Real world Redux - Sample chapter

Jakob Lind

Chapter 2 - Connecting to React

Understanding the ins and outs of connecting to React is one of the trickiest part of Redux.

There is a lot of new stuff to learn at the same time. You both need to understand `mapStateToProps`, `mapDispatchToProps` and `connect`. You also need to learn about new concepts such as higher-order components and presentational/container pattern.

You will learn about most of it in this chapter. Learning this takes some time. You should revisit this chapter after some time because it's difficult to grasp everything in one read.

Recap of how to use connect function

In the previous chapter you learned how to use it. First you wrapped the root component in a `Provider` component that gets the store as props.

```
import { Provider, connect } from "react-redux";

// ... create store here

ReactDOM.render(
  <Provider store={store}>
    <AppContainer />
  </Provider>,
  document.getElementById("root")
);
```

And then you use `connect` function to give your React component access to the Redux store and dispatching Redux actions.

```
// ...create an App component

const mapStateToProps = state => {
  return { counter: state.counter };
};
const mapDispatchToProps = {};
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

You have probably still a bunch of unanswered questions that we will clear out in this chapter.

The mysterious double parenthesis ()()

When you looked at the example of `connect` the first time you probably thought to yourself: "hm... wtf?".

```
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

The first part looks familiar, it calls a function `connect` and passes in two variables. But then... another set of parenthesis? What does that even mean?

In JavaScript, a function can return *another* function. Look at this simple example:

```
function getFunction() {
  return function() {
    console.log("hello world");
  };
}
```

When you call `getFunction` all that happens is that another function is returned. You can assign the returned function to a variable:

```
const helloWorldFunction = getFunction();
```

You can now call `helloWorldFunction` just like any other function:

```
helloWorldFunction();
```

And it would print out "hello world".

Back to `connect`.

```
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

When `connect(..)` is called it returns *another* function that is called immediately by the second set of parameters.

This is another way to write the function call that is more explicit:

```
const returnedFunction = connect(mapStateToProps, mapDispatchToProps);
returnedFunction(App)
```

Why did they design it like this? Wouldn't it be easier just to put all arguments in the first function?

One reason could be to make it possible to reuse one `connect` for several components. Something like this:

```
const connected = connect(
  mapStateToProps,
  mapDispatchToProps
);

const IndexPageContainer = connected(
  IndexPage
);
const ProfilePageContainer = connected(
  ProfilePage
);
```

In my experience it's not very common to write code like this because most times each component need their own `mapStateToProps` and

`mapDispatchToProps`. But it's good to know that it's possible. This is an example of Redux high flexibility.

The mapStateToProps function

`mapStateToProps` is passed into `connect` function as you just saw:

```
const AppContainer = connect(
  mapStateToProps, // <--- This
  mapDispatchToProps
)(App);
```

This is how `mapStateToProps` look like from the previous chapter:

```
const mapStateToProps = state => {
  return { counter: state.counter }
}
```

It's a function that takes `state` as argument and returns an object with a `counter`.

The purpose of the `mapStateToProps` function is to give the component access to the Redux state via props.

`mapStateToProps` is automatically called by Redux internals every time your state changes. It gets the new Redux state as input parameter and it returns an object that is passed into the connected component as props.

In this example, `counter` will be passed into the connected component and can be accessed from the component as `this.props.counter`.

```
class App extends React.Component {
  render() {
    return (
      <div>
        State from Redux:{" "}
        {this.props.counter}
      </div>
    );
  }
}
```

> **Don't forget** Your React components only gets Redux state from `this.props` - you never need to put any Redux state in `this.state`.

mapStateToProps decouples React from Redux

`mapStateToProps` is a function that takes in the Redux state and converts it to a format that is easy for your React components to work with. This means that the structure of the state that your React components gets is not the same structure as the Redux state.

Now, why would you need this function? Why not pass in the whole Redux state into your React components?

Let's look at a real-world example.

Let's say you have a `ProfilePicture` component that displays a profile picture.

If you don't use `connect`, then you would pass in the whole store to

this component

```
let store = createStore(
  reducer,
  initialState
);
// ...
<ProfilePicture store={store} />;
```

Now your `ProfilePicture` component must know about the structure of the store. It must know how to fetch the profile picture URL from that store (for example `store.user.profilePicture.url`).

```
const ProfilePicture = ({ store }) => (
  <img
    src={
      store
        .getState()
        .user
        .profilePicture
        .url
    }
  />
);
```

Your components should only be concerned about displaying stuff. Now it also has to be concerned about how to find data in the state.

The bad thing about this solution is that you cannot reuse the component in other places of your application. If you want to display the profile picture of something that is not a user, for example an event, you would need to create a new component for that because the path to the event picture is different than the path to the user picture.

`mapStateToProps` to the rescue!

Now let's implement `ProfilePic` as a component that only does presentation of data. You will use the connect function with `mapStateToProps` function.

This time it's the `mapStateToProps` function that would know where to find the profile picture. And it would send only the URL to the `ProfilePicture` component.

