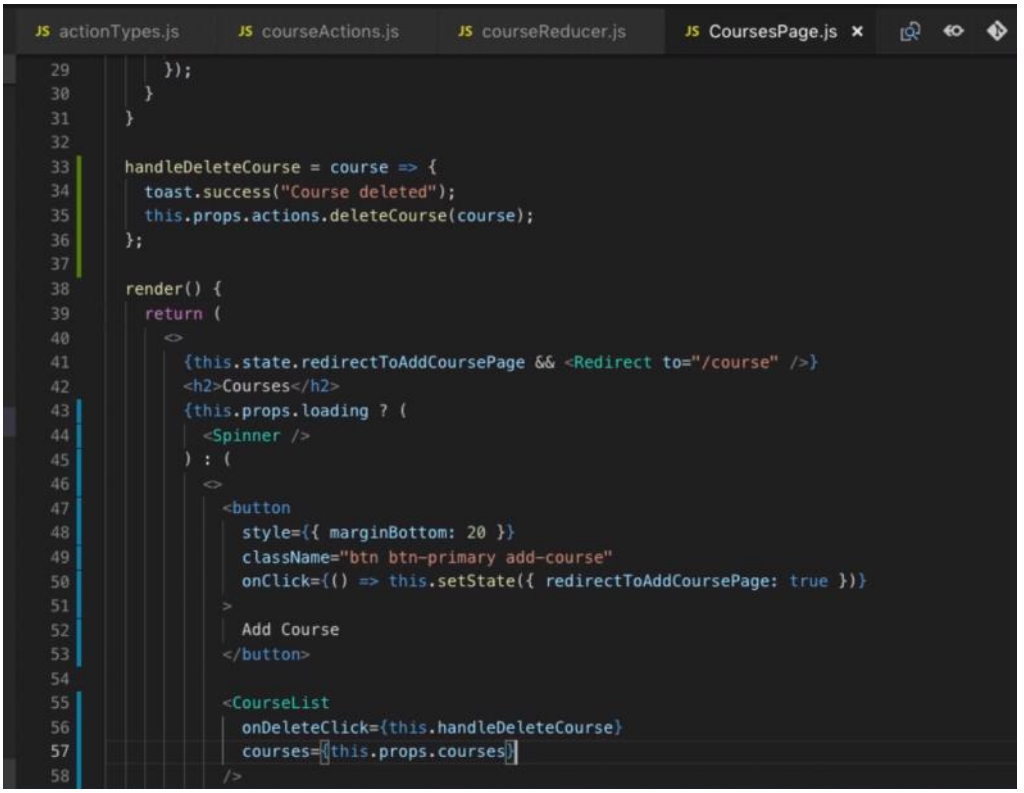
- actionTypes.js | courseActions.js | courseReducer.js



```
import * as types from "../actions/actionTypes";
import initialState from "./initialState";

export default function courseReducer(state = initialState.courses, action) {
  switch (action.type) {
    case types.CREATE_COURSE_SUCCESS:
      return [...state, { ...action.course }];
    case types.UPDATE_COURSE_SUCCESS:
      return state.map(course =>
        course.id === action.course.id ? action.course : course
      );
    case types.LOAD_COURSES_SUCCESS:
      return action.courses;
    case types.DELETE_COURSE_OPTIMISTIC:
      return state.filter(course => course.id !== action.course.id);
    default:
      return state;
  }
}
```

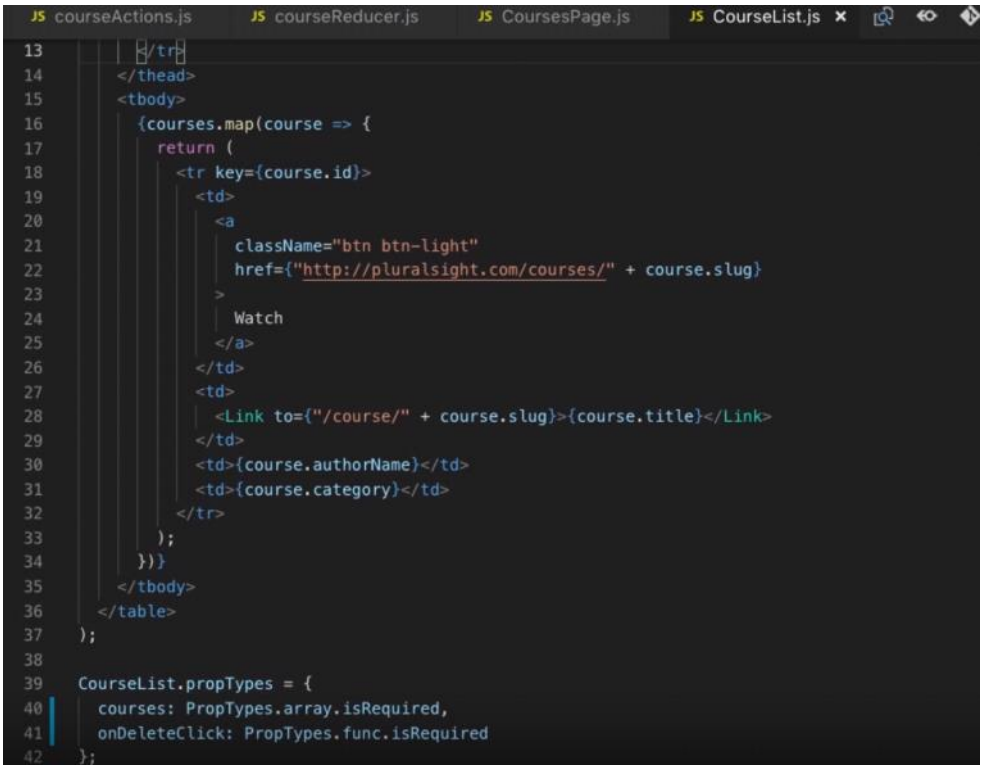
- 
```
actionTypes.js | courseActions.js | courseReducer.js | CoursesPage.js

  actions: PropTypes.object.isRequired,
  loading: PropTypes.bool.isRequired
};

function mapStateToProps(state) {
  return {
    courses:
      state.authors.length === 0
        ? []
        : state.courses.map(course => {
            return {
              ...course,
              authorName: state.authors.find(a => a.id === course.authorId).name
            };
          }),
    authors: state.authors,
    loading: state.apiCallsInProgress > 0
  };
}

function mapDispatchToProps(dispatch) {
  return {
    actions: {
      loadCourses: bindActionCreators(courseActions.loadCourses, dispatch),
      loadAuthors: bindActionCreators(authorActions.loadAuthors, dispatch),
      deleteCourse: bindActionCreators(courseActions.deleteCourse, dispatch)
    }
  };
}
```
- 
```
actionTypes.js | courseActions.js | courseReducer.js | CoursesPage.js

    });
  }

  if (authors.length === 0) {
    actions.loadAuthors().catch(error => {
      alert("Loading authors failed" + error);
    });
  }
}

handleDeleteCourse = course => {
  toast.success("Course deleted");
  this.props.actions.deleteCourse(course);
};

render() {
  return (
    <>
```
- 
```
actionTypes.js | courseActions.js | courseReducer.js | CoursesPage.js

import React from "react";
import { connect } from "react-redux";
import * as courseActions from "../../redux/actions/courseActions";
import * as authorActions from "../../redux/actions/authorActions";
import PropTypes from "prop-types";
import { bindActionCreators } from "redux";
import CourseList from "./CourseList";
import { Redirect } from "react-router-dom";
import Spinner from "../common/Spinner";
import { toast } from "react-toastify";

class CoursesPage extends React.Component {
  state = {
    redirectToAddCoursePage: false
  };
```

- actionTypes.js | courseActions.js | courseReducer.js | CoursesPage.js



```
      });
    }
  }

  handleDeleteCourse = course => {
    toast.success("Course deleted");
    this.props.actions.deleteCourse(course);
  };

  render() {
    return (
      <>
        {this.state.redirectToAddCoursePage && <Redirect to="/course" />}
        <h2>Courses</h2>
        {this.props.loading ? (
          <Spinner />
        ) : (
          <>
            <button
              style={{ marginBottom: 20 }}
              className="btn btn-primary add-course"
              onClick={() => this.setState({ redirectToAddCoursePage: true })}
            >
              Add Course
            </button>

            <CourseList
              onDeleteClick={this.handleDeleteCourse}
              courses={this.props.courses}
            />
```

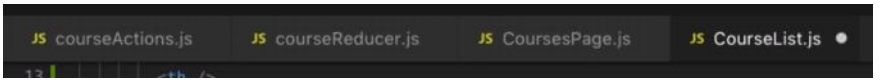
- courseActions.js | courseReducer.js | CoursesPage.js | CourseList.js

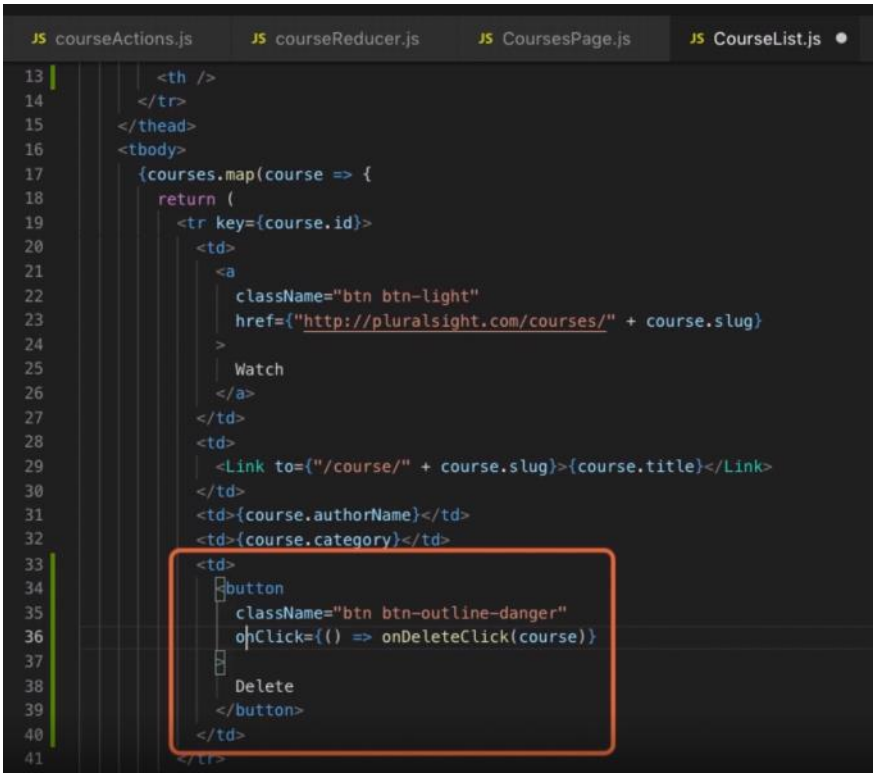


```
      </tr>
    </thead>
    <tbody>
      {courses.map(course => {
        return (
          <tr key={course.id}>
            <td>
              <a
                className="btn btn-light"
                href={"http://pluralsight.com/courses/" + course.slug}
              >
                Watch
              </a>
            </td>
            <td>
              <Link to={"/course/" + course.slug}>{course.title}</Link>
            </td>
            <td>{course.authorName}</td>
            <td>{course.category}</td>
          </tr>
        );
      })}
    </tbody>
  </table>
);

CourseList.propTypes = {
  courses: PropTypes.array.isRequired,
  onDeleteClick: PropTypes.func.isRequired
};
```

- courseActions.js | courseReducer.js | CoursesPage.js | CourseList.js

```
import React from "react";
import PropTypes from "prop-types";
import { Link } from "react-router-dom";

const CourseList = ({ courses, onDeleteClick }) => (
  <table className="table">
    <thead>
```



- 
courseActions.js
courseReducer.js
CoursesPage.js
CourseList.js
<th />
</tr>
</thead>
<tbody>
{courses.map(course => {
return (
<tr key={course.id}>
<td>
<a
className="btn btn-light"
href={"http://pluralsight.com/courses/" + course.slug}
>
Watch

</td>
<td>
<Link to={"/course/" + course.slug}>{course.title}</Link>
</td>
<td>{course.authorName}</td>
<td>{course.category}</td>
<td>
<button
className="btn btn-outline-danger"
onClick={() => onDeleteClick(course)}
>
Delete
</button>
</td>
- 
handleDeleteCourse = course => {
toast.success("Course deleted");
this.props.actions.deleteCourse(course);
};
render() {
return (
<>
Optimstic tradeoff:
+ Better user experience when call succeeds
- Confusing user experience if call fails
className="btn btn-primary add-course"
- 
CoursesPage.js
}
handleDeleteCourse = course => {
toast.success("Course deleted");
this.props.actions.deleteCourse(course).catch(error => {
toast.error("Delete failed. " + error.message, { autoClose: false });
});
};

- **Async/Await**

Async/Await

```
async function handleSaveCourse(course) {
  try {
    const courseId = await saveCourse(course);
    return courseId;
  } catch (error) {
    console.log(error);
  }
};
```

Async/Await uses promises behind the scenes, so it can easily interact with promise-based code.

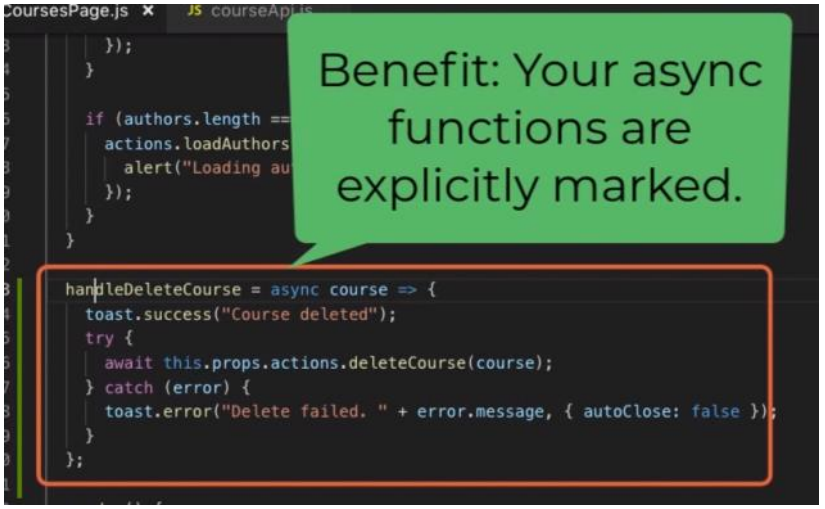
Async/Await

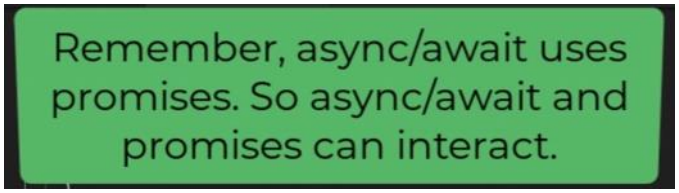
```
async function handleSaveCourse(course) {
  try {
    const courseId = await saveCourse(course);
    return courseId;
  } catch (error) {
    console.log(error);
  }
};
```

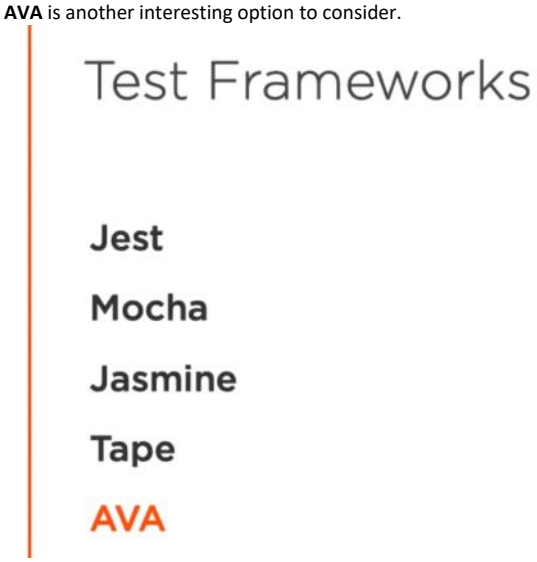
The function will pause execution and continue when the async call completes.

```
async function handleSaveCourse(course) {
  try {
    const courseId = await saveCourse(course);
    return courseId;
  } catch (error) {
    console.log(error);
  }
};
```

This returns a promise.

