Redux.js

How Redux Works

leandro.frigerio@devugees.org

Contents

By using Redux we’ll implement a central data store in our application. The store contains the state of the application and is the source of truth for component “Counter”.

Reimplement the state management of our Counter App using Redux

- Import Redux **in Counter.js**
- Add MapStateToProps function **in Counter.js**
- Connect the component to Redux **in Counter.js**
- Create the Store **in App.js**
- Create the Reducer **in App.js**
- Provide the Store **in App.js**
- Create the actions **in App.js**
- Dispatch the actions **in Counter.js**

How to Redux

After learning a bit about React and getting into Redux, can be really confusing understand how it all works.

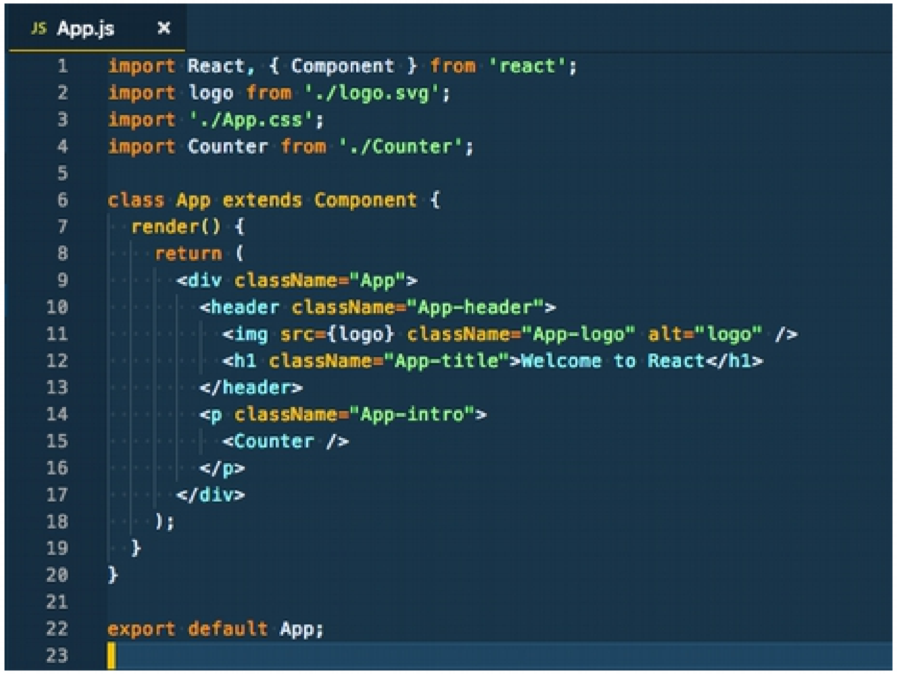
Actions, reducers, action creators, middleware, pure functions, immutability… check

Quick Review:

- the Redux **store** is like a brain: it's in charge for orchestrating all the moving parts in Redux
- the **state** of the application lives as a single, immutable object within the store
- as soon as an **action** is dispatched the reducer is called
- the **reducer** accept a state and an action and returns the **next state**

