COLLEGIO TECHNOLOGIES INC.

REACT, REDUX AND AJAX

REACT AND REDUX: THE PERFECT MATCH!

- Now that we have some experience with setting up a Redux store, as well as dispatching actions and creating reducers to handle dispatched actions, we need to see how we can connect it with our React apps.

- Also, now that our data layer is essentially separate from our application logic, we can concentrate any AJAX calls to outside servers into here, instead of all over your app!

SETTING UP REDUX WITH REACT

- As is the case with just about everything we've added to our React project, you're going to want to organize your Redux functions so that you can have easier access and separation of functionality.

- It is recommended that you have separate folders in which to store your Redux reducers, actions, and selectors, and stores.

- Once you have everything broken out, you will need to install a library called **react-redux**
  - **npm install react-redux**

- The react-redux library makes extensive use of what are known as **Higher Order Components**.

HIGHER ORDER COMPONENTS

- A ***Higher Order Component*** is a component that renders another component.
- The primary advantages for using a Higher Order Components are:
  - Code Reuse
  - Render Hijacking
  - Prop Manipulation
  - Abstraction of the State

```
import React from 'react';
import ReactDOM from 'react-dom';

const NameInfo = (props) => (
    <div>
        <h1>Hey There!</h1>
        <p>Your name is: {props.yourName}</p>
    </div>
);

const withShareWarning = (WrappedComponent) => {
    return (props) => (
        <div>
            <p>Sharing your name on the internet may lead to undesirable results!</p>
            <WrappedComponent {...props} />
        </div>
    );
}

const AdminNameInfo = withShareWarning(NameInfo);

ReactDOM.render(<AdminNameInfo yourName="Rob" />,  document.getElementById('app'));
```

SETTING UP REACT AND REDUX

- Inside of the react-redux library, we will essentially only be using two functions:
  - The **Provider** component
  - The connect function

- The provider component will be declared once when we are setting up our store in Redux, and the connect functions will be used for each component that needs to connect to the store.

THE PROVIDER COMPONENT

- As mentioned, the Provider component will be declared inside of your app's entry point. If you are following along with the course code, this will be the **/src/app.js** file.

- The Provider component is a Higher Order Component that will provide your Redux store to the entire app.

- Once it is imported into your project, you simply create the Provider Component, add your Redux store as a prop, and include your Wrapper Component, and you will have access to your Redux store throughout your app!

```
import { Provider } from 'react-redux';
import configStore from './store/configStore';

// create the Redux Store (using our imported store)
const store = configStore();

const appTemplate = (
    <Provider store={store}>
        <MainRouter />
    </Provider>
);

ReactDOM.render(appTemplate, document.getElementById('app'));
```

THE CONNECT FUNCTION

- The second part to using the React-Redux library is the **connect()** function.
  - This is the function that we use to specify which specific data we should have access to inside of our component.

- After importing the connect function from the react-redux library, we will be using the connect() function to pass in our mapped state variables, and from the resulting function, we will pass in our component.
  - This sounds a little convoluted, but after several examples, it should become clear.

```
import React from 'react';
import { connect } from 'react-redux';

const PlayersListPage = (props) => (
    <div className="container">
        <h1>Players List</h1>
        {props.players.length}
    </div>
);

// we use the connect function to map our state, then call our component!
const ConnectedPlayersListPage = connect((state) => {
    return {
        players: state.players
    }
})(PlayersListPage);

export default ConnectedPlayersListPage;
```

THE CONNECT FUNCTION (CONTINUED)

- Furthermore, we can simplify our connect code so that it is even more concise:
  - We don't need to declare **ConnectedPlayersListPage**, as we can just export this as the default.
  - Typically, the state that is passed into the component as a props is abstracted out as well (typically called **mapStateToProps**).

```
import React from 'react';
import { connect } from 'react-redux';

const PlayersListPage = (props) => (
    <div className="container">
        <h1>Players List</h1>
        {props.players.length}
    </div>
);

const mapStateToProps = (state) => {
    return {
        players: state.players
    }
}
export default connect(mapStateToProps)(PlayersListPage);
```