- 

- 

- **Testing React**
  - **Jest** -> By Facebook, Easy to setup and most popular testing framework for React. Jest also comes bundled with Create React App.
  - **Mocha** is also popular because it's highly configurable and has a large ecosystem of support.
  - **Jasmine** is similar to Mocha and Jest, but Mocha tends to be more configurable, so some people prefer Mocha over Jasmine.
  - **Tape** is arguably the leanest and simplest library of the bunch. Its simplicity and minimal configuration are its key strengths. Tape also avoids the global variable configuration that exists in Jest, Mocha, and Jasmine.
  - **AVA** is another interesting option to consider.
  - 

  - We're going to use **Jest** because it's popular, easy to configure, fast, and offers an excellent command-line interface.
  - Jest also has some unique features like Snapshot testing that allow us to easily generate and store serialized snapshots to help protect us from regressions. Helper libraries make testing React and Redux easier.

- 

Helper Libraries

React Test Utils

Enzyme

React testing library

- **React Test Utils** is a library that's specifically for testing React components. It's built and supported by Facebook. This library is powerful, but unfortunately it has a rather verbose API.

React Test Utils

- 

Specifically for React

Built by Facebook

Verbose API

- React Test Utils offers two ways to render your React components.
  - **Shallow Render**
    - The Shallow Render lets us render just the component that we're testing without rendering any of its children. That's why it's called Shallow Render. It renders only one level deep.
    - Shallow rendering is useful to constrain yourself to testing a component as a unit and ensure that your tests aren't indirectly asserting on behavior of child components.
    - When testing with shallowRender, no DOM is required.
    - Shallow Render returns an object that mirrors what we expect to see in a real DOM.
    - But sometimes you want to simulateinteractions with the DOM, such as clicks and change events. That's when Shallow Render is no longer sufficient.

  - **renderIntoDocument**
    - It actually renders the component into the DOM. So this function requires a DOM to be present.
    - Although this doesn't mean that you have to fire up an actual browser to use it.
    - Libraries like JSDOM offer a simulated DOM that runs a node. That allows you to interact with the DOM as though you're in a real browser.
    - Finally, by rendering into the document, you can simulate multiple interactions.

React Test Utils: Two Rendering Options

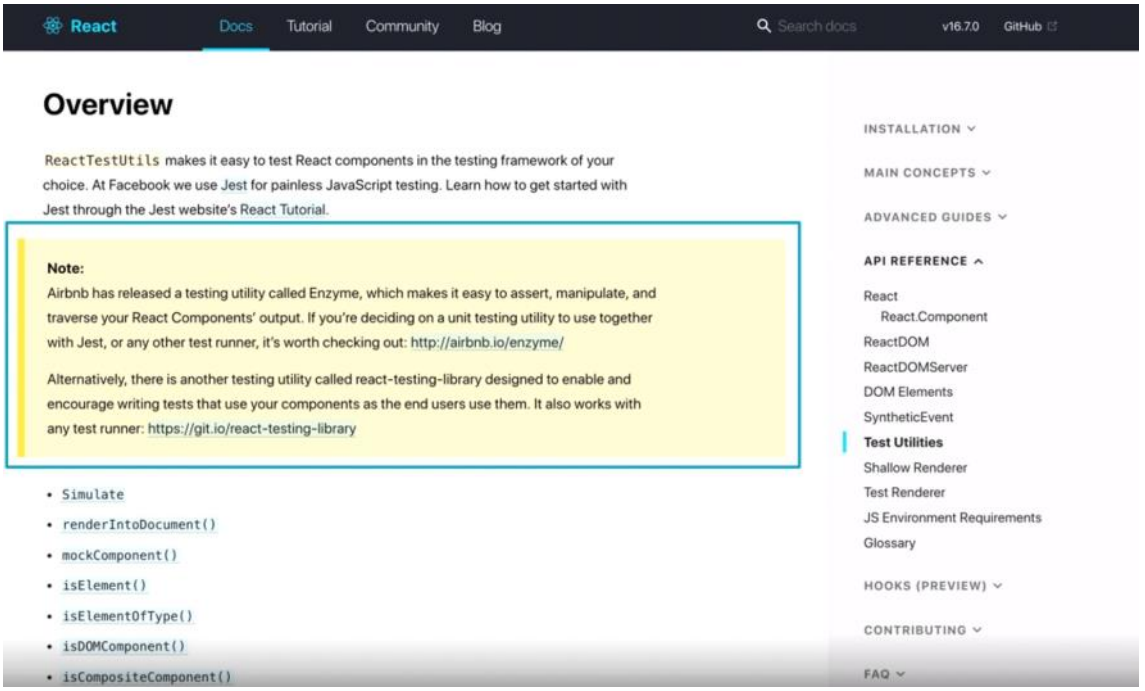
shallowRender	renderIntoDocument
Render single component	Render component and children
No DOM Required	DOM Required
Fast and Simple	Supports simulating interactions

- Below are the functions for querying and interacting with the DOM.
  - FindRenderDOMComponentWithTag is for finding a specific DOM element.
  - ScryRenderedDOMComponents finds components by tag name.
  - And once you have a reference to an element, you can do things like simulate interactions.
  - You could simulate clicks, keypresses, etc..

React Test Utils: DOM Interactions

- findRenderedDOMComponentWithTag
- scryRenderedDOMComponentsWithTag
- Simulate
  - Clicks
  - Keypresses
  - Etc.

More: reactjs.org/docs/test-utils.html

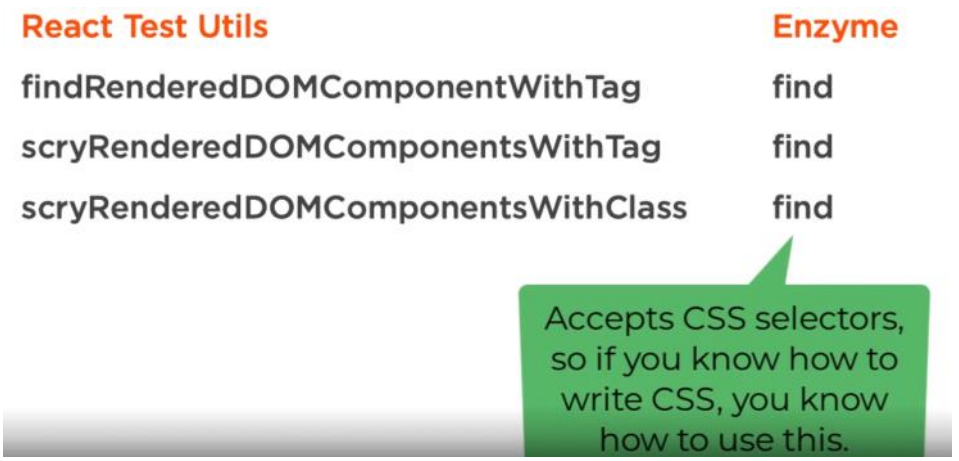


- The ReactTestUtils documentation recommends two alternatives, **Enzyme** or **React Testing Library**.
- **Enzyme**

Why Enzyme?

React Test Utils	Enzyme
findRenderedDOMComponentWithTag	find
scryRenderedDOMComponentsWithTag	find
scryRenderedDOMComponentsWithClass	find

- With Enzyme, you just call find, and you pass it a css selector to be able to get what you want.
- Enzyme is using React Test Utils behind the scenes, but it creates a much friendlier API on top of it.
- Enzyme's find function is very versatile because it accepts CSS selectors, so if you already know how to write CSS, or if you've worked on jQuery in the past, since jQuery also accepts CSS selectors, it's quite easy to learn Enzyme.



- And, of course, Enzyme's API offers much more than just find.
- Enzyme is an abstraction. Just like a theater production behind the scenes, there's a lot going on.
- Ultimately, Enzyme is calling React Test Utils behind the scenes, and it uses JSDOM to create an in-memory DOM to simulate the browser.
- It also uses a library called Cheerio that provides those CSS selectors that is just mentioned.

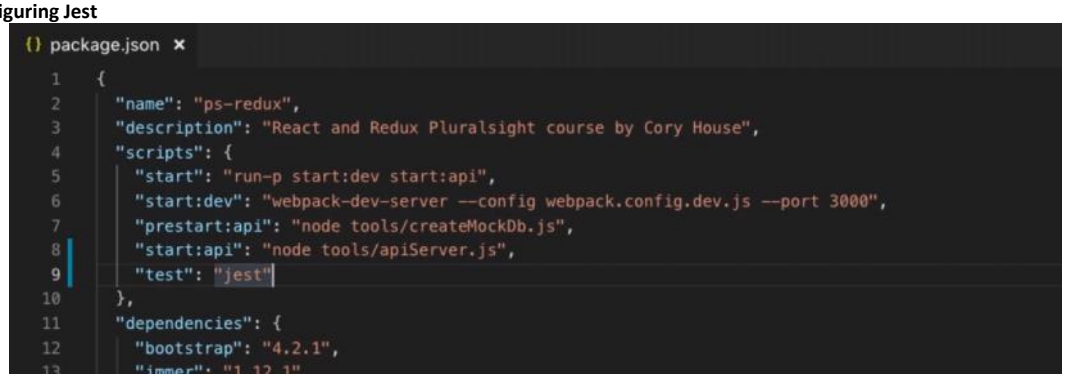
Enzyme Is An Abstraction

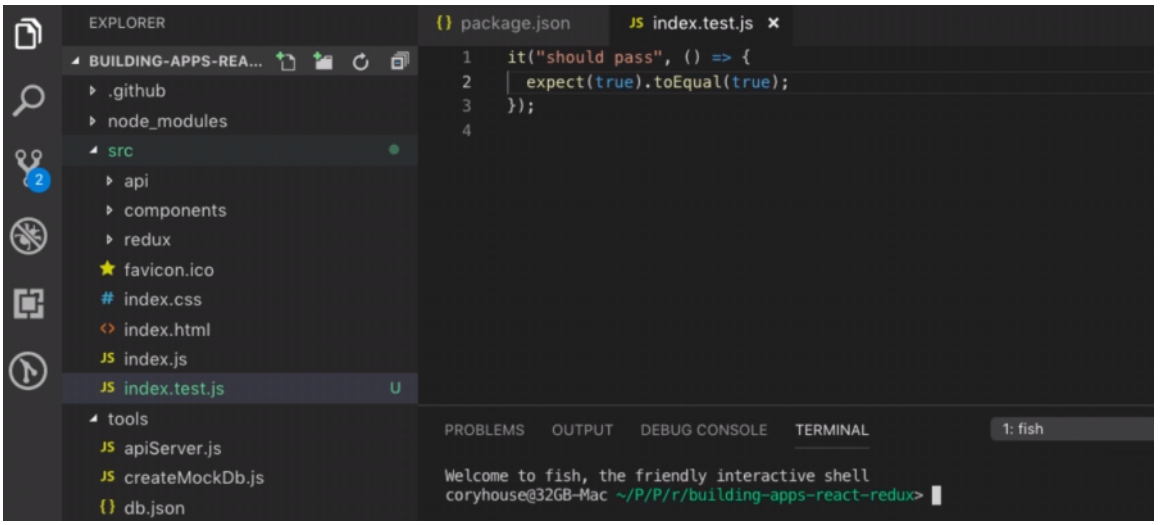
Behind the scenes
- React Test Utils
- JSDOM (In-memory DOM)
- Cheerio (Fast jQuery style selectors)

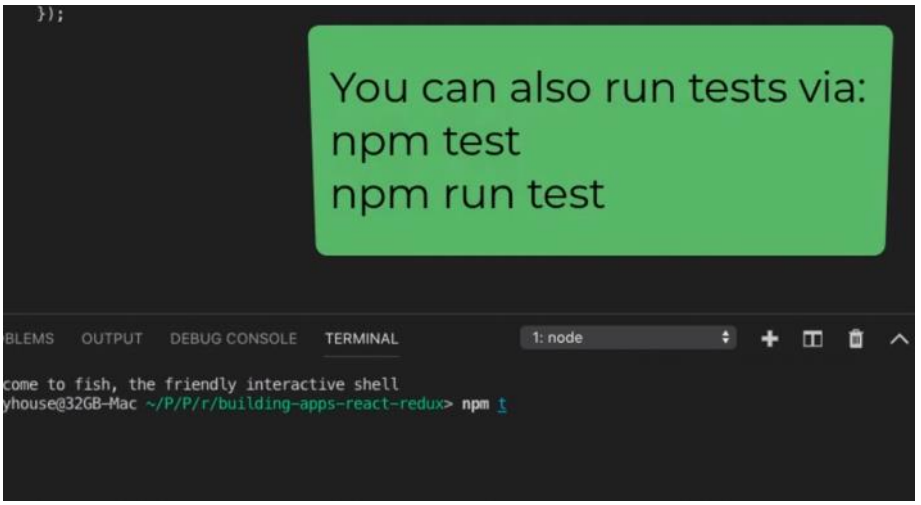
- With Enzyme, these three libraries come together to create a powerful and concise testing API and environment without having to configure them to work together, and without having to open an actual browser to run our tests.

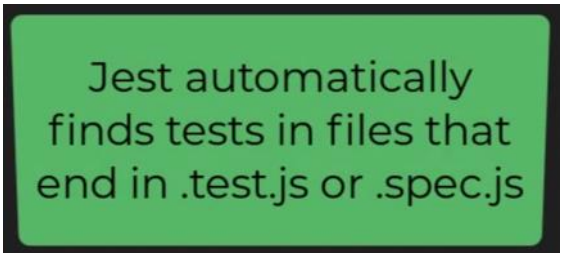
- **React Testing Library**
  - React Testing Library is an alternative to Enzyme.
  - It offers a much smaller API and helps encourage writing tests that resemble the way that your software is used.
  - This tends to lead to tests that are less brittle and helps encourage writing accessible applications.

- Like Enzyme, we can simulate interactions with our React components without opening a browser.
- Both Enzyme and react-testing-library are great options for testing React components.