Example: no Redux but plain State management

We create an example app with create-react-app and we implement a simple counter

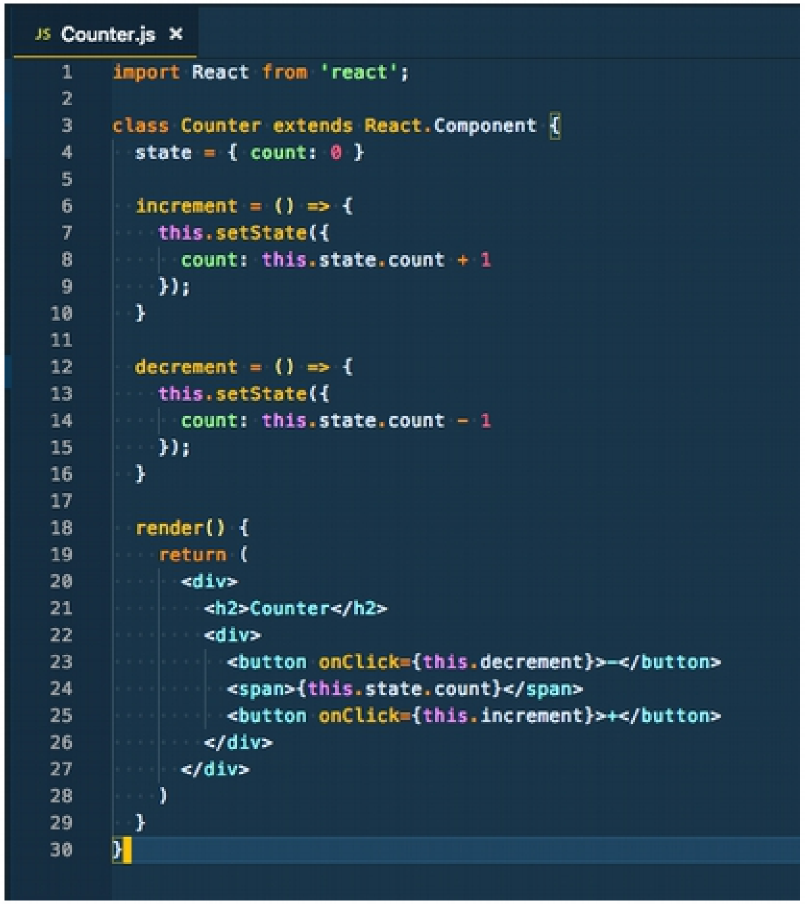


App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          <Counter />
        </p>
      </div>
    );
  }
}

export default App;
```

Example: Counter.js

- The count state is stored in the top level Counter component
- When the user clicks “+”, the button’s onClick handler is called, which is bound to the increment function in the Counter component.
- The increment function updates the state with the new count.
- Because state was changed, React re-renders the Counter component (and its children), and the new counter value is displayed.



```
import React from 'react';

class Counter extends React.Component {
  state = { count: 0 }

  increment = () => {
    this.setState({
      count: this.state.count + 1
    });
  }

  decrement = () => {
    this.setState({
      count: this.state.count - 1
    });
  }