INTERLUDE: INSTALLING REACT DEVELOPER TOOLS

- Before we move on, we should download a third party extension for our browser called **React Developer Tools**.
  - Chrome: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en
  - Firefox: https://addons.mozilla.org/en-US/firefox/addon/react-devtools/
  - Any other browser: You're on your own! ;)
  - **Note:** You may need to restart your browser once it is installed.
- React Dev Tools are a useful extension to your developer tools that will allow you to see what your code looks like from a component perspective, and let's you see data about each specific component, such as props and state values.

THE CONNECT FUNCTION AND DISPATCHING ACTIONS

- In addition to any state values passed down via **mapStateToProps()**, if you use React Dev Tools you will notice that you also have access to the **dispatch()** function for dispatching actions.
  - This is a feature of **connect()**, and allows you to be able to make updates to any necessary state values.

- Please note, if you are using a text field to input any prop values, you will also need to declare an **onChange()** event to update the values. If you do not include an **onChange()** event, the field will be read-only.

EXAMPLE: UPDATING STATE USING INPUTS AND ONCHANGE()

```
import React from 'react';
import { connect } from 'react-redux';
import { setFilterText } from '../actions/filters';

const PlayersListFilters = (props) => (
    <div>
        <input type="text" value={props.filters.text} onChange={(e) => {
            props.dispatch(setFilterText(e.target.value));
        }} />
    </div>
);

const mapStateToProps = (state) => {
    return {
        filters: state.filters
    };
};

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

USING CONNECT() WITHOUT PASSING STATE

- Sometimes, you may only need to use the **dispatch()** function within a component, and have no need to declare state.
  - This is perfectly fine, and is common practice in React. When you export your component, simply export it and leave the first set of arguments blank.

```
import React from 'react';
import { connect } from 'react-redux';
import { removePlayer } from '../actions/players';

const PlayersListItem = ({ dispatch, id, name, sport_type, skill_level, gender, message }) => (
    <div>
        <h3>{name} ({gender})</h3>
        <h4>{skill_level}</h4>
        <p>{message}</p>
        <button onClick={() => {
            dispatch(removePlayer({ id }));
        }}>Remove</button>
    </div>
);

export default connect()(PlayersListItem);
```

EXERCISE: SETTING UP FILTERING AND ADDING NEW PLAYERS

- Now that we know how we can connect up some info, let's try to set up our app to display, filter and add new players using Redux.

- Use **props.history.push('/')** to redirect your page to the homepage using react-router. You can change the value to redirect to other internal locations too!

- ***Side Note:*** We don't cover any date-related functionality in this app, but it is fairly common that you'll want to set up a date-picker to use inside of your app. AirBnB has developed an open-source React date picker that is great for serving this purpose:
  - ***https://github.com/airbnb/react-dates***
  - In addition, you may want to use moment.js to configure your date info if you don't already: ***http://momentjs.com/***

SYNCHRONOUS AND ASYNCHRONOUS ACTION GENERATORS

- So far, whenever we've worked with redux, any actions that we've generated have been synchronous actions; that is, they are performed, and then a value is returned, but the system waits until it is returned.

- However, in order to use external RESTful calls, which are asynchronous and return promises, we need to have a way in which to handle whatever is eventually sent back from the action generator call.

- In order to handle asynchronous actions in redux, we're going to use another library, called **Redux Thunk**.
  - **yarn add redux-thunk**

REDUX THUNK

- Redux Thunk allows us to bend the rules of action generators by allowing us to return functions from our action generators, which by default should only return actions.
  - The returned function will then be called with **dispatch()**, allowing us to make calls to other actions.
- Redux Thunk is what's known as **middleware** for redux, and in order to use it, we'll also need to include the **applyMiddleware** function from redux.
  - **applyMiddleware** is used as the third argument inside of the **createStore** function.