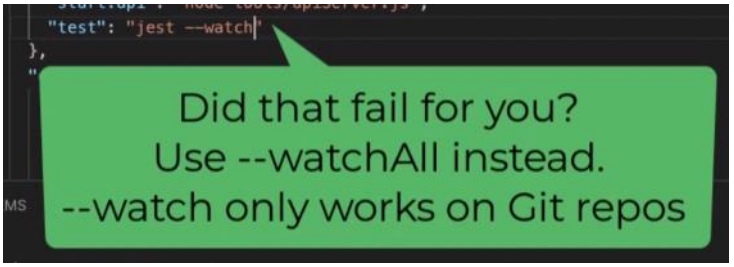
- **Configuring Jest**
  - 

  - 

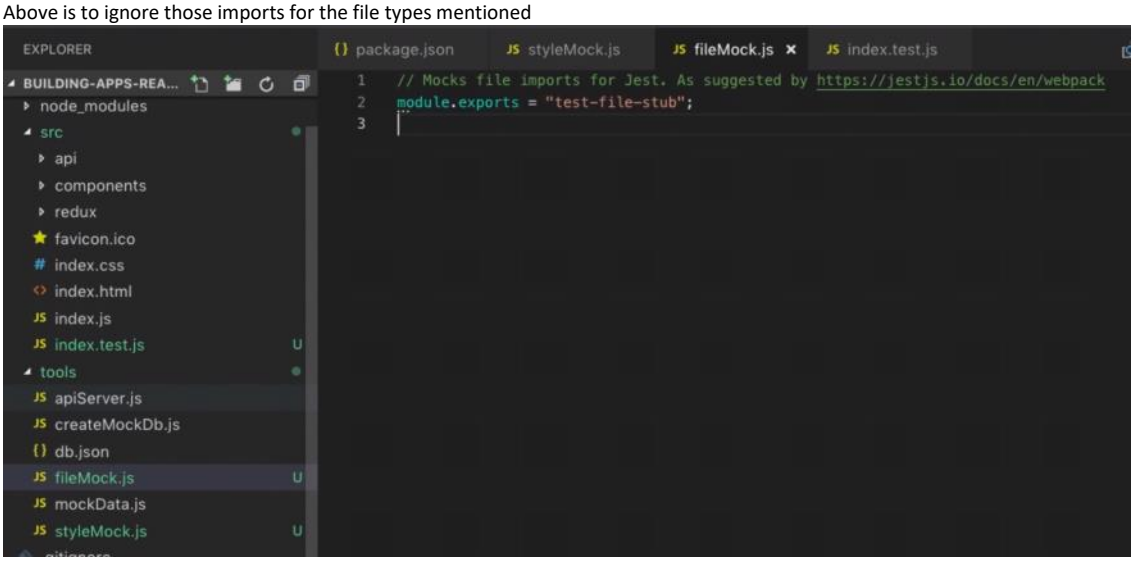
  - 

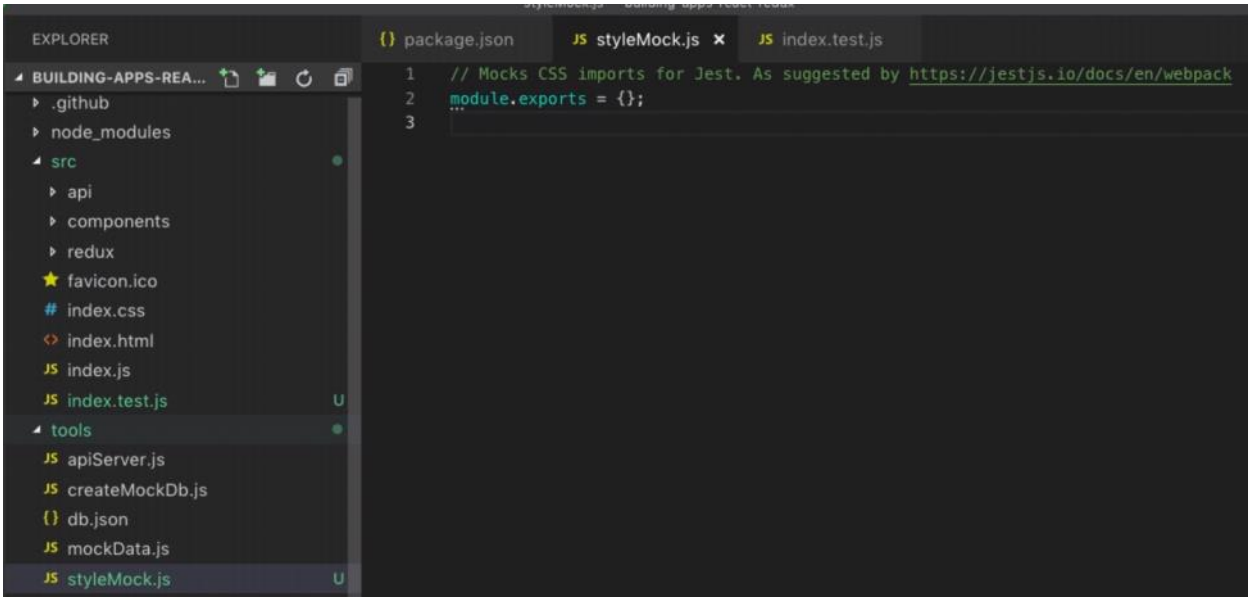
  - 

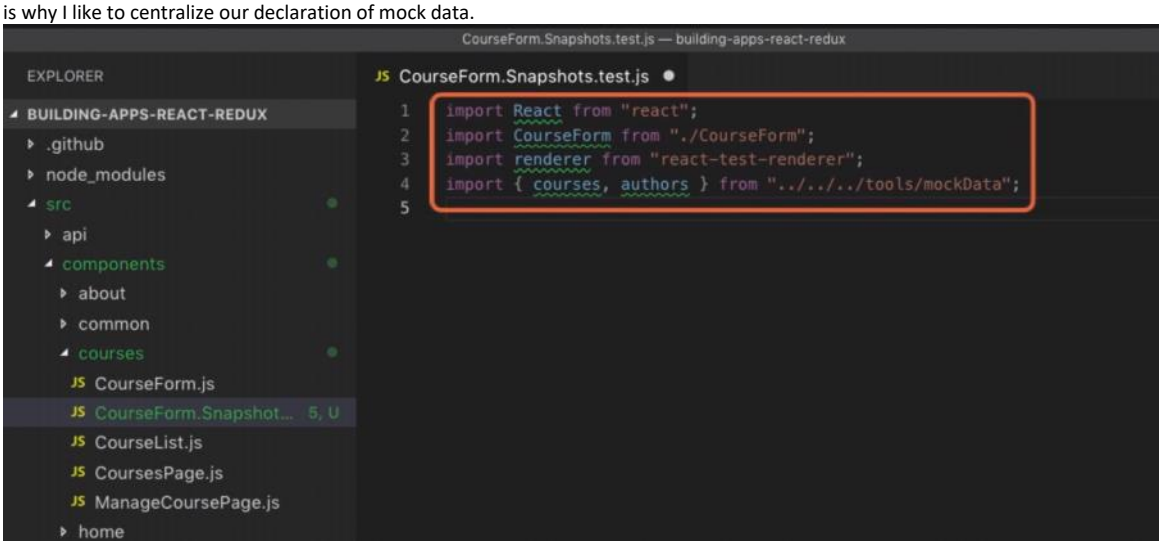
- 

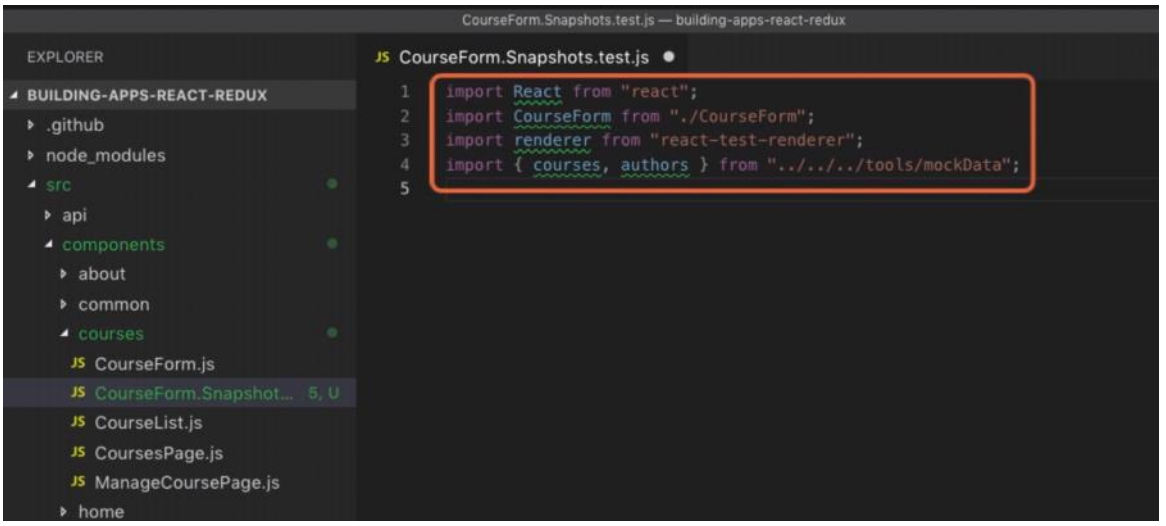
- 

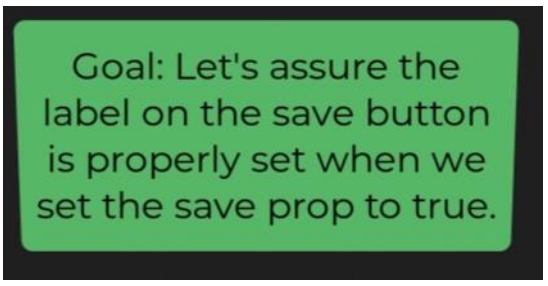
- 

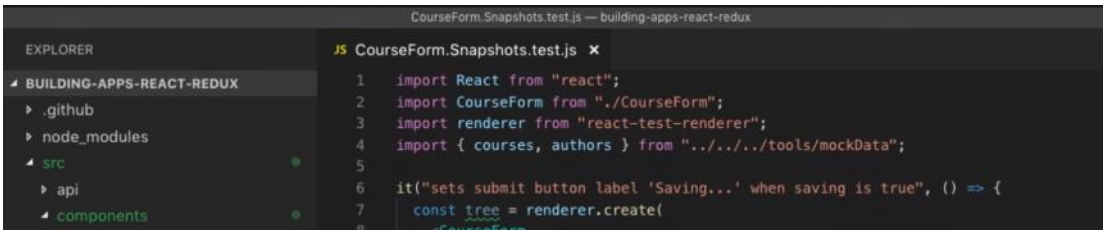
- Above is to ignore those imports for the file types mentioned
- 

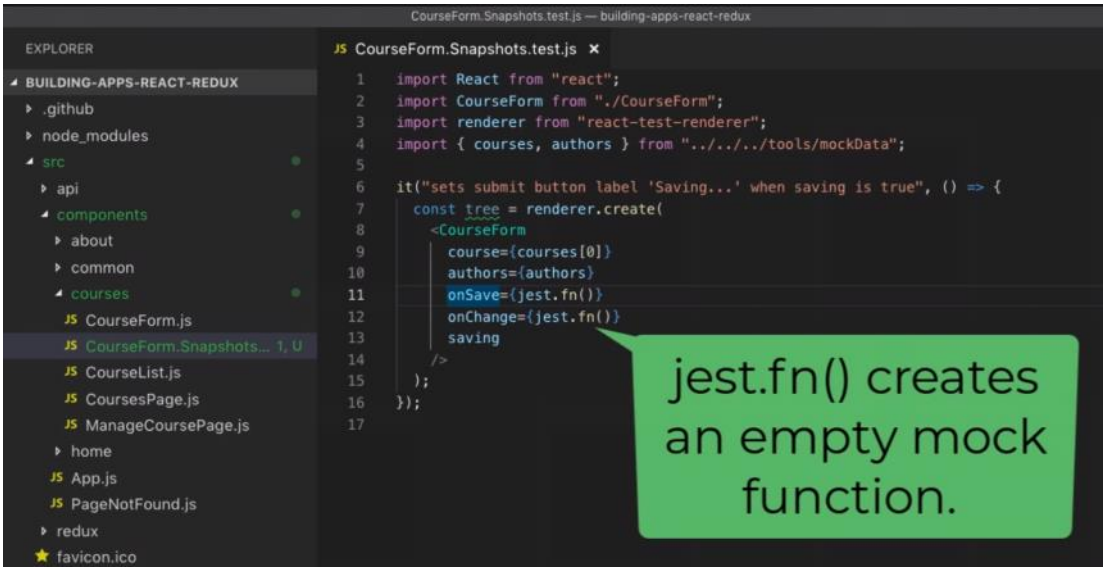
- 

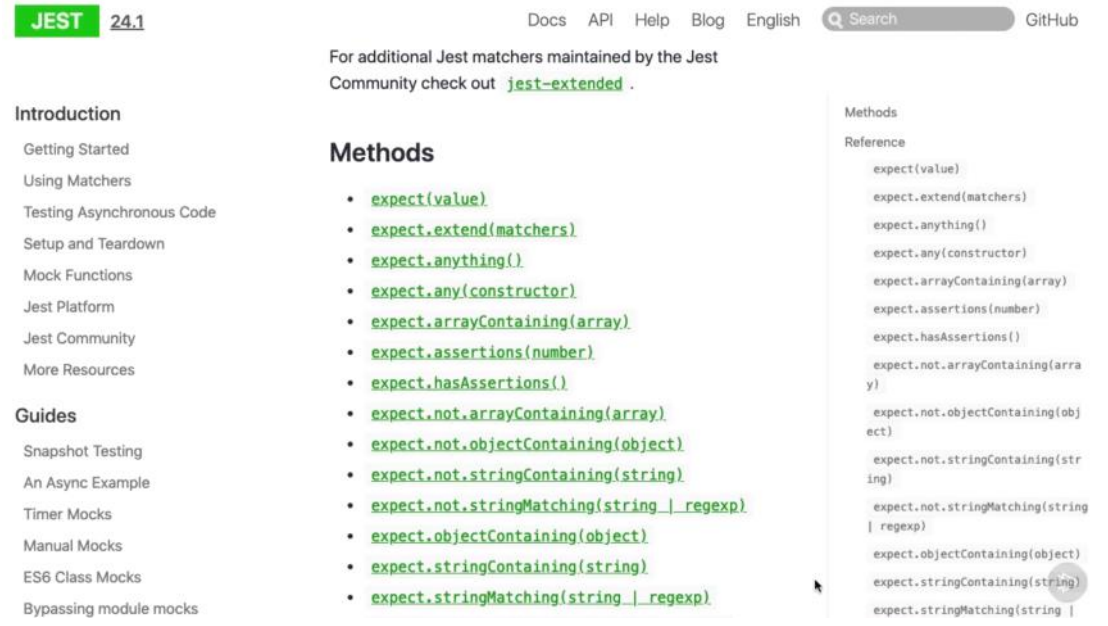
- **Test React with Jest Snapshot Tests**
  - Jest offers a rather unique feature called snapshot testing.
  - Snapshots store a record of a component's output.
  - So, snapshots can be useful for documenting expected output and regression testing protection.
  - Under the courses folder, let's create a new file "CourseForm.Snapshots.test.js".
  - To clarify, We don't have to have snapshots in the file name.
  - First, we need to import four items. We need to import React, since we're going to run a React component.
  - Our system under test will be the CourseForm.
  - We're going to use react-test-renderer to render our component, and I'm pulling in some mock course and author data for use in our test. This is why I like to centralize our declaration of mock data.
  - 