  render() {
    return (
      <div>
        <h2>Counter</h2>
        <div>
          <button onClick={this.decrement}>-</button>
          <span>{this.state.count}</span>
          <button onClick={this.increment}>+</button>
        </div>
      </div>
    )
  }
}
```

Planning: what I'll do

- Import Redux **in Counter.js**
- Add MapStateToProps function **in Counter.js**
- Connect the component to Redux **in Counter.js**
- Create the Store **in App.js**
- Create the Reducer **in App.js**
- Provide the Store **in App.js**
- Create the actions **in App.js**
- Dispatch the actions **in Counter.js**

Install Redux

- Install Redux

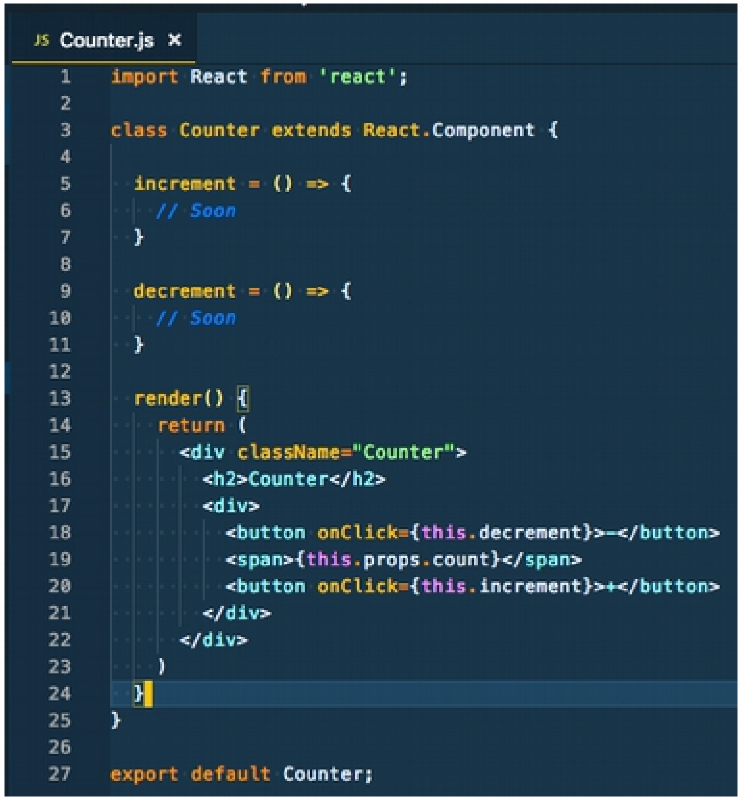
  yarn add redux react-redux

What are we doing? Redux keeps the state of your app in a single store.
Then, you can extract parts of that state and plug it into your components as props. This lets you keep data in one global place (the store) and feed it directly to any component in the app, without the gymnastics of passing props down multiple levels.

Side note: you'll often see the words "state" and "store" used interchangably.
Technically, the state is the data, and the store is where it's kept.

Remove the state management

- Changes in the Counter.js
- Remove the state management I'm going to replace with Redux
- Translate state in props (note: this won’t work yet, of course, because the Counter is not receiving a count prop. We’re gonna use Redux to inject that.)



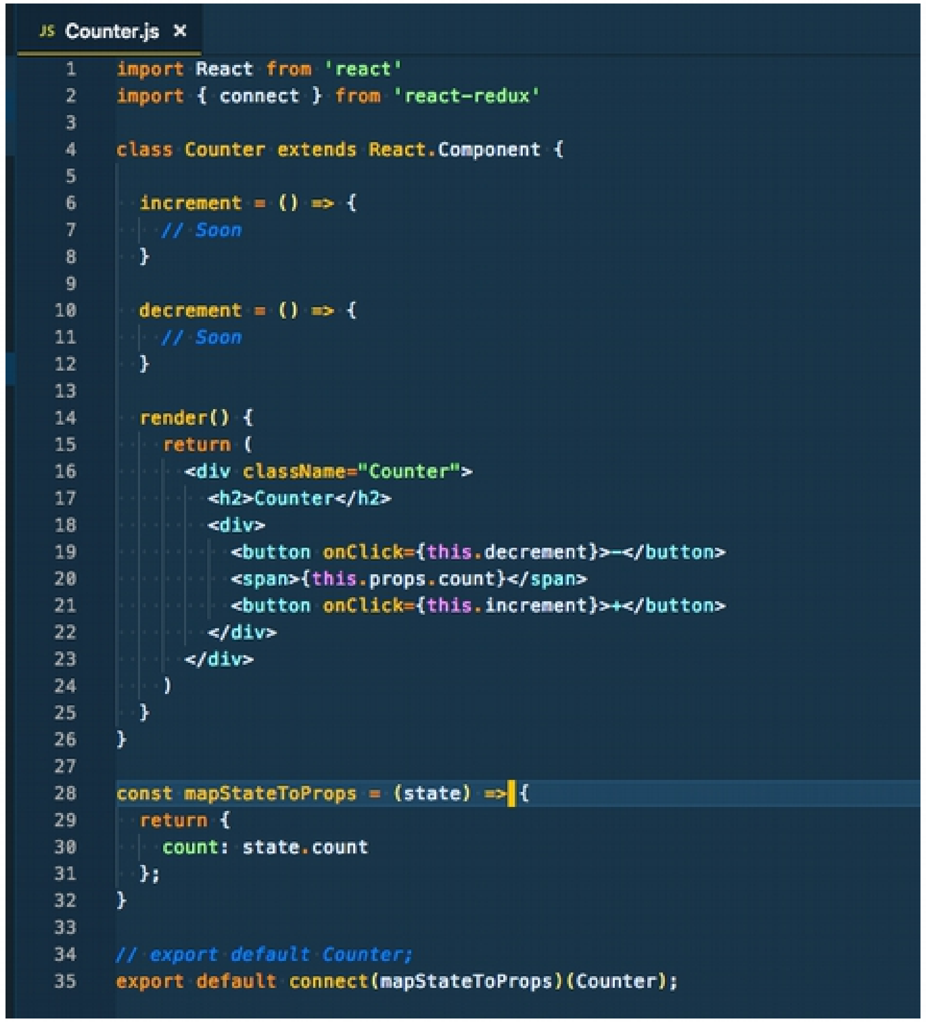
Counter.js ×