```
const ProfilePicture = ({ imgUrl }) => (
  <img src={imgUrl} />
);

const mapStateToProps = state => {
  return {
    imgUrl: state.user.profilePicture.url
  };
};

connect(mapStateToProps)(ProfilePicture);
```

The `ProfilePicture` component now only accepts `imgUrl` as props, and you can also use it in other parts of your application:

```
<ProfilePicture imgUrl={eventImgUrl}/>
```

The Redux code is decoupled from your React components.

The mapDispatchToProps function

The `mapDispatchToProps` function is passed into `connect` together with `mapStateToProps` that you just learned about.

```
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps // <--- This
)(App);
```

Instead of mapping state to the component like `mapStateToProps` does, `mapDispatchToProps` maps actions to the component. The component can access the actions via props.

Let's revisit the implementation from earlier:

```
const mapDispatchToProps = dispatch => {
  return {
    increaseCounter: () => {
      dispatch({
        type: "INCREASE_COUNTER"
      })
    }
  }
}
```

It's a function that gets `dispatch` as input parameter. It returns an object where the keys are mapped as props to the wrapping component. In this case it will give the component `increaseCounter` as props. `dispatch` is a function that dispatches actions to your reducer.

This results in that you can dispatch Redux actions from your React component by calling the function `this.props.increaseCounter()` like this:

```
class App extends React.Component {
  render() {
    return (
      <div>
```

```
        State from Redux:{" "}
        {this.props.counter}
        <button
          onClick={() =>
            this.props.increaseCounter()
          }
        >
          increase
        </button>
      </div>
    );
  }
}
```

So everytime you press the button it will call the `props.increaseCounter` which then will call `dispatch({ type: "INCREASE_COUNTER"})`. `dispatch` is a function in the store you created earlier with `createStore`. The store is passed to your component via `Provider` component where passed the store as props.

Presentational and container components

If you have read about Redux before you have probably heard about presentational/container component pattern. It have probably made you confused about what it is and when to use it.

You have actually already coded one presentational and one container component.

When calling the `connect` method you get a new component back.

```
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

The component you get back (that we call `AppContainer`) gets stuff from the Redux store and injects it to the `App` component. *This component is a container component.* All components that are returned from `connect` are container components.

All other React components you have written are presentational components. Presentational components are components that has the purpuse to describe how things should look.

This is an overview from the official Redux docs:

	Presentational components	Container components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

You don't have to worry about presentational/container components in the early stages of your application development process. At the start, you should only focus on writing regular React components and get things to work.

If you start planning for a presentational/container architecture before you have written any code, there is a big chance that one of the following will happen:

- You get confused and get stuck
- You will write unnecessarily complicated code because you will write presentation/container components that are not really needed.

When your app grows you might want to look into refactoring it to use more container components. That means using `connect` in your app in more places than the root.

We will continue the discussion on presentational/container components in the Patterns chapter.

Connecting to React in a larger app (many connects)

So far we have just looked at a simple example where our complete app consists of one component.

Real apps are not like that. They are deep heirarchies of complex React components.

If you only connect at the root (`App` component) like we have done so far in the book, then you must pass down props deep in the heirarchy to pass data from the Redux store to all components. Look at this example where we want to display the username inside a box up in the header:

```
const LoginArea = ({ username }) => (
  <div>
    welcome {username}{" "}
    <a href="/logout">Logout?</a>
  </div>
);

const Header = ({ username }) => (
  <LoginArea username={username} />
);

class App extends React.Component {
  render() {
    return (
      <div>
        <Header
          username={this.props.username}
        />
      </div>
    );
  }
}

const mapStateToProps = state => {
  return { username: state.user.username };
};

const mapDispatchToProps = //...
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

Now we need to pass `username` to the `Header` component even though that component doesn't use `username` for anything. All it does is that it passes it down to the `LoginArea`. This is called "prop drilling".

Prop drilling is a feature of React and is good in many cases. But as

the application grows you might drill through so many layers that the code gets difficult to work with.

There is a way to avoid prop drilling. It's possible to `connect` to any React component. We can connect directly to the `LoginArea`:

```
const LoginArea = ({ username }) => (
  <div>
    welcome {username}{" "}
    <a href="/logout">Logout?</a>
  </div>
);

const mapStateToProps = state => {
  return { username: state.user.username };
};

const LoginAreaContainer = connect(
  mapStateToProps
)(LoginArea);


const Header = () => (
  <LoginAreaContainer />
);

