React Redux Basics

Hello World React Redux

STEP 1: create the react-redux directory w/ the *npx*

In the outer directory type: *npx create-react-app react_redux_initial_demo*

STEP 2: view the skeletal files:

Index.js

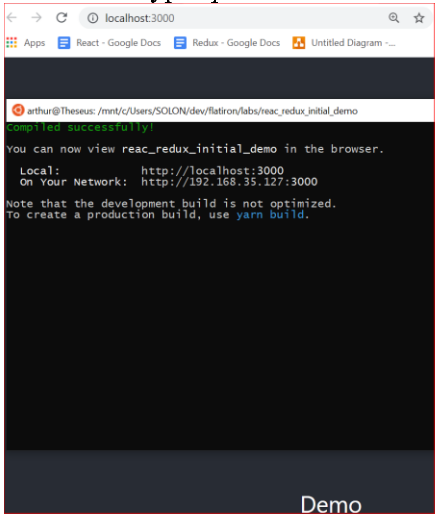
```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

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

App.js

```
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        Demo
      </header>
    </div>
  );
}

export default App;
```

STEP 3: view the running app

cd into the directory *react_redux_initial_demo* & then type *npm start* to view the demo page



```
Compiled successfully!

You can now view reac_redux_initial_demo in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.35.127:3000

Note that the development build is not optimized.
To create a production build, use yarn build.
```

STEP 4: install redux & react-redux

Normally, to install Redux into a React application, you need to install two packages, `redux` and `react-redux` by running `npm install redux && npm install react-redux`

STEP 5: Setup Store

We plan to use Redux to initialize our store and pass it down to our top-level container component.

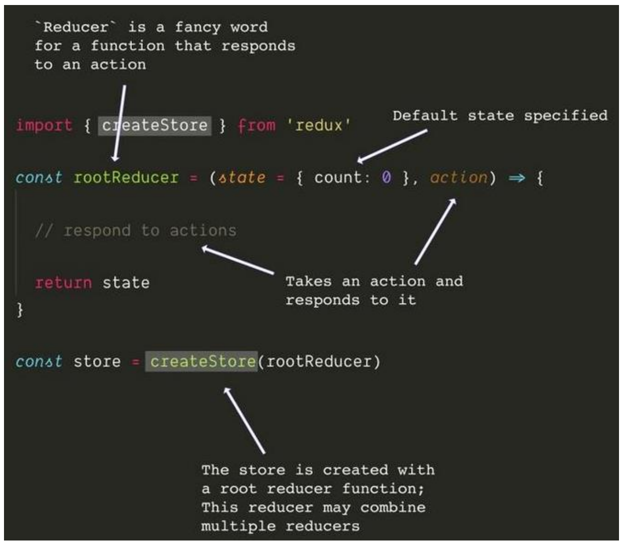
STEP 5.1: import createStore

Redux provides a function, *createStore()*, that, when invoked, returns an instance of the Redux store for us. So we can use that method to create a store.
We want to *import createStore()* in our *src/index.js* file, where *ReactDOM* renders our application.

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

Our application state lives in a central Redux store.
That store is created with a function called a *reducer*.

STEP 5.1.1: create a reducer



`Reducer` is a fancy word for a function that responds to an action

```
import { createStore } from 'redux'

const rootReducer = (state = { count: 0 }, action) => {

  // respond to actions

  return state
}

const store = createStore(rootReducer)
```

Default state specified

Takes an action and responds to it

The store is created with a root reducer function; This reducer may combine multiple reducers

A reducer takes in a *state* and an *action* and then returns the same state *or* returns a new state.
A reducer is just a fancy word for a function that modifies a state by acting upon an action.

```
const rootReducer =(state, action) =>{
// respond to action & return a new state or a new state
  return state;
}
```

STEP 5.2: initialize createStore