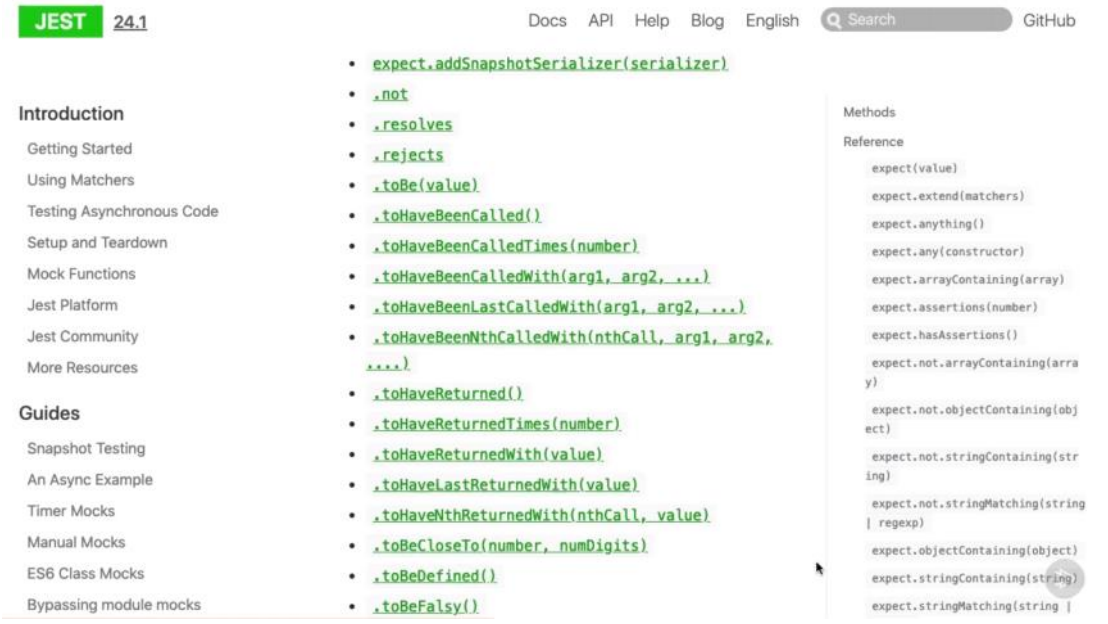
  - 

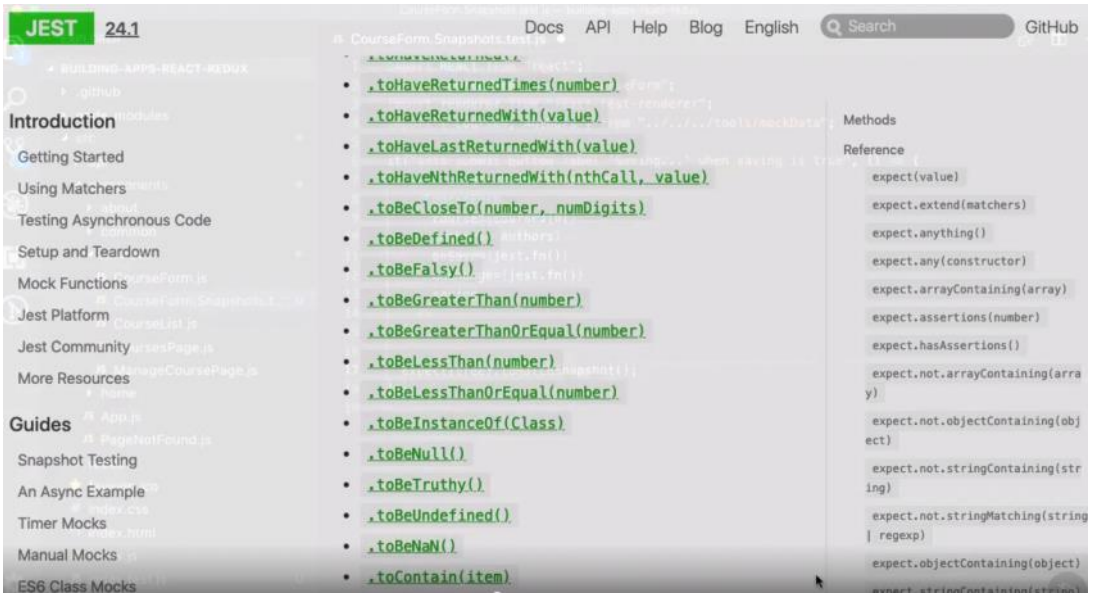
  - 

  - 


CourseForm.Snapshots.test.js — building-apps-react-redux
EXPLORER
BUILDING-APPS-REACT-REDUX
.github
node_modules
src
api
components
about
common
courses
CourseForm.js
CourseForm.Snapshots... 1, U
CourseList.js
CoursesPage.js
ManageCoursePage.js
home
App.js
PageNotFound.js
redux
favicon.ico
CourseForm.Snapshots.test.js
import React from "react";
import CourseForm from "./CourseForm";
import renderer from "react-test-renderer";
import { courses, authors } from "../../../tools/mockData";
it("sets submit button label 'Saving...' when saving is true", () => {
const tree = renderer.create(
<CourseForm
course={courses[0]}
authors={authors}
onSave={jest.fn()}
onChange={jest.fn()}
saving
/>
);
});
jest.fn() creates an empty mock function.


JEST 24.1
Docs API Help Blog English Search GitHub
For additional Jest matchers maintained by the Jest Community check out jest-extended .
Introduction
Getting Started
Using Matchers
Testing Asynchronous Code
Setup and Teardown
Mock Functions
Jest Platform
Jest Community
More Resources
Guides
Snapshot Testing
An Async Example
Timer Mocks
Manual Mocks
ES6 Class Mocks
Bypassing module mocks
Methods
expect(value)
expect.extend(matchers)
expect.anything()
expect.any(constructor)
expect.arrayContaining(array)
expect.assertions(number)
expect.hasAssertions()
expect.not.arrayContaining(array)
expect.not.objectContaining(object)
expect.not.stringContaining(string)
expect.not.stringMatching(string | regexp)
expect.objectContaining(object)
expect.stringContaining(string)
expect.stringMatching(string | regexp)
Methods
Reference
expect(value)
expect.extend(matchers)
expect.anything()
expect.any(constructor)
expect.arrayContaining(array)
expect.assertions(number)
expect.hasAssertions()
expect.not.arrayContaining(array)
expect.not.objectContaining(object)
expect.not.stringContaining(string)
expect.not.stringMatching(string | regexp)
expect.objectContaining(object)
expect.stringContaining(string)
expect.stringMatching(string |


JEST 24.1
Docs API Help Blog English Search GitHub
expect.addSnapshotSerializer(serializer)
.not
.resolves
.rejects
.toBe(value)
.toHaveBeenCalled()
.toHaveBeenCalledTimes(number)
.toHaveBeenCalledWith(arg1, arg2, ...)
.toHaveBeenLastCalledWith(arg1, arg2, ...)
.toHaveBeenNthCalledWith(nthCall, arg1, arg2, ...)
.toHaveReturned()
.toHaveReturnedTimes(number)
.toHaveReturnedWith(value)
.toHaveLastReturnedWith(value)
.toHaveNthReturnedWith(nthCall, value)
.toBeCloseTo(number, numDigits)
.toBeDefined()
.toBeFalsy()
Introduction
Getting Started
Using Matchers
Testing Asynchronous Code
Setup and Teardown
Mock Functions
Jest Platform
Jest Community
More Resources
Guides
Snapshot Testing
An Async Example
Timer Mocks
Manual Mocks
ES6 Class Mocks
Bypassing module mocks
Methods
Reference
expect(value)
expect.extend(matchers)
expect.anything()
expect.any(constructor)
expect.arrayContaining(array)
expect.assertions(number)
expect.hasAssertions()
expect.not.arrayContaining(array)
expect.not.objectContaining(object)
expect.not.stringContaining(string)
expect.not.stringMatching(string | regexp)
expect.objectContaining(object)
expect.stringContaining(string)
expect.stringMatching(string |

- 

- CourseForm.Snapshots.test.js — building-apps-react-redux

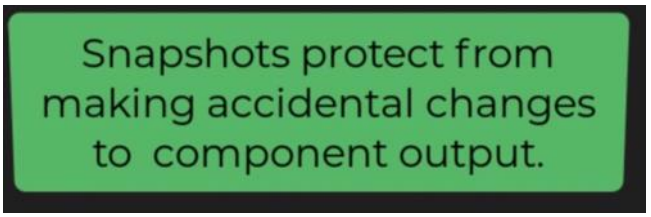
CourseForm.Snapshots.test.js

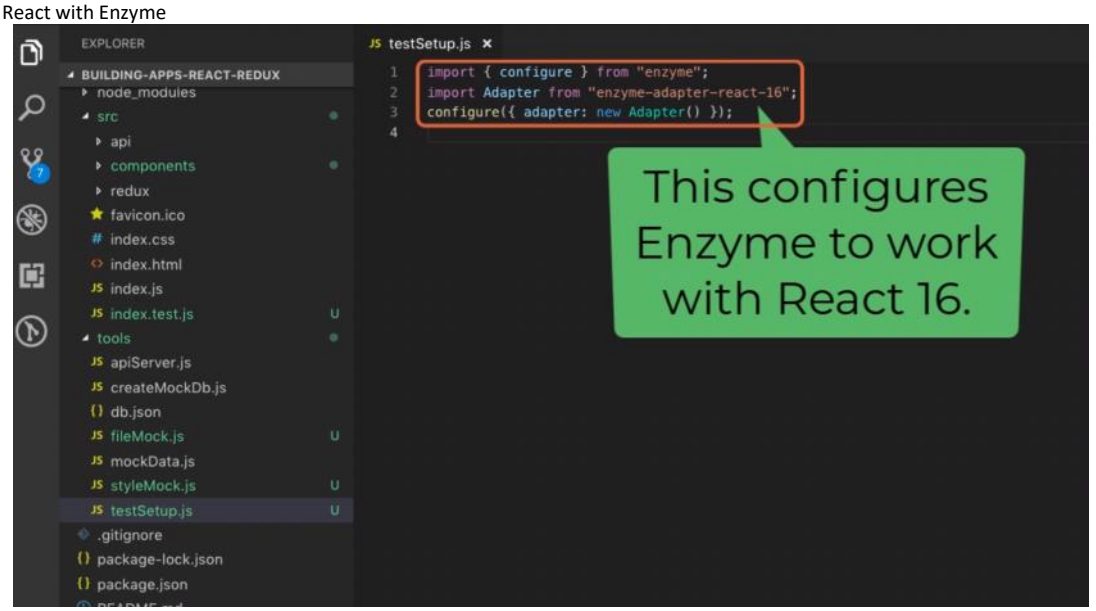
```
import React from "react";
import CourseForm from "./CourseForm";
import renderer from "react-test-renderer";
import { courses, authors } from "../../../tools/mockData";

it("sets submit button label 'Saving...' when saving is true", () => {
  const tree = renderer.create(
    <CourseForm
      course={courses[0]}
      authors={authors}
      onSave={jest.fn()}
      onChange={jest.fn()}
      saving
    />
  );

  expect(tree).toMatchSnapshot();
});

```

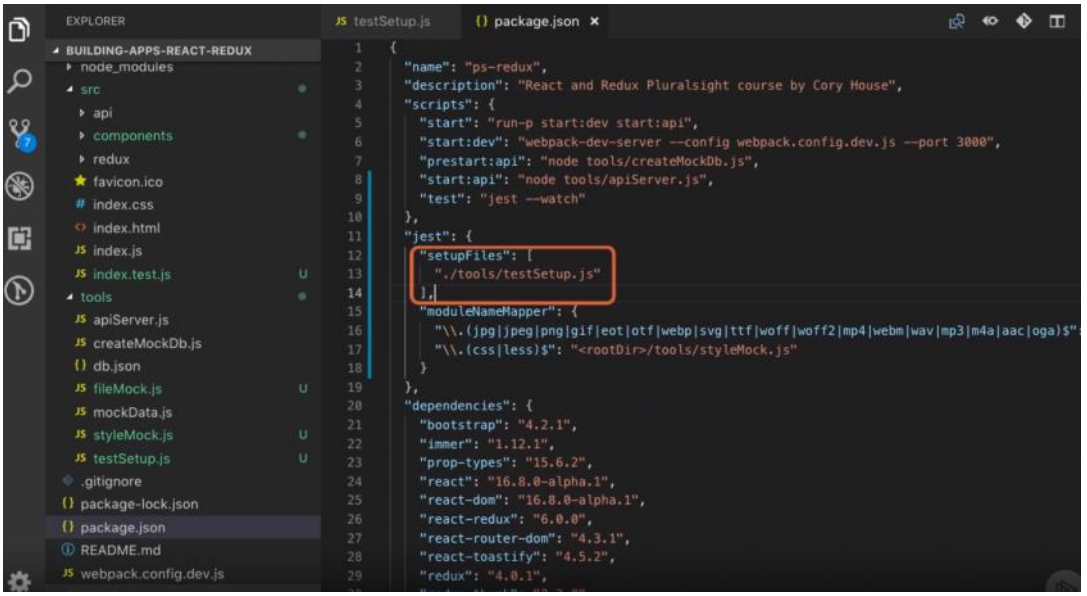
- CourseForm.Snapshots.test.js | CourseForm.Snapshots.test.js.snap

```
it("sets submit button label 'Saving...' when saving is true", () => {
  const tree = renderer.create(
    <CourseForm
      course={courses[0]}
      authors={authors}
      onSave={jest.fn()}
      onChange={jest.fn()}
      saving
    />
  );

  expect(tree).toMatchSnapshot();
});


it("sets submit button label 'Save' when saving is false", () => {
    const tree = renderer.create(
        <CourseForm
            course={courses[0]}
            authors={authors}
            onSave={jest.fn()}
            onChange={jest.fn()}
            saving={false}
        />
    );

    expect(tree).toMatchSnapshot();
});
```