/// .. implemenation of App etc..
```

Then you avoid passing props deep down in the heirarchy!

Now you might think to yourself "So should I connect to every component that wants to use data from the Redux state and never pass down props?". No that is most likely not the best idea. Sometime it makes sense to pass down props, sometimes it does not. There is a balance there. It's the part of the job as a developer to find the balance that

makes the code most readable and maintainable.

A good idea is to start with one connect on the root component. When the code base grows you extend with more connects. You add connect when you see a clear need for it.

A deep dive into the implementation of connect

When I was learning about Redux, the `connect` function felt like magic to me. How can a call to `dispatch` in `mapDispatchToProps` trigger a call in the reducer?

When I looked at the implementation of `connect` it all clicked. Now it's time to do the same for you and hopefully you it will all click for you to!

You don't need to code along to this part. The main take away of this section is understanding how `connect` really works under the hood.

So far you have learned that `connect` returns another function:

```
function connect(
  mapStateToProps,
  mapDispatchToProps
) {
  return function(WrappedComponent) {
    //.. do something here
  };
}
```

How does the rest of the implementation look? Let's implement it step-by-step.

If we take a look at how the connect function is used, you can see that it's a function that takes the React component as input, and outputs a new React component.

```
const AppContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(App);
```

It takes the `App` component as input and returns a new component called `AppContainer`.

Let's make the implementation return a new component that wraps the `WrappedComponent`:

```
function connect(
  mapStateToProps,
  mapDispatchToProps
) {
  return function(WrappedComponent) {
    //we return a Wrapper component:
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
          />
        );
      }
    };
  };
}
```

So now your function returns the exact component you sent as input (here named `WrappedComponent`) with an empty wrapper component around it.

Now we want to pass down props containing data from the store to `WrappedComponent`. You will do it like this:

1. Get the state from the store with `store.getState()`
2. Call the function `mapStateToProps` that has been passed in
3. Set the returned data as props in the newly created component

It looks like this:

```
function connect(
  mapStateToProps,
  mapDispatchToProps
) {
  return function(WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(
              store.getState(),
              this.props
            )}
          />
        );
      }
    };
  };
}
```

So you have a function which takes a React component as input and returns a new enhanced React component. This actually has a fancy name: Higher order component (HOC). It's a pattern that is commonly used in advanced React applications and React libraries because it allows you to elegantly reuse logic in React components. Read more

about this in the patterns chapter

You also want your React component to be able to call actions with the dispatcher. To do that, you pass down the actions from the `mapDispatchToProps` function as props to your components in a similar way that you did in the previous step.

```
function connect(
  mapStateToProps,
  mapDispatchToProps
) {
  return function(WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(
              store.getState(),
              this.props
            )}
            {...mapDispatchToProps(
              store.dispatch,
              this.props
            )}
          />
        );
      }
    };
  };
}
```

Did you notice that we access `store` but it's not defined anywhere?

To be able to use `connect` you must wrap your root component in a `Provider` component. The `Provider` injects the store into all your components.

```
ReactDOM.render(
  <Provider store={store}>
    <AppContainer />
  </Provider>,
  document.getElementById("root")
);
```

I'm not going into the details on how `Provider` injects the store to the components. But a hint is that it uses the React context API.

In the implementation of `connect` you now have a way to map the data from the store to props of your components. You can also dispatch actions from your component.

It's just one more thing remaining now. You need to make sure you don't miss any updates from your store by subscribing to it.

```
function connect(
  mapStateToProps,
  mapDispatchToProps
) {
  return function(WrappedComponent) {
    return class extends React.Component {
      render() {
        return (
          <WrappedComponent
            {...this.props}
            {...mapStateToProps(
              store.getState(),
              this.props
            )}
            {...mapDispatchToProps(
              store.dispatch,
              this.props
            )}
          />
```

```
        );
      }
      componentDidMount() {
        this.unsubscribe = store.subscribe(
          this.handleChange.bind(this)
        );
      }
      componentWillUnmount() {
        this.unsubscribe();
      }
      handleChange() {
        this.forceUpdate();
      }
    };
  };
}
```

Now you have subscribed for changes in your store with a callback function `handleChange`. Your callback function calls the React function `forceUpdate` which forces a re-render of your component.

This is more or less the whole implementation of the `connect` function. You never need to implement this yourself in your own project but it's very useful to know how it works under the hood.

Connect gives us optimizations

In the implementation of `connect` we force an update of the React component *every time* there is a change in the Redux store. Even if there is a change that your component doesn't care about.

The real implementation of `connect` is smarter than that. It adds a

bunch of optimizations that makes sure the React components are not rendered when they don't have to.

Thanks for reading this sample

In this sample chapter you have taken a deep dive into the `connect` function.

Hungry for more? In the full book you will learn:

- The Redux basics
- How to structure the format of your Actions
- Reducers that are easy to work with
- File structure for a scalable app
- How to organize complex state
- What code belongs in Redux and what code belongs in React
- Do Ajax the right way with async actions
- Tools and libraries to make you more productive
- Using middlewares to maximize code reuse

Buy the book today!