Redux and React: An Introduction

What is redux?

Redux is a flux based state container for handling javascript application state. It is a popular choice for storing application state mainly due to its three defining principles:

- A single object tree stores all of your application state
- State is readonly and changes are triggered by actions
- The state can only be manipulated by pure functions that are triggered by your actions

Actions

An action is simply an object that describes a change you want to make to your state. These are somewhat similar to event objects.

A standard pattern for actions is the following structure:

```
const action = {
  type: 'ACTION_TYPE',
  payload: 'Some data'
};
```

The type is similar to an event type and is required for all actions, and the payload is the data that will be used to transform our state.

Not all actions need a payload though, as some actions like incrementing a number do not require any additional data e.g.

```
const increment = {
  type: 'INCREMENT'
};
```

Action Creators

Action creators are simply a function that allow us to abstract away the creation of actions, allowing us to easily dispatch an action without having to define all of its properties.

Here’s an example of a simple action creator:

```
export const ADD_NUMBER = 'ADD_NUMBER';

export const addNumber = (number) => ({
  type: ADD_NUMBER,
  payload: number
});
```

Now we can use this later to quickly create an action with some additional data attached to it e.g.

```
const action = addNumber(7);
```

Reducers

A reducer is the pure function that we will use to transform our store state. Reducers are triggered whenever an action is dispatched and receive both the current state of that reducer (which will be undefined to begin with) and the action that was dispatched.

Here’s a simple example of a reducer that keeps track of a number and handles the “add number” action that we defined above.

```
import { ADD_NUMBER } from './our-actions-file';

export const count = (state = 0, action) => {
  switch (action.type) {
    case ADD_NUMBER:
      return state + action.payload;
    default:
      return state;
  }
};
```

There are several things are important to understand when defining a reducer:

- Our state will be undefined to begin with, so we'll want to give this a sensible default value (0 in this case)
- Our reducer cannot return an undefined value
- Our reducer will be triggered by any action that is dispatched, so we should return the existing state if the action is not relevant to this reducer (that's what the default case is for).

Creating A Store

Now that you understand the basics of actions and reducers we can actually put them to use and create a store.

In this case we are going to create a simple store that only contains our single "count" reducer.

```
import { createStore } from 'redux';
import { count } from './our-reducers-file';

export const store = createStore(count);
```

With this example our "count" reducer will make up the entirety of our store state, so calling the method `store.getState()` will simply return a number. Let's talk a bit about some of the available store methods…

Store Methods

We wont actually need to call any of these methods ourselves, (and I'd actually avoid this at all costs), as the tools we'll cover shortly will handle this for us, but for the purposes of describing how the store composes our state, and how actions are dispatched it's important to cover briefly.

getState

`store.getState()` is pretty self explanatory - it simply returns the current state of the store.

dispatch

`store.dispatch()` is the method that is used to dispatch an action and subsequently trigger our reducers.

If we were to manually dispatch our “add number” action we would do so in the following way:

```
store.dispatch(addNumber(7));
```

This would cause our “count” reducer to then be called with the current store state and our “add number” action.

Note that we are not passing the action creator itself to the dispatch function, but instead the action that is returned by it.

Combining Reducers

For most applications we are going to want to store more than a single number, which we can then access from an object tree. In order to save us a lot of hassle handling all of the store state in a single reducer we can use a function provided by redux to combine our reducers into an object tree.

```
import { combineReducers, createStore } from 'redux';
import { count, someOtherReducer } from './our-reducers-file';

export const store = createStore(combineReducers({
  count,
  someOtherReducer
}));
```

What this is actually doing behind the scenes is creating another function that calls all our our reducers with the state that is relevant to them. It’s basically like a magical parent reducer.

If we were to over simplify how this works it’d look something like the following:

```
const combineReducers = (reducers) => {
  return (state = {}, action) => {
    const newState = {};

    for (let key in reducers) {
      const reducer = reducers[key];

      newState[key] = reducer(state[key], action);
    }

    return newState;
  };
}
```

Note how all of the reducers are called with the same action.

After we combine our reducers, calling `store.getState()` would return something like this:

```
const state = {
  count: 0,
  someOtherReducer: 'Some value'
};
```

This allows use to access each of our reducers state individually e.g.

```
const count = state.count;
```

But this isn't exactly how we'll be doing things. As I mentioned before, we wont be manually calling `getState` or `dispatch`.

Provider

Here's where we start to integrate redux with our react application.

To do so we'll also need to install a module called react-redux.

React redux provides several tools that allow us to easily access store state and dispatch actions from our react components.