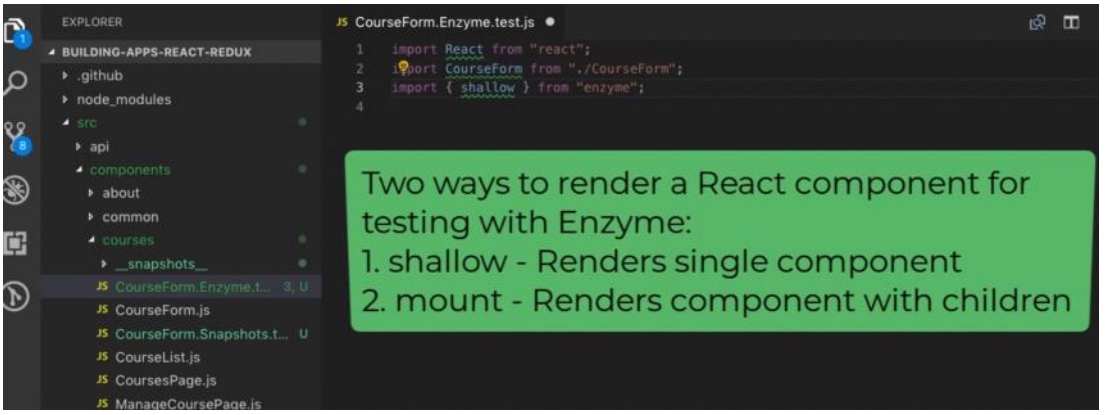
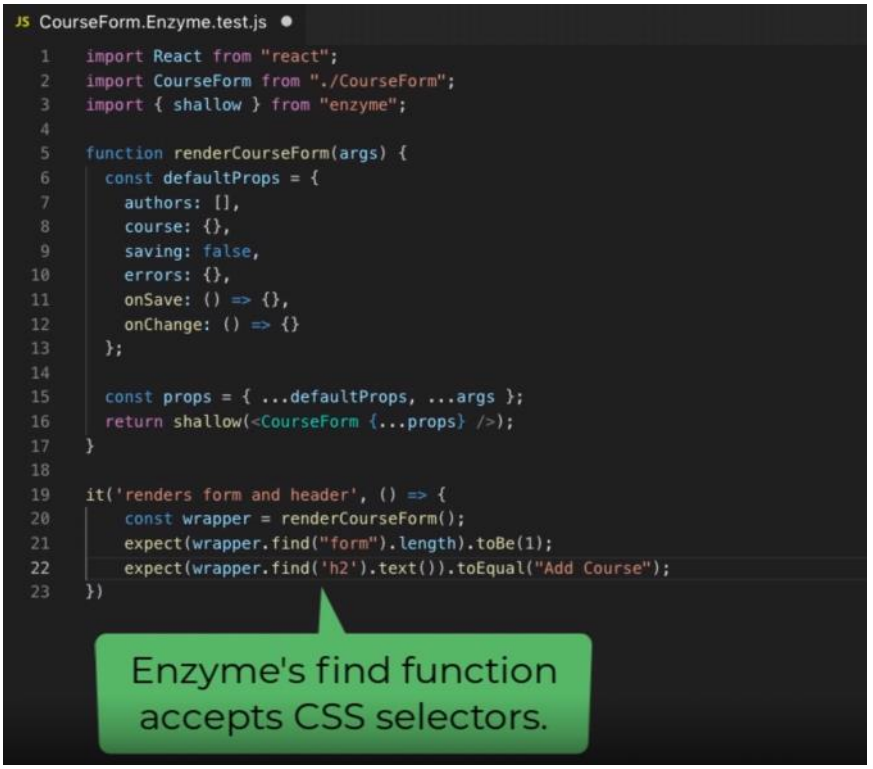
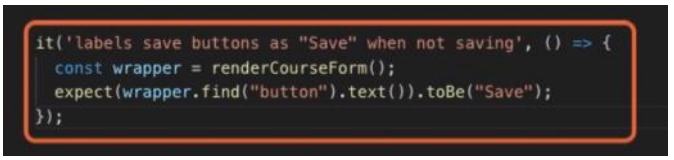
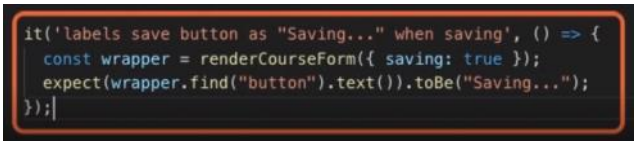
```
    );

    expect(tree).toMatchSnapshot();
});
```

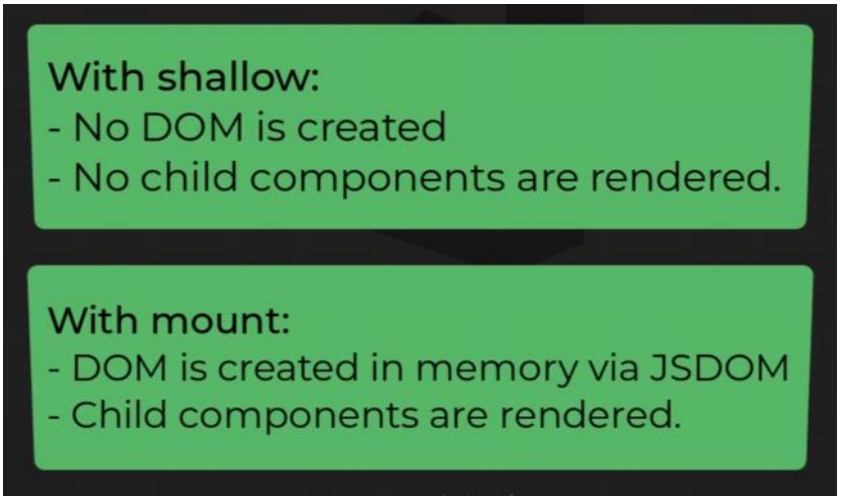
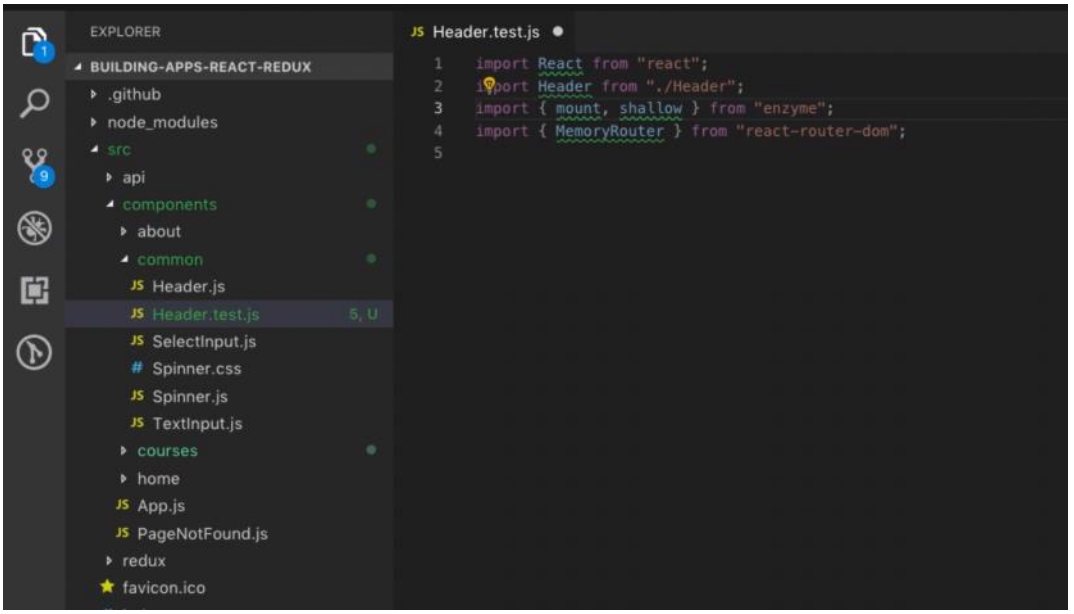
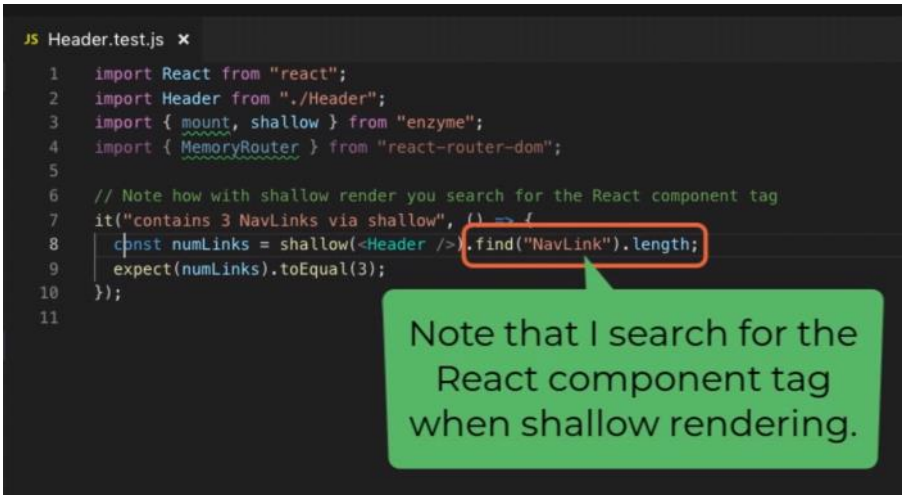
- 

- Test React with Enzyme
  - 

  - 

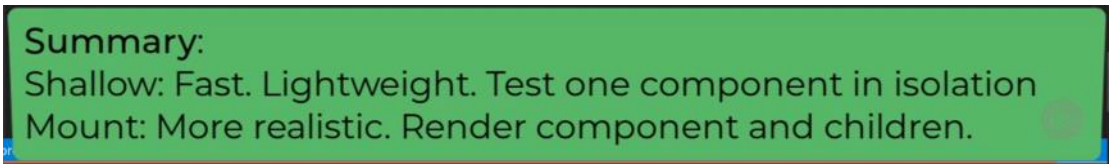
  - Configuring Enzyme requires pulling in an adapter for the particular version of React that we're using. This configures Enzyme to work with React 16. As newer versions of React are released, we'll have to use the corresponding adapter for that version.
  - We also need to tell Jest to call this adapter, so in package.json and we need to add one more section here in the Jest config.
  - Jest will run any items that we declare under the setupFiles array. We call the test setupFile that we just configure for Enzyme.

- 
EXPLORER
BUILDING-APPS-REACT-REDUX
.github
node_modules
src
api
components
about
common
courses
__snapshots__
CourseForm.Enyzme.t... 3, U
CourseForm.js
CourseForm.Snapshots.t... U
CourseList.js
CoursesPage.js
ManageCoursePage.js
CourseForm.Enzyme.test.js
import React from "react";
import CourseForm from "./CourseForm";
import { shallow } from "enzyme";
Two ways to render a React component for testing with Enzyme:
1. shallow - Renders single component
2. mount - Renders component with children
- 
CourseForm.Enzyme.test.js
import React from "react";
import CourseForm from "./CourseForm";
import { shallow } from "enzyme";
function renderCourseForm(args) {
const defaultProps = {
authors: [],
course: {},
saving: false,
errors: {},
onSave: () => {},
onChange: () => {}
};
const props = { ...defaultProps, ...args };
return shallow(<CourseForm {...props} />);
}
it('renders form and header', () => {
const wrapper = renderCourseForm();
expect(wrapper.find("form").length).toBe(1);
expect(wrapper.find('h2').text()).toEqual("Add Course");
})
Enzyme's find function accepts CSS selectors.
- 
Examples:
ID: find('#firstname')
Class: find('.wrapper')
Tag: find('h1')
- 
it('labels save buttons as "Save" when not saving', () => {
const wrapper = renderCourseForm();
expect(wrapper.find("button").text()).toBe("Save");
});
- 
it('labels save button as "Saving..." when saving', () => {
const wrapper = renderCourseForm({ saving: true });
expect(wrapper.find("button").text()).toBe("Saving...");
});

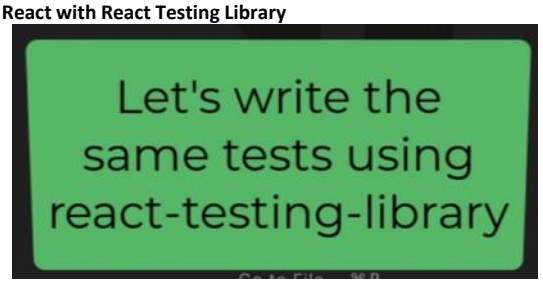
- 
With shallow:
- No DOM is created
- No child components are rendered.
With mount:
- DOM is created in memory via JSDOM
- Child components are rendered.
- 
EXPLORER
BUILDING-APPS-REACT-REDUX
.github
node_modules
src
api
components
about
common
Header.js
Header.test.js
SelectInput.js
Spinner.css
Spinner.js
TextInput.js
courses
home
App.js
PageNotFound.js
redux
favicon.ico
Header.test.js
import React from "react";
import Header from "./Header";
import { mount, shallow } from "enzyme";
import { MemoryRouter } from "react-router-dom";
- 
Header.test.js
import React from "react";
import Header from "./Header";
import { mount, shallow } from "enzyme";
import { MemoryRouter } from "react-router-dom";
// Note how with shallow render you search for the React component tag
it("contains 3 NavLinks via shallow", () => {
const numLinks = shallow(<Header />).find("NavLink").length;
expect(numLinks).toEqual(3);
});
Note that I search for the React component tag when shallow rendering.

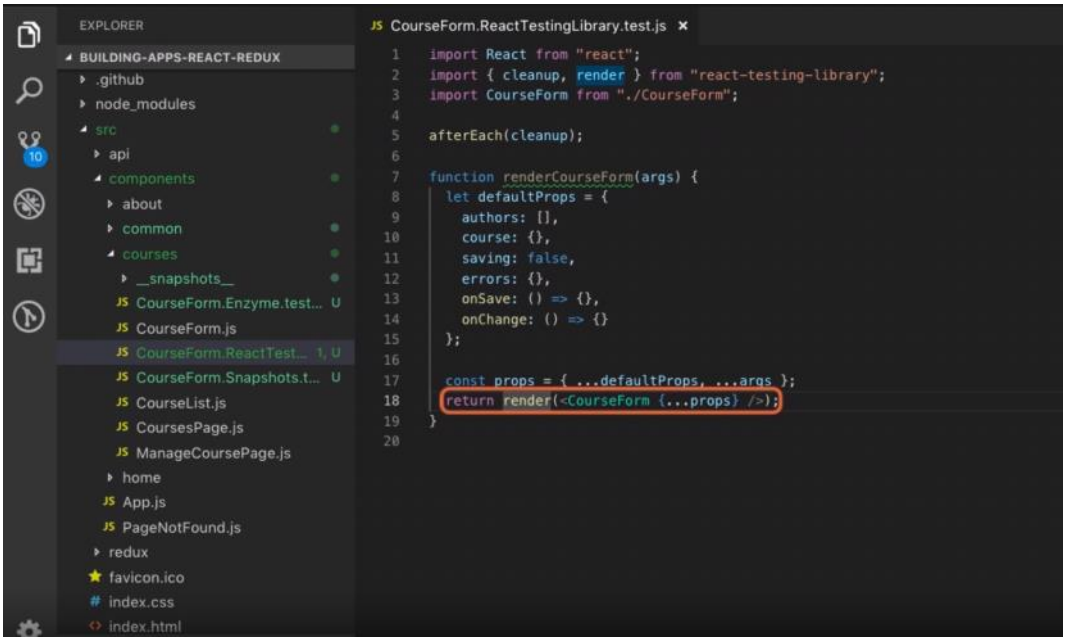
- Header.test.js

```
import React from "react";
import Header from "./Header";
import { mount, shallow } from "enzyme";
import { MemoryRouter } from "react-router-dom";

// Note how with shallow render you search for the React component tag
it("contains 3 NavLinks via shallow", () => {
  const numLinks = shallow(<Header />).find("NavLink").length;
  expect(numLinks).toEqual(3);
});

// Note how with mount you search for the final rendered HTML since it generates the final DOM.
// We also need to pull in React Router's memoryRouter for testing since the Header expects to have React Router's props passed in
it("contains 3 anchors via mount", () => {
  const numAnchors = mount(
    <MemoryRouter>
      <Header />
    </MemoryRouter>
  ).find("a").length;

  expect(numAnchors).toEqual(3);
});
```

