**Step 1: Import Redux NPM packages**

npm install redux react-redux

**Step 2: Create a Reducer**

A reducer is a pure function that accepts 2 parameters: state and action. State can be anything, including objects. Action is an object with type property that specifies the type of action as a string. Let's create a countReducer in src/index.js

**src/index.js**

**const** countReducer = **function** (state = 0, action) {

**switch** (action.type) {

**case** "INCREMENT":

**return** state + 1;

**case** "DECREMENT":

**return** state - 1;

**default**:

**return** state;

}

};

The state is initalized as 0 (when the app starts). We handle 3 types of actions, with the default case returning the original state unchanged. The other two cases modify the value of the state and return it as a "new" state (this is important for [immutability](https://redux.js.org/recipes/structuring-reducers/immutable-update-patterns) when the state is an object).

**Step 2: Create a Redux Store**

The store holds the state as one big global object known as a state tree. The store allows us to:  
- **dispatch** actions to modify the state  
- **subscribe** to recieve notification about state changes  
- **retrieve** the entire state tree  
Let's import and create the redux store and use our reducer to initialize it:

**src/index.js**

...

import { createStore } **from** 'redux';

...

let store = createStore(countReducer);

...

In our case, the entire state tree is just a single reducer. Nothing complex. We will learn how to create more complex state tree in later tutorials.

**Step 3: Wrap the Main App Component with Provider**

Now, we will connect redux to react using the NPM library react-redux. Let's import the <Provider />, and wrap our main app component with it. Also, pass our previously created store into the <Provider />'s store prop.

**src/index.js**

...

import { Provider } **from** 'react-redux';

...

const App = () => (

<Provider store={store}>

<h1>Helloworld React & Redux!</h1>

</Provider>

);

ReactDOM.render(<App />, document.getElementById('root'));

The <Provider/> will supply our entire component tree with the global state tree.

**Step 4: Create and Connect a Container Component**

In the world of React & Redux, *Container (smart) Components* are responsible for pulling state from the Redux store, transforming it, and passing it down to *Presentational (dumb) Components*. Let's convert our <h1> tag into a *Container Component*.

**src/index.js**

...

import { Provider, connect } **from** 'react-redux';

...

const Component = () => <h1>Helloworld React & Redux!</h1>;

**const** Container = connect()(Component);

**const** App = () => (

<Provider store={store}>

<Container />

</Provider>

);

...

We import the connect function from react-redux. We refactor out our original <h1> component into Component. Then, we create a new container component, called Container. The first () invokes the connect function and binds to the Redux store. Then, we invoke the returned function our *Presentational Component* called Component. Now, we've offically connected our Component to Redux, but it doesn't do anything special right now.

**Step 5: Select and Transform State from Redux Store**

Let's use our Container component to select the state and optionally, transform it.

**src/index.js**

...

const mapStateToProps = state => {

**return** {

count: state

};

};

**const** Container = connect(mapStateToProps)(Component);

...

Here, we define a new function called mapStateToProps that literally **maps** or links the state from the Redux store to the component props we wish to pass to our downstream component. In this case, we convert the state (which is just a number from countReducer) to a prop called count. We should do any necessary transformations here.

**Step 6: Use the State in the Presentational Component**

The count prop is now being passed to our Component. Let's declare it as our parameter, and add it to the JSX. The sore responsibility of the *Presentational Component* is to convert props into JSX with little or no logic.

**src/index.js**

**const** Component = ({count}) => <h1>Helloworld React & Redux! {count}</h1>;

Refresh the page and you should now see 0 next to the heading. We are receiving state from Redux store now! But how do we change it? Let take a look at actions.

**Step 7: Add Buttons to our Presentational Component**

Now, we're going to add two buttons in our *Presentational Component* that increment and decrement the count.

**src/index.js**

**const** Component = ({count, handleIncrementClick, handleDecrementClick}) => (

<div>

<h1>Helloworld React & Redux! {count}</h1>

<button onClick={handleDecrementClick}>Decrement</button>

<button onClick={handleIncrementClick}>Increment</button>

</div>

);

Notice that we are passing the two click handlers as props to the two buttons. We will provide these callbacks from the Container for dispatching actions to the Redux store. These buttons are not functional until we do this.

**Step 8: Pass Callback that Dispatch Actions to Store**

It's time to map our store dispatch to callback functions. Here's the change:

**src/index.js**

**const** mapDispatchToProps = dispatch => {

**return** {

handleIncrementClick: () => dispatch({ type: 'INCREMENT' }),

handleDecrementClick: () => dispatch({type: 'DECREMENT'})

}

};

**const** Container = connect(mapStateToProps, mapDispatchToProps)(Component);

We pass a second function called mapDispatchToProps to our connect function in the Container component. This function maps the dispatch function from the Redux store to the registered callbacks. These callbacks are named as the return object's property, and passed to the downstream component as props (handleIncrementClick and handleDecrementClick). Now, it should work! We can modify the state using the buttons!

**Step 9 (optional): Refactor the Code**

Let's move the similar code into separate files to keep the project tidy and maintained. Let's create a separate file for the the *Container* component, *Presentational* component, store initialization, and reducer. Also, let's put all the counter code into a single directory, because as the project grows, we will create other components with their own reducers, containers, and presentational components. The final directory structure should look like this:

src

├── configure-store.js

├── counter

│   ├── component.js

│   ├── container.js

│   └── reducer.js

├── index.html

└── index.js

And here's the code in each file:

**src/counter/component.js**

**import** React **from** 'react';

**export** **const** Component = ({ count, handleIncrementClick, handleDecrementClick }) => (

<div>

<h1>Helloworld React & Redux! {count}</h1>

<button onClick={handleDecrementClick}>Decrement</button>

<button onClick={handleIncrementClick}>Increment</button>

</div>

);

**src/counter/container.js**

**import** { connect } **from** 'react-redux';

**import** { Component } **from** './component';

**const** mapStateToPros = state => {

**return** {

count: state

};

};

**const** mapDispatchToProps = dispatch => {

**return** {

handleIncrementClick: () => dispatch({ type: 'INCREMENT' }),

handleDecrementClick: () => dispatch({ type: 'DECREMENT' })

}

};

**export** **const** Container = connect(mapStateToProps, mapDispatchToProps)(Component);

**src/counter/reducer.js**

**export** **const** countReducer = **function** (state = 0, action) {

**switch** (action.type) {

**case** "INCREMENT":

**return** state + 1;

**case** "DECREMENT":

**return** state - 1;

**default**:

**return** state;

}

};

**src/configure-store.js**

**import** { createStore } **from** 'redux';

**import** { countReducer } **from** './counter/reducer';

**export** **const** store = createStore(countReducer);

**src/index.js**

**import** React **from** 'react';

**import** ReactDOM **from** 'react-dom';

**import** { Provider } **from** 'react-redux';

**import** { store } **from** './configure-store';

**import** { Container } **from** './counter/container';

**const** App = () => (

<Provider store={store}>

<Container />

</Provider>

);

ReactDOM.render(<App />, document.getElementById('root'));