```
const store = createStore(rootReducer);
```

Our Index.js looks so:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { createStore } from 'redux';

const rootReducer =(state, action) =>{return state;}
const store = createStore(rootReducer);
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

STEP 6: Pass Store to the top-level Component

The *React Redux* library gives access to a component called the ***Provider***. The ***Provider*** is a component that comes from our *React Redux* library. It wraps around our **App** component. It does two things for us. The **first** is that it will alert our **Redux** app when there has been a change in state, and this will re-render our **React** app.

STEP 6.1: import Provider

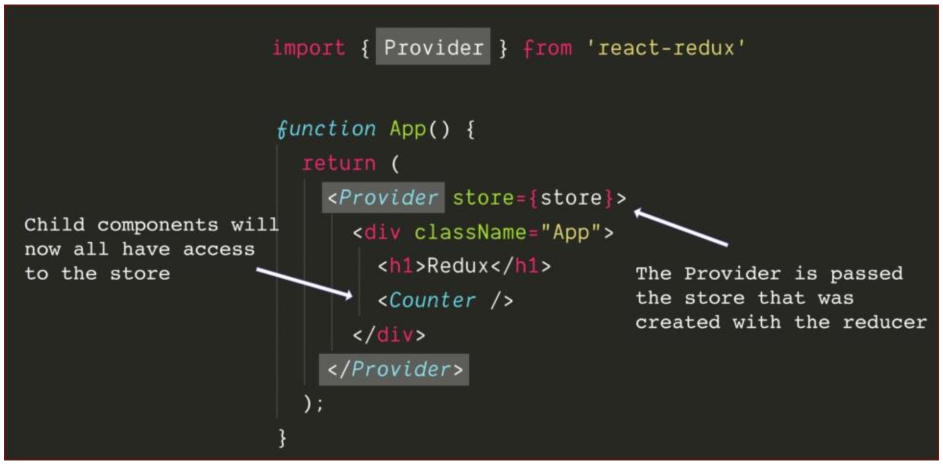
```
import { Provider } from 'react-redux';
```

STEP 6.2: Wrap App with Provider & pass Provider store obj

```
// added Provider to wrap around App
  <Provider store={store}>
      <App />
  </Provider> ,
```

The *index.js* file now looks so:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { createStore } from 'redux'; // this allows us to create store
import { Provider } from 'react-redux';
// we pass the store obj to top level container Provider

const rootReducer =(state, action) =>{
  return state;
}

const store = createStore(rootReducer);

// added Provider to wrap around App
ReactDOM.render(
  <Provider store={store}>
      <App />
  </Provider> ,
  document.getElementById('root')
);
```

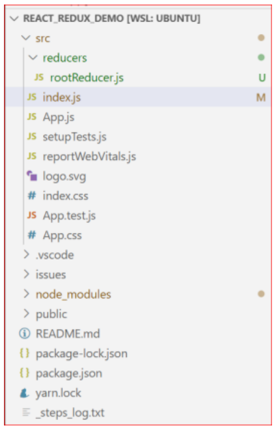


```
import { Provider } from 'react-redux'

function App() {
  return (
    <Provider store={store}>
      <div className="App">
        <h1>Redux</h1>
        <Counter />
      </div>
    </Provider>
  );
}
```

Child components will now all have access to the store
The Provider is passed the store that was created with the reducer

STEP 6.3 (optional): Have rootReducer abstracted out for OO

Now to make this cleaner and more OO we can extract *rootReducer* out and then place it in the *reducers* folder with the other reducers.



```
REACT_REDUX_DEMO [WSL: UBUNTU]
  src
    reducers
      rootReducer.js   U
    index.js   M
    App.js
    setupTests.js
    reportWebVitals.js
    logo.svg
    index.css
    App.test.js
    App.css
  .vscode
  issues
  node_modules
  public
  README.md
  package-lock.json
  package.json
  yarn.lock
  _steps_log.txt
```

In *reducers/rootReducer.js*

```
export default function rootReducer (state, action) {
    return state;
  }
```

In *Index.js* add:

```
import rootReducer from './reducers/rootReducer.js';
```

So now our index.js looks so:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { createStore } from 'redux'; // this allows us to create store
import { Provider } from 'react-redux';
// we pass the store obj to top level container Provider
import rootReducer from './reducers/rootReducer.js';

const store = createStore(rootReducer);

const rootElement =   document.getElementById('root')
// added Provider to wrap around App
ReactDOM.render(
  <Provider store={store}>
      <App />
  </Provider> ,
  rootElement
);
```

STEP 7: Get the data out of the Store/ State

Suppose our store has the data or variable *counter* which we plan to increment in our app from the browser.

So first we add it to the reducer since it is the function that modifies the state's data or variables by heeding the action parameter we pass to it. So now our *rreducers/reducer.js* looks so:

STEP 7.1: Define a variable /data in store