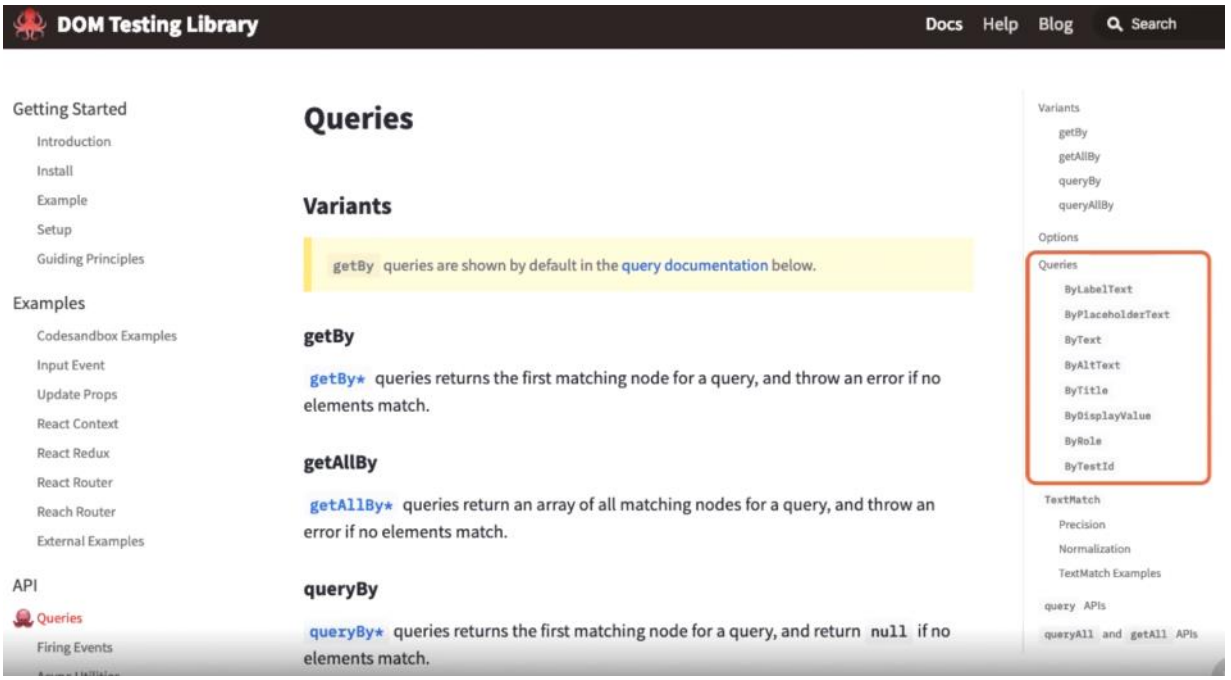
- 

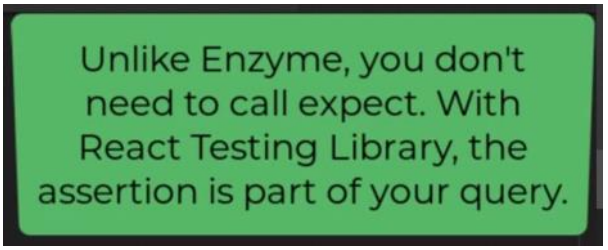
- So use mount if we want to test the final DOM, use refs, or test interactions with child components.

- **Test React with React Testing Library**
  - 

  - 

-
```
20
21  it("should render Add Course header", () => {
22    const { getByText } = renderCourseForm();
23    getByText("Add Course");
24  });
25
```
  - The react-testing-library docs provide a number of different methods for querying our React component's DOM.
  - But in this case, we're looking for particular text. So this will search through our React component's output and look for the text Add Course.
  - The getByText function has an assertion built in, so if it doesn't find the text that we pass, then it will fail.
-
```
it('should label save button as "Save" when not saving', () => {
  const { getByText } = renderCourseForm();
  getByText("Save");
});
```
- 

- 

- 

-
```
it('should label save button as "Saving..." when saving', () => {
  const { getByText, debug } = renderCourseForm({ saving: true });
  debug();
  getByText("Saving...");
});
```
  - The above is the way to debug in react testing library

- **Testing Redux**
  - Connected components
  - Redux
    - Action creators
    - Thunks
    - Reducers
    - Store
  - Testing Connected Components
    - **Test markup**
      - Given a certain set of props, we should expect to get certain output.
      - For container components, the service area should be minimal here since we'll save our exploration of testing markup for the presentation components.
      - Remember, markup belongs in presentation components, so ideally the only JSX in a container component is a reference to a child component.
    - **Test behavior**
  - **Testing container components**

Testing Container Components

They're wrapped in a call to connect! What do I do?

- Our container components should have very little markup anyway, since that should be handled by our presentation components as much as possible in order to separate concerns.
- The tricky thing about testing container components is they're all wrapped in a call to connect. And the connect function assumes that our app is ultimately wrapped in a provider component, so our container components don't export the component that we wrote. Instead, they export the component wrapped in a call to connect.

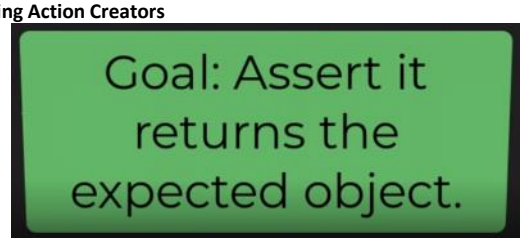
Testing Container Components

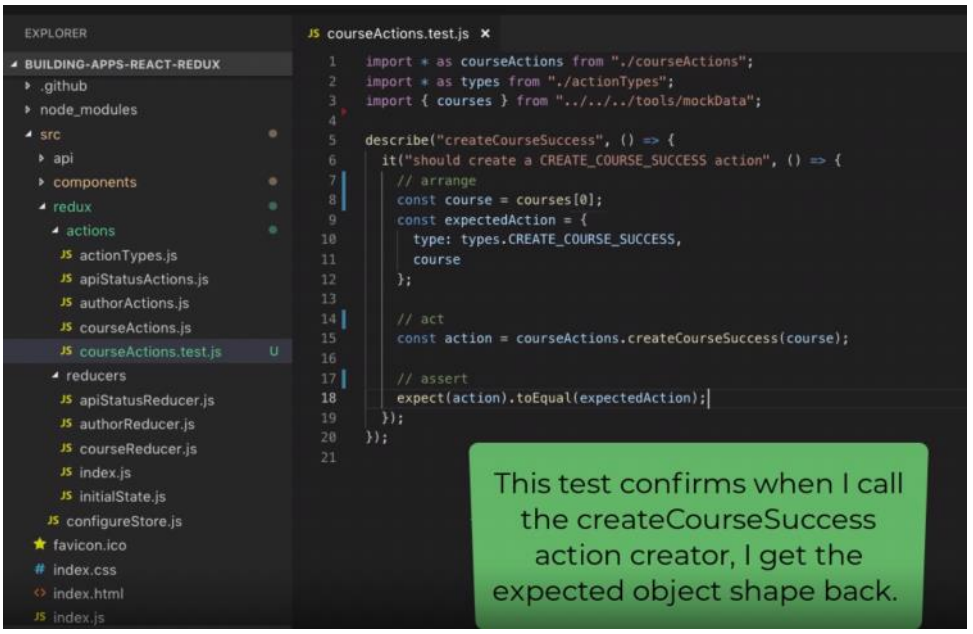
They're wrapped in a call to connect! What do I do?

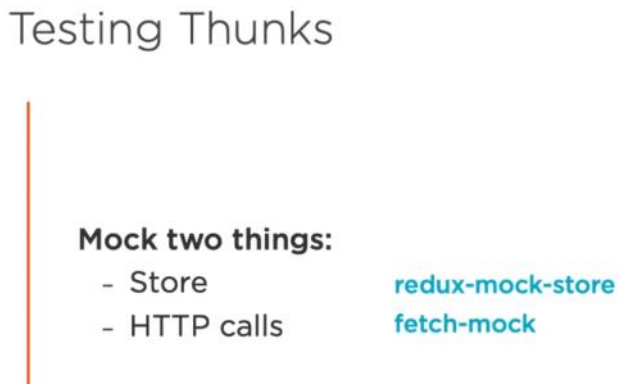
1. Wrap with <Provider>
2. Add named export for unconnected component

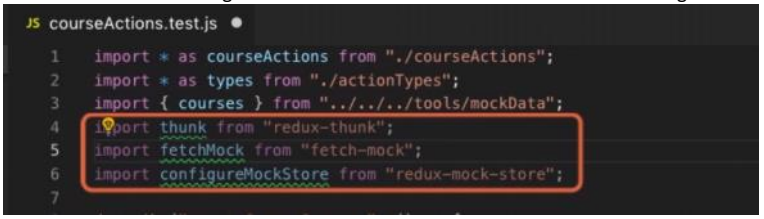
- There are two ways that we can handle this.
  - First, we can wrap the container component with React Redux's provider component within our test. So with this approach, you actually reference the store and pass it to the provider and compose your component under the test. The advantage of this approach is you can actually create a custom store for the test. So this approach is useful if you want to test the Redux-related

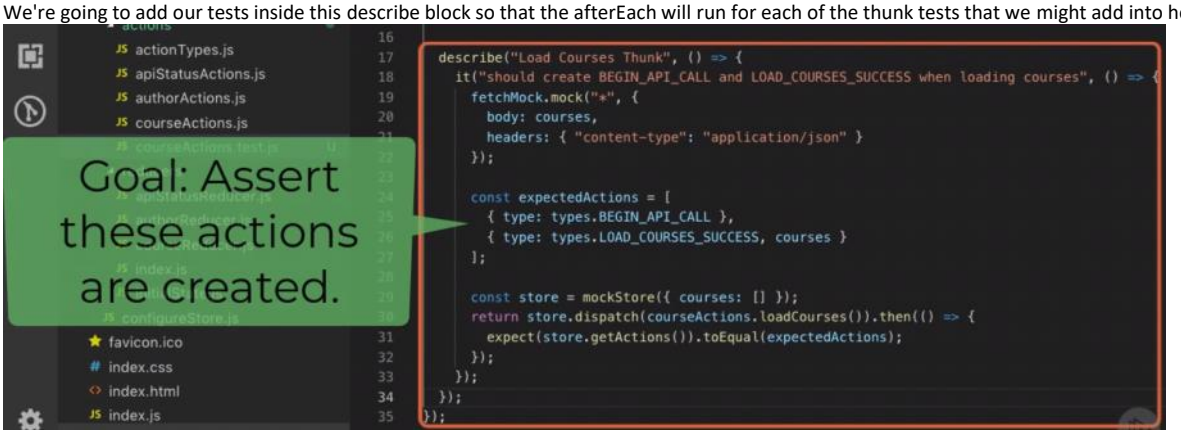
portions of your component.
- If you're merely interested in testing the component's rendering and local state-related behaviors, you can simply add a named export for the unconnected plain version of your component. As you'll see, if you forget to do one of these two things when trying to test a container component, you will at least get a handy error message that guides you in this direction.

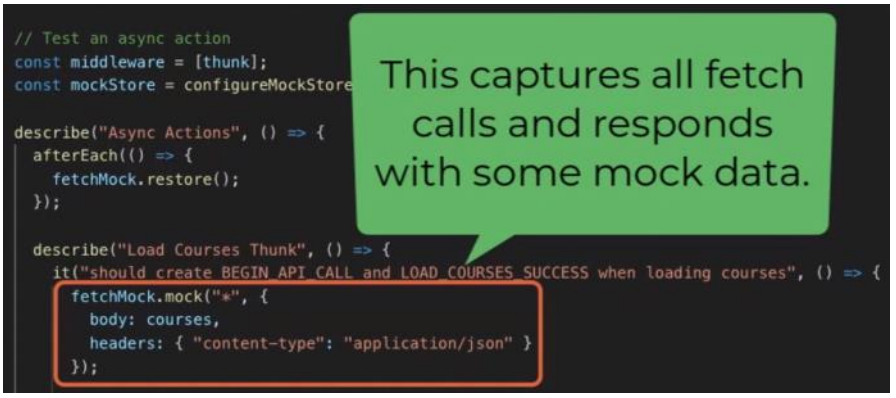
- **Testing Action Creators**
  - 

  - 

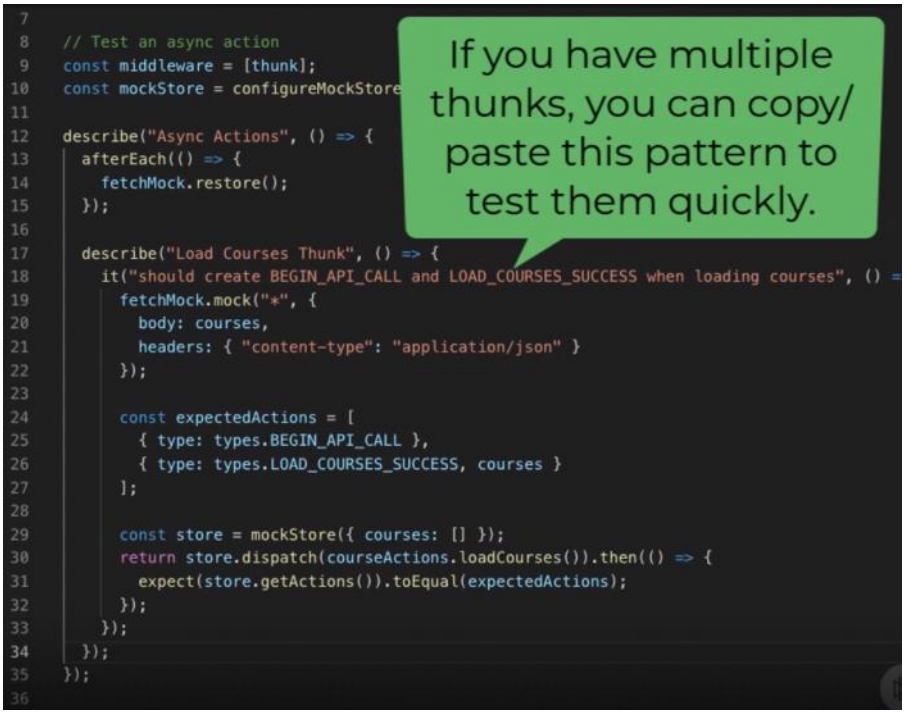
- **Testing Thunks**
  - 

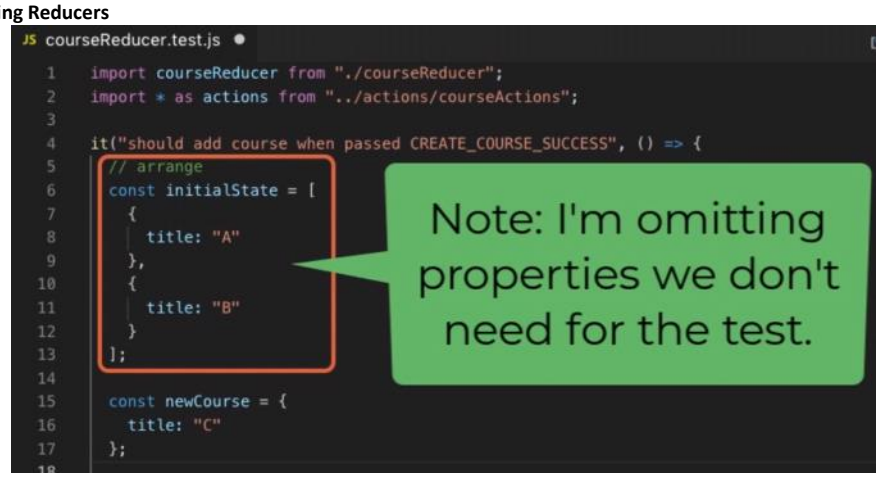
  - Thunks handle asynchrony.
  - They often dispatch multiple actions and often interact with APIs. This makes them a little trickier to test.
  - To test a thunk, it's going to require some mocking. We need to mock two things;
    - our Redux store and
    - any API calls that we make.
  - We'll mock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