The provider is a react component that sits at the root level of your app, and allows any of its children access to the store (which we supply to it as a prop) via context and the `connect` function (which we'll get to in a second).

```
import React from 'react'
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { store } from './our-store-file';
import Counter from './somewhere-else';

const App = () => (
  <Provider store={store}>
    <Counter />
  </Provider>
);

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

You can also manually supply the store to a connected component (which we'll cover in a second), which is useful when testing components. I would not, however, recommend giving any of your components direct access to the store in your application.

Connecting A Component

The final piece of the puzzle is the `connect` function provided by react-redux. This allows us to map parts of the store state to a component, and at the same time, automatically wrap our actions with the `dispatch` method so that we don't have to worry about calling it ourselves.

The main benefits of using a provider with this `connect` function are that an application can be provided an entirely different store when needed, which is very useful for server side rendering, but I wont be covering that today.

Here's a simple component that allows us to display a number and add to it:

```
import React, { Component } from 'react';
import { addNumber } from './our-actions-file';

class Counter extends Component {
  onAddClick = () => {
    this.props.addNumber(7);
  }

  render () {
    return (
      <div>
        Count: {this.props.count}
        <button onClick={this.onAddClick}>
          Add 7!
        </button>
      </div>
    );
  }
}
```

Right now, if this component was rendered like in the above provider example, it wont have access to either of the props `count` or `addNumber` as we are not providing them, but after we connect the component, these state values and actions will be mapped to its props.

Here we're going to connect our component:

```
const mapStateToProps = ({count}) => ({
  count
});

export default connect(mapStateToProps, { addNumber })(Counter);
```

Let's break this down a bit…

Firstly, `connect` is a function that returns another function. Connect takes 2 optional arguments; `mapStateToProps`, and `mapDispatchToProps`; and returns a function that takes our component as an argument.

So what are `mapStateToProps` and `mapDispatchToProps`?

mapStateToProps

`mapStateToProps` is a function that will be called when our component mounts, updates, or our store state is changed. All this does is extract the state that we want from the store and return it as an object. The `connect` function then provides these values as props to our component so that we can access them with `this.props.count` for example.

mapDispatchToProps

`mapDispatchToProps`, which in this case is simply an object containing our action creator (but can also be a function that allows you to do some more complex stuff), wraps each of our actions with `dispatch` so that when called with `this.props.addNumber(7)`, for example, automatically dispatches our action. Similarly to `mapStateToProps`, the `connect` function provides these values to our component so they can be accessed as props.

An over simplified example of what happens to our `mapDispatchToProps` behind the scenes would look something like this:

```
const mapDispatchToProps = (actionCreators) => {
  const dispatchedActionCreators = {};

  for (let key in actionCreators) {
    const actionCreator = actionCreators[key];

    dispatchedActionCreators[key] = (...args) => {
      // For the purposes of this example `dispatch` magically comes out of nowhere
      dispatch(actionCreator(...args));
    };
  }

  return dispatchedActionCreators;
};
```

An alternative to using a `mapDispatchToProps` function or object, is to provide nothing e.g.

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

This might seem like an odd thing to do at first, but if we do not provide dispatch props then the store's `dispatch` method is automatically provided as a prop, so we can manually dispatch actions.

I know I said that we shouldn't need to call dispatch directly on the store ourselves, but in this case it is fine because we are not accessing the store directly. It's being provided by the `connect` function.

Some may prefer this approach as you can avoid shadowing variable names when destructuring actions from props that are also imported, which can occasionally result in calling the wrong function, like in the following example:

```
import React, { Component } from 'react';
import { addNumber } from './our-actions-file';

class Counter extends Component {
  onAddClick = () => {
    const { addNumber } = this.props;

    // This shares a variable name with the imported action creator
    // Sometimes the action creator may be accidentally called instead of the
dispatch version
    addNumber(7);
  }

  render () {
    return (
      <div>
        Count: {this.props.count}
        <button onClick={this.onAddClick}>
          Add 7!
        </button>
      </div>
    );
  }
}
```

```
export default connect(mapStateToProps, { addNumber })(Counter);
```

An example that uses the `dispatch` prop rather than `mapDispatchToProps`:

```
import React, { Component } from 'react';
import { addNumber } from './our-actions-file';

class Counter extends Component {
  onAddClick = () => {
    // Now we are directly referencing our imported action creator
    // And manually dispatching it with the dispatch prop provided by connect
    this.props.dispatch(addNumber(7));
  }

  render () {
    return (
      <div>
        Count: {this.props.count}
        <button onClick={this.onAddClick}>
          Add 7!
        </button>
      </div>
    );
  }
}

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