```
import React from 'react';

class Counter extends React.Component {

  increment = () => {
    // Soon
  }

  decrement = () => {
    // Soon
  }

  render() {
    return (
      <div className="Counter">
        <h2>Counter</h2>
        <div>
          <button onClick={this.decrement}>-</button>
          <span>{this.props.count}</span>
          <button onClick={this.increment}>+</button>
        </div>
      </div>
    )
  }
}

export default Counter;
```

Add Redux

- import { connect } from 'react-redux'
- Add mapStateToProps function
- Connect the component with Redux

This will fail with an error: we're tyring to connect the Counter but we haven't yet initialized the Store and wrapped the root component in a Provider. Since connect pulls data from the Redux store, and we haven’t set up a store or told the app how to find it, this error is pretty logical. Redux has no dang idea what’s going on right now.

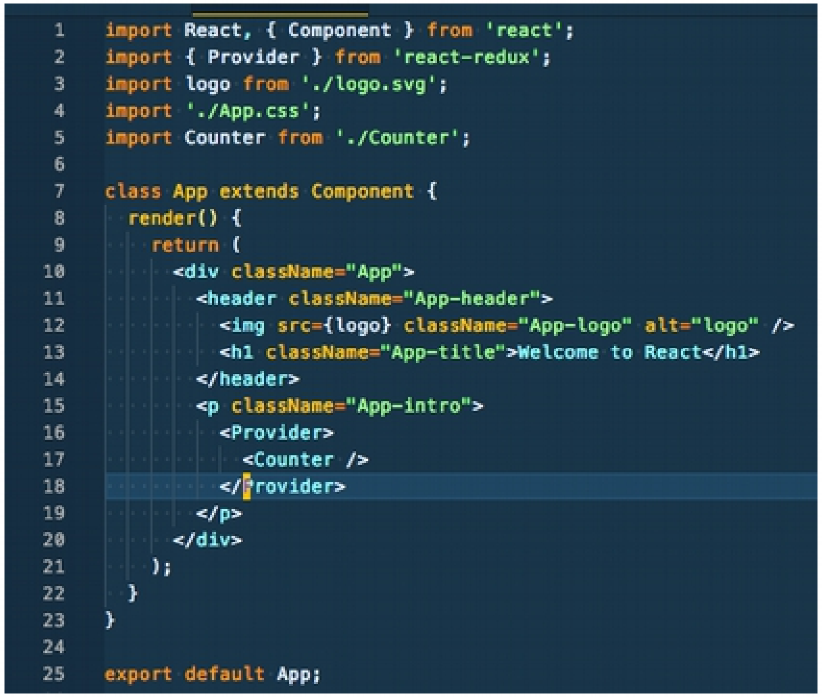


```
import React from 'react'
import { connect } from 'react-redux'

class Counter extends React.Component {

  increment = () => {
    // Soon
  }

  decrement = () => {
    // Soon
  }

  render() {
    return (
      <div className="Counter">
        <h2>Counter</h2>
        <div>
          <button onClick={this.decrement}>-</button>
          <span>{this.props.count}</span>
          <button onClick={this.increment}>+</button>
        </div>
      </div>
    )
  }
}

const mapStateToProps = (state) => {
  return {
    count: state.count
  };
}

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

Could not find "store" in either the context or props of "Connect(Counter)". Either wrap the root component in a <Provider>, or explicitly pass "store" as a prop to "Connect(Counter)".

Provide a Store

Redux holds the global state for the entire app, and by wrapping the entire app with the **Provider** component from react-redux, every component in the app tree will be able to use connect to access the Redux store if it wants to.

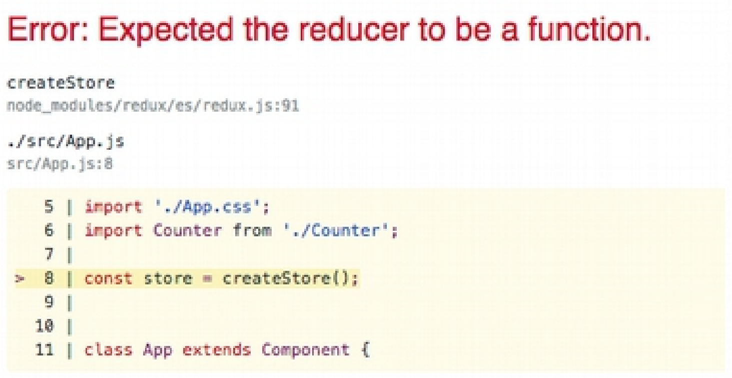


```
import React, { Component } from 'react';
import { Provider } from 'react-redux';
import logo from './logo.svg';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          <Provider>
            <Counter />
          </Provider>
        </p>
      </div>
    );
  }
}

export default App;
```