```
export default function rootReducer
(
    state =
    {
        count : 0
    }
    ,
    action
)
{
    return state;
}
```

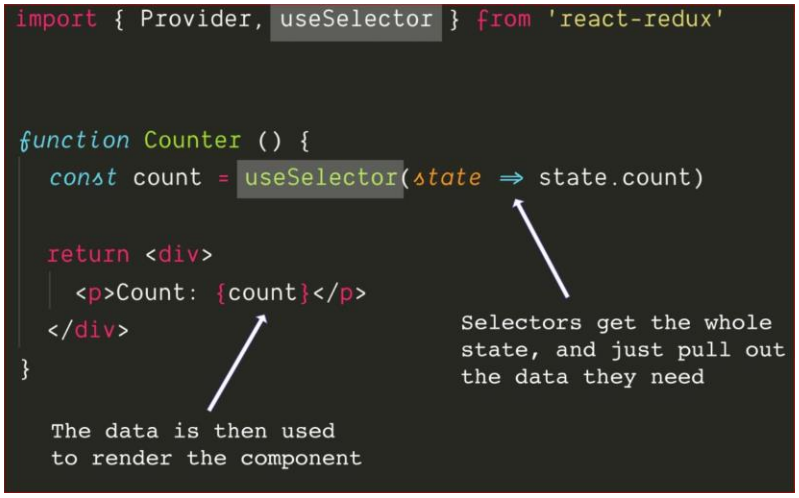
STEP 7.2: Use Selectors to extract variables from store

To get data out of store *useSelector* hook from react-redux.

'*Selector*' is just a fancy word for function that takes data out of store.

useSelector takes a *callback* which gegets the entire redux state & you just pick out what you need for that component.

```
import { useSelector } from 'react-redux';
```



```
import { Provider, useSelector } from 'react-redux'


function Counter () {
  const count = useSelector(state => state.count)

  return <div>
    <p>Count: {count}</p>
  </div>
}
```

Selectors get the whole state, and just pull out the data they need
The data is then used to render the component

```
function Counter () {
  const count = useSelector(state => state.count)
  return (
          <div>
            <p>Count: {count}</p>
          </div>
        );
}
```

Our *App.js* now looks so:

```
import './App.css';
import { useSelector } from 'react-redux';

function Counter () {
  const count = useSelector(state => state.count)

  return (
          <div>
            <p>Count: {count}</p>
          </div>
        );
}

function App() {
  return (
    <Counter />
  );
}

export default App;
```

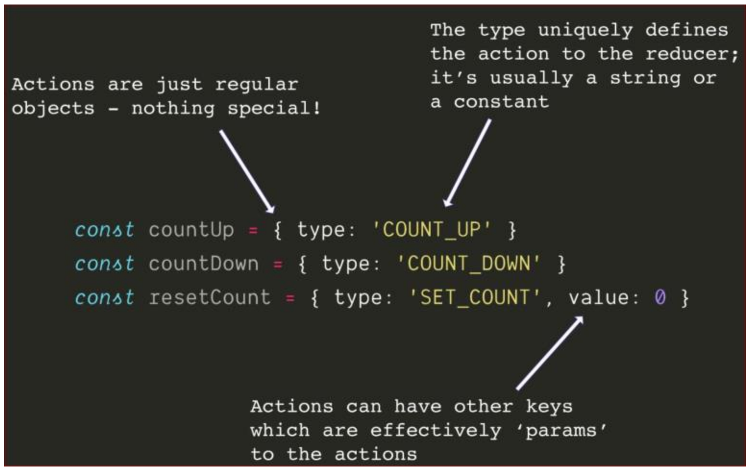
Step 8: Change the State/ Store

A changed state is returned by the *reducer* (function which acts upon a *state* by acting upon an *action*). To review this is what our reducer looks like:

```
export default function rootReducer
(
    state =
    {
        count : 0
    }
    ,
    action
)
{
    return state;
}
```

Step 8.1: Create an action

In order to modify the state the reducer function requires an action object.



```
const countUp = { type: 'COUNT_UP' }
const countDown = { type: 'COUNT_DOWN' }
const resetCount = { type: 'SET_COUNT', value: 0 }
```

Actions are just regular objects - nothing special!
The type uniquely defines the action to the reducer; it's usually a string or a constant
Actions can have other keys which are effectively 'params' to the actions

Actions are plain JS objects. All actions should have a "*type*" key. They must also have additional keys (*parameters*).

The key "*type*" uniquely defines the action to the reducer. It's usually a string or constant.

Actions can have other keys which are effectively *"params"* to the actions.