```
const store = createStore(
    combineReducers({
        players: playersReducer,
        filters: filtersReducer,
        selectedPlayer: selectedPlayerReducer,
        auth: authReducer
    }),
    {},
    applyMiddleware(ReduxThunk)
);
```

REDUX THUNK

- Once you have Redux Thunk set up in your project, you can use it within your action generators by returning a function instead of an object containing an action. The function will contain a **dispatch** argument, that is what you will use for calling other actions.
    - The **dispatch** function should be used within the **.then()** or **.catch()** functions returned from the promise.

- One very cool feature of using Redux Thunk: we're not just limited to one dispatch action, and in fact can perform multiple dispatches from the same function!

```
export const loginUser = ({ email, password }) => {
    return (dispatch) => {
        authLoading(dispatch);

        firebase.auth().signInWithEmailAndPassword(email, password)
            .then((user) => {
                loginUserSuccess(dispatch, user);
            })
            .catch(() => {
                firebase.auth().createUserWithEmailAndPassword(email, password)
                    .then((user) => {
                        loginUserSuccess(dispatch, user);
                    })
                    .catch((err) => {
                        loginUserError(dispatch, err.message);
                    })
            });
        };
};
```

HANDLING ERRORS INSIDE OF YOUR REDUX THUNK ACTIONS

- Remember, when we are working with Redux Thunk, we are returning a function instead of an object.
    - If your function contains a promise, and the promise code happens to create an error, there is a chance that the error can happen silently.
        - This is because, if you have a catch function attached to your promise, any errors with the AJAX call **or** the associated callback will default to the **.catch()** function.

- If you are custom handling your error, adding a **console.log** function call during development to print the error can provide you with better details of the error context.

REDUX THUNK CONCLUSION

- We'll be using Redux Thunk moving forward to perform any asynchronous calls. By using it, we can pass functions instead of objects.
  - This is something that you'll be using often when developing React apps (how many apps do you know of that don't reach out to an external server?)

- Finally, now that we know how to handle async API calls, how can we actually make the calls?

RESTFUL HTTP CALLS WITH AXIOS

- Finally, we'll take a look at how we can work with external data via RESTful HTTP calls using a third party library known as ***axios***.
  - **yarn add axios**
  - **import axios from 'axios';**

- Once you have the axios library imported, you can make GET, POST, and DELETE calls using **axios.get()**, **axios.push()** and **axios.delete()**
  - These functions return promises, which are handled by using the **.then()** function, which contains the returned response information from the API call. We'll be interested in **response.data** to see any returned values.

```
axios.get('http://your-data-url/')
    .then((res) => {
        res.data.players.forEach((player) => {
            store.dispatch(addPlayer(player));
        });
    });
```

CONCLUSION: REACT

- That's it! You now have all of the basic building blocks that you need to build an interactive React application!
- In our final lesson, we'll be going over making sure that your code will always work, even when updates are made to your app, by adding a testing suite to your React application.
- We will also be going over how you can set up a production copy of your app, and release it on a web server!

RESOURCES

- **Lesson Video:** *https://youtu.be/MRZh33MUN3Q*
- **Lesson Code:** https://github.com/collegio/intro_react_redux2
- **Completed Lesson Code:** https://github.com/collegio/intro_react_redux2_completed
- **React Developer Tools:**
  - **Chrome -** https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en
  - **Firefox -** https://addons.mozilla.org/en-US/firefox/addon/react-devtools/
- **Redux Thunk:**
  - **Official Site:** https://github.com/gaearon/redux-thunk
- **Axios:**
  - **Official Site -** https://github.com/axios/axios
  - **alligator.io Tutorial on Axios -** https://alligator.io/react/axios-react/
- **Date Picker and Time Display:**
  - **AirBnB Date Picker -** https://github.com/airbnb/react-dates
  - **Moment.js -** http://momentjs.com/

DISCLAIMER

- Copyright 2021 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.