Create the Store

Redux comes with a handy function that creates stores, and it’s called createStore. Yep. Let’s make a store and pass it to Provider:

- import { createStore } from 'redux'
- const store = createStore()
- <Provider store={store}>



```
Error: Expected the reducer to be a function.

createStore
node_modules/redux/es/redux.js:91

./src/App.js
src/App.js:8

   5 | import './App.css';
   6 | import Counter from './Counter';
   7 |
>  8 | const store = createStore();
   9 |
  10 |
  11 | class App extends Component {
```

Another error, but different this time: Redux: it’s not very smart. You might expect that by creating a store, it would give you a nice default value for the state inside that store. Maybe an empty object? But no: Redux makes zero assumptions about the shape of your state.

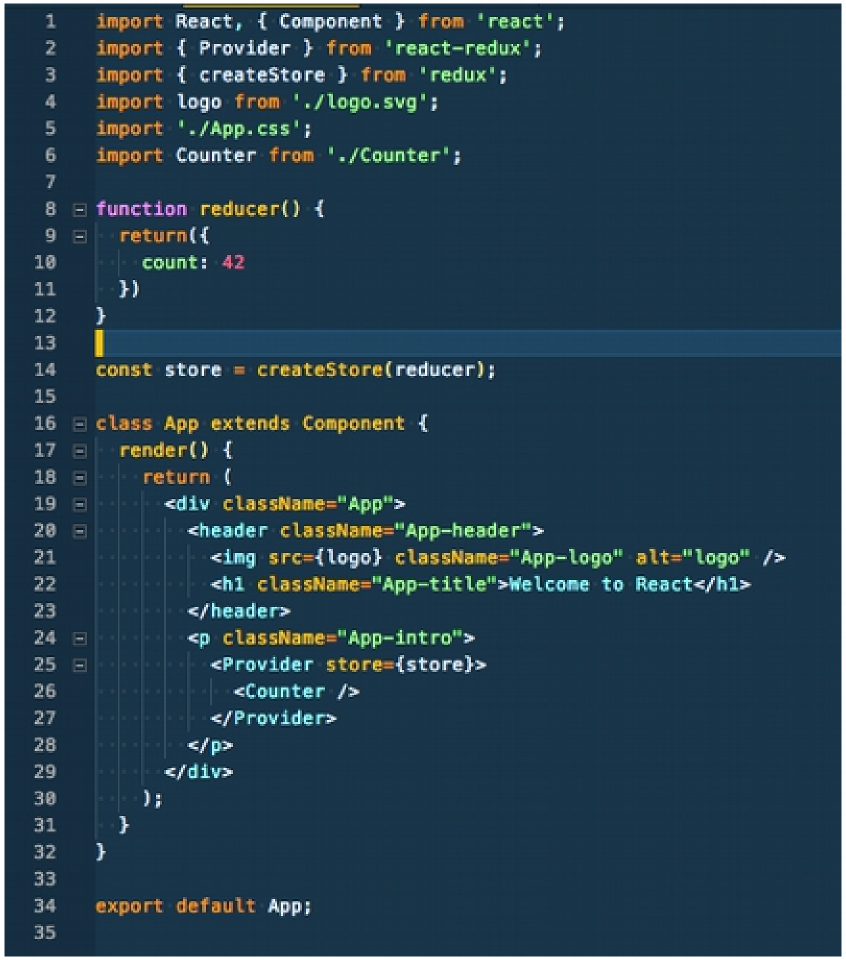
Create a Reducer

We have to provide a function that will return the state: a reducer

The reducer is expected to return the state. It's actually supposed to take the current state and return the new state: we'll implement it soon.

Now the App is working again… it's not doing so much but no error.

