References:

- How to build an eCommerce Website using React-Redux

REACTDom

`react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an "escape hatch" to get outside of the React model. (Though this is not commonly needed for most components)

So called DOM-specific methods:

1. `ReactDOM.render(element, container[, callback])`
    - This renders a *React element* into the *DOM* int he supplied container and returns a *references* to the component.
    - If *React element* has been previously rendered than it will update it.
    - If (optional) `callback` is provided it will be called after *React element* has been rendered or updated

What are Redux Reducers for?

Important about them:

- They should be designed to make them a *single source of Truth.* (i.e. stored in a single place so that if we need to access that stat anywhere else in in our app then we can do so by a single reference to this source of truth)
- Every state can have their own **reducer**.
- To alleviate a mess we can combine them in one **root reduce**, which ends up defining the store
-

Implementing Actions

Once you have designed your reducers you can jump into implementing actions. *Actions will make sure we make correct calls to our API*

Actions seem to *serve* API calls. Ideally it does so through three defined stages:

1. Loading State
2. Success
3. Failure

They are supposed to provide assurance of correct data flow from the API.

Once of the examples:

```
export function fetchSearchData(args) {
return async (dispatch) => {
```

```
// Initiate loading state
dispatch({
type: FETCH_SEARCH_DATA
});
try {
// Call the API
const result = await fetchSearchData(args.pageCount, args.itemsPerPage);

// Update payload in reducer on success
dispatch({
type: FETCH_SEARCH_SUCCESS,
payload: result,
currentPage: args.pageCount
});
} catch (err) {
// Update error in reducer on failure
dispatch({
type: FETCH_SEARCH_FAILURE,
error: err
});
}
};
}
```

Integrating React with Redux(React-Redux)

`react-redux` is its package

Integration

`react-redux`'s `provider` class: It helps letting allowing the app access all of the states provided by it

```
// src/index.js // //
...
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
const App = <h1>React-Redux eCommerce app</h1>;
ReactDOM.render(
<Provider store={store}>
{ App }
</Provider> ,
document.getElementById(‘root’)
);
```

`Connect` Component

THe `Connect` Component allows us to retrieve data or change it by dispatching an action

Example

```
import React, { Component } from 'react';
import { connect } from 'react-redux'
import fetchSearchData from './action/fetchSearchData';
import SearchData from './SearchData';
const Search = (props) => (
<SearchData
search={props.search}
fetchSearchData={props.fetchSearchData}
/>
);
const mapStateToProps = (state) => ({
search: state.header.search.payload
});
const mapDispatchToProps = {
fetchSearchData
};
export default connect(mapStateToProps, mapDispatchToProps)(Search)
```

You can see in the code above that both `mapStateToProps` and `mapDispatchToProps` are functions that are provided a *state*

Sidenotes

- JavaScript's `export` statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs **`with the import`** statement.