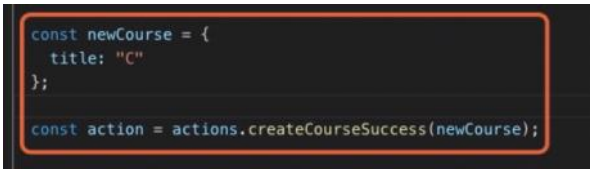
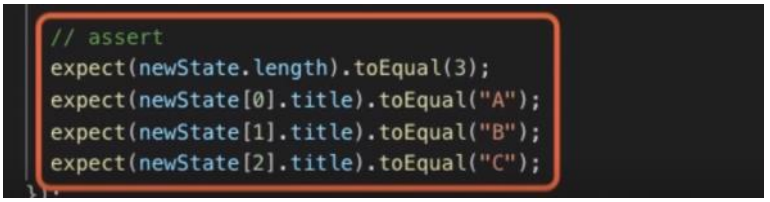
-
```
// Test an async action
const middleware = [thunk];
const mockStore = configureMockStore(middleware);
```
-
```
describe("Async Actions", () => {
    afterEach(() => {
        fetchMock.restore();
    });
});
```
- We are going to place tests for Async Actions within this describe block, and every test that we put inside of here, this particular afterEach will run, reusing fetchMock to mock the fetch call that happens in our thunks.
- To keep our tests atomic, it's important to run fetchMock.restore after each test. This initializes fetchMock for each test.
- We're going to add our tests inside this describe block so that the afterEach will run for each of the thunk tests that we might add into here.
- 

- 

-
```
const expectedActions = [
  { type: types.BEGIN_API_CALL },
  { type: types.LOAD_COURSES_SUCCESS, courses }
];
```
-
```
const store = mockStore({ courses: [] });
```
-
```
return store.dispatch(courseActions.loadCourses()).then(() => {
  expect(store.getActions()).toEqual(expectedActions);
});
});
});
```

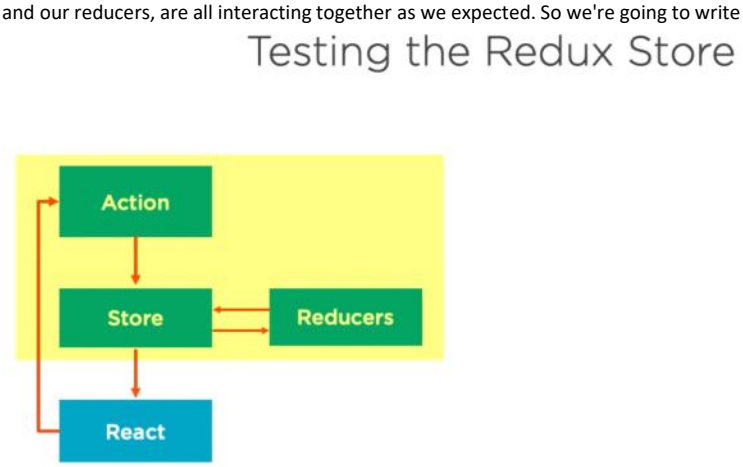
- 

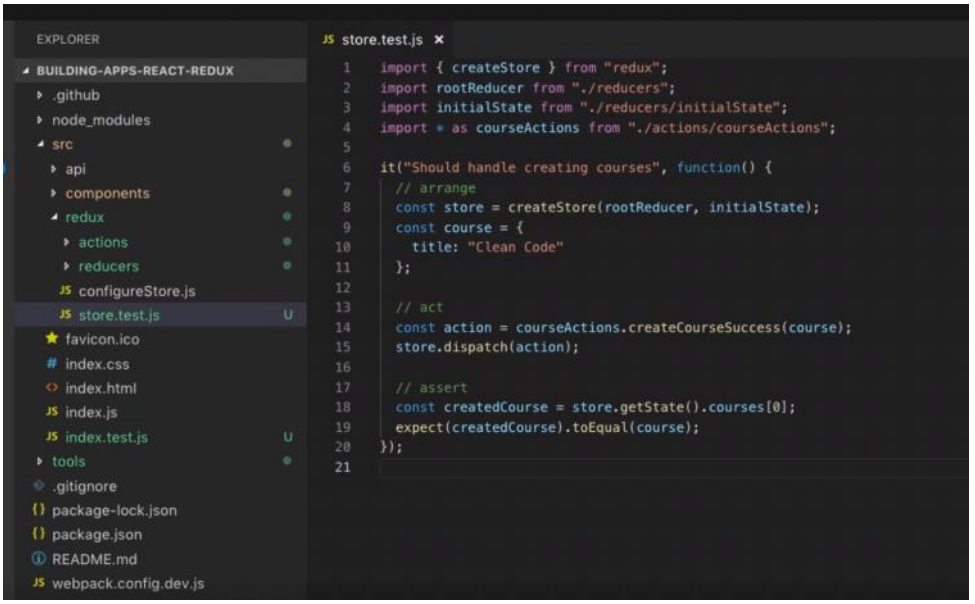
- **Testing Reducers**
  - 

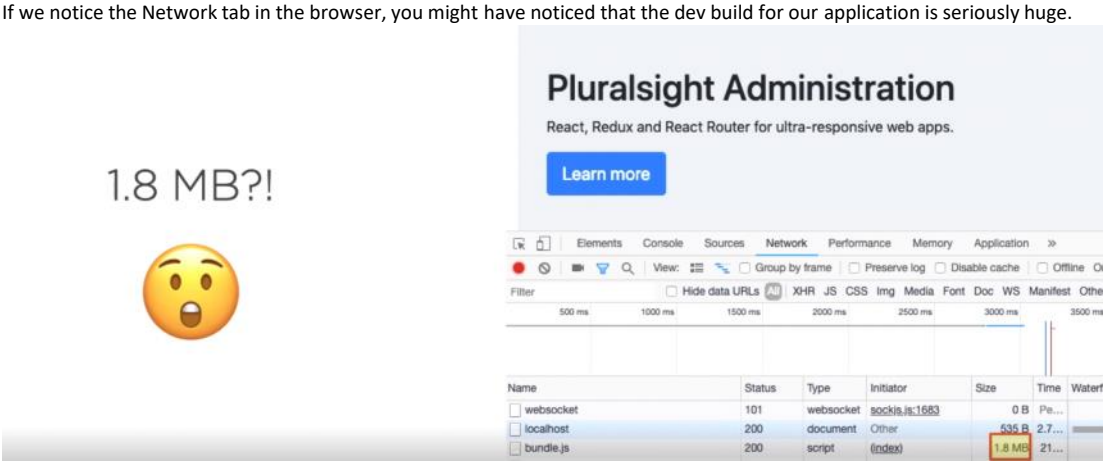
  - 
  - 
  - 

```
it("should update course when passed UPDATE_COURSE_SUCCESS", () => {
  // arrange
  const initialState = [
    { id: 1, title: "A" },
    { id: 2, title: "B" },
    { id: 3, title: "C" }
  ];

  const course = { id: 2, title: "New Title" };
  const action = actions.updateCourseSuccess(course);

  // act
  const newState = courseReducer(initialState, action);
  const updatedCourse = newState.find(a => a.id == course.id);
  const untouchedCourse = newState.find(a => a.id == 1);

  // assert
  expect(updatedCourse.title).toEqual("New Title");
  expect(untouchedCourse.title).toEqual("A");
  expect(newState.length).toEqual(3);
});
```

- **Testing Store**
  - When testing the store, we're really writing an integration test rather than a unit test because our goal is to assure that our actions, the store, and our reducers, are all interacting together as we expected. So we're going to write some tests for the interaction of these three pieces



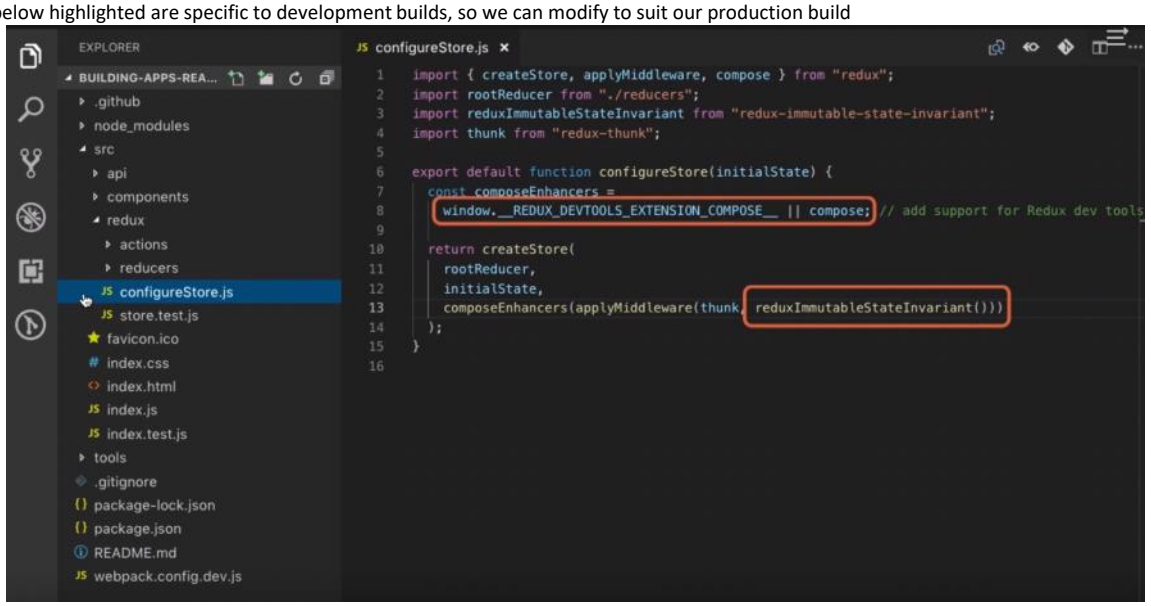
- 

- **Production Builds**
  - If we notice the Network tab in the browser, you might have noticed that the dev build for our application is seriously huge.
  - 

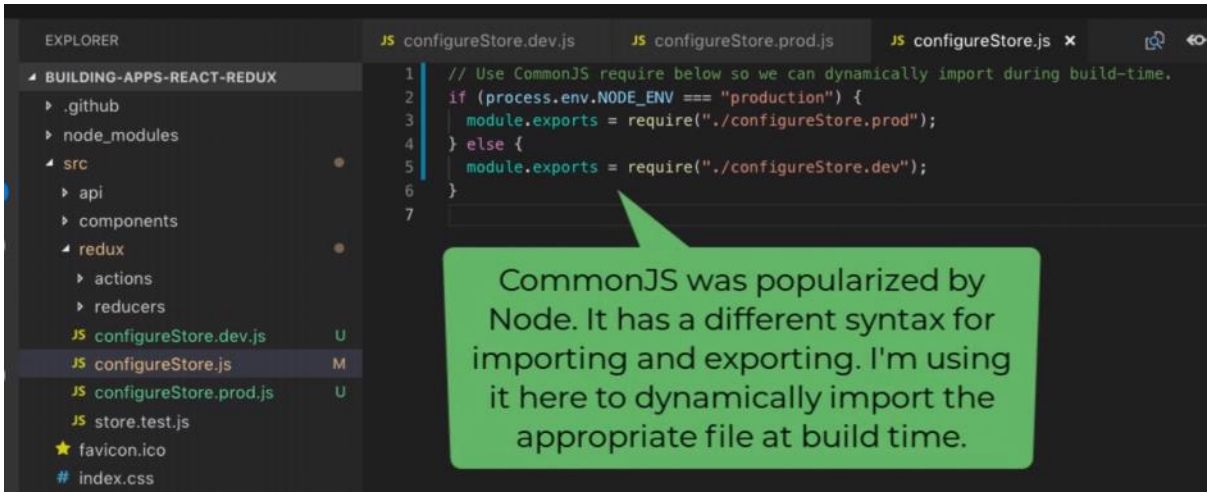
  - 

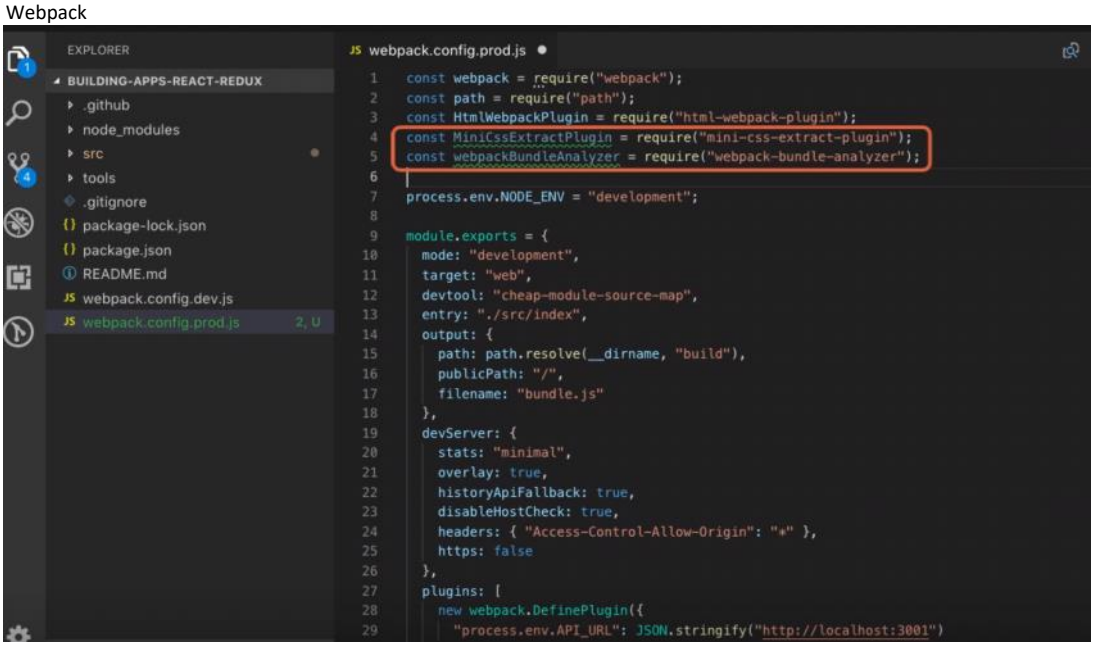
Production build process

- Lint and runs tests
- Bundle and minify JS and CSS
- Generate JS and CSS sourcemaps
- Exclude dev-specific concerns
- Build React in production mode
- Generate bundle report
- Run the build on a local webserver