So far so good!



```
import React, { Component } from 'react';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import logo from './logo.svg';
import './App.css';
import Counter from './Counter';

function reducer() {
  return({
    count: 42
  })
}

const store = createStore(reducer);

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          <Provider store={store}>
            <Counter />
          </Provider>
        </p>
      </div>
    );
  }
}

export default App;
```

The Story So Far

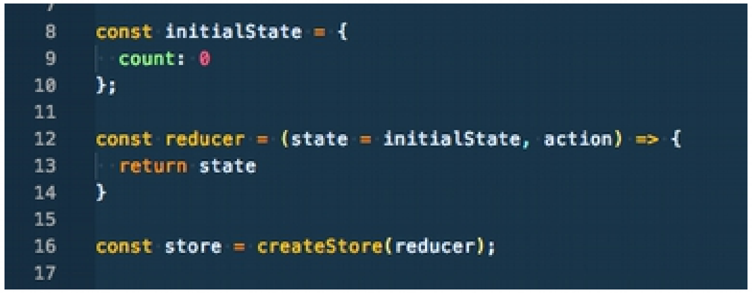
- We wrote a **mapStateToProps** function that does what the name says: transforms the Redux state into an object containing props.
- We connected the **Redux** store to our Counter component with the connect function from react-redux, using the **mapStateToProps** function to configure how the connection works.
- We created a **reducer** function to tell Redux what our state should look like.
- We used the ingeniously-named **createStore** function to create a store, and passed it the reducer.
- We wrapped our whole app in the **Provider** component that comes with react-redux, and passed it our store as a prop.
- The app works flawlessly, except the fact that the counter is stuck at **42**.

Implement the reducer

We need to properly implement the reducer we just created:

- the reducer takes the current state and an action, and then it returns the new state:
- the very first time Redux calls this function, it will pass undefined as the state. That is your cue to return the initial state. (console.log to debug it!)

  It’s common to write the initial state above the reducer, and use ES6’s default argument feature to provide a value for the state argument when it’s undefined.

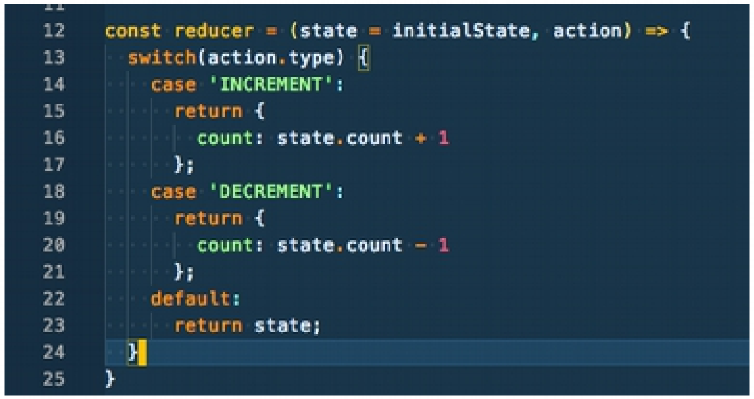


```
const initialState = {
  count: 0
};

const reducer = (state = initialState, action) => {
  return state
}

const store = createStore(reducer);
```

Actions

An “action” is a JS object that describes a change that we want to make.
The only requirement is that the object needs to have a **type** property, and its value should be a string.

Tthe **reducer**’s job is to take the current state and an action and figure out the new state. So if the reducer received an action like **{ type: "INCREMENT" }**, what might you want to return as the new state?



```
const reducer = (state = initialState, action) => {
  switch(action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1
      };
    case 'DECREMENT':
      return {
        count: state.count - 1
      };
    default:
      return state;
  }
}
```

Always Return a State

You’ll notice that there’s always the fallback case where all it does is return state. This is important, because Redux can (will) call your reducer with actions that it doesn’t know what to do with. In fact, the very first action you’ll receive is **{ type: "@@redux/INIT" }**. Try putting a console.log(action) above the switch and see.

Remember that the reducer’s job is to return a new state, even if that state is unchanged from the current one. You never want to go from “having a state” to “state = undefined”, right? That’s what would happen if you left off the default case. Don’t do that.