```
  const countUp = { type: 'COUNT_UP' }
```

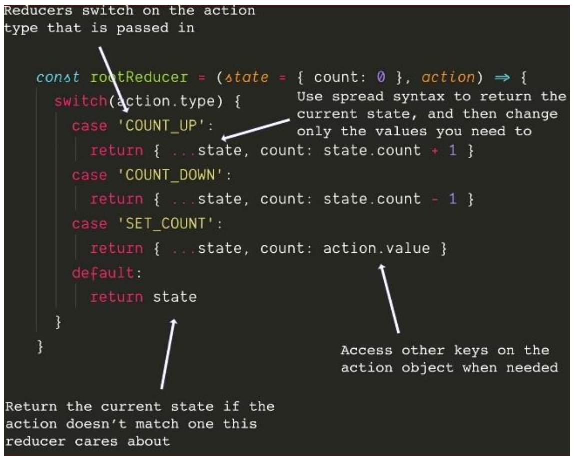
Actions are not called but "*dispatched*" to the reducers. The action "*type*" is what tells the reducer what to do (return a new state or the old one). Actions are objects sent to the store , where they are run through reducers.

```
import './App.css';
import { useSelector } from 'react-redux';

function Counter () {
  const count = useSelector(state => state.count)
  const countUp = { type: 'COUNT_UP' }
  return (
          <div>
            <p>Count: {count}</p>
          </div>
        );
}

function App() {
  return (
    <Counter />
  );
}

export default App;
```

Step 8.2: Write your reducer

To change the data in the store, first write your reducer.
Reducers are often written with switch case statements but don't have to be. They just have to take in an action and state

and return a new state.



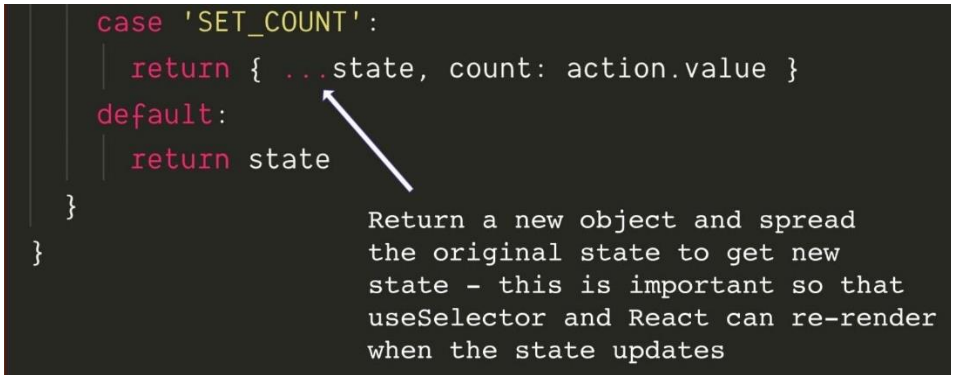
```
const rootReducer = (state = { count: 0 }, action) => {
  switch(action.type) {
    case 'COUNT_UP':
      return { ...state, count: state.count + 1 }
    case 'COUNT_DOWN':
      return { ...state, count: state.count - 1 }
    case 'SET_COUNT':
      return { ...state, count: action.value }
    default:
      return state
  }
}
```

Reducers switch on the action type that is passed in
Use spread syntax to return the current state, and then change only the values you need to
Access other keys on the action object when needed
Return the current state if the action doesn't match one this reducer cares about

In *reducers/rootReducer.js*

```
export default function rootReducer
(
    state =
    {
        count : 0
    }
    ,
    action
)
{
    if (action.type=="COUNT_UP"){
        return {...state, count: state.count +1};
    }
    return state;
}
```

It's **important** ***reducers return a NEW state object (and not mutate the old one) so that your components will re-render when something changes***. Don't set state values in reducers only ever return a new state object with changed values.



```
      case 'SET_COUNT':
        return { ...state, count: action.value }
      default:
        return state
    }
}
```

Return a new object and spread the original state to get new state - this is important so that useSelector and React can re-render when the state updates

Step 8.3: Dispatch an action to a reducer

Import :

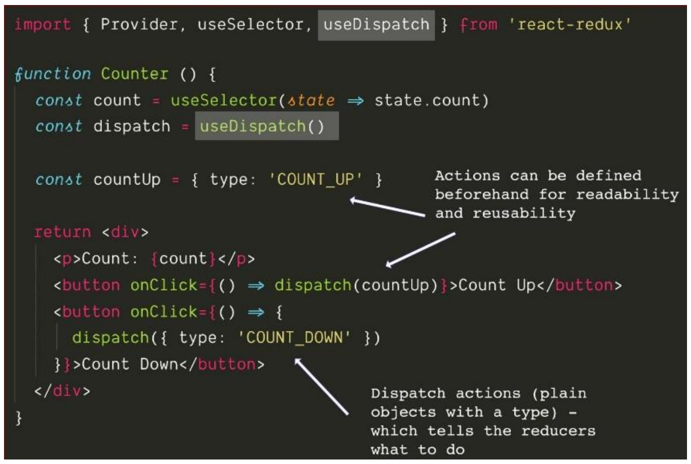
```
import { useSelector, useDispatch } from 'react-redux';
```

Then use a variable for dispatching actions:

```
  const dispatch = useDispatch();
```

Finally use it so in App.js:

```
import './App.css';
import { useSelector, useDispatch } from 'react-redux';

function Counter () {
  const count = useSelector(state => state.count);
  const countUp = { type: 'COUNT_UP' };
  const dispatch = useDispatch();

  return (
        <div>
          <p>Count: {count}</p>
          <button onClick={() => dispatch(countUp)}>Count Up</button>
        </div>
      );
}

function App() {
  return (
    <Counter />
  );
}

export default App;
```

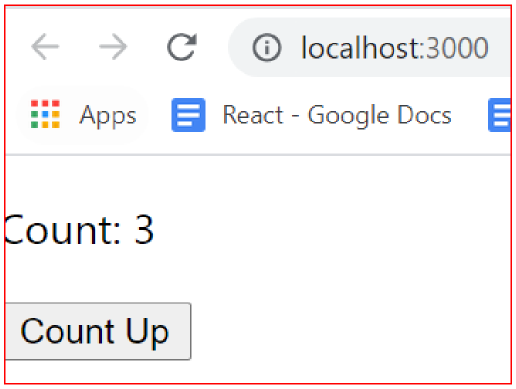


```
import { Provider, useSelector, useDispatch } from 'react-redux'

function Counter () {
  const count = useSelector(state => state.count)
  const dispatch = useDispatch()

  const countUp = { type: 'COUNT_UP' }

  return <div>
    <p>Count: {count}</p>
    <button onClick={() => dispatch(countUp)}>Count Up</button>
    <button onClick={() => {
      dispatch({ type: 'COUNT_DOWN' })
    }}>Count Down</button>
  </div>
}
```

Actions can be defined beforehand for readability and reusability

Dispatch actions (plain objects with a type) - which tells the reducers what to do

To dispatch an action, use the "*useDispatch*" hook from react-redux. Call "*useDispatch*" with an action object, which will run through the reducers, and will potentially change the state.



The Source

My source code *index.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { createStore } from 'redux'; // this allows us to create store
import { Provider } from 'react-redux';
// we pass the store obj to top level container Provider
import rootReducer from './reducers/rootReducer.js';

const store = createStore(rootReducer);

const rootElement =   document.getElementById('root')
// added Provider to wrap around App
ReactDOM.render(
  <Provider store={store}>
      <App />
  </Provider> ,
  rootElement
);
```

App.js

```
import './App.css';
import { useSelector, useDispatch } from 'react-redux';

function Counter () {
  const count = useSelector(state => state.count);
  const countUp = { type: 'COUNT_UP' };
  const dispatch = useDispatch();

  return (
          <div>
            <p>Count: {count}</p>
            <button onClick={() => dispatch(countUp)}>Count Up</button>
          </div>
        );
}

function App() {
  return (
    <Counter />
  );
}

export default App;
```

reducers/rootreducer.js

```
export default function rootReducer
(
    state =
    {
        count : 0
    }
    ,
    action
)
{
    if (action.type=="COUNT_UP"){
        return {...state, count: state.count +1};
    }
    return state;
}
```

The entirety of code in one monolithic piece:

```
import React from "react";
import ReactDOM from "react-dom";
import "./styles.css";

import { Provider, useSelector, useDispatch } from 'react-redux'

import { createStore } from 'redux'

const rootReducer = (state = { count: 0 }, action) => {
  switch(action.type) {
    case 'COUNT_UP':
      return { ...state, count: state.count + 1 }
    default:
      return state
  }
}

const store = createStore(rootReducer)

function Counter () {
  const count = useSelector(state => state.count)
  const dispatch = useDispatch()

  const countUp = { type: 'COUNT_UP' }

  return <div>
    <p>Count: {count}</p>
    <button onClick={() => dispatch(countUp)}>Count Up</button>
  </div>
}

function App() {
  return (
    <Provider store={store}>
      <div className="App">
        <h1>Redux</h1>
        <Counter />
      </div>
    </Provider>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

References:

Learn - Intro To Redux Library Codealong
React Redux in 10 tweets Source Code by Chris Richards [1]