- The below highlighted are specific to development builds, so we can modify to suit our production build
  - 

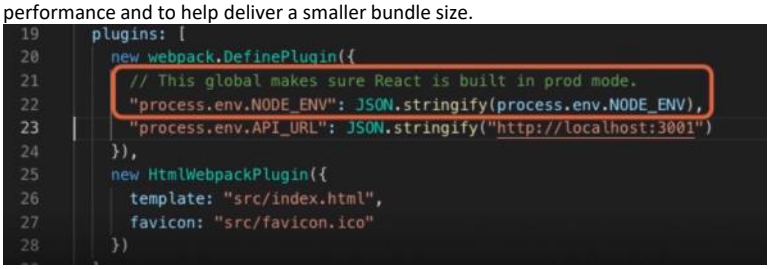
  - We can rename the above file to configureStore.dev.js and create a new one for production build
  - 

- 

- Setup Webpack
  - 

  - We're going to use the MiniCssExtractPlugin, which will minify our CSS and extract it to a separate file.
  - We are also going to add in webpackBundleAnalyzer, which will create a handy report of what's in our bundle.
  - 

  - 

  - We can remove the devServer section.
  - The define plugin lets us define variables that are then made available to the libraries that web pack is building. React looks for this to determine if it should be built in production mode. Production mode omits development-specific features like property types for performance and to help deliver a smaller bundle size.
  - 

  - With webpackBundleAnalyzer configuration, Webpack will automatically display a report of what's in our bundle when the build is completed.

-
```
plugins: [
  // Display bundle stats
  new webpackBundleAnalyzer.BundleAnalyzerPlugin({ analyzerMode: "static" }),
```

- For production, we want to minify our CSS and extract it to a separate file. This plugin is MiniCssExtractPlugin. Webpack will pick the name for us and add a hash to it. This way the filename will only change when our CSS changes. So this supports setting far expires headers on your web server so your users only have to reload this file when it changes.
-
```
plugins: [
  // Display bundle stats
  new webpackBundleAnalyzer.BundleAnalyzerPlugin({ analyzerMode: "static" }),

  new MiniCssExtractPlugin({
    filename: "[name].[contenthash].css"
  }),

  new webpack.DefinePlugin({
    // This global makes sure React is built in prod mode.
```

- HtmlWebPackPlugin, which we've already been using in development. This plugin performs a number of functions. It generates our index.html, and it adds a reference to our JavaScript bundle and CSS bundle into the HTML forums.
- And this is handy, since the JS and CSS filenames will change over time, since they contain hashes so that we can cache them for a long time on our web server.
- For production, let's add a few more settings. With these extra settings, it also minifies our HTML, removes comments, and much more. These are all little performance enhancements to keep our HTML file as small as possible.
-
```
new HtmlWebpackPlugin({
  template: "src/index.html",
  favicon: "src/favicon.ico",
  minify: {
    // see https://github.com/kangax/html-minifier#options-quick-reference
    removeComments: true,
    collapseWhitespace: true,
    removeRedundantAttributes: true,
    useShortDoctype: true,
    removeEmptyAttributes: true,
    removeStyleLinkTypeAttributes: true,
    keepClosingSlash: true,
    minifyJS: true,
    minifyCSS: true,
    minifyURLs: true
  }
```

- For our final change, let's scroll down to our CSS config. I'm going to replace what we have here in our use statement with this.
- MiniCssExtractPlugin will extract our CSS to a separate file using the css-loader that you see here. And it will generate a sourceMap for debugging purposes.
- Down here, we're also using postcss, which can perform a variety of processing on CSS. We're using the cssnano postcss plugin to minify our CSS. Remember that loaders runfrom the bottom up. So the postcss-loader will run, and then the css-loader will take over, generate our sourceMap, and extract to a separate file
-
```
module: {
  rules: [
    {
      test: /\.(js|jsx)$/,
      exclude: /node_modules/,
      use: ["babel-loader", "eslint-loader"]
    },
    {
      test: /(\.css)$/,
      use: [
        MiniCssExtractPlugin.loader,
        {
          loader: "css-loader",
          options: {
            sourceMap: true
          }
        },
        {
          loader: "postcss-loader",
          options: {
            plugins: () => [require("cssnano")],
            sourceMap: true
          }
        }
```

-
```
module: {
  rules: [
    {
      test: /\.(js|jsx)$/,
      exclude: /node_modules/,
      use: ["babel-loader", "eslint-loader"]
    },
    {
      test: /(\.css)$/,
      use: [
        MiniCssExtractPlugin.loader,
        {
          loader: "css-loader",
          options: {
            sourceMap: true
          }
        },
        {
          loader: "postcss-loader",
          options: {
            plugins: () => [require("cssnano")],
```

Loaders run from the bottom up. So the postcss-loader will run first.

- **Set up npm scripts**
  - **build** - This script will run Webpack and pass the config of webpack.config.prod.js. This script will write to the build folder.
  - **clean:build** - So before we run the production build, it's a good idea to delete the previous build folder. This script we'll call the rimraf package, which is a cross-platform-friendly way to delete files, and we'll delete the build folder, and then we'll recreate the build folder using the mkdir command, which runs on all operating systems.
  - **test:ci** - It's also a good idea to run our tests before the production build, but our current test script runs our tests in watch mode. We only need our test to run one time before the build to assure that they passed, so let's create a dedicated script that runs the test just once. Ci stands for continuous integration because this would also likely be a script you'd run on a continuous integration server. This calls jest without watch mode enabled, since we only want to run the test one time when we build. We'll use this to assure that we can't deploy an app with broken tests.
  - **prebuild** - This script will use run-p to run multiple npm scripts in parallel. Run-p is a command that comes with the npm run all package. We want to run the clean:build script, as well as the test.ci script before we run our build. So this will run both of these scripts at the same time, and if our tests fail, then the build will fail.
  - **serve:build** - It would also be handy to be able to run our production build locally to assure that the build runs properly. This will use a lightweight web server called http-server to serve what we write to the build folder.
  - **postbuild** - For convenience, we can run this server after our build is complete. Like our prebuild script, this will run after the build script by convention because it has the same name, but with post on the front. We'll use run-p again to run a few scripts at the same time. We want to start our API and also serve our build.
  -
```
{} package.json ×
{
  "name": "ps-redux",
  "description": "React and Redux Pluralsight course by Cory House",
  "scripts": {
    "start": "run-p start:dev start:api",
    "start:dev": "webpack-dev-server --config webpack.config.dev.js --port 3000",
    "prestart:api": "node tools/createMockDb.js",
    "start:api": "node tools/apiServer.js",
    "test": "jest --watch",
    "test:ci": "jest",
    "clean:build": "rimraf ./build && mkdir build",
    "prebuild": "run-p clean:build test:ci",
    "build": "webpack --config webpack.config.prod.js",
    "postbuild": "run-p start:api serve:build",
    "serve:build": "http-server ./build"
  },
```

Transcripts

Thursday, August 29, 2019

Production Builds

Intro

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

Production Build Plan Overview

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

Set up Production Redux Store

In this final module, we're going to prepare our application for production by creating an automated build that does the following. Runs our tests and lints our code, bundles and minifies our JavaScript and our CSS, generates source maps for both, so that we can debug any production issues, excludes dev-specific concerns, such as dev-only Redux store configurations, it should build React inproduction mode so that dev-specific features like property-type validation are eliminated for optimal performance, and the build will generate a bundle report so that we can see our app size and clearly see what packages are in our app's build. Finally, it will run our production build locally using a local webserver. This sounds like a lot of work, but you'll be surprised how little code we write. Let'smake it happen. To begin preparing our app for production, let's configure our Redux store for production. Our current store contains code that we don't want to run in production, such as reduxImmutableStateInvariant, and the reduxDevTools configuration. So let'screate a separate Redux store configuration for development and production. To begin, let's rename our existing configureStore toconfigureStore.dev .js. So this will remain our development configuration. Now let's copy the contents of this file and create a new file in the same folder called configureStore.prod .js. And we'll paste in the development configuration as our starting point. Our production configuration will not need Redux dev tools, so we'll take that out. This means we also no longer have thecomposeEnhancers function, so we can remove that. This means we can also jump to the top and remove the compose import, as well as the import for reduxImmutableStateInvariant. And now the only middleware that we want to apply will be Redux thunk. So our production store configuration is much simpler. Now we just need to add a little code to call the proper store configuration based on our environment. So, let's create another file in this same directory called configureStore.js. For this file, I'm going to use CommonJS so that we can dynamically import during build time. We're going to look at the Node environment to determine whether to load the production configuration or the dev configuration. This pattern assures that our dev-related store configuration code isn't included in our production bundle. Now you might be wondering where the Node environment is configured. We'll take care of that in the next clip as we configure Web Pack for production.

Set up Webpack

We already have a development webpack.config, and our production webpack.config will be quite similar, but it will differ in some key ways. Let's begin by copying our existing webpack.config, and then pasting it out here at the root. But I will rename it to webpack.config .prod .js. To begin, we need to

import two more items at the top. We're going to use the MiniCssExtractPlugin, which will minify our CSS and extract it to a separate file. I'm also going to add in webpackBundleAnalyzer, which will create a handy report of what's in our bundle. Down here on line 7, we want to set the Node environment to production. Remember the code that we wrote in the previous clip is looking for this setting. On line 10, let's set the mode to production as well. This configures web pack with some useful defaults for production builds. For production, we'll use the recommended production source map, which is source-map. This is a little slower to create than the source map setting that we were using for development, but it's higher quality, so for production builds, it's worth the wait. Our entry point will remain the same, and our output settings will also remain the same, but unlike the dev config, web pack will actually write physical files to the build directory since we're in production mode. We don't need this dev server section for production, so we can remove all of that. Now let's tweak our plugin configuration. Remember, plugins enhance web pack's power. Under the define plugin, I'm going to add one more setting. The define plugin lets us define variables that are then made available to the libraries that web pack is building. React looks for this to determine if it should be built in production mode. Production mode omits development-specific features like property types for performance and to help deliver a smaller bundle size. There's a few other plugins that I'd like to configure for our production build. We'll add them up here at the top. First, let's add webpackBundleAnalyzer. This is already imported up above. With this configuration, Webpack will automatically display a report of what's in our bundle when the build is completed. For production, we want to minify our CSS and extract it to a separate file. So let'sadd the MiniCssExtractPlugin. Webpack will pick the name for us and add a hash to it. This way the filename will only change when our CSS changes. So this supports setting far expires headers on your web server so your users only have to reload this file when it changes. The last plugin to configure is HtmlWebPackPlugin, which we've already been using in development. This plugin performs a number of functions. It generates our index.html, and it adds a reference to our JavaScript bundle and CSS bundle into the HTML forus. And this is handy, since the JS and CSS filenames will change over time, since they contain hashes so that we can cache them for a long time on our web server. For production, let's add a few more settings. With these extra settings, it also minifies our HTML, removes comments, and much more. These are all little performance enhancements to keep our HTML file as small as possible. For our final change, let's scroll down to our CSS config. I'm going to replace what we have here in our use statement with this. It's admittedly clunky to type, so I'm going to paste it in and then talk through. MiniCssExtractPlugin will extract our CSS to a separate file using the css-loader that you see here. And it will generate a sourceMap for debugging purposes. Down here, we're also using postcss, which can perform a variety of processing on CSS. We're using the cssnano postcss plugin to minify our CSS. Remember that loaders runfrom the bottom up. So the postcss-loader will run, and then the css-loader will take over, generate our sourceMap, and extract to a separate file. And that's it. Now that we have this config ready, of course we need to call it. Sounds like a job for an npm script. Let's do that next.

Set up npm Scripts

To handle our production build, we need a few more npm scripts. First, let's add a script called build. This script will run Webpack and pass the config of webpack.config .prod .js. Add my comma on the previous line. This script will write to the build folder, so before we run the production build, it's a good idea to delete the previous build folder. So, let's add a script for that. We'll call it clean:build. This script we'll call the rimraf package, which is a cross-platform-friendly way to delete files, and we'll delete the build folder, and then we'll recreate the build folder using the mkdir command, which runs on all operating systems. It's also a good idea to run our tests before the production build, but our current test script runs our tests in watch mode. We only need our test to run one time before the build to assure that they passed, so let's create a dedicated script that runs the test just once. We'll call it test:ci. Ci stands forcontinuous integration because this would also likely be a script you'd run on a continuous integration server. This calls jest without watch mode enabled, since we only want to run the test one time when we build. We'll use this to assure that we can't deploy an app with broken tests. Now let's create a script called prebuild. I'm going to place it up above the build script, although it's not required to sit here. I just like to put prescripts before the related script. This script will use run-p to run multiple npm scripts in parallel. Run-p is a command that comes with the npm run all package. We want

to run the clean:build script, as well as the test.ci script before we run our build. So this will run both of these scripts at the same time, and if our tests fail, then the build will fail. We're almost done. It would also be handy to be able to run our production build locally to assure that the build runs properly. So, let's add a script that will serve our production build. We'll call it serve:build, and this will use a lightweight web server called http-server to serve what we write to thebuild folder. For convenience, we can run this server after our build is complete. We can do that by creating a script called postbuild. Like our prebuild script, this will run after the build script by convention because it has the same name, but with post on the front. We'll use run-p again to run a few scripts at the same time. We want to start our API and also serve our build. Of course, for a real production app, you'd want to hit a different API. Now that we have our script set up, let's give it a shot in the next clip.

Run Production Build and Review Bundle Content

Okay, hit Save on your package.json and cross your fingers because it's time to see if this thing bursts into flames. Say npm run build. We can see all our tests run, and they all pass successfully.

Then Webpack begins running our build. Running Webpack could take a moment, because remember it's minifying our code, which is a slow process that we only need to do for production. When our build is done, a tab opens automatically that contains our report from webpack-bundle-analyzer. Bundle-analyzer shows what's in our bundle. We can see that we're running the production version of React. The size of these boxes represents the size of the file. And for the big moment of truth, take your mouse and hover over bundle.js. Now we can see that our production build is only 64 KB. That's a huge improvement! Now that we're properly bundling and minifying our code, our app is only 63K g-zipped. That's about the size of this cat JPEG! Crazy! This lightweight result is the clear benefit of choosing focused libraries like React, React Router, and Redux. Nice work! And since each box represents its relative size in the bundle, we can see that React is about half of our bundle. We can also see thatRedux is quite a small dependency, only 2.2 K g-zipped. In fact, it's smaller than the react-toastify library that we're using to show toast, which is 5.6 K g-zipped. Over here on the right is the source code that we've written, which totals up to about 12 K. And as a sidenote, you could further improve the bundle size by code splitting your app. Check out React.lazy for how to do that.

Run Production App Locally

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

Final Challenges

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

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

From <https://app.pluralsight.com/library/courses/react-redux-react-router-es6/transcript>