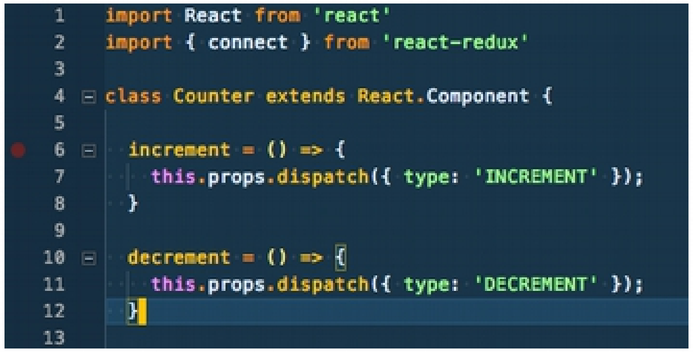
NO WAY: state.count++ OR state.count--
You also can’t do things like state.foo = 7, or state.items.push(newItem), or delete state.something.

Where Do Actions Come From?

Now, we need to use the actions we implemented in the switch of the reducer. The Actions are **dispatched**, with a handy function called **dispatch**.

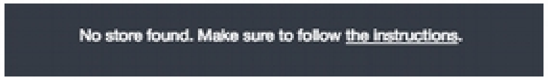
The dispatch function is provided by the instance of the Redux store. That is to say, you can’t just import { dispatch } and be on your way.

As luck would have it, the connect function has our back. In addition to injecting the result of **mapStateToProps** as props, connect also injects the dispatch function as a prop.

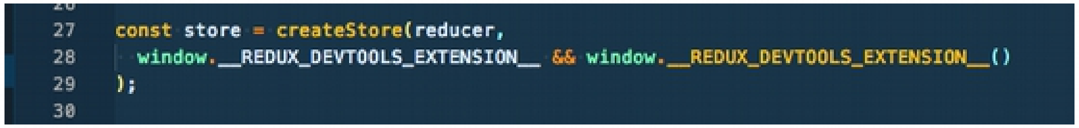


```
import React from 'react'
import { connect } from 'react-redux'

class Counter extends React.Component {

  increment = () => {
    this.props.dispatch({ type: 'INCREMENT' });
  }

  decrement = () => {
    this.props.dispatch({ type: 'DECREMENT' });
  }

```

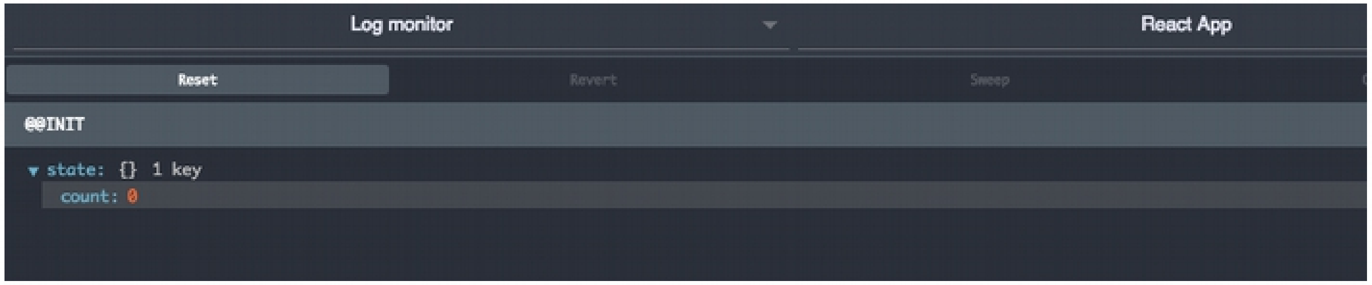
Devtools!


No store found. Make sure to follow the instructions.

Now the counter is working with Redux, we want to implement the DevTools: in the createStore we'll add


```
const store = createStore(reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

And we'll see it working...



```
@@INIT
▾ state: {} 1 key
    count: 0
```

Using the Store

Beside the counter we want to now how many time the user clicks:

- We'll add clicks to the Store
- We'll debug it with the Redux Devtools
- We'll publish the value near to the counter

Addon

If you want to replace the default SVG of create-react-app with the Redux SVG, you'll find here what you are looking for

https://raw.githubusercontent.com/reactjs/redux/master/logo